Query Processing for Composite Objects

1993

Hanxiong CHEN

95301562
Abstract

This dissertation, entitled "Query Processing for Composite Objects", is a contribution on query processing in advanced database systems.

In order to support advanced applications effectively, many new data management concepts, such as the object-oriented and extensible database models, the relational extension model, etc., have been developed in the last decade. These concepts offer very promising prospects for the modeling and handling of collections of advanced application objects, and so on. Associated with these new concepts, the problems which have arisen from the view point of query processing are,

- How to optimize ADT/user-defined functions?
- How to retrieve composite objects more efficiently?

To address these problems, two types of approach are proposed in this thesis. One is to optimize a query including computationally "expensive" ADT functions. A new cost model is developed, and several algorithms are designed and analyzed based on this cost model. The second approach involves new buffer allocation strategies to reduce the I/O cost in traversing composite objects, by dynamically using the references among the objects. This thus enhances the performance of retrieving composite objects. The result of qualitative analysis is confirmed in the simulation.
Contents

1 Introduction

1.1 Overview and Contributions .................................. 1

1.2 Related Works .................................................. 7

1.2.1 Advanced Data Models ...................................... 7

1.2.2 Buffer Management in DBMS ................................ 9

1.2.3 Query Processing .......................................... 10

1.3 Organization .................................................... 11

2 Query Processing in Advanced Applications .................. 13

2.1 Support ADT in a Relational Environment .................. 13

2.1.1 Queries including ADT functions ........................... 14

2.1.2 Traditional Optimization ................................... 15

2.2 Composite Objects ............................................ 17

2.2.1 Querying Composite Objects ................................. 18

2.2.2 Buffering: Review .......................................... 19

2.2.3 Clustering: Techniques and Disadvantages .............. 22
<table>
<thead>
<tr>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.4 Prefetching and Caching: Techniques and Disadvantages</td>
</tr>
<tr>
<td>2.3 Summary</td>
</tr>
<tr>
<td>3 Optimizing Queries Including ADT Functions</td>
</tr>
<tr>
<td>3.1 Motivation and Overview</td>
</tr>
<tr>
<td>3.2 The Decomposition Approach</td>
</tr>
<tr>
<td>3.2.1 Terminologies</td>
</tr>
<tr>
<td>3.2.2 Cost Model</td>
</tr>
<tr>
<td>3.2.3 Algorithm</td>
</tr>
<tr>
<td>3.2.4 Example</td>
</tr>
<tr>
<td>3.3 The Approach Using (Semi)-Join</td>
</tr>
<tr>
<td>3.3.1 Cost Model and Algorithm</td>
</tr>
<tr>
<td>3.3.2 General Discussion</td>
</tr>
<tr>
<td>3.4 Summary</td>
</tr>
<tr>
<td>4 New Buffer Allocation Approaches</td>
</tr>
<tr>
<td>4.1 Introduction</td>
</tr>
<tr>
<td>4.2 Access Patterns</td>
</tr>
<tr>
<td>4.3 LRU-S</td>
</tr>
<tr>
<td>4.3.1 Terminology</td>
</tr>
<tr>
<td>4.3.2 LRU-S Algorithm</td>
</tr>
<tr>
<td>4.4 Formal Analysis</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Example of Composite Object. ...................... 19
2.2 An Example of Buffer Performance \((m = n = 10)\)  21
2.3 An Example of Buffer Performance (pure loop). ... 22
3.1 The Image of Decomposition. ....................... 30
3.2 Example Relations. ................................. 32
3.3 Result of Join and Semijoin. ........................ 33
3.4 Selectivities and False Drops. ...................... 37
3.5 The Search Space. ................................. 41
3.6 An Example of Decomposition. ...................... 44
3.7 The Three Methods. ................................. 46
3.8 Expensive ADT Function, Low Image Size. .......... 48
3.9 Expensive ADT Function, Medium Image Size ....... 49
3.10 Expensive ADT Function, High Image Size .......... 50
3.11 Cheap ADT Function. .............................. 51
4.1 Examples of Access Patterns. ...................... 62
4.2 An Example of LRU-S. .............................. 68
LIST OF FIGURES

4.3 An Unusual Phenomenon of LRU-S. . . . . . . . . . 74
4.4 An Example of the Data Set . . . . . . . . . . . . . . . . 76
4.5 Loop of $Acs_r.Acs_s$ ($|Acs_s| = 30, |Acs_r| = 25$) . . . . 77
4.6 Loop of $Acs_r.Acs_s$ ($|Acs_s| = 30, |Acs_r| = 40$) . . . . 78
4.7 Loop of $Acs_r.Acs_s$ ($|Acs_s| \approx 45, |Acs_r| \approx 30$) . . . 79
4.8 Loop of $Acs_r.Acs_s$ ($|Acs_s| \approx 14, |Acs_r| \approx 9$) . . . . 80
4.9 Loop of $Acs_r.Acs_s$ ($|Acs_s| \approx |Acs_r| \approx 18$) . . . . 81
4.10 Looping Access (Shared 30%, Number of nodes \approx 22) 82
4.11 Looping Access (Shared 50%, Number of nodes \approx 40) 83
4.12 Looping Access (Shared 50%, Number of nodes \approx 40) 84
4.13 Random Access . . . . . . . . . . . . . . . . . . . . . . . . . 85

5.1 An Example of Storage of Composite Objects. . . . . 92
Chapter 1

Introduction

1.1 Overview and Contributions

Traditional database systems, especially relational database systems, have proved to be successful in supporting standard data processing applications. Relational database systems have provided users with a modeling methodology independent of the details of physical implementation [Astr76, Codd70, Date83, Date86a, Date86b, Maie85, Ullm88, Ullm89]. However, as pointed out by many researchers [Hard87b, Hugh91, Kemp87b, Kent79, Lori82, Spon84, Maie89] involved in database system design, relational database management systems are not designed and implemented to support advanced applications in a straightforward fashion. For example, they cannot manage CAD objects as whole database entities. The scheme design in the relational database system is imposed by the CAD data model and it is not powerful enough to constitute a natural mapping from CAD objects onto tuples or relations in the relational database [Kemp87a].
To support advanced applications such as CAD, many different data management concepts have been studied in recent years. Several famous concepts include the following:

- Semantic Data Models
- Object-Oriented Models
- Extensible DBMSs
- Relational Extensions

It has been pointed out that these paradigms must have many common functions, such as data models for composite objects, programming languages that support permanence, object management systems, and so on. The various aspects of these paradigms are summarized in Appendix B. Here, concentrating on query processing, we are interested in the following common properties of these paradigms:

- Support of new data types hence new operators
- Processing of composite objects

A system which supports abstract data types (ADT) [Gutt78] which may be user-defined raises many problems such as how to express and implement these types and the functions associated with them. From the viewpoint of query processing, how to optimize a query including such functions remains a problem. This is because the encapsulation of ADT/user-defined data types keeps the information from the query processor, and thus makes it difficult to
1.1. OVERVIEW AND CONTRIBUTIONS

optimize queries including such functions. Similarly, query processing sub-system in systems treating composite objects are still not adequately optimized: for applications which involve a great deal of associative searching for objects in secondary memory, such functions have difficulty even keeping up with the performance achieved by, for example, relational DBMSs. Yet it is essential that object-oriented systems will yield at least the same performance achieved by relational systems, otherwise their acceptance in the engineering field will be jeopardized even though they provide higher functionality through type extensibility and type-associated operations that model context-specific behavior. Engineers are generally not willing to trade performance for extra functionality and expressive power. Of course, there are major similarities between query processing in relational DBMSs and object bases. Therefore, the large body on knowledge of relational optimization techniques can serve as a basis for further development. However, the full potential of the object-oriented paradigm can only be exploited for optimization if new access support techniques are tailored specifically for the object-oriented model(s) and not merely assimilated from the relational model. Access support relations constitute one such approach.

Clustering is an effective mechanism for retrieving complex objects. Many variant clustering schemes have been suggested to improve the performance of OODBMSs. Two issues may compromise the effectiveness of a clustered structure, i.e., object updates and multiple relationships. Updates may destroy the initially clustered structure, and in a multiple relationship environment, clustering
objects based on one relationship may sacrifice others.

Due to the nature of supported applications, OODBMSs need efficient mechanisms for the retrieval of composite objects and navigation along the semantic links among objects. Object clustering is effective when the primary access pattern follows the primary relationship used for clustering, but is not so effective when multiple relationships are involved [Chng91b]. Object buffering provides a possible solution for compensating the limitations of clustering schemes. [Chng91a] investigates the advantages of using an additional layer of object-based buffer pools on top of a traditional page-based buffer pool, and proposes a profile-based buffering scheme to optimize the efficiency of object buffer pools.

Consequently, here we consider the query process in the query processing sub-system in two layers: the layer of query process (query rewriting), and the layer of buffer allocation. For the former, we will concentrate on ADT functions with high computational costs and propose two kinds of algorithms which rewrite the query to reduce the amount of ADT function computation; for the latter, a number of buffer allocation approaches which take into consideration the information on references among objects, and which are thus more flexible and more effective, are discussed for buffer management in OODBs.

To address the former problem of supporting ADT, we will discuss two schemes for optimizing computationally expensive ADT functions in the context of optimizing ADT queries. These two schemes, called "decomposition" and "using (semi)join," are based
on the observation that an ADT function is computationally expensive. The decomposition algorithm replaces a computationally expensive ADT function with a cost-effective (or, computationally “cheap”) equivalent sequence of simpler functions. Iterative reduction of an operand relation can significantly lessen the total computational cost of the given ADT function. Several lemmas are developed to find the optimal (cost minimal) decomposition. Based on these, a decomposition algorithm is proposed and proved to give an optimal decomposition.

An optimization method that treats ADT selections in the same way as joins has also been proposed [Chen90, Yaji91]. As an extension of this, we have found that using semi-joins can sometimes effectively reduce the number of evaluations of ADT functions and consequently lead to the reduction of the total cost. This effectiveness expressly appears under the assumption that the computational cost of ADT selections is far more expensive than that of joins.

We clarify the conditions and their corresponding query evaluation plan (abbreviated, QEP), showing that under certain conditions, the corresponding QEP is “cheapest” in terms of evaluation cost. We then extend the analysis to general cases, in which more than two objects are considered. An algorithm has been designed to determine the performing order of joins.

To address the latter problem of processing composite object, we treat nested objects on which the imposed Is-Part-Of relationship is implemented using composite references. As discussed by
Kim [Kim88, Kim89b], the composite references are further divided into two categories, exclusive and shared (composite) references. A query on a set of composite objects often scans composite objects successively. At a certain stage of scanning, composite object references are traversed successively. Consequently, sequences of composite object traversals within shared objects appear repeatedly in one query.

In order to improve the performance of the buffer management system in such a situation, the reference information among the accessed objects should not be disregarded. This observation encourage us to consider the possibility of utilizing the reference information among components of composite objects in the buffer management system, especially for the buffer allocation. Conventionally, it is not possible for the buffer management system to have access to the reference information in composite objects in the buffer, and this has been considered beyond the scope of the responsibilities of the buffer management system. We believe that it is useful to make the reference information available to the buffer management system because 1) this enhances buffer management performance and 2) the implementation is feasible. For example, we can store the reference information about the relating objects of an object in the header of the storage for the object, and collect this information into a special area of the memory when the object is accessed. Alternatively, we can also collect the reference information of all database objects and store it separately, and read and keep this information in the memory for buffer management. As a basic step toward this objective, the new buffer allocation ap-
proaches LRU-S and LRU-D have been formally defined and tested using a certain benchmark to show their efficiency. In addition, we analyze the properties of our strategies and test their efficiency using a certain benchmark.

1.2 Related Works

This thesis draws on research from the following areas of computer science: advanced database models, buffer management in DBMSs, and query processing. Several models/systems which have influenced this work are summarized in Appendix B. The sequel part of this section review the relevant aspects of these areas.

1.2.1 Advanced Data Models

It is commonly recognized that the relational data model [Codd70], with its flat representation of data results in a semantic mismatch between the entities being modeled and the underlying DBMS [Kent79]. A number of approaches have been advanced to remedy this problem. One approach has been to modify the relational model to provide it with more power[Codd79]. More recent work has aimed at including user defined data types within the relational model [Atki87, Osbo86, Rowe87, Ston83, Ston84, Ston86a, Ston86b]

An orthogonal approach has been the development of new and more powerful data models generally classified as semantic data models. Most semantic models are based on the concepts of aggre-
gation, generalization and classification [Smit77a, Smit77b]. Well
known examples are the E-R model [Chen76], Daplex [Ship81],
IFO [Abit88] and others. An overview of this entire field can be
found in [Rich87].

Extensible model [Bato88, Care88, Yu87, Yu90] try to provide
a ready means of constructing a database systems tuned or biased
toward a particular domain of applications. Much of the work
in this field can be characterized as software engineering: trying
to produce clean specifications of database components and their
interfaces, so that those components can be easily interchanged or
modified.

Object-oriented database model [Banc88, Frey88, Hard88,
Hugh91, Katz89, Keta89, Lécl88, Lécl89] offer very promising
prospects in modeling and handling advanced application objects.
One of the most important feature of object-oriented models is that
they support structural data [Adib88, Bato84, Bato85, Pare88].

All the data models mentioned above have influenced the cur-
rent research. The data model used in this thesis includes fea-
tures of extended relational models, e.g., the concept of ADTs and
user-defined data types; features of object-oriented models, such
as structural data; features of extensible models, e.g., extension of
methods for supporting new data types.

In the notation of queries, the algebras of normalized and non-
normalized relational models are used. [Chen92b, Chen93] have
led directly to this work.
1.2. RELATED WORKS

1.2.2 Buffer Management in DBMS

Traditional buffer management in DBMS utilizes or inherits the policies of buffer management in operating systems such as LRU (Least Recent Used), LIFO (Last In First Out), FIFO (First In First Out), and so on. Yet efforts have been made to find efficient policies and new approaches, such as MRU (Most Recent Used) [Ng91], DBMIN [Chou85] and Hot Spot theory[Sacc82, Sacc86], have been developed. A survey of database buffer management done by Effelsberg [Effe84] and Sacco et al [Sacc86] makes a good reference. It is pointed out that these approaches are not sufficient for supporting advanced applications, and as alternatives, clustering [Schö89] and prefetching [Palm91] are proposed. Nevertheless, there is no one buffer allocation policy which is efficient in every case. In other words, access patterns or query patterns are the most effective factors that influence the performance.

The policy proposed in this thesis is based on the considerations of traditional LRU. Similar to clustering, it takes into consideration the reference structures among composite objects. Finally, the Hot Spot theory helps us in analysing our simulation results. The HyperModel Benchmark proposed by Anderson [Ande91] enlightened us in that the database used in our simulation has the same structure as Anderson's one.
1.2.3 Query Processing

Query processing has many components including query representation, transformation, optimization and evaluation. A good overview of these issues and their interrelationships in the context of the relational model is provided by Jarke [Jark84]. For distributed systems, in addition to the query processing of the central relational model, the cost of communication among the sites where the database is stored must be taken into account [Chu82, Hevn79, Mack86, Yu84]. Much work done in [Chen90, Chen92a, Yaji91] emphasized the importance of query processing in non-traditional systems.

Two types of query transformations are common. One generates query expressions for which it is easier to prove certain properties such as safety. Methods for rewriting relational calculus expressions to obtain domain independent formulas are presented in [Geld87]. Domain independence implies safety by insuring that a query can be answered by examining only the relations it references, not the (possibly infinite) domains the attributes of these relations are drawn from [Geld87]. A similar technique using set operators instead of universal quantification and negation is given in [Ozso89].

The second type of transformation generates query expressions which are less costly to evaluate. These are often referred to as logical transformations. Jarke [Jark84] and Talbot [Talb84] identify many such transformations for the relational data model. Shaw identifies a limited number of logical transformations for an object
The ability to define precisely the nature and pre-conditions of logical transformations has led to their application using rule-based systems. This technique can be used both to ameliorate queries [Grae87, Haas89] and to translate them into executable access plans [Frey87, Lohm88]. In fact, two types of rules — transformation rules and implementation rules — proposed by Freytag [Frey87] correspond exactly to two types of transformation mentioned above.

Almost all of the works referenced here have been directly or indirectly applied in this thesis. We use logical equivalence preserving transformations to improve query expressions and define them with conditions which would be suitable for rule-based transformation systems. The join enumeration algorithm of [Mish92] is modified and extended for generating our approaches. The idea of semijoin [Yu84, ChYu92] is directly combined with our algorithm in the second approach for optimizing queries including ADT functions.

1.3 Organization

In Chapter 2, the motivation for introducing the concepts of ADT and composite objects for supporting advanced applications is explained in detail. Then, new phenomena that ADT functions are computational expensive and composite objects are accessed together, which arise with the introduction of such concepts when processing new queries are investigated. It is also pointed out that
such queries cannot be effectively treated using traditional strategies.

Concentrating on the processing of queries including ADT functions, especially selections with ADT functions as predicates, Chapter 3 presents two approaches: reduction by decomposition and reduction by (semi)join. Algorithms corresponding to these two approaches are designed and analyzed.

We propose LRU-S, a new buffer allocation strategy in Chapter 4. Qualitative analysis is developed to identify the situations under which LRU-S is beneficial. This strategy, combined with traditional clustering, is expected to enhance the efficiency of traversing composite objects. A simulation is designed and the empirical results, which confirm the above observation, are illustrated and discussed.

Chapter 5 discusses some directions for future research such as ideas on the implementation and then summarizes this thesis.
Chapter 2

Query Processing in Advanced Applications

2.1 Support ADT in a Relational Environment

An ADT in the database context has been discussed in [Atki87, Jian89, Osbo86, Rich87, Ston88c]. In this chapter, we first revise the ADT model.

Definition 2.1 (ADT) An ADT \( \alpha \) is modeled as \( \alpha = (D_\alpha, F_\alpha) \), where \( D_\alpha \) is the specification of its domain and \( F_\alpha \) is a set of ADT functions defined on it.

Definition 2.2 (ADT Function) An ADT function \( f \) is a function mapping instances of ADT into a primitive value, or

\[
f : (D_1 \times D_2 \times \ldots \times D_n) \rightarrow D_p,
\]

and \( \forall i (D_i \in D_P \cup D_{ADT}) \land \exists i (D_i \in D_{ADT}) \land D_p \in D_P \), where \( D_P \) and \( D_{ADT} \) are two sets of primitive domains and ADT domains, respectively.
For example, graphs are defined as an ADT \( \text{GRAPH} = (D_{\text{GRAPH}}, F_{\text{GRAPH}}) \), where \( D_{\text{GRAPH}} = (V, E) \), \( V \) and \( E \) are sets of nodes and edges of a graph, respectively, and \( F_{\text{GRAPH}} = \{ \text{isomorphic}, \#\text{node}, \#\text{edge}, \ldots \} \). In an extended relational data model with ADT domains, a relational schema is specified by \( R = (A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n) \), where \( A_i \) is an attribute of \( R \) with \( D_i \) as its domain, and \( D_i \) may be an ADT. The specification \( D_\alpha \) and the definition for each ADT function in \( F_\alpha \) are given by the DBI (database implementor) [Bato88].

### 2.1.1 Queries including ADT functions

Introducing ADT provides many new features for an extended relational model. The implementation of ADTs is always a matter of concern, and support for the operation on such ADTs should be considered. For example, displaying a graph of ADT \( \text{GRAPH} \) is a considerable operation. These matters, in turn, bring new features into query and query processing. Queries in an extended relational model can be defined in many ways. Concentrating on the cost of ADT functions, in this thesis, we will adopt the SQL-like [Date87] definition and focus on the basic form, shown in the following,

```
Select attributes
From extended-relations
Where ADT-predicates.
```

Such queries can be expressed using the traditional notation of \textit{projection} (\( \Pi \)), \textit{selection} (\( \sigma \)), and \textit{join} (\( \mathcal{X} \)), if the semantics of the extension of ADT is defined. The following gives a definition of
extension of ADT to these operations.

**ADT Projection.** An *ADT projection* $\Pi_A$ is a projection on an attribute $A$, where $D_A \in D_{ADT}$, and $D_A$ is the domain of $A$.

**ADT Selection.** An *ADT selection* $\sigma_f$ is a selection with an *ADT predicate* $f$, which is a special case of the ADT function with the following definition:

$$f : (D_1 \times D_1) \cup (D_1 \times D_2) \cup (D_2 \times D_1) \rightarrow \text{Boolean},$$

where $D_1 \in D_{ADT} \land D_2 \in D_P$.

**ADT Join.** An *ADT join* $\Join$ is a join with an ADT predicate $f$ as described above.

Similar to a conventional selection $\sigma_{A=\text{"c"}}R$, an ADT selection is specified as, e.g., $\sigma_{\text{isomorphic}(G,g)}R$, where $G$ is an ADT (GRAPH) field of $R$ and $g$ is a "constant" GRAPH. Although we consider only binary ADT functions for compatibility with the conventional selection predicates, it is easy to extend our discussion to n-ary ADT predicates.

### 2.1.2 Traditional Optimization

The query processor undertakes the responsibility of rewriting and/or optimizing a query and produces a (cheapest) QEP. In [Date86a], Date identifies the four broad stages of the overall optimization process as follows:

- Cast the query into some internal representation
• Convert to canonical form

• Choose candidate low-level procedures

• Generate query plans and choose the cheapest

In the first two stages, there should not be any essential difference between query processing in a relational model and an extended relational model; while, there will be differences in processing in the latter two stages. Utilizing the query processor of a relational model to a query in extended relational model will not be efficient in certain cases.

In engineering databases, we are faced with a data model that allows for extension of the basic types by some form of data abstraction. This is a powerful modeling facility, but one which has severe implications regarding how we may do query optimization. No longer is a single simple model with a limited set of carefully studied operators available for use. Each user-defined type effectively introduces a brand new algebra. How then are we to understand the properties of these algebras in order to allow the same degree of optimization as was possible in the relational world?

For example, in a chemical database system, the support of ADT $GRAPH$ as described above is necessary to represent the structure of chemical compounds. Accordingly, ADT functions such as displaying a graph, calculating the number of nodes in a graph, comparing two graphs, etc., are required.

Example
Consider the following relation:
2.2. **COMPOSITE OBJECTS**

Compound(Name, Graph, ...)  
where the relation named Compound has attributes Name and Graph, etc., and the domain of Graph is an ADT with which a set of user-defined ADT functions is associated.

```
select Compound.Name  
from Compound  
where isomorphic(Compound.Graph, const(graph0)).
```

The above query determines the names of compounds whose chemical structure graphs are isomorphic to a given graph graph0, where the term const(graph0) gives a "constant" graph.

The computational cost of the ADT function isomorphic is not negligible compared with its I/O cost. A conventional optimizer is designed based on the assumption that the computational cost is negligible, and produces a query plan which applies isomorphic to the whole relation. This impairs system performance seriously.

### 2.2 Composite Objects

In this section we will first explain the composite object model, then discuss some features in processing composite object queries.

**Definition (composite object)** A composite object can be described as a DAG $G_{comp} = (N_{comp}, E_{comp})$. $E_{comp} \subseteq N_{comp} \times N_{comp}$ represents the object reference augmented with the Is-Part-Of relationship. We say "o_i directly references o_j", denoted as $o_i \triangleright o_j$, if $(o_i, o_j) \in E_{comp}$. The relation '≽' is defined as follows:

(1) if $o_i \triangleright o_j$ then $o_i \succ o_j$, 

(2) if $o_i \succ o_j \wedge o_j \succ o_k$ then $o_i \succ o_k$.

$\succ$ is transitive, but is supposed to be asymmetric here; which implies that the relation $\succ$ is a partial order and the structure of the composite object is a DAG. We sometimes call $G_{\text{comp}}$ a composite object schema.

Suppose that a composite object $o$ whose schema is $O$ is given; an object identifier $id$ is given to the composite object $o$, which may include object references to objects whose schemas are given by $O_j$ ($1 \leq j \leq m$). Under the constraints imposed on the composite object schema, the composite object forms a DAG.

### 2.2.1 Querying Composite Objects

Queries in OODBs have been studied by many researchers [Kim88, Kim89b, Kemp90b, Shaw90b]. Here, as shown in Figure 2.1, we explain the query of composite objects using a modified examples from their papers. Consider the object queries in which the same part of reference sequences such as $O_1.O_2...O_n$ will appear more than once in the select clause and in the where clause combined by logic operator AND/OR. That is to say, generally, the objects of a composite object will be frequently accessed if they are specified more than once in a given object query. The situation will be more complicated if considering the sharing among objects.

**Example.** Select all vehicles whose weight is greater than 2000, and which is manufactured by a company whose president is under 50 or which has a computer research division.
(Vehicle select :V (:V Weight > 2000
and (:V ref :C
 (:C President Age < 50
or (:C Divisions some :D
 (:D Name = "Computer"
and (:D Goal = "Research")))))))

Figure 2.1: Example of Composite Object.

2.2.2 Buffering: Review

The strategies of LRU, MRU and LIFO have been studied extensively in the field of Operating Systems, and hence are not explained here. For comparing the behaviors of the above, we consider a simple example in the following.
Example. Perform the Cartesian product (or loop join) between two relations $R$ and $S$ in a buffer with size $s$, where $R$ and $S$ are stored in pages $p_1, p_2, \ldots, p_m$ and $q_1, q_2, \ldots, q_n$, respectively. \footnote{It is suggested in [Mish92] that a loop-join be performed in the following way: allocate $s - 1$ pages of the buffers for the smaller relation (outer relation) and one page for another relation (inner relation), read each tuple of the inner relation once for all the tuples in the outer relation.}

Figure 2.2 shows an example of rates of page fault with various buffer size $s$. As a comparison, Figure 2.3 shows another example for a ten-time pure loop of ten pages, e.g., repeating $\{p_1p_2\ldots p_{10}\}$ ten times. LRU is criticized for its pool performance for loop-join when sufficient buffers\footnote{Called a "hot point," which is $\min(m, n) + 1$ or 11 in Figure 2.2, and 10 in Figure 2.3, respectively.} are not allocated. On the other hand, it is easy to understand that LIFO and MRU behave poorly in cases of "warm starting," in which case the buffer is filled with some other pages, and of non-loop accesses.

To extend the buffer allocation strategies for database systems, the theory of Hot Spots is proposed. DBMIN [Chou85] and Hot Set [Sacc82, Sacc86] are two examples. A few objects in a database account for a large share of all database accesses. These objects are called hot spots. The ability to determine which objects are hot spots opens the door to a variety of performance improvements such as reorganization, migration, and replication of data. [Sale92] presents some techniques that can be used to identify hot spots by analyzing a string of references and collecting statistics. Depending on the length of the reference string and the amount of space available for the analysis, each technique will have a non-zero prob-
2.2. COMPOSITE OBJECTS

Figure 2.2: An Example of Buffer Performance ($m = n = 10$).

ability of false diagnosis, i.e., mistaking "cold" items for hot spots and vice versa. [Sale92] also compares the techniques analytically and shows the tradeoffs among time, space and the probability of false diagnoses.

The policies surveyed above do not take into consideration the semantic relationships among the data accessed. Object-oriented database systems with such buffer mechanisms are criticized for poor performance partially due to their "object-at-a-time" processing. Many approaches have been developed to address this problem. We will introduce two of these in the following sections.
2.2.3 Clustering: Techniques and Disadvantages

Due to the nature of supported applications, OODBMSs need efficient mechanisms for the retrieval of composite objects and the navigation along semantic links between objects. Clustering is the most frequently mentioned strategy for processing queries of composite objects in such situations [Bane88b, Benz90, Chng91b, Horn87, Kato91, Kell91, Tsan92, Will91]. For example, Keller used this method to enhance the performance on set-oriented applications [Kell91]; Zdonik [Horn87] and Carey et al [Care91] took it as a way to reduce communication between a server and clients in a client-server architecture. It has been shown that the DAG clustering scheme is an effective strategy for improving performance [Bane88b].
The basic idea of clustering is to pack together relevant objects onto the same page or contiguous pages.

1. Storing an object with its subobjects transfers a package of related objects that are almost always accessed together.

2. Storing all instances of a type together is used to satisfy queries requiring the search of all objects of a type.

3. Partitioning based on property values is similar to indexing. In using properties, specific values, such as "red", or numeric intervals, such as $0 < n < 3$, may be specified. This method allows a client to separate objects containing a property value of particular interest into one segment.

4. One object per segment is intended for very large objects, since these are costly to transfer and tend to be accessed individually.

However, clustering has its own disadvantages. First of all, clustering is effective only for a fixed number of query patterns. Object clustering is effective when the primary access pattern follows the primary relationship used for clustering. Secondly, clustering is powerless for large objects; the effect of clustering when object sizes are relatively small compared with page size is shown in [Kim87a]. However, for large objects (long fields) such as discussed in EXODUS [Care88, Care90], GemStone [Cope84], Starburst [Haas90] and many hypertext systems where an object occupies one whole page or even more than one page, the approach of clustering loses
its power. Thirdly, clustering is static; this means that no structural information is considered once the clustering is completed. Trying to address such problem, in the O2 system, a dynamic clustering has been proposed [Benz90]. The algorithm which implements a set of clustering strategies for the O2 system takes as input the access patterns of objects given by the user. The input provides information about the frequency of accesses from one object to another. The algorithm determines the clustering strategy, which is in fact a set of PTs (placement trees) on the corresponding COG (Composition Object Graph) based on the TIG (Type Inheritance Graph). The overhead of re-clustering or dynamic clustering is heavy. Finally, it is impossible to cluster objects based over different relationships; hence clustering is not so effective when multiple relationships are involved [Chng91b].

At least three types of structural relationships have been found to be useful for clustering: configuration, version, and correspondence [Chan89]. Because of the nature of the algorithms used in application programs such as CAD tools, some relationship links may be traversed more frequently than others during certain periods of time. That is, different relationship links have different weights. This approach, although remedied the static disadvantages somehow, is not thought of as providing a solution. It can not even solve the problem of the following simple example: how to do the clustering if an object is intensively shared by many objects. Redundantly storing this shared object in each cluster does not seem a viable answer.

Object buffering provides a possible way to compensate for the
limitation of clustering schemes. [Chng91a] investigates the advantages of using an additional layer of object-based buffer pools on top of a traditional page-based buffer pool, and proposes a profile-based buffering scheme to optimize the efficiency of object buffer pools. A performance evaluation based on the semantic implementation of profiles was conducted. However, it is still static and did not solve all the problems mentioned above.

2.2.4 Prefetching and Caching: Techniques and Disadvantages

In relation to the techniques of clustering, certain applications queries can be concentrated on certain patterns. Therefore, one can prefetch the relevant objects utilizing the information about these patterns. Faloutsos et al. [Falo91] used predictive loading and Carey [Care91] used caching to reduce communication among the server and the clients in a server-client architecture. These works are types of prefetching. Similar to clustering, prefetching has its own disadvantages. One is that it "inherits" some of the disadvantages of clustering discussed above, such as staticity and effectiveness for a fixed number of query patterns; taking the Input/Output between the primary memory and disc as the factor of measuring the performance, prefetching is effective only for small objects. Another disadvantage of prefetching is that it has to manage the profiles of query patterns.
2.3 Summary

The traditional query processing mechanism has been introduced here, especially the techniques used at the stage of query rewriting and of buffer management strategies in advanced database systems. In summary, none of these strategies meets all the requirements of advanced applications, but each does have some advantages over the others. This implies that important concepts in each method need to be extended.

In the discussion of processing ADT functions, we will concentrate on the query processing matter in the relational approach and investigate how to make the system better suited for use in advanced applications. The reason why we chose the approach of extending the relational model is as follows.

- The relational database system has a well-understood formal basis that facilitates effective database design and query processing.

- Relational database technology is well established. The theory and practice of object-oriented database technology is always built on top of the basic concepts and/or technologies of relational database [Hafe88, Kita89, Thom86]. Specially, some object-oriented databases can be categorized as NF².

- To date, the query languages for object-oriented databases e.g., the queries, query algebra and query processing proposed in [Shaw89, Dave90], have not exceeded the framework provided by relational systems.
Chapter 3

Optimizing Queries Including ADT Functions

3.1 Motivation and Overview

The extension of employing ADTs [Ston88c, Jian89] introduces a new dimension to query optimization. Conventional query processing focuses on the cost of input and output between main and secondary memory. In contrast, to process queries involving ADT functions, the computing cost of the ADT functions must be taken into consideration, since it is often the case that they dominate I/O costs. As mentioned in the previous chapter, suppose that an ADT function isomorphic is defined to determine isomorphism between two graphs. The function isomorphic is computationally very expensive, and is not negligible in query processing.

In order in processing queries of non-traditional applications efficiently, many approaches such as the extensible system [Bato88] and the optimizer generator [Grae87, Care88] have been proposed.
It has also been proposed that a rule-based approach to the optimization of ADT queries is very promising [Frey87, Lohm88]. In particular, decomposition rules are considered to reduce estimated costs significantly [Haas89]. However, these past works have ignored the possibility of optimizing queries including ADT functions by replacing the expensive ADT function with a cost effective but equivalent sequence of simpler functions.

This section discusses such a decomposition scheme for ADT functions in the context of optimizing ADT queries. We extend the optimizer so that it also takes into consideration the cost of evaluating the functions. When a query includes ADT functions whose computation makes up a considerable portion of the query processing time, it is wise to avoid computing such functions as much as possible. Fortunately, most computationally "expensive" ADT functions have some necessary conditions which can be implemented instead as a collection of computationally "cheap" ADT functions. This property can be used for screening the candidates which have no chance of satisfying the predicate including the expensive ADT functions. For the resulting small set of candidates, if necessary, the expensive ADT functions are applied to guarantee the sufficient conditions.

Recalling the example given in the previous chapter, it is well known that isomorphic graphs have the same number of nodes. In other words, for two graphs x and y, the trivial function nodeEQ(x, y) gives a necessary condition of isomorphic(x, y). Similar to nodeEQ, several necessary conditions are given as follows.
3.1. MOTIVATION AND OVERVIEW

$\text{isomorphic}(x, y) \Rightarrow \text{edgeEQ}(x, y)$  \hspace{1cm} (f_1)
$\land \text{nodeEQ}(x, y)$  \hspace{1cm} (f_2)
$\land \text{sameDegreeNodeEQ}(x, y)$  \hspace{1cm} (f_3)
$\land \text{maxNextDegreeEQ}(x, y)$  \hspace{1cm} (f_4)
$\land \text{samePathLengthEQ}(x, y)$  \hspace{1cm} (f_5)
$\land \text{sameCutset}(x, y)$  \hspace{1cm} (f_6)

These functions, which check whether two graphs have same number of nodes, same number of edges, same cut-set, and so on, enforce the necessary conditions of isomorphic, and discard tuples which cannot meet the selection condition. Instead of applying isomorphic to the entire relation, we can apply the above functions, prior to the application of isomorphic. Intuitively, the function with lower cost and higher screening capacity should be applied first. Thus, the order of application is important. A query evaluation plan, which is logically equivalent to the original query and more efficient, is given below.

$$
\Pi_{\text{Name}} (\sigma_{\text{isomorphic}}(\text{Graph}, \text{const}(\text{graph}_0)) \land 
(\sigma_{\text{maxNextDegreeEQ}}(\text{Graph}, \text{const}(\text{graph}_0)) \land 
(\sigma_{\text{sameDegreeNodeEQ}}(\text{Graph}, \text{const}(\text{graph}_0)) \land 
(\sigma_{\text{nodeEQ}}(\text{Graph}, \text{const}(\text{graph}_0)) \land 
(\sigma_{\text{edgeEQ}}(\text{Graph}, \text{const}(\text{graph}_0)) \text{Compound}))))).
$$

The image of decomposition is illustrated in Figure 3.1

In addition, based on the observation in the previous section, the computational complexity of an ADT function may be far more ex-
pensive than that of joins. This means that the definitive factor in the cost of query evaluation will shift from joins to ADT selections for cases of this kind. Therefore, the extension of ADT’s gives rise to new problems for conventional query optimization strategies, which are derived based on the assumption that the cost of a selection operation is much cheaper than that of a join operation. By contrast, selections involving evaluation of a computationally expensive ADT function may be more costly than joins. In such cases, a new query optimization strategy focusing on ADT selection is needed.

In order to derive query optimization strategies that correspond
to the emergence of selections in which a computationally expensive ADT function is involved, [Chen90, Yaji91] have proposed optimization methods that treat ADT selections in the same way as joins, and as an extension of them, we found that using semijoin [Ullm89, ChYu92] can sometimes effectively reduce the number of evaluations of the ADT functions and consequently lead to the reduction of the total cost. This effectiveness seems to appear when the computational cost of ADT selections is far more expensive than that of joins.

In [Yaji91] the possibility has been demonstrated that the total query processing cost is reduced if other joins are performed before some costly ADT selections. In other words, the strategies in [Yaji91] are chosen from between two candidates: the conventional query optimization strategy of "pushing selection down" and a new strategy by which some joins are performed before costly ADT selections if the join selectivity is less than a given value. But as the following discussion shows, we have found that in some cases an additional execution of semi-join can be more effective in the sense of reducing the number of ADT instances evaluated during the computation of selection.

As background for our research, we shall consider two relations in a chemical DBMS [Fuji91], shown in the following Figure 3.2:

The relation named *Structure* has attributes *Class* and *Graph*, etc., and the domain of *Graph* is an ADT with which is associated a set of user-defined ADT functions. *Compound* is another relation which has attributes *Name*, *Class*, etc. A typical query concerning
these two relations would be:

\[
\begin{align*}
\text{select} & \quad \text{Compound.Name} \\
\text{from} & \quad \text{Compound} \\
\text{where} & \quad \text{isomorphic(Compound.Graph, const(grapho))},
\end{align*}
\]

which would determine the names of compounds whose chemical structure graphs are isomorphic to a given graph $graph_0$. Here, the term $\text{const}(graph_0)$ returns a "constant" graph.

A conventional optimizer designed according to the principle of "pushing selections down" will unconditionally produce a QEP as follows:

\[
\Pi_{\text{Name}}(\sigma_{\text{isomorphic(Graph, const(grapho))}} \text{Structure}) \bowtie \text{Compound}
\]
However, since the ADT function *isomorphic* is very computational expensive, the key point of the query optimization is to reduce the number of evaluations of the ADT function as much as possible. The following QEP has been suggested by [Chen90, Yaji91] when the join selectivity is comparatively low.

\[ \Pi_{\text{Name}} \sigma_{\text{isomorphic}([\text{Graph, const}([\text{graph}_0])])}(\text{Structure \bowtie Compound}) \]

Moreover, we have observed that in certain cases semi-join can be more effective for reducing the set of candidates, on which the expensive ADT function is to be applied. For the relations *Structure* and *Compound* mentioned above, the results of join and semi-join are shown in Figure 3.3

(a). Structure \bowtie Compound

| Name | Class | Graph | ...
|------|-------|-------|
| cn1  | 1     | g₁    | ...
| cn1  | 1     | g₁    | ...
| cn1  | 1     | g₁    | ...
| cn2  | 1     | g₂    | ...
| cn2  | 1     | g₂    | ...
| cn2  | 1     | g₂    | ...

(b). Structure \bowtie Compound

| Class | Graph | ...
|-------|-------|
| 1     | g₁    | ...
| 1     | g₂    | ...

Figure 3.3: Result of Join and Semijoin.

As we can see, there are six tuples in the result of join and two in that of semi-join. Although the number of different instances for attribute *Graph* are in fact both two, the ADT function has to be evaluated six times if the selection is applied to the result of join since there is no simple way to identify an ADT instance, while the
ADT function computations need be performed only twice if the result of semi-join is chosen as the operand of the selection.

Therefore, a third QEP with the least number of evaluations of ADT selection can be proposed in this case as follows:

\[ \Pi \text{Name} \ (\sigma_{\text{isomorphic(Graph, const(grapho))}}) \]
\[ (\text{Structure} \bowtie \text{Compound}) \bowtie \text{Compound} \]

Generally speaking, if the degree of duplication of the attribute value on which the join operation is applied is rather high, it is possible to execute semi-join first to reach the least number of candidates on which the computationally expensive ADT function need be applied. Even though the join operation has to be executed twice according to this strategy, the total cost will be decreased.

3.2 The Decomposition Approach

3.2.1 Terminologies

In this section, we formalize a query processing scheme based on the decomposition of a selection operation involving a computationally expensive ADT function. We then give an algorithm for determining an optimal query evaluation plan.

We assume that for a given ADT predicate, a set of ADT predicates \{f_1, f_2, ..., f_n\} (called a specification of f) is given by the DBI such that each ADT predicate \( f_i \) in the specification is a necessary condition of \( f \), and the set forms the sufficient condition of \( f \). This means \( f \) is logically equivalent to \( f_1 \land f_2 \land ... \land f_n \).
Definition 3.1. Suppose \( f \) has the specification \( \{ f_1, f_2, \ldots, f_n \} \). A distinct sequence: \( \tau = (f_{k_1}, f_{k_2}, \ldots, f_{k_t}) \), where each element of \( \tau \) is an element of the specification, is called a decomposition of \( f \) if \( f \) is logically equivalent to \( f_{k_1} \land f_{k_2} \land \ldots \land f_{k_t} \). Here \( t \leq n \), that is, some elements of the specification may not appear in \( \tau \). We define operators ‘\( \in \)’ and ‘\( - \)’ on the decomposition \( \tau = (f_1, \ldots, f_m) \) in the following way:

- \( f_k \) is an element of \( \tau \), or \( f_k \in \tau \), iff \( f_k \in \{ f_1, \ldots, f_m \} \),
- \( \tau - \{ f_k \} = (f_1, \ldots, f_{k-1}, f_{k+1}, \ldots, f_m) \),
- \( \tau - \{ f_{k_1}, f_{k_2}, \ldots f_{k_t} \} = \tau - \{ f_{k_1} \} - \{ f_{k_2} \} - \ldots - \{ f_{k_t} \} \).

Definition 3.2. If \( \tau = (f_1, f_2, \ldots, f_l) \) is a decomposition of \( f \), then a sequence of selections \( (\sigma_{f_1} \ldots (\sigma_{f_2}(\sigma_{f_1}))), \ldots \) is called the decomposition of ADT selection \( \sigma_f \), or,

\[
\sigma_{(f_1,f_2,\ldots,f_l)} = (\sigma_{f_1} \ldots (\sigma_{f_2}(\sigma_{f_1}))) \ldots .
\]

The optimal evaluation order of the \( \sigma_{f_i} \)'s depends on the computational cost of the functions and their reduction ratio. We introduce two kinds of reduction ratios. One is “selectivity,” which indicates the reduction ratio of applying \( \sigma_f \) to a relation: \( I = |\sigma_f R|/|R| \), where \( |R| \) denotes the cardinality of \( R \). Applying \( \sigma_f \) to \( R \) divides \( R \) into two parts: \( f._{TRUE} \) consisting of the tuples of \( R \) which satisfy \( f \), and \( f._{FALSE} \) which is the complement of \( f._{TRUE} \). Then \( |f._{TRUE}| = I \ast |R| \), and \( |f._{FALSE}| = (1 - I) \ast |R| \). Another reduction ratio is related to a “false drop.” Let \( f_i \) be an ADT predicate in the specification of \( f \). If we apply \( \sigma_{f_i} \) to \( R \), all the tuples in \( f._{TRUE} \) satisfy \( f_i \), while \( f._{FALSE} \) is further divided...
into \( f_i \_TRUE \) and \( f_i \_FALSE \). We define

\[
d_i = \frac{|f_i \_TRUE|}{|f_i \_FALSE|}
\]

as the false drop of \( f_i \). The number of tuples in \( \sigma_{f_i} R \) is

\[
|f \_TRUE| + |f_i \_TRUE| = I * |R| + d_i * (1 - I) * |R|
\]

Figure 3.4 illustrates these relationships.

Now, let the computational cost of \( f \) and \( f_i \) be \( \lambda_0 \) and \( \lambda_i \), respectively. The evaluation cost of \( \sigma_f R \) is then \( \lambda_0 * |R| \), while that of \( \sigma_{(f_i,f)} R \) is

\[
\lambda_i |R| + \lambda_0 |\sigma_{f_i} R| = (I * (\lambda_i + \lambda_0) + (1 - I) * (\lambda_i + d_i \lambda_0)) * |R| \quad (3.1)
\]

Obviously, (3.1) is much lower than \( \lambda_0 |R| \) when \( \lambda_0 \gg \lambda_i \). It is noteworthy that for the decomposition \((f_1, f_2, ..., f_t)\) of \( f \), \( d_t = 0 \) holds because \( f_1 \land f_2 \land ... \land f_t \) should be logically equivalent to \( f \).

### 3.2.2 Cost Model

In this section, we provide an algorithm for finding the optimal decomposition. The algorithm decomposes an ADT predicate \( f \) with high computational cost into the most inexpensive sequence of ADT predicates that is logically equivalent to \( f \). Comparison among costs of possible decompositions of a given ADT selection is based only on the cost of in-memory operations for computing ADT predicates. The I/O cost remains constant even if the application order of these functions is changed, and hence is omitted in our
3.2. THE DECOMPOSITION APPROACH

Figure 3.4: Selectivities and False Drops.

cost model. An evaluation cost of $\sigma_{\tau}$, where $\tau = (f_1, f_2, ..., f_t)$, is denoted by $\text{Cost}((f_1, f_2, ..., f_t))$ or $\text{Cost}(\tau)$.

In the following discussion, we assume that the $f_i$'s in the specification of $f$ are independent of one another, and no index is employed on the ADT field. We extend (3.1) to the general case $\tau = (f_1, ..., f_t)$. Using the notation in the previous section and omitting the common factor $R$, the evaluation cost of the decomposition $\sigma_{\tau}$ is

$$\text{Cost}(\tau) = I \times (\lambda_1 + \lambda_2 + ... + \lambda_t) + (1 - I) \times (\lambda_1 + d_1\lambda_2 + d_1d_2\lambda_3 + ... + d_1d_2...d_{t-1}\lambda_t) \quad (3.2)$$

We next compare the costs of two different decompositions with the same set of elements. For the decomposition $\tau_1 = ...
(f_1, f_2, ..., f_i, f_{i+1}, ..., f_t) = \tau, \text{ from (3.2),}

\text{Cost(} \tau_1 \text{) = } I \ast (\lambda_1 + \lambda_2 + \ldots + \lambda_t) + (1 - I)(\lambda_1 + d_1\lambda_2 + \ldots + d_1d_2...d_{i-1}\lambda_i + \ldots + d_1d_2...d_{t-1}\lambda_t).

While, for \tau_2 = (f_1, f_2, ..., f_{i+1}, f_i, ..., f_t),

\text{Cost(} \tau_2 \text{) = } I \ast (\lambda_1 + \lambda_2 + \ldots + \lambda_t) + (1 - I)(\lambda_1 + d_1\lambda_2 + \ldots + d_1d_2...d_{i-1}\lambda_i + \ldots + d_1d_2...d_{t-1}\lambda_t).

We have

\frac{\text{Cost(} \tau_1 \text{) - Cost(} \tau_2 \text{)}}{(d_1d_2...d_{i-1}) \ast (1 - I)} = \frac{\lambda_i}{(1 - d_i)} - \frac{\lambda_{i+1}}{(1 - d_{i+1})} \ast (1 - d_i)(1 - d_{i+1}).

By defining the rank p_k of f_k as

p_k = \frac{\lambda_k}{1 - d_k},

obviously p_i < p_{i+1} implies Cost(\tau_1) < Cost(\tau_2).

From this it is evident that Lemma 1 follows (Proof given in detail in Appendix A).

Lemma 1. Let f_1, ..., f_n and p_1, ..., p_n be functions of ADT predicates in the specification of f and their ranks, respectively. If decomposition (f_1, ..., f_n) satisfies p_i \leq p_j, for any 1 \leq i < j \leq n, then

\text{Cost(} (f_1, ..., f_n) \text{)} = \text{min}\{\text{Cost(} (f_{i_1}, ..., f_{i_n}) \text{)} | < i_1, ..., i_n > \text{ is a permutation of } < 1, ..., n >\}.
3.2. THE DECOMPOSITION APPROACH

be optimal. It is possible that by eliminating certain \( f_i \)'s from the decomposition, the evaluation cost can be reduced further. In the following we try to determine such \( f_i \)'s.

**Lemma 2.** Let \( \tau = (f_1, \ldots, f_i) \) be a decomposition of \( f \) which meets the condition of Lemma 1. Then,

\[
\text{Cost}(\tau) \leq \text{Cost}(\tau - \{f_i\})
\]

implies

\[
\text{Cost}(\tau - \{f_j\}) \leq \text{Cost}(\tau - \{f_j\} - \{f_i\})
\]

for all \( j, j \neq i \)

**Proof.** We prove the lemma for the case \( j < i \), (in fact, only this case is used in our algorithm). Suppose (3.4) fails while (3.3) holds. Then,

\[
\text{Cost}(\tau - \{f_j\}) - \text{Cost}(\tau - \{f_j\} - \{f_i\}) = I \ast \lambda_i + (1 - I)(d_1 \ldots d_{i-1}/d_j)(\lambda_i

- (1 - d_i)(\lambda_{i+1} + \ldots + d_{i+1} \ldots d_{t-1} \lambda_t)) > 0,
\]

while

\[
\text{Cost}(\tau - \{f_i\}) - \text{Cost}(\tau) = -I \ast \lambda_i - (1 - I)d_1 \ldots d_{i-1}(\lambda_i - (1 - d_i)(\lambda_{i+1} + \ldots + d_{i+1} \ldots d_{t-1} \lambda_t)) \geq 0.
\]

Adding the two expressions above and noting that

\[(1 - I)(d_1 \ldots d_{i-1}(1 - d_j)/d_j) > 0,\]
we have
\[ \frac{\lambda_i}{(1 - d_i)} > \lambda_{i+1} + d_{i+1}\lambda_{i+2} + \ldots + d_{i+1}d_{t-1}\lambda_t \] (3.5)

From the definition of \( p_i \) and \( p_{i+1} \leq p_{i+2} \),
\[
\lambda_{i+1} + d_{i+1}\lambda_{i+2} + \ldots + d_{i+1}d_{t-1}\lambda_t \\
\geq \lambda_{i+1} + d_{i+1}(1 - d_{i+2})\lambda_{i+2}/(1 - d_{i+1}) \\
= \lambda_{i+1}(1 - d_{i+1}d_{i+2})/(1 - d_{i+1}) \\
= (1 - d_{i+1}d_{i+2})p_{i+1}.
\]

By repeating this process on the right hand side of (3.5) (A proof is given in Appendix A using mathematical deduction),
\[
\lambda_{i+1} + d_{i+1}\lambda_{i+2} + \ldots + d_{i+1}d_{t-1}\lambda_t \\
\geq (1 - d_{i+1}d_{i+2}\ldots d_t)p_{i+1} \\
= p_{i+1} \quad \text{(from} d_t = 0). \]

Thus, (3.5) becomes
\[
p_i = \frac{\lambda_i}{(1 - d_i)} > p_{i+1}.
\]

This contradicts the assumption that \( p_i \leq p_{i+1} \), hence (3.4) holds. The proof for case \( j > i \) is given in detail in Appendix A.

\[ \square \]

This lemma means that if the elimination of \( f_i \) from \( \tau \) increases the cost, then the elimination of \( f_i \) from a subsequence of \( \tau \) also increases the cost. This fact can be utilized to reduce the search space of determining the optimal decomposition.
3.2.3 Algorithm

In the following, we give the decomposition algorithm. In the procedure Search, the search space is a binary tree, as shown in Figure 3.5. The node labeled $f_k$ indicates that the ADT predicate $f_k$ should be deleted from the decomposition $\tau$, while the node labeled $\neg f_k$ indicates the contrary. A path in the tree denotes an (intermediate) result. For example, a path $(f_5, \neg f_4)$ stands for $\tau - \{f_5\}$.

![Figure 3.5: The Search Space.](image)

**Algorithm (Decomposition).** Suppose that the specification of an ADT function $f$ is $\{f_1, f_2, ..., f_i, ..., f_n\}$, for which the computational costs and false drops are known.

1. Sort $f, f_1, f_2, ..., f_i, ..., f_n$ in an ascending order of their ranks, $p_i$. For simplicity, let the result be $\tau = (f_1, f_2, ..., f_n, f_{n+1})$, where there is an $m$, $1 \leq m \leq n + 1$, such that $f = f_m$. 
2. Suppose that $f_i$ ($1 \leq t \leq n + 1$) satisfies

$$p_i = \min\{p_i \mid 1 \leq i \leq n + 1 \land d_i = 0\}.$$ 

Discard those $f_i$'s whose ranks $p_i$'s are not smaller than $p_t$. $\tau$ becomes $(f_1, f_2, \ldots, f_t)$.

3. Define

$$S = \{f_i \mid i \neq t \land f_i \in \tau \land \text{Cost}(\tau - \{f_i\}) < \text{Cost}(\tau)\}$$

$$= \{f_{i_1}, f_{i_2}, \ldots, f_{i_k}\},$$

where $l_i < l_{i+1}$ (hence $p_i \leq p_{i+1}$) for $1 \leq i < k$.

4. Output $\text{Search}(\tau, k)$.

The procedure $\text{Search}(\tau, i)$ is defined as follows.

$$\text{Search}(\tau, i)$$

if $i = 0$ then return $\tau$;

if $\text{Cost}(\tau - \{f_i\}) < \text{Cost}(\tau)$ then

return $\text{minCost}(\text{Search}(\tau - \{f_i\}, i - 1), \text{Search}(\tau, i - 1))$

else return $\text{Search}(\tau, i - 1)$;

and

$$\text{minCost}(\tau_1, \tau_2) = \text{"if Cost}(\tau_1) < \text{Cost}(\tau_2) \text{ then } \tau_1 \text{ else } \tau_2"}.$$

Theorem. The algorithm above gives the optimal decomposition of $f$.

Sketch of the proof

1. By Lemma 1, the optimal decomposition must be of the form $(f_{i_1}, f_{i_2}, \ldots, f_i, f_t)$.
2. By Lemma 2, the optimal decomposition can only be derived by deleting a subset of $S$ from $\tau$, and the search of the algorithm covers the whole search space except the nodes with a higher evaluation cost.\[\Box\]

Note that we add $f$ into the function set given in the specification in Step 1. The reason for this is that if in Step 2, $p_t > \text{"the rank of } f\text{"}$, then $\text{Cost}(f_{i_1}, f_{i_2}, ..., f_{i_t}, f) > \text{Cost}(f_{i_1}, f_{i_2}, ..., f_{i_t}, f)$. The complexity of the algorithm is $O(T \cdot 2^k)$, where $T$ is the cost of calculating the value of the expression

$$\text{Cost}(\tau) = I \ast (\lambda_1 + \lambda_2 + \ldots + \lambda_t)$$

$$+(1 - I)(\lambda_1 + d_1\lambda_2 + d_1d_2\lambda_3 + \ldots + d_1d_2\ldots d_{t-1}\lambda_t).$$

$O(T \cdot 2^k)$ is small for the original ADT predicate with a small number of functions in the specification. On the other hand, if the ADT predicate has a large number of functions in the specification, denoted by $n$, the naive algorithm to find the optimal subset and order has a complexity of $O(2^n \cdot n!)$. The algorithm above reduces the cost from $O(T \cdot 2^n \cdot n!)$ to $O(T \cdot 2^k)$ for some $k < n$, extending the practical applicability of our optimization scheme. Furthermore, since the ranks of ADT predicates are considered to be fairly stable, the optimal decomposition for a given ADT predicate does not change frequently. This means that once the optimal decomposition is derived for a given ADT predicate, the DBMS can cache the optimization result.
3.2.4 Example

Recall the example query given in Section 2.1.2, where we gave the resulting query evaluation plan by decomposing isomorphic by the algorithm. The process of deriving the decomposition using applying the algorithm is illustrated in Figure 3.6. There, $(A_x \sim\ A_y)$ means that the incidence matrices $A_x$ and $A_y$ of graphs $x$ and
y are isomorphic, i.e., $A_y$ is derived from $A_x$ by permuting the rows and columns. This is then used as an implementation of the ADT predicate *isomorphic*. The algorithm gives the query evaluation plan shown in Section 1. The evaluation plan reduces the evaluation cost of the original query from 1000 to 125.5. We can see that $n=7$ and $k=2$, and that the cost of the algorithm itself is $O(T \ast 2^2)$. This cost is negligible when compared with the computational cost of *isomorphic*.

### 3.3 The Approach Using (Semi)-Join

#### 3.3.1 Cost Model and Algorithm

We compare the costs of the following QEPs in a memory-resident model, the intuitive image is shown in Figure 3.7.

- **method 1:** $(\sigma_{f(R_1.A, \text{const})} R_1) \bowtie_B R_2$
- **method 2:** $\sigma_{f(R_1.A, \text{const})} (R_1 \bowtie_B R_2)$
- **method 3:** $(\sigma_{f(R_1.A, \text{const})} (R_1 \bowtie_B R_2)) \bowtie_B R_2$

The costs for different methods are denoted as $C_1$, $C_2$ and $C_3$ respectively, and we use the following notation for the cost computation.

- $R_i$: Relation $i$,
- $t_i$: Number of tuples of $R_i$, also denoted as $||R_i||$,
- $D_{iA} = \Pi_A R_i$,
- $d_{iA} = ||D_{iA}||$: Image size of attribute $A$ in $R_i$, 
\[ s_{ij} = \frac{t_{ij}}{t_i t_j} \]: where \( R_i = R'_i \bowtie R_j \),

\( \lambda \): Cost of calculating \( f \) once.

\( I_f = ||\sigma_{f(A, c)}R_i||/||R_i|| \): Selectivity of \( f \).

Using the parameters listed above, the cost of the three QEPs can be described as follows:

\[ C_1 = \lambda t_1 + I_f t_1 t_2 \]

\[ C_2 = t_1 t_2 + \lambda t_1 t_2 s_{12} \]

\[ C_3 = t_2 \ln(t_2) + t_1 d_{2B} + \lambda t_1 d_{2B} s_{12} + t_1 d_{2B} s_{12} I_f t_2 \]

Based on the assumption that the computational cost of the ADT function is far more expensive than that of the join, we consider the computational cost of the ADT function to be a dominant factor in the total cost. Hence the following discussion focuses on the coefficient of the \( \lambda \).

(1) \( t_1 \) for \( C_1 \)

(2) \( t_1 t_2 s_{12} \) for \( C_2 \)
3.3. *THE APPROACH USING (SEMI)-JOIN*

(3) \( t_1 d_{2B}s_{12} \) for \( C_3 \)

As for (2) and (3), \( t_1 d_{2B}s_{12} \) is straightforward. consequently,

\[ C_3 \leq C_2 \quad (\text{while } \lambda \gg 1) \]

for (1) and (3), it is clear that

\[ t_1 d_{2B}s_{12} \leq t_1 \]

because the cardinality of the result of the semijoin is not greater than that of the first relation participating in the semijoin.

\[ C_3 \leq C_1 \quad (\text{while } \lambda \gg 1) \]

Based on the analysis above, an algorithm is proposed as follows:

if \( d_{2B}/t_2 < \alpha < 1 \) then

execute method 3

else if \( t_2s_{12} \leq 1 \) then

execute method 2

else

execute method 1

To evaluate the execution costs using the different methods mentioned above, a lot of simulations have been performed and an analysis of some of the results are presented here. We assume that the two participating relations both have cardinality of 1000. The parameters considered in the simulations are join selectivity \( s_{12} \), image size \( d_{2B} \) and the computation cost of the ADT function \( \lambda \). In the following figures, the horizontal axis represents the join selectivity, while the vertical represents the total cost.
[Figure 3.8] This is the case in which $d_{2B}$ is rather small (about 30% of $t_2$). Here method 3 showed a confident result. As mentioned above, when the product of $s_{12}$ and $d_{2B}$ is rather small, method 3 will be the best choice. As a result, even if the selectivity is not so small, the low image size $d_{2B}$ will maintain confidence in the performance of method 3. Note that $\lambda$ here is rather too large to satisfy our basic assumption.

[Figure 3.9] Note that all conditions are the same as in [Figure 3.8], except that the image size $d_{2B}$ is greater than that in [Figure 3.8] (about 60% of $t_2$). From the results we found that the interval in which method 3 showed the best effect became narrower due to the increase in $d_{2B}$.

[Figure 3.10] This is the case in which $d_{2B}$ is very close to $t_2$ (about 99% of $t_2$). From the our investigations we found that
3.3. THE APPROACH USING (SEMI)-JOIN

the results of method 3 are almost identical to that of method 2. However, in actuality in cases of this kind method 2 should be chosen, because the reduce join of method 3 has no effect at all and join has to be performed twice meaninglessly. This result supports the condition of \( \frac{d_{2B}}{t_2} < \alpha < 1 \) in the algorithm above. That is, when the value of \( \frac{d_{2B}}{t_2} \) is close to 1, method 2 should be chosen rather than method 3.

[Figure 3.11] This is the case in which \( \lambda \) is not as great compared to the cost of join (Here \( \lambda \) is 100 and we assume one comparison of join is 1). This means our assumption is not satisfied. From the result we found that method 3 is not as good as method 1.
3.3.2 General Discussion

The basic idea we have present above is that we can use semijoin to reduce the number of the candidates on which the expensive ADT selection is to be applied, in return for which, some join operations must be performed twice. This means that some expensive operations ADT selection are substituted for by other comparatively cheap join operations. In the case of two relations the discussion is quite simple. But when we consider the general case, the form of queries is as follows:

\[ R_1 \bowtie R_2 \ldots R_{i-1} \bowtie \sigma_f R_i \bowtie R_{i+1} \ldots \bowtie R_n. \]

The discussion becomes difficult because if we maintain the same approach of pursuing the least evaluation of the ADT selections, we will probably have to do a considerable number of join operations.
repeatedly which can not be ignored as in the case of two relations.

Therefore, we will change our policy for the general case, pursuing a local least evaluation of ADT selections rather than a total least evaluation. The reducer of $R_i$ (meaning the relation that is used to reduce the cardinality of $R_i$) is chosen only from between the adjacent relations $R_{i-1}$ and $R_{i+1}$. The algorithm given above is used separately for the two pairs of relations, $R_i - R_{i-1}$ and $R_i - R_{i+1}$. Consequently, two methods are selected for the two pairs. A comparison between these two methods is done and the better method is chosen.

If the decision of the above procedure is method 3, indicating that executing semi-join with a reducer is a good choice, then the algorithm proposed in [Yaji91], which is an extension of KBZ [KBZ86], will be applied. This algorithm takes a tree query $Q$
including $m$ nodes to which an ADT selection are applied as its input, and returns the optimal $QLPT$ (Quasi Linear Processing Tree) for $Q$ as its output. Because semi-join can be thought as the join with the result of the projection of the second relation, we add a reducer node as an adjacent node of node $R_i$ to make our problem agrees with the input form of the algorithm.

**Algorithm**

1. Execution of the algorithm in [Yaji91]. As the result of this, $QLPT_1$ is returned.

2. Using the algorithm above on two pairs of relations, $R_i - R_{i-1}$ and $R_i - R_{i+1}$. The two results are compared and the better of the two (pairs of relation and method) is chosen.

3. If the method chosen in (1) is not method 3, end and output $QLPT_1$.

4. If the method chosen in (1) is method 3, make the reducer node the projection of the chosen relation (adjacent node to the node $R_i$) and put it into the tree query as an adjacent node to the node $R_i$. Execute the algorithm in [Yaji91] once again. The result is noted as $QLPT_2$.

5. Compare $\text{cost}(\text{make} - \text{reducer}) + \text{cost}(QLPT_2)$ and $\text{cost}(QLPT_1)$, choose the better one as output and end.
3.4 Summary

We have proposed and described a new scheme for optimizing queries containing ADT functions, based on the concept of decomposition, and semijoin.

An algorithm has been proposed of finding the optimal decomposition of ADT functions. An analytical study based on this model has shown the advantages of our method. An experimental study using a chemical database has shown the feasibility of this method. New methods and an extension of the cost model pose the following open problems:

1. For an actual ADT, the rank of each ADT predicate should vary depending on the sizes of the ADT instances.

2. Since the efficiency of the decomposition depends on the specification (see Section 2) of the ADT predicate given by the DBI, tools that aid DBI in building an efficient specification are necessary.

Another algorithm of using semijoin to reduce the amount of computation of expensive ADT function, has also been discussed. Experiments practiced for the simplest case help us to clarify the property of most of the parameters, and lead to the establishing of the cost model. The algorithm for treating queries with generalized linear form indicates the best QEP. This approach remains the following problems:

1. In linear form, how far will the semijoin be effective?
2. Analysis for more general query forms such as "star" and "tree," is necessary.
Chapter 4

New Buffer Allocation Approaches

4.1 Introduction

Recently, it has become clear that composite objects play a key role in varieties of advanced database applications [Bane88, Care88]. The composite objects mentioned here are the nested objects on which the Is-Part-Of relationship is imposed. The Is-Part-Of relationship is represented by a composite reference, an object reference augmented with the Is-Part-Of relationship. As mentioned in [Kim89b], composite references are further divided into two categories, the exclusive and shared composite references. A shared composite reference from o1 to o2 means that o1 has o2 as its part and o2 is possibly to be shared by o1 and others. In addition to the hierarchical structures of the composite objects, queries on the composite objects usually follow the references repeatedly, as the query examples shown in [Bane88, Kim89b] illustrate. In the support of the composite objects in buffer management systems, the
above mentioned features, which are summarized in the following, must be taken into consideration.

- Two kinds of references are generally supported;
- Objects may be shared;
- Navigation through the two references is a main feature in object queries;
- The reference sequences, which will be accessed more than one time in a given query, are not negligible.

This situation mentioned here is quite different from conventional query processing in relational databases, and the buffer management strategies used in relational databases are less effective in OODBMS. In order to improve the performance of the buffer management system in object-oriented database systems, one of the most important things is to make the reference information available in the buffers. Kemper has mentioned the importance of object access along reference chains leading from one object to another [Kemp90]. We have also observed that many applications tend to traverse one composite object continuously before accessing other composite objects, but during the traversal of one composite object, the number of accesses of other composite objects is not negligible. These observations encourage us to consider the possibility of utilizing the reference information among components of composite objects in buffer management systems, especially for the buffer allocation. Conventionally, it is not permitted for buffer
management systems to see the reference information among composite objects in the buffers, and this has been considered out of the scope of the responsibilities of the buffer management system. We however shall adopt the position that making the reference information available to the buffer management system is harmless.

Related works in relational database management systems are summarized below. LRU (least recently used) and MRU (most recently used) are well-known for their simplicity and effectiveness. Effelsberg discussed the availability of buffers at runtime in [Effe84], and Chou investigated the access patterns of queries in [Chou85]. In order to improve the performance of LRU and MRU, Ng tried to take both into account, and demonstrated the possibility of variable buffer allocation [Ng91]. However, no method was suggested for finding the optimal solutions for the buffer allocation strategy in the paper. Faloutsos introduced two predictors, throughput and effective disk utilization, for predicting the performance of a system, and the buffer allocation decision was made based upon these [Fal91]. The predictors are determined by the availabilities of buffers at runtime, the characteristics of the query, and the dynamic workload on the systems. As pointed out in [Corn89], in relational database systems query optimization and buffer allocation should not be treated independently. The reason is that the holding time and/or transaction response time are generally dependent on both buffer allocation and query optimization strategies. Cornell focused in particular on multiple query processing simultaneously, and an optimization method was developed based on an integer programming model combined with a
queueing model [Corn89].

In most object-oriented database management systems, two kinds of references, composite object references and object references, are generally supported. The former are used to construct composite objects, whereas the latter are used to describe relationships between distinct composite objects [Kim89b]. In Figure 2.1, the former and the latter types of reference are illustrated using solid and dotted lines with arrow heads, respectively. Since the two references influence query processing strategies, the final goal of our research is to support both efficiently in the buffer management system. However, at this stage of progress in our research, our approach is to manage the composite object references in the buffer management system. Note that in the following discussion, we use the term “reference” to mean composite reference.

The idea of supporting the references in the buffer management system results in typical query processing, in which all the conditions imposed on composite objects are applied to a certain composite object at the same time, with this composite object being retrieved if it satisfies all of the conditions before checking other composite objects. Under this query process, it is often the case that units of composite objects will be accessed together.

To informally describe the benefits of our approach, an example of a composite object access sequence:

(#vehicle11 #body13 #chassis14 #cpart2 #designer7 #door15
 #vehicle21 #body13 #chassis14 #cpart2 ...)
is provided to access two composite objects #vehicle11 and #vehicle21 shown in Figure 2.1. As shown, the two composite objects share the component object #body13. Suppose that a buffer can hold up to 5 objects. The buffer becomes (#vehicle11 #body13 #chassis14 #cpart2 #designer7) after the first five objects are accessed. The leftmost object in the buffer is the bottom object, which will be swapped out first. When object #door15 is accessed, the bottom object #vehicle11 is swapped out, and the buffer becomes (#body13 #chassis14 #designer7 #division7 #door15).

For the following access of object #vehicle21, the conventional LRU policy simply swaps out the bottom object #body13. In this way, consecutive object faults will occur when #body13 and #chassis14 are accessed for the second time. And the same phenomenon repeats in the traversal of every composite object. MRU, the optimal replacement policy for looping-references [Ng91], does not work well since this is not a loop access, rather, as the traversal is going on, the objects being accessed change. As mentioned above, because the objects which contain related objects in buffer will usually be accessed in turn, it is worth while keeping all related objects together in the buffer, rather than swapping out some of the objects which are about to be used.

Our approach is described below. When the first five objects are accessed, the buffer becomes Chan(#vehicle11 #body13 #chassis14 #cpart2 #designer7) and #vehicle11 has to be swapped out for #door15, since there are no objects directly or indirectly referenced by #door15 in the buffer. Following that, when #vehicle21 is accessed, our approach is to swap out object #designer7 in-
stead of #body13. This is because there exists an object reference between objects #vehicle21 and #body13, and the buffer then becomes (#body13 #chassis14 #cpart2 #door15 #vehicle21), and it is easy to see that this policy saves objects #body13 and #chassis14 in the buffer when they are needed. The objects which are directly or indirectly referenced by #vehicle11 can be found out in Figure 2.1.

4.2 Access Patterns

Before discussing our buffer allocation strategies, we will define the query evaluation strategy we adopt. We also define several basic access patterns in composite object systems. In the following discussion, the symbols $O$, $o$, $A$ and $a$ stand for composite object schema, composite object, attribute and attribute value, respectively. With regard to composite objects, we will use the notations given in the definition in Section 2.2. The query algebra for relational database systems is very regular, allowing page accessing properties to be easily understandable. In [Chou85], [Ng91] and [Falo91], it is shown that sequential, random, and looping accesses cover all query operations, such as join, index and scan access, and so on. In the following, the basic patterns are introduced for operations in an object-oriented database system. However, it is often the case that an access of queries in an object-oriented database system does not simply correspond to a single pattern, but to a combination of these patterns. In object-oriented database systems, the structure operations on objects are traversals among
reference graphs. In [Bane88b], Banerjee et al suggested three access patterns. The first two patterns are “breadth first search” and “depth first search” which we feel to be less possible in a real composite object traversal; rather, back-tracking is needed.¹

We will concentrate on depth first traversals, with or without back tracking, and random traversals, which are defined as follows.

**Depth First Traversal:** For a composite object $o$, a depth first traversal on $o$, $DFT(o)$, is an access sequence

$$DFT(o) = o DFT(o_1) \ldots DFT(o_n)$$

such that $(o, o_k) \in E_{\text{comp}}$ for $1 \leq k \leq n$.

**Back-tracking Depth First Traversal:** For a composite object $o$, a back tracking depth first traversal on $o$, $DFT_{BT}(o)$, is an access sequence

$$DFT_{BT}(o) = o DFT_{BT}(o_1) o \ldots o DFT_{BT}(o_n) o$$

such that $(o, o_k) \in E_{\text{comp}}$ for $1 \leq k \leq n$.

**Random Traversal:** For a composite object $o$, a random traversal on $o$, $RT(o)$, is an access sequence $RT(o) = o_1 o_2 \ldots o_n$ such that $o = o_1$ and for $1 \leq k \leq n - 1$, $o_k \triangleright o_{k+1}$ or $o_{k+1} \triangleright o_k$.

Figure 4.1 shows several traversals. Figure 4.1 (b), (c) and (d) illustrate the depth first traversal, the back-tracking depth first traversal, and the random traversal, respectively. Sample access sequences “abdece,” “abdbabaceca,” and “abecabdb” correspond to the three types of traversal.

¹The example given by Chan et al [Chan92] confirms this assumption.
The third traversal in [Bane88b] is the "child-depth-first traversal," a hybrid of the breadth-first and depth-first traversals, shown in the following.

**PROCEDURE Children-Depth-First Traversal (P):**

IF node P was not previously visited THEN

DO

Visit node P;

Visit ALL previously unvisited children of P;

For EACH child C of P

CALL Children-Depth-First Traversal (C)

END;

END PROCEDURE.

A query in an object-oriented database system is a sequence of
4.2. ACCESS PATTERNS

traversals. Considering the basic traversal described above, we define access patterns as follow below; a query can be expressed as a combination of such patterns.

**Access Sequence:** An "access sequence" $Acs$ is a sequence of objects $Acs = o_1 o_2 ... o_n$, where $o_i$'s are not necessarily distinct. This means that we first access object $o_1$, then access $o_2$, and so on.

**Sequential Object Access** A "sequential object access" $Acss$ is a composite object traversal $TR(o) = oTR(o_1)...TR(o_m)$.

**Loop Object Access** A "loop object access" $Acsl$ is a sequence $(s_1, ..., s_k)$, where each $s_i$ is a $Acss$ $TR(o^{(i)})$, and all $o^{(i)}$'s have the same composite object schema $O$.

**Random Object Access** A "random object access" $Acsr$ is a $Acs$ whose elements appear in the sequence randomly.

When we search an object without using an index, we must start at well-known objects, which are often the roots. Such access comes under the sequential object access. The above mentioned "child-depth-first" traversal and "Backward/forward queries" by [Kemp90] are just special cases of this. On the other hand, if an index can be used to locate an object directly, we can then access the object directly. In this case, the access comes under random object access pattern. The accesses given above with their special cases cover most of the operations in composite object queries.
4.3 LRU-S

As mentioned above, in the conventional database system the basic access patterns are well known. This makes it easy to analyze the effect of buffer management policies, like LRU, MRU, and so on. Though MRU is known to be efficient for loop access, it collapses when the access is a sequence formed by some random/sequential accesses followed by a loop access. Similarly, LRU loses its efficiency if the access has the form \( s_1 s_0 s_2 s_0 \ldots \), where \( s_i \)'s are sequences of objects and \( s_i \neq s_j \) for \( i \neq j \).

This motivates us to consider a buffer management which takes such reference information into account. Suppose that the buffer manager knows the reference information of each object; it is then possible to keep objects, which are going to be accessed, from being swapped out. To implement this idea, a new buffer management strategy LRU-S is proposed here. LRU-S, with \( S \) meaning "structure", is LRU with an extension which takes into consideration the structures of data. In the following subsections, a formal definition of LRU-S is given, then, qualitative analysis is developed to identify the situations under which LRU-S is beneficial and when it breaks down.

4.3.1 Terminology

In order to formalize the movement of objects in the buffer, the DAG defined in the previous section is revised here. \( N \) stands for a set of nodes of a DAG and the DAG is constructed using the
4.3. **LRU-S**

partial order $\prec$, which is transitive but not reflective. In the object reference interpretation, $a \triangleright b$ means that the node $a$ directly or indirectly refers to the node $b$.

For notational convenience, the reflective extension of $\prec$, denoted as $\preceq$, is defined as

$$a \preceq b \overset{\Delta}{=} a = b \lor a \prec b.$$  

For $a, b \in N$, $\neg(a \prec b \lor a \triangleright b)$ is denoted as $a \perp b$. A set of sequences whose element is in $N$ is denoted as $N^*$. An object access is a sequence of nodes $r = r_1 \ldots r_m$ for $r_i \in N$.

The status of a buffer is a sequence $b = b_1 b_2 \ldots b_s$ for $b_i \in N$, where the tail $b_s$ is the most recently used node. Parentheses are used to articulate an expression to make it comprehensible. Operations on a sequence are summarized as follows. Given a sequence $s$, $s_i$ indicates the $i$'th member of $s$, and the first $k$ elements sub-sequence of $s$ is denoted as $s^{(k)}$. $|s|$ is the length of a sequence $s$ (hence $s^{(|s|)}$ is obviously $s$ itself). $\varepsilon$ is an empty sequence. The meanings of the following notations are obvious.

\[
\begin{align*}
  n \in s \overset{\Delta}{=} n \in \{s_1, \ldots, s_m\}, & \quad n \notin s \overset{\Delta}{=} n \notin \{s_1, \ldots, s_m\}, \\
  hd(s) = s_1, & \quad tl(s) = s_2 \ldots, s_m, \\
  n \perp s \overset{\Delta}{=} \forall i(n \perp s_i), & \quad s \perp n \overset{\Delta}{=} \forall i(s_i \perp n).
\end{align*}
\]

For example, for a buffer $b$, $hd(b)$ gives the head element of $b$, which is the "most recent used" node in, say, LRU; while $tl(b)$ is the status of $b$ after swapping out $hd(b)$. To utilize the information associated with elements in a sequence,

$$s \downarrow_i \overset{\Delta}{=} s_1 s_2 \ldots, s_i$$
and

\[ s \uparrow f \triangleq s \downarrow \neg f \]

are introduced. Here, \( s_i \) satisfies \( f \) for \( i = 1 \ldots t \), and \( l_i < l_{i+1} \) for \( i = 1, \ldots, t-1 \). As an example, because \( \lambda x.x < n \) is a function giving such \( x \) that \( x < n \), recalling the definition of \( < \) it becomes obvious that \( (b \downarrow_{\lambda x.x < n}) \) gives a sequence extracted from the sequence \( b \) whose nodes are directly or indirectly referenced by \( n \).

### 4.3.2 LRU-S Algorithm

Assume that the available buffer size is \( s_{\text{max}} \). Starting from a buffer list \( b \), if the buffer list reaches a state \( b' \) after the reference of an object access \( r \), then the resulting list \( b' \) is denoted as \( b' = b|_r \). The resulting buffer \( b|_n \) is defined as follows:

**Case 1:** If there is a \( k \) such that \( n = b_k \), namely, \( n \in b \), then we define

\[ b|_n = (b \uparrow_{\lambda x.x < n})(b \downarrow_{\lambda x.x < n})n \]

**Case 2:** There is no \( k \) such that \( n = b_k \), namely, \( n \notin b \), then

**Case 2-1:** If the buffer is not full (\( |b| < s_{\text{max}} \)), we define

\[ b|_n = (b \uparrow_{\lambda x.x < n})(b \downarrow_{\lambda x.x < n})n \]

**Case 2-2:** If the buffer is full (\( |b| = s_{\text{max}} \)), then we define

\[ b|_n = \begin{cases} 
  \text{tl}(b)n, & \text{if } (b \uparrow_{\lambda x.x < n}) = \varepsilon \\
  (b'' \downarrow_{\lambda x.x < n'})(b'' \uparrow_{\lambda x.x < n'})(b \downarrow_{\lambda x.x < n})n, & \text{otherwise} 
\end{cases} \]

where \( b'' = (b \uparrow_{\lambda x.x < n}) \) and \( n' = \text{hd}(b'') \).
4.3. \textit{LRU-S}

Figure 4.2 gives an intuitive illustration of LRU-S. Suppose that the object to be accessed is object $n$, and that the buffer stack is $b$, the bottom object of which is the candidate for being swapped out. The following gives an explanation of LRU-S.

1. If $n$ is found in $b$, put it to top of $b$, then, goto 4

2. If $n$ is not found in $b$ and $b$ is not full? then put $n$ to top of $b$, and goto 4.

3. If $n$ is not found in $b$ and $b$ is full? then first check each object in $b$ from the bottom, moving the first object $e$ that is not referenced by $n$ to the bottom of $b$ (If such an $e$ not found, then naturally select the bottom object of $b$ as the swapped out object). Secondly, move all objects in $b$ which are referenced by $e$ but not $n$ to the bottom but one (the bottom is $e$) in $b$, keeping the order as is. Finally, swap out the bottom object ($e$) and read $n$ into top of $b$. End.

4. Extract all objects in $b$ which are referenced by $n$ and put them on the top but $n$ of $b$, keeping the order as is. End.

For each node $n$ in the access sequence, the algorithm checks the nodes in the buffer to find the reference relationship. Accordingly, the complexity of the algorithm is $O(s_{\text{max}} \cdot |Acs|)$ times of doing the comparison $\prec (\preceq)$, where $|Acs|$ is the length of the access sequence. For convenience of comparison, the formal definition of traditional LRU and MRU are also listed in the following.
Figure 4.2: An Example of LRU-S.

**LRU**

\[
b_n = \begin{cases} 
(b \downarrow_{\lambda x. x \in b})n, & \text{if } n \in b \\
bn, & \text{if } n \notin b \land |b| \neq s_{max} \\
(tl(b))n, & \text{otherwise}
\end{cases}
\]

**MRU**

\[
b_n = \begin{cases} 
n(b \downarrow_{\lambda x. x \in b}), & \text{if } n \in b \\
bn, & \text{if } n \notin b \land |b| \neq s_{max} \\
n(tl(b)), & \text{otherwise}
\end{cases}
\]
4.4 Formal Analysis

In this section, we give a qualitative analyst of LRU-S. Let the set of all the resulting lists from an empty list $\varepsilon$ be $B$. In the following, some properties of LRU-S are given. We only prove the first, since the proofs for the other properties are similar.

The following property shows that LRU-S does not "waste" buffer space.

**Property 1** Each node appears in a buffer list at most once. That is, $\forall b \in B \ (n \in b \Rightarrow \exists! j, b_j = n)$,

where $\exists!$ means "uniquely exists".

Proof:
We prove this remark using mathematical induction.
It is trivial when $b = \varepsilon$.
Suppose that the consequence is true for $b$. After accessing node $n'$, $b$ becomes $b'$ ($b' = b|_n$), and we will prove that the consequence is true for $b'$, following the cases of the definition of LRU-S.

Case 1:

(a) If $n = n'$. According to the definition,

$$b' = (b \uparrow_{\lambda x.x \leq n'})(b \downarrow_{\lambda x.x < n'})n'.$$

By the assumption, $n \notin (b \uparrow_{\lambda x.x \leq n'})$, and $n \notin (b \downarrow_{\lambda x.x < n'})$, and accordingly, $n$ uniquely appears at the tail of $b'$.

(b) If $n \neq n'$.

If $n \in b$, then
In either case, \( n \) appears uniquely in \((b \downarrow_{\lambda x.x < n'})\) because of the assumption.

Otherwise,

\[
(n \not< n') \Rightarrow n \notin (b \uparrow_{\lambda x.x \leq n'}), n \in (b \downarrow_{\lambda x.x < n'})
\]

and \( n \) appears uniquely in \((b \uparrow_{\lambda x.x \leq n'})\) because of the assumption. Hence, \( n \notin b' \).

Case 2 is trivial. \(\Box\)

**Property 2** The most recently accessed node is at the tail of the buffer list. That is, for any buffer \( b \) and node \( n \), \( b|n = b'n \) for some \( b' \).

These two properties above are common to LRU, MRU, and LRU-S. The next four properties serve to show the relationship between an access sequence and remaining objects.

**Property 3** For \( b, t \in N^* \), if \( t \perp b \) and \( s_{max} \geq |t| \), then \( t_i \in b|t \)

Since \( t \) is irrelevant to \( b \), the elements in \( b \) are not moved to the right of \( t \). Therefore, after \( t \) is accessed, any element in \( t \) is left in \( b|t \). This means that the original contents of \( b \) are swapped out before \( t \), and guarantees that \( t_i \) remains in the buffer. In the other words, LRU-S works for irrelevant objects (say, random accesses) just as LRU does.
4.4. FORMAL ANALYSIS

Property 4 For $b \in N^*$, $t$, $w_i \in N^*$, if $b \perp t$, $t \perp w_i$, $\forall i \; t_i \leq t_1$, and $s_{\text{max}} \geq |t| + \max_i |w_i|$, then

$$t_k \in b|w_0tw_1tw_2...tw_{k-1}$$

Property 5 For $b \in N^*$, $t$, $u \in N^*$, $w_i \in N^*$, for $b \perp t$, $b \perp u$, $u \perp w_i$, $b \perp w_i$, and

$s_{\text{max}} \geq |t| + |u|$, $s_{\text{max}} \geq |u| + \sum_{1 \leq j \leq k-1} |w_j|, \forall i \; t_i \leq t_1$, then

$$t_k \in b|ut_1uw_1ut_2w_2...w_{k-1}$$

If the buffer is large enough, LRU-S behaves exactly like LRU and MRU. However, within the interval $(\max_i |w_i|, |t| + \max_i |w_i|)$ of buffer size, LRU-S shows a higher probability of keeping the necessary objects in the buffer than LRU does. This probability comes from the utilization of the reference information among objects in $t$. LRU-S increases the chances of keeping $t_i$ in the buffer even if $w_j$’s are accessed between $t_{j-1}$ and $t_j$.

To explain these properties in detail, we consider a simple ideal example: an access sequence of form $sr_1sr_2...sr_k$, where $s$ is an $Acs_s$, and $r_i$ is an $Acs_r$, and $r_1$, $r_2$, ...$r_k$ are different from one another. Suppose that $|s| = x$ and $|r_i| = y$. Let $x > y$ and let the buffer size $|b| = x + \alpha$, $0 < \alpha < y$. Using the notation of the \textit{first k elements sub-sequence} of $s$, $s^{(k)}$ (again, obviously $s^{(x)} = s$, $r^{(y)}_i = r_i$), the behaviors of LRU and LRU-S are compared in the following,
CHAPTER 4. NEW BUFFER ALLOCATION APPROACHES

<table>
<thead>
<tr>
<th>Access point</th>
<th>LRU</th>
<th>LRU-S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$s^{(1)}$</td>
<td>$s^{(1)}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$x + \alpha$</td>
<td>$sr_1^{(\alpha)}$</td>
<td>$sr_1^{(\alpha)}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$x + y$</td>
<td>$s^{(x+\alpha-y)r_1}$</td>
<td>$s^{(x+\alpha-y)r_1}$</td>
</tr>
<tr>
<td>* $x + y + 1$</td>
<td>$s^{(x+\alpha-y-1)r_1s^{(1)}}$</td>
<td>$r_1^{(y-1)}s^{(x+\alpha-y+1)}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>* $x + y + i$</td>
<td>$s^{(x+\alpha-y-i)r_1s^{(i)}}$</td>
<td>$r_1^{(y-i)}s^{(x+\alpha-y+i)}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$x + y + x$</td>
<td>$r_1s^{(x+\alpha-y)}$</td>
<td>$r_1^{(\alpha)}s$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$x + y + x + y$</td>
<td>$s^{(x+\alpha-y)r_2}$</td>
<td>$s^{(x+\alpha-y)r_2}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

From the assumption that $y < x < |b| = x + \alpha < x + y$, at access point $x + y + i$, the numbers of (possibly to be repeatedly accessed) elements of $s$ remaining in the buffer are $x + \alpha - y$ for LRU and $x + \alpha - y + i$ for LRU-S, respectively. Noting that the above discussion holds true even if we exchange $x$ and $y$, we are led to the conclusion that LRU-S performs better than LRU for repeats of $Acs_s$ following a $Acs_{r_i}$ when the buffer size is within the interval $(\max\{|s|, |r_i|\}, |s| + |r_i|)$. More precisely, in the table above, although both LRU and LRU-S raise $x+y$ faults for the first iteration ($sr_1$), LRU and LRU-S raise $x+y$ and $x+y-(|b|-y)$ faults respectively for each $i$-th iteration ($sr_i$, $i > 1$). LRU-S performs better because $|b| > x > y$ hence $x + y - (|b| - y) < x + y$. As
4.4. FORMAL ANALYSIS

an example, we count the number of faults when access $s$ for the second times. In the table above, it can be seen that faults raised at access points $x + y + i$ for $i = 1, \ldots, y - \alpha$. After access points $x + y + (y - \alpha)$, the whole $s$ is in the buffer and no faults raised. The access of each node of $r_2$ raises faults. Therefore, the total number of fault for the second iteration is $(y - \alpha) + y = x + y - (|b| - y)$.

Property 6 For

$b, t, u, w \in N^*, b \perp t, b \perp u, b \perp w, u \perp w, t \perp u, t \perp w, \forall i t_i < t_1,$

and $|t| - 1 + |u| + |w| \geq s_{max}$, then

$$u_1 \notin b|t|_1 w$$

Property 6 shows an unfavorable case to LRU-S, in the cases that the second access of $t_1$ moves the useless nodes $t_2 \ldots t_m$ to the right of the buffer, and pushes the useful nodes $u_1 \ldots$ to the left. This property means that if the shared common sequence $t_1 \ldots t_m$ is not traversed entirely after the root node $t_1$ is accessed, it is wise to avoid accessing the root node $t_1$ at all.

We found an example for which LRU-S performs unusually. For this special access sequence, the node fault rises while the buffer size increases. Figure 4.3 illustrates the example:

Here, we suppose that only 2 $\triangleright 3$, with the other nodes irrelevant to one another. The first line is the access sequence, the two numbers '3' and '4' in the first column are the buffer sizes, each column of the right block represents the status of the buffer stack after the corresponding node of the first line is accessed. The bottom element of the buffer stack is the first one to be swapped.
out when necessary. "*" indicates "fault"; we can see that when the buffer size is increased by one, the number of faults rises. This phenomenon is due to the swapping of node 2 when node 5 is accessed (see "(*)" in the table). In the case of buffer size 3, before node 5 is accessed, the parent node of 3 (that is, 2) is swapped out already, and LRU-S behaves exactly the same as LRU does. By contrast, in the case of buffer size 4, node 2 happens to be at the bottom of the buffer stack and is swapped out for node 5, which is being accessed. Furthermore, because $2 \succ 3$ is found, according to the algorithm of LRU-S, node 3 is drawn to the bottom of the buffer stack, which causes an additional fault when node 3 is accessed the next time.

![Figure 4.3: An Unusual Phenomenon of LRU-S.](image)

<table>
<thead>
<tr>
<th>list</th>
<th>2</th>
<th>4</th>
<th>7</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

| 4    | 2 | 4 | 7 | 3 | 5 | 6 | 3 |
|      | 2 | 4 | 7 | 3 | 5 | 6 | 3 |
|      | 2 | 4 | 7 | 3 | 5 | 6 | 3 |
|      | 2 | 3 | 4 | 7 |   |   |   |
|      | * | * | * | * | * | (c) | * |
4.5 Simulation

In order to evaluate the performance of the LRU-S, experimental results are explained in this section.

4.5.1 Data Set

The data model used is similar to the one proposed in Anderson’s benchmark [Ande91], except that there is no implementation of refTo/refFrom. The database is a rooted tree, as shown in Figure 4.4, where each internal (non-leaf) node at level \( k \) of the tree has five nodes of level \( k + 1 \) as its children, and five nodes of level \( k + 1 \) as it parts. On the other hand, any node of level \( k \) (\( k > 0 \)) can be a child of only one node of level \( k - 1 \), while it may be a part of more than one node of \( k - 1 \).

The parameter considered in the experiments is buffer size. Because I/O plays a predominant role, we measure the performance of buffer management strategies by the rate of object faults, which is the ratio of the number of object faults to the number of object accesses. The following experiments compare LRU-S with both of the well-known LRU and MRU methods. In each figure, the horizontal axis expresses buffer size, while the vertical axis expresses the rate of object faults in percentages.

4.5.2 Empirical Results and Discussions

Experiment 1: Loop of Acs, Acs.
As illustrated in Section 4.2, a query on a certain part of a composite object, which may involve references to component objects if certain conditions are satisfied, gives an object access which is basically a loop of \( Acs_r, Acs_s \). In addition to the buffer size, the average length of an iterator \( Acs_r, Acs_s \) also affects the results. As was pointed out in properties 4 and 5 in the previous section, if the buffer size is within a certain interval, useful objects are kept in the buffer by the LRU-S. By contrast, under LRU or MRU, such useful objects are simply swapped out. We call the interval in which LRU-S performs better the “desired interval”. The desired interval of the buffer size in these experiments is \((s_1, s_2)\), where \(s_1\) and \(s_2\) are approximately equal to \(\max\{|Acs_s|,|Acs_r|\}\) and \(|Acs_r|+|Acs_s|\).
respectively. Figure 4.5 and 4.6 give two extreme results of the typical access pattern: the repeat of an $Acs_s$ following an $Acs_{ri}$. These two access sequences are obtained in the same way: one traversal of a subtree of the database shown in Figure 4.4 is taken as the $Acs_s$, and access sequences to some of the other nodes are taken as the sum of all $Acs_{ri}$. To make the explanation understandable, the $Part/Partof$ relationships are not considered here, and any element of $Acs_{ri}$ is not taken from any ancestor of the subtree being taken as the $Acs_s$. This guaranteeing that $Acs_s$ and $Acs_{ri}$ do not intersect or affect one another. According to the formal analysis in Section 4.4, LRU-S performs better than LRU-S when a buffer is of a size within the interval $(\max\{|Acs_s|, |Acs_{ri}|\}, |Acs_s| + |Acs_{ri}|)$. From Figure 4.5 and 4.6, we can readily see that LRU-S performs better than LRU in (30, 56) and (40, 70), respectively. The "hot point" of
LRU clearly appears in both figures. We also add the curve which is the theoretical result of LRU-S based on the analysis given in Section 4.4 to Figure 4.5. Focusing on the desired interval (30, 56), it can be easily seen that the result of experiment gives good agreement with the theoretical one.

Figure 4.7, 4.9 and 4.9 show more other results about the desired intervals. For example, in Figure 4.7, LRU-S behaves better than both LRU and MRU in the interval (45, 80). Here, (45, 80) is approximately equal to the desired interval (|Ac,|, |Ac,| + |Ar|). Experiments were also conducted for different |Ac,| and |Ar| values, and the results shown in Figure 4.8 and 4.9. As predicted, proof that LRU-S behaves better than LRU and MRU in the desired interval can be found in Figure 4.8 and Figure 4.9. These
4.5. SIMULATION

Figure 4.7: Loop of $Acs_s, Acs_r$ (\(|Acs_s| \approx 45, |Acs_r| \approx 30\))

figures different from the previous two in that the \textit{Part/Partof} relationships are active here. These relationships affect the performance because the condition "$Acs_s$ and $Acs_r$ do not intersect one another" is not satisfied. The three strategies behave similarly outside the desired interval. This means that LRU-S is not less effective than LRU or MRU, even outside the desired interval in which it is more effective. Finally, the relationship \textit{Part/Partof} affect the results in that because of sharing, $Acs_s$ and $Acs_r$ do not intersect clearly. This in turn affect the desired interval and the shapes of the result curves in the figures. The effect of the relationship \textit{Part/Partof} must be investigated further.
Experiments 2: Looping Access

Data in these experiments is taken by scanning objects of (say) \( k' \)th level of the database shown in Figure 4.4, and at the same time traversing the subtrees rooted at these \( k' \)th objects. In order to tuning the rate of shared, number of \textit{Part/Partof} links in the database are changed. The rate of shared objects and buffer size are considered in this experiment. Using the definition of "reference" in Section 4.4, we calculate the rate of shared objects in the following way: for each component object \( o_i \) of a composite object, the number, say \( d \), of direct references to \( o_i \) is counted. Then \( \max\{d - 1, 0\} \) is named the "referenced degree" of \( o_i \). The average rate of shared objects is calculated by averaging the referenced degrees over all composite and component objects. As mentioned
above, composite object traversal is a special case of sequential access. In this case, the desired interval is approximately \((s \times r, s)\), where \(r\) is the rate of shared objects and \(s\) is equal to the average number of component objects referenced by a composite object.

Figure 4.10 and 4.11 illustrate two results: Figure 4.10 the case for a rate of shared objects of 30% and Figure 4.11 the case for a rate of shared object of 50%. In Figure 4.10 LRU-S explicitly behaves better than LRU and MRU. In Figure 4.11, LRU-S behaves better than LRU in the interval, and is not less effective than MRU. It can be seen that MRU is affected by the location where the common part of the access sequence appears. If the common part appears in the head of the access sequence, then MRU should perform like LRU-S, which is the case of Figure 4.11. Otherwise, as shown in
Figure 4.10, LRU-S works better than MRU. Changing the average rate of shared objects affects the desired interval. Finally, if the buffer is not initially empty but contains something irrelevant to the access sequence, then LRU and LRU-S will be affected slightly while MRU is seriously degraded. This consequence is confirmed by comparing Figure 4.10 and Figure 4.11.

![Graph](image)

**Figure 4.10: Looping Access (Shared 30%, Number of nodes ≈ 22)**

Figure 4.12 shows a result of an access of another kind of loop. For a tree database of height 4 as shown in Figure 4.4, we repeatedly and randomly access the objects of level 2, and traversal the subtree every time the of level 2 is touched. It can be seen that LRU-S always performs better than LRU, and almost better than MRU. From this figure, we can see one evident advantages of LRU-S over LRU, which is that LRU-S is not affected by the "hot points,"
while LRU is affected heavily.

Experiment 3: Sequential and Random Access

A single $Ac_n$ behaves like a random access and makes no difference between LRU, MRU or LRU-S. LRU-S is not less effective for $Ac_n$ even if there is no reference information available. Similarly, LRU-S is not less effective even if the paths in $Ac_n$'s are randomly distributed. The reason can be found from property 3 given in the Section 4.4. An result is given in Figure 4.13. In this experiment, the access sequence is obtained by randomly selecting 2000 nodes from the database shown in Figure 4.4.
Figure 4.12: Looping Access (Shared 50%, Number of nodes ≈ 40)

4.6 Discussion on Extension of LRU-S

As discussed in the previous sections, LRU-S takes into consideration the reference information between objects. However, LRU-S does not distinguishes the component objects. In order to exactly express locality, we introduce the concept of “distance”. Two distances between composite objects are introduced.
4.6. DISCUSSION ON EXTENSION OF LRU-S

4.6.1 The Concept of Distances

Definition (distance) The distance of objects $a, b$ is defined as follows:

$$
\text{dist}(a, b) = \begin{cases} 
0, & \text{if } a = b \\
1, & \text{if } a \triangleright b \lor b \triangleright a \\
\min\{\text{dist}(a, c) \mid \text{dist}(c, b) = 1\} + 1, & \text{if } \{\text{dist}(a, c) \mid \text{dist}(c, b) = 1\} \neq \emptyset \\
\infty, & \text{otherwise.}
\end{cases}
$$

$$
\text{dist}^\triangleright(a, b) = \begin{cases} 
0, & \text{if } a = b \\
1, & \text{if } a \triangleright b \\
\min\{\text{dist}^\triangleright(a, c) \mid c \triangleright b\} + 1, & \text{if } \{\text{dist}^\triangleright(a, c) \mid c \triangleright b\} \neq \emptyset \\
\infty, & \text{otherwise.}
\end{cases}
$$

Figure 4.13: Random Access
It must be noted that the "distance" we mention differs from the mathematical concept. That is, it may not satisfy the axioms of distance. For example, $\text{dist}^r(a, b) = \text{dist}^r(b, a)$ does not always hold true, and so on.

4.6.2 LRU-D

Using the notations above, we now define a new buffer allocation strategy LRU-D. LRU-D is important to keep the objects, which have already been accessed and will be accessed soon, in the buffer. Combining an appropriate buffer allocation strategy with a clustering technique will improve the performance of the clustering technique. The effect is to be investigated in details.

$$b|_n = \begin{cases} 
    b \downarrow_{x \cdot \text{dist}(x, n) = \infty} \ldots \downarrow_{x \cdot \text{dist}(x, n) = 1} n, \\
    \text{if } n \in b \\
    t \downarrow_{b \downarrow_{x \cdot \text{dist}(x, n) = \infty} \ldots \downarrow_{x \cdot \text{dist}(x, n) = 1} n, \\
    \text{otherwise}
\end{cases}$$

For an $\text{Acs} = n_1 \ldots n_k$, suppose that the distinct nodes in $\text{Acs}$ are $n_{l_1} \ldots n_{l_m}$, and $n_{l_i}$ repeats $s_i$ times in $\text{Acs}$, for $i = 1 \ldots m$ (hence $\sum_{i=1}^m s_i = k$), and suppose that for each $i$, $1 \leq i \leq m - 1$, $s_i \geq s_{i+1}$. Then, the optimal buffer strategy should keep $n_{l_i} \ldots n_{l_i}$ in the buffer if the buffer size, $t$, is smaller than $m$. On the other hand, "local optimal" only guarantees that when a node $n_i$ is claimed, the buffer allocation strategy does not swap out $n_{i+1}$ if the buffer of a reasonable size is allocated.
**Property 1.** LRU-D is local optimal for a random traversal $RT$ and for a back-tracking depth first traversal $DFT_{BT}$, if a buffer of a reasonable size is allocated.

We will only explain the case of $DFT_{BT}$. According to the access patterns mentioned above, if the current claimed node is $n_i$, then $n_{i+1}$ should be in $\{b \downarrow_{x, \text{dist}(x,n_i)=1}\}$, if $n_{i+1}$ is in the buffer. By the definition of LRU-D, this $n_{i+1}$ is kept in the buffer $b$, if possible.

### 4.7 Summary

A new strategy, LRU-S, which take into consideration the reference information among objects is proposed for buffer management of an object-oriented database system. The formal definition of LRU-S makes it possible to explicate some of its properties. Using these properties, we determined the intervals of buffer size in which LRU-S exhibits its features, and discovered the situations for which LRU-S is not suitable. The unfavorable cases can be easily avoided by slightly modifying the query pattern. Consequently, LRU-S behaves in a manner comparable to LRU for conventional applications, and works better than both LRU and MRU in many cases of possible composite object queries. The results of the experiments above confirm this conclusion. This approach opens up a new direction for improving the performance of buffer management systems in object oriented database systems.

We will further investigate the effect of the rate of shared objects on the behaviors of LRU-S. Since the performance of a buffer
replacement strategy depends heavily on the query patterns, alternative query patterns should be further investigated. This motivates us to analyze the distribution of query patterns in object oriented database systems.
Chapter 5

Conclusions

5.1 Discussions on Implementation Issues

An example system which supports ADT is ADT-INGRES [Ston83, Ston86c]. INGRES, a relational database system developed at the University of California, Berkeley [Ston76], took the approach of support ADTs as the columns of a relation. INGRES has been extended to allow the database designer to specify new column types and operators on these data types using user defined procedures that obey a specialized protocol. This use of ADTs is a generalization of database experts.

ADT-INGRES provides a novel way of specifying new data types and corresponding operators in a database management system. However, it does not provide any additional support for handling hierarchical data structures that occur frequently in engineering applications. ADT-INGRES provides some facilities for behavioral object orientation by allowing the database user to define application-specific operations. However, these operations are
quite tedious to implement because internally, the model is not structurally object oriented.

Extensible systems must allow (sophisticated) programmers to add many kinds of extensions to the capabilities of the base system. A rule-based optimizer meets the requirement of supporting these features, because of its high extensibility [Frey87, Haas89, Lohm88, Ston87a]. Using the notation in [Frey87], Our approach of “decomposition” and “using join/semijoin” can be combined into a rule-based optimizer.

\[
\text{Isomorphic}(G, g_0) \xrightarrow{C} \\
\text{(nodeEQ}(G, g_0) \land \text{edgeEQ}(G, g_0) \land \ldots)\ldots
\]

Furthermore, the use of indices as a way of implementing efficient access to abstract data types can also be considered. An index is really nothing more than a precompiled search. In other words, we store lists of all objects that would be returned if we were to search a set for all occurrences of objects that match some criterion. Usually this criterion requires that a key value be equal to a given constant. By using the index, we avoid having to redo the search for these objects. With abstract data types, we could potentially base an index on the results of applying various functions. That is, we could keep a list of all objects \(x\) that would return a given value \(v\) if a function \(f\) were to be applied to \(x\). A set of these lists for all possible \(v\) would be an index.

\[
I_f = \{(v_i, (x_1, ..., x_n)) | f(x_i) = v_i, 1 \leq i \leq n\}
\]

Notice that \(f\) is a function of a single argument. Of course, it is conceivable that it will also be necessary at times to deal with the
case of more general functions with multiple arguments. However, restricting our attention to functions of one argument, which result in no side-effects, simplifies our approach to query processing.

Our data abstraction facilities allow property values of ADT to be computed in arbitrary ways. In many cases, this operation simply looks up the value of the property from a stored piece of the representation. However, it is possible for this operation to compute a complex function based on several pieces of the representation. This conveniently allows a straightforward way to incorporate derived properties into the model. It potentially complicates our ability to analyze query expressions since we cannot always transform them directly into simple manipulations of the underlying storage.

The existence of an index by-passes this problem. If we have the value of a property stored in an index, it becomes irrelevant at retrieval time how this value was derived. It becomes available by performing a lookup on the index. Suppose that there exists an index on the file of the number of nodes and edges of ADT \textit{GRAPH}, then the rule given above can be further rewritten as

\[
\cdots \text{Isomorphic}(G, g_0) \xrightarrow{C} \\
\cdots ((x_G = \text{node}(g_0)) \land y_G = \text{edge}(g_0) \land \cdots)...
\]

Where

\[
C = ((\text{Ind}(x_G, T(G), G, \text{node})) \land (\text{Ind}(y_G, T(G), G, \text{edge})))
\]

On the other hand, LRU-S and LRU-D manage the object buffer, based on the assumption that all objects are of uniform size. One way to release this restrain is to combine LRU-S and
LRU-D with the technique of clustering, which is shown in Figure 5.1. In this figure, (a) is a schema of a composite object, (b) is an example instance of the composite object, and (c) shows the corresponding pages of instance (b), clustered using a “reasonable” clustering.

As mentioned in Chapter 2, we concentrated on relational models. Associated with the implementation of LRU-S and LRU-D, one way to store the objects is based on nested relational model. The nested relational model, also sometimes called NF$^2$ (non-first normal form relational model), is a generalization of the traditional relational model without the first normal form restraints. Attributes of a nested relation can have non-atomic values. Their values can be relations which can hence be viewed as subrelations of the nested relation. This model was first proposed by Makinouchi [Maki77]. Then a number of researchers
5.1. DISCUSSIONS ON IMPLEMENTATION ISSUES

([Hafe88, Kita89, Leve92, Maki77, Thom86]) have given formulations for the nested relational model.

The nested relational model implicitly incorporates references to tuples of different relations. It supports composite attribute types, which can be either tuple valued. A composite attribute could also be a list of elements. All these structures can be arbitrarily nested. Thus it is really a hybrid of the relational and the hierarchical data model. However, th nested relational models including the NF$^2$ model do not provide for the definition of application-specific operations, and the support of recursive data structures such as composite objects discussed in this thesis.

The problem in implementing LRU-S and LRU-D is

how to manage the reference information among the objects?

Basically, it is meaningless if the reference information about an object remains unknown until the object is read into the buffer. Therefore, one way we under consideration is to separate an object into a data part and a pointer part. The pointer part stores the reference information and is kept in the buffer.

To reduce the overhead of managing the reference information, the following approaches are to be considered.

- Limiting the reference information to the buffer, a reasonable approach considering the theory of hot-spots.
- Compression of reference information;
- Sharing of the reference information with other operations, resulting in a relative reduction of the cost; for example, the
pointers mentioned above can be used for doing graph operations.

5.2 Summary

The primary contributions of this thesis can be summarized as follows:

- Uniforming the discussion of the ADT and composite objects in an extended relational model.

- Establishment of a cost model for processing queries including "expensive" ADT functions.

- Proposal of the approaches of "decomposition" and "using join/semijoin" to optimize the queries mentioned above.

- Clarification of the influence of reference relationships among the component objects of a composite object on buffer allocation.

- Proposal of LRU-S
Appendix A

Proof of the Consequences in Chapter 3

A.1 Proof of Lemma 1

We are to prove that if $p_1 < p_2 < \ldots < p_n$, then

$$\text{cost}((f_{k_1}, \ldots, f_{k_i}, \ldots f_{k_n})) \geq \text{cost}((f_1, \ldots f_n)).$$

Since there must be a $i$ such that $k_i = 1$, let $f_1$ be the function with the smallest rank: $p_1 < p_i$, $\forall i \neq 1$. From the discussion that "$p_i < p_{i+1}$ implies $\text{Cost}(\tau 1) < \text{Cost}(\tau 2)$", we have:

$$\begin{align*}
\text{cost}((f_{k_1}, \ldots, f_{k_{i-1}}, f_{k_i}, f_{k_{i+1}}, \ldots, f_{k_n})) \\
= \text{cost}((f_{k_1}, \ldots, f_{k_{i-1}}, f_1, f_{k_{i+1}}, \ldots, f_{k_n})) \\
\geq \text{cost}((f_{k_1}, \ldots, f_1, f_{k_{i-1}}, f_{k_{i+1}}, \ldots, f_{k_n})) \\
\geq \ldots.. \\
\geq \text{cost}((f_1, f_{k_1}, \ldots f_{k_{i-1}}, f_{k_{i+1}}, \ldots, f_{k_n})).
\end{align*}$$

Similarly, there is a $j$ such that $k_j = 2$, and $\forall s$, $s \neq i$, $s \neq j$, $p_1 \leq p_2 \leq p_k$. So, in the same manner, we can get (without losing
generality, we can assume that \( j > i \)

\[
\text{cost}((f_{k_1}, \ldots, f_{k_{i-1}}, f_{k_i}, f_{k_{i+1}}, \ldots, f_{k_n})) \\
\geq \text{cost}((f_1, f_{k_1}, \ldots, f_{k_{i-1}}, f_{k_{i+1}}, \ldots, f_{k_n})) \\
\geq \cdots \\
\geq \text{cost}((f_1, f_2, f_{k_1}, \ldots, f_{k_{i-1}}, f_{k_{i+1}}, \ldots, f_{k_{j-1}}, \ldots, f_{k_{j+1}}, \ldots, f_{k_n})).
\]

Repeat this procedure, we finally reach

\[
\text{cost}((f_{k_1}, \ldots, f_{k_{i-1}}, f_{k_i}, f_{k_{i+1}}, \ldots, f_{k_n})) \\
\geq \cdots \\
\geq \text{cost}((f_1, f_2, \ldots, f_n))
\]

### A.2 Proof of Lemma 2 (Supplement)

For the case \( j < i \), the proof of

\[
\lambda_{i+1} + d_{i+1}\lambda_{i+2} + \ldots + d_{i+1}\cdots d_{i+k-1}\lambda_{i+k} \geq p_{i+1}(1 - d_{i+1}\cdots d_{i+k})
\]

is given in the following using mathematical deduction:

From the definition of \( p_i \) and \( p_{i+1} \leq p_{i+2} \),

\[
\lambda_{i+1} + d_{i+1}\lambda_{i+2} \geq \lambda_{i+1} + d_{i+1}(1 - d_{i+2})\lambda_{i+1}/(1 - d_{i+1}) \\
= \lambda_{i+1}(1 - d_{i+1}d_{i+2})/(1 - d_{i+1}) \\
= (1 - d_{i+1}d_{i+2})p_{i+1}
\]

Suppose that

\[
s_k = \lambda_{i+1} + d_{i+1}\lambda_{i+2} + \ldots + d_{i+1}\cdots d_{i+k-1}\lambda_{i+k} \\
\geq p_{i+1}(1 - d_{i+1}\cdots d_{i+k})
\]
then

\[ s_{k+1} = s_k + d_{i+1}\ldots d_{i+k} \lambda_{i+k+1} \]
\[ \geq p_{i+1}(1 - d_{i+1}\ldots d_{i+k}) + d_{i+1}\ldots d_{i+k+1} p_{i+k+1}(1 - d_{i+k+1}) \]
\[ \geq p_{i+1}(1 - d_{i+1}\ldots d_{i+k}) + d_{i+1}\ldots d_{i+k+1} p_{i+1}(1 - d_{i+k+1}) \quad (p_{i+k+1} \geq p_{i+1}) \]
\[ = (1 - d_{i+1}\ldots d_{i+k+1}) p_{i+1} \]

Proving Lemma 2 for case \( j > i \).

In order to simplify the proof, we introduce a composite function \( \hat{f} \). A composite function \( \hat{f} \) of \((f_i, \ldots, f_j)\) is a function with the evaluation cost

\[ \hat{\lambda} = (\lambda_i + d_i \ast \lambda_{i+1} + \ldots + d_i\ldots d_{j-1} \lambda_j), \]

and the false drop \( \hat{d} = d_i\ldots d_{j-1} \). Let \( \omega(t) = I \ast (\lambda_1 + \ldots + \lambda_t) \), then

\[ \text{Cost}(\tau) = \omega(t) + (1 - I)(\lambda_1 + d_1 \lambda_2 + d_1 d_2 \lambda_3 + \ldots + d_1 d_2\ldots d_{t-1} \lambda_t) \]
\[ = \omega(t) + (1 - I)(\lambda_1 + d_1 \lambda_2 + \ldots + d_1\ldots d_{i-2} \lambda_{i-1}) \]
\[ + d_1\ldots d_{i-1}(\lambda_i + d_i \lambda_{i+1} + \ldots + d_{j-1} \lambda_j) \]
\[ + d_1\ldots d_{i-1} d_i\ldots d_j \lambda_{j+1} + \ldots + d_{i+1}\ldots d_{t-1} \lambda_t) \]
\[ = \text{Cost}((f_1, \ldots, f_{i-1}, \hat{f}, f_{j+1}, \ldots, f_t)). \]

The rank \( \hat{p} \) of \( \hat{f} \) is \( \hat{\lambda}/(1 - \hat{d}) \).

\[ \hat{p} = \frac{\hat{\lambda}}{1 - \hat{d}} \]
\[ = (\lambda_i + d_i \ast \lambda_{i+1} + \ldots + d_i\ldots d_{j-1} \lambda_j)/(1 - d_i\ldots d_j). \]
\[ \leq (\frac{\lambda_{i+1}}{1 - d_{i+1}}(1 - d_i) + d_i \lambda_{i+1} + \ldots)/(1 - d_i\ldots d_j) \]
\[ = (\frac{\lambda_{i+1}}{1 - d_{i+1}}(1 - d_i d_{i+1}) + \ldots)/(1 - d_i\ldots d_j) \]
\[
\frac{\lambda_j}{1 - d_j} (1 - d_i...d_j)/(1 - d_i...d_j) = p_j.
\]

The omitted part based on the following general expression:
\[
\forall k, i < k \leq j,
\]
\[
\frac{\lambda_{k-1}}{1 - d_{k-1}} (1 - d_i...d_{k-1}) + d_i...d_{k-1}\lambda_k 
\leq \frac{\lambda_k}{1 - d_k} (1 - d_i...d_{k-1}) + d_i...d_{k-1}\lambda_k 
= \frac{\lambda_k}{1 - d_k} (1 - d_i...d_k)
\]

Similarly \( p_i \leq \hat{p} \).

Now, define \( \hat{f}_1 = (f_1, ..., f_{i-1}) \), \( \hat{f}_2 = (f_{i+1}, ..., f_{j-1}) \), and \( \hat{f}_3 = (f_{j+1}, ..., f_t) \), for \( \tau = (f_1, ..., f_t) \). Suppose that Lemma 2 does not hold, that means, "(3.3) \( \rightarrow \) (3.4)" is false. We transform this expression as follows,
\[
\neg((3.3) \rightarrow (3.4)) \iff \neg((\neg(3.3) \lor (3.4))) 
\iff ((3.3) \land \neg(3.4))).
\]

This is to say,
\[
\text{Cost}(\tau - \{f_j\}) > \text{Cost}(\tau - \{f_i\} - \{f_j\}) \quad \text{(3.3).}
\]

and
\[
\text{Cost}(\tau) \leq \text{Cost}(\tau - \{f_i\}). \quad \text{\neg(3.4).}
\]

Using the notation mentioned above, this assumption can be expressed as
\[
\text{Cost}(\hat{f}_1, f_i, \hat{f}_2, \hat{f}_3) > \text{Cost}(\hat{f}_1, \hat{f}_2, \hat{f}_3)
\]
A.3. PROOF OF THEOREM

and

\[ \text{Cost}( (\hat{f}_1, f_i, \hat{f}_2, f_j, \hat{f}_3) \leq \text{Cost}(\hat{f}_1, \hat{f}_2, f_j, \hat{f}_3). \]

That is,

\[
\begin{align*}
\text{Cost}( (\hat{f}_1, f_i, \hat{f}_2, f_j, \hat{f}_3) - \text{Cost}(\hat{f}_1, \hat{f}_2, f_j, \hat{f}_3) \\
= I \cdot \lambda_i + (1 - I)\hat{d}_1(\lambda_i - (1 - d_i)(\hat{\lambda}_2 + \hat{d}_2 \lambda_j)) \\
> 0 \quad (A.1)
\end{align*}
\]

and

\[
\begin{align*}
\text{Cost}(\hat{f}_1, \hat{f}_2, f_j, \hat{f}_3) - \text{Cost}(\hat{f}_1, f_i, \hat{f}_2, f_j, \hat{f}_3) \\
= -I \cdot \lambda_i + (1 - I)\hat{d}_1(\lambda_i - (1 - d_i)(\hat{\lambda}_2 + \hat{d}_2 \lambda_j + \hat{d}_2 \hat{d}_j \lambda_3)) \\
\geq 0 \quad (A.2)
\end{align*}
\]

Hence,

\[
(A.1) + (A.2) = (1 - I)\hat{d}_1(1 - d_i) \lambda_j - (1 - d_j) \hat{\lambda}_3 \\
> 0.
\]

However, because \((1 - I)\hat{d}_1(1 - d_i) \hat{d}_2 > 0\), we must have

\[
\frac{\lambda_j}{(1 - d_j)} > \hat{\lambda}_3.
\]

This contradicts the assumption that \(p_j < \hat{p}_3\). So, the assumption that "\(\neg((3.3)) \rightarrow (3.4))\)" cannot be true. In other words, Lemma 2 holds.

A.3 Proof of Theorem

We are to prove:
1. The optimal decomposition must be of the form 
\((f_{i_1}, f_{i_2}, ..., f_{i_k}, f)\).

2. The optimal decomposition can only be yielded from a \(\tau - E_0\) where any element of \(E_0\) belongs to \(S\) — the result of step 3 of the algorithm.

3. The search of the algorithm covers the whole search space (the binary tree shown in Fig3.2) except the nodes with higher evaluation cost.

By 1) and 2), it is certain that the optimal decomposition can be found in the search space of the algorithm. By 3) (as well as 1) and 2)), the result of the algorithm is the optimal one.

1). Evidently, if \(\tau\) is an optimal decomposition of \((f, f_1, ..., f_n)\), then by Lemma 1, \(\tau\) must be of the form

\[ (f_{i_1}, f_{i_2}, ..., f_{i_k}, f, f_{i_{k+1}}, ...) \]

with their ranks satisfy

\[ p_{i_1} \leq p_{i_2} \leq ... \leq p_{i_k} \leq p_0 \leq p_{i_{k+1}} \leq ... \]

In addition, let \(C_k = Cost((f_1, ..., f_k))\), then repeating

\[ C_{i+1} - C_i = I * \lambda_{i+1} + (1 - I) * d_i * \lambda_{i+1} \]

leads to

\[ \forall i < j, C_i \equiv Cost((f_1, ..., f_i)) \leq Cost((f_1, ..., f_j)) \equiv C_j. \]

Especially, \(\text{Cost}\)

\[ (f_{i_1}, f_{i_2}, ..., f_{i_k}, f) \leq \text{Cost}(f_{i_1}, f_{i_2}, ..., f_{i_k}, f_{i_{k+1}}, ...). \]
Hence the optimal sequence found in the result of step 1 and 2 of algorithm will also be the global optimal one.

2). Suppose that \( \tau - E_0 \) is the result of our algorithm, and \( \tau - E'_0 \) is the optimal decomposition strictly better than \( \tau - E_0 \). Define \( E'_{01} = E_0 \cap S, E'_{02} = E_0 \cap (\neg S) \). If \( E'_{02} = \phi \), then \( E'_0 \) is in the search space of the algorithm, and \( \tau - E_0 \) must be better than \( \tau - E'_0 \).

Suppose \( E'_{02} = (f_{e_1}, ..., f_{e_k}) \). From the definition of \( S \),

\[
\text{Cost}(\tau) < \text{Cost}(\tau - \{f_{e_1}\}).
\]

If

\[
\text{Cost}(\tau - E'_{01}) \leq \text{Cost}(\tau - E'_{01} - \{f_{e_1}\})
\]

\[
\text{Cost}(\tau - E'_{01} - \{f_{e_1}\}) \leq \text{Cost}(\tau - E'_{01} - \{f_{e_1}\} - \{f_{e_2}\})
\]

... ...

\[
\text{Cost}(\tau - E'_{01} - \{f_{e_1}\} - ... - \{f_{e_{k-1}}\}) \\
\leq \text{Cost}(\tau - E'_{01} - \{f_{e_1}\} - ... - \{f_{e_k}\}),
\]

then adding these expresses together gives

\[
\text{Cost}(\tau - E'_{01}) \leq \text{Cost}(\tau - E'_{01} - \{f_{e_1}\} - ... - \{f_{e_k}\})
\]

and

\[
\text{Cost}(\tau - E_0) \leq \text{Cost}(\tau - E'_0).
\]

This contradicts the assumption that \( E'_0 \) is the optimal sequence. So there is at least one \( j \) such that

\[
\text{Cost}(\tau - E'_{01} - \{f_{e_1}\} - ... - \{f_{e_{j-1}}\}) > \text{Cost}(\tau - E'_{01} - \{f_{e_1}\} - ... - \{f_{e_j}\}).
\]

By repeatedly applying (3.4) of Lemma 2,

\[
\text{Cost}(\tau) > \text{Cost}(\tau - \{f_{e_j}\}).
\]
This leads to \( f_{e_j} \in s \), and contradicts to the definition of \( E_0' \).

3). The whole search space can be expressed as a binary tree shown in Fig 3.2. The algorithm recursively searches the whole tree except those subtrees which are cut at step 5) of the algorithm by the condition

\[
\text{Cost}(\tau - E - f_{i_j}) > \text{Cost}(\tau - E).
\]

On the other hand, by 2) of Lemma 2, there will not be any \( f_{i_j} \) in such subtrees which satisfies

\[
\text{Cost}(\tau - E - f_{i_j} - f_{i_j}) < \text{Cost}(\tau - E - f_{i_j}).
\]

Consequently, the optimal result will not be lost.
Appendix B

Survey of Advanced DBS Proposals

With respect to the support advanced applications, many paradigms have been studied recently. There has been an emergence of numerous data models, and numerous approaches with the identical aim of providing increased expressiveness to the modeler and incorporating a richer set of semantics into databases.

B.1 Semantic Data Models

The semantic data models [Bati86, Peck88] can be categorized as structural approaches. Although the relational data model has provided users with a modeling methodology independent of the details of physical implementation, the relational data model does not offer a sufficiently rich conceptual model for handling applications that can not be mapped onto tables in a natural way. The semantic data models which have been proposed attempt to cap-
ture more semantics in database systems for supporting CAD data model, computer graphics design, etc.

On the other hand, the abstractions such as generalization and aggregation, which had been adopted by psychologists and AI researchers, were first introduced into database design by Smith and Smith [Smit77a, Smit77b]. Generalization [Smit77a] is the means by which differences among similar objects are ignored to form a higher order object where the similarities can be emphasized. Aggregation [Smit77b] is the means by which the relationships between low-level objects can be considered as a higher level object. In 1984, a modeling construct called "molecular object" was introduced by Batory and Buchmann [Bato84, Bato85]. This concept does not deal with columns of a relation, but rather treats collections of heterogeneous tuples as single objects. These molecular objects have the property that they are given different representations at different levels of abstraction. At higher levels, a molecular object is represented by a single tuple. At lower levels, it is represented by a set of interrelated tuples in different relations. To a large extent, these semantic models have been proved to be useful approaches to specify the CAD applications.

Interest in semantic data models began in the mid-seventies and was, to some extent, a reaction to the simplicity of the relational model. In this model, any semantics that could not be expressed directly in terms of relations would have to be embedded in the application code. The semantic data models include higher-level modeling constructs so that a database designer can assert more information about the data in a way that the database system
There were no commercial implementations of these semantic data models during the period of active research. However, semantic data models have had a strong historical influence on object-oriented databases, for which there is currently a strong commercial interest.

SDMs include object identity and usually a type hierarchy. They do not commonly include the notions of data abstraction or of user-defined operations (i.e., methods). It is the addition of these last two features that distinguish object-oriented data models from SDMs. It is data abstraction and operations that allow object-oriented models to be extended in arbitrary ways. SDMs have at times been called structurally object-oriented (as opposed to behaviorally object-oriented). Because they lack data abstraction and operations, they are not extensible models. As a result, we tend to see many built-in modeling facilities.

One of the goals of SDM research was to produce a semantically accurate data model. That is, for each entity or idea in the application space, there should be a single natural choice of modeling construct.

The SDM was designed for implementation as a more expressive interface over a relational system. The designers attempted to distill their experience in designing databases to select the most common data semantics and to incorporate these into a single data model. It is only recently that a subset of these models has been made into a product by Unisys called SIM. A toy implementation
was used by Dennis McLeod to build an Interaction Formulation Advisor Program (IFAP). It presumes some users do not have sufficient knowledge of the database contents to form a SQL query. The IFAP leads the user through a dialog about information stored in the database through questions concerning user interest. This dialog is driven by the high-degree of semantics captured in an SDM schema. In this way, the SDM schema and the IFAP interface provided their user with a very convenient semantic data dictionary. The SDM schema provides documentation on database structure that can be exploited by a conversational interface.

Daplex [Ship81] was developed at Computer Corporation of America as a response to models like the SDM. It attempts to express concepts like those introduced in the SDM in a "concise conceptual framework." In other words, it uses a notation of mathematical functions as the basis upon which higher-level semantic concepts can be built. The Daplex view is that everything is a function. Even persistent names of data items are modeled as functions. e.g., the name Intersections is a zero-argument function that evaluates to a set of Intersections. Daplexs adopts this view in order to introduce simplicity and uniformity into a complex modeling environments like that proposed by the SDM. The Daplex query language is cast in terms of a built-in set of iterators that apply a predicate to a set of values.

It should be noted that although SDM and Daplex were both data models and not full programming languages, they each inspired a database programming language based on them. DIAL was the language that extended SDM, and ADAPLEX [Smit83]
was a programming language based on ADA and Daplex. TAXIS [Myl080], which was developed during the same period as the other semantic models, also has several built-in, high-level modeling primitives (e.g., properties, classes, and subclasses), but it adds other concepts (e.g., meta-classes treating operations and transactions as objects) that bear a striking similarity to object-oriented systems.

B.2 OODBS

The fact that the notion of objects in CAD, VLSI and computer graphics applications does not correspond closely to either tuples or relations forces designers to introduce another object, in addition to tuples and relations, into the relational database system. Fortunately, however, the object-oriented database systems proposed by many authors [Banc88, Frey88, Hard88, Horn87, Katz89, Kemp87a, Keta89, Khos88, Lécl88, Maie89] as a new concept for supporting engineering applications offer very promising prospects for modeling and handling a collection of CAD objects. The notion of “complex object”[Lori82] in System R[Astr76], or of “composite object”[Kim87b, Kim89b] in ORION[Kim88b] reflects the fact that objects in the engineering world are composed of parts that may among themselves involve a variety of other relationships. This provides foundation for treating a complex CAD object description as a whole database entity. The complex object can be thought of as a hierarchically organized collection of data. However, the restriction on complex objects is that any child tuple can not be
shared by more than one parent tuple. The composite object provided in ORION explicitly captures and enforces the IS-PART-OF integrity constraint between child and parent pairs of objects in a hierarchical collection of objects. The database scheme in ORION is based on class hierarchies. However, the CAD process changes dynamically from time to time. Thus the cost of scheme evolution in ORION is likely to be very high.

Object-oriented database systems can be categorized into two basic approaches, structural and behavioral [Kemp87a]. The central notion of the structural approach originated from the "complex object" proposed by Lone[Lori82] and the "molecule object" proposed by Batory[Bato84, Bato85] and many others[Adib88, Pare88, Ston84]. All these researchers have emphasized the fact that objects are composed of parts that may among themselves involve a variety of other relationships. Therefore, the database system has to provide facilities for mapping complex objects onto database structures and for retrieving these objects as unit entities.

The structural systems (e.g., SDM) raise the question of whether behavior should be embedded at the database level or in a layer outside the database. An question is whether by putting the behavior inside the database, we can achieve any improvement in performance. The structural systems were running their semantic layer in the workstation before the interpreters and storage managers of behavioral systems were running on multiple machines. Therefore, distribute applications were running on structural systems earlier than they were on behavioral systems. GemStone [Cope84] was one of the simplest object-oriented models. It is based on Smalltalk,
with very few extensions. Orion also has a Smalltalk-like model, but with many more extensions; e.g., it has a facility for complex objects that is built-in and that affects the semantics of fundamental notions such as creation and deletion. An important issue in the design of such a model is how much semantics is needed in the database interpreter, as well as how much should be built out of the interpreters primitives at the application level.

In contrast to the structural approach, the behavioral approach to databases has originated from the notion of abstract data types [Belk86, Gutt84, Inag84a, Inag84b, Inag84c, Inag84d, Jian89, Ston83, Ston86a] in programming languages. The behavioral approach is used here, by applying a set of procedures to identify or recognize CAD objects stored in the database, as the form in which they originally existed in CAD applications. That is to say, a database system should provide constructs to define manipulation of application objects in a manner that is familiar to designers. Therefore, the database system should allow users to convey meanings associated with application-specific operations such as boolean operations and geometrical operations in the CSG scheme[Requ80].

GemStone incorporates object identity and encapsulation via data abstraction that defines an external interface as a set of messages. It also supports inheritance. The features added by GemStone to Smalltalk to create a database are that it is disk base and provides concurrency control and recovery, constraints, secondary storage management, queries, index support, authorization, and a structured name space. GemStone incorporates a query language, but queries are formed over the instance variables of an object.
Although this approach avoids some of the problems of querying abstract types, it violates the principle of encapsulation. It should not be possible to write any external programs, including queries, that depend on the representation (i.e., instance variables) of an objects.

Orion supports automatic type extents. That is, for any type T the system maintains the set of all T's. Systems vary substantially in this regard. GemStone, for example, does not support automatic extents. Other languages, such as O_2 [Banc88], will support them as options.

Orion has concentrated on the area of schema evolution and has produced a set of rules that must be obeyed in order for a type hierarchy to remain consistent in the face of change.

IRIS [Fish87], an experimental system from Hewlett-Packard, is the most relational of all the approaches represented in this section. It was designed to support persistence in multiple programming languages, whereas O_2 was designed to support multiple database languages for writing methods.

IRIS uses a relational storage manager similar to the Relational Storage idsSystem (RSS) component of System R [Astr76]. It supports a separate query language, based on SQL, called OSQL. An issue here is whether queries should be captured by a separate language, as in IRIS, or whether they should be part of a single language, as in GemStone.

The IRIS model is based on objects, types and operations. Types are collections of objects. Other models treat types as pro-
tocols specification. This distinction leads to interesting differences in the view of inheritance in a system. Within the IRIS type hierarchy, it is possible to constrain a set of subtypes to be mutually exclusive. Such a constraint is not a typical programming-language feature for types (e.g., GemStone does not support it). It is a feature that is most similar to knowledge-representation languages.

IRIS allows several types of functions. A stored function is a table that explicitly records the mappings between the input objects and the output. A derived function is specified as a query whose result is a virtual table. The query is evaluated whenever the function is invoked. In this way, derived functions are similar to views. A foreign function is written in some programming language such as C or LISP. These last functions are outside the control of the system and can, therefore, have arbitrary side effects.

IRIS allows the type graph to change dynamically. Types can be added or deleted, and objects can change their type. This kind of capability is useful for modeling real-world situations, but it introduces safety problems with respect to type checking.

O₂ is the most formally described of all these systems. It supports identity, types with encapsulation, inheritance, and operation overloading with dynamic binding.

The authors defined O₂ by first describing a consistent set of objects (i.e., no dangling references and only one object with a given id). Next, they define a type structure and what it means for this to be consistent. Finally, they provide an interpretation that maps each type onto a subset of objects.
$O_2$ makes a distinction between values and objects. Objects have identity and can, therefore, be referenced from many contexts, whereas values do not. A value can be embedded in an object or in another value, and as such is guaranteed to have no outside reference.

Data objects are pairs of (id, value), and values are primitive, tuple, or set. Update is supported by changing the id-to-value binding. Tuples and sets can refer to other object.

In $O_2$, a type describes a minimal behavior for an object. An object that is an instance of type $T$ must support all the behavior associated with $T$, but may actually support more. For example, ("John", 20) is an instance of the type tuple (Name: String, age:int) and also an instance of the type (tuple(name: String)). Subtypes are defined explicitly. That is, the system does not infer the subtype relationship by looking at the signatures for types. The user must declare the fact which asserts that the subtype extension is a subset of the supertype extension.

Encore [Encore, Kogu90, Reis86, Skar87, Shaw89, Shaw90, Shaw90b, Zdon88] is another un-complete object-oriented database system of Brown University. Encore is targeted at large scale software engineering applications which are involved in data modeling. Encore provides an abstract type system for modeling objects in the object store. In Encore an abstract type is defined as set of property types to model the attributes of an object and a set of operation types to model the behavior of the object. The property type encapsulates the storage mechanism of the attribute value
and allows the add of additional properties to the property itself. Encore provides a powerful query model which may seem familiar to relational system users. All of the query primitives are methods of a set type. Specifically, it supports primitives for Select, Image, Project, OJoin (Object-oriented Join), and the usual set operations of Union, Intersection, Difference, as well as some operations specific to the OODB environment of Nest, UnNest, and Flatten [Shaw90].

B.3 Relational Extensions

The ER(Entity-Relational) model which unified features of the traditional models was proposed by Chen [Chen76], to facilitate the incorporation of semantic information. In 1979, an extension of Codd's relational model RM/T[Codd79] was proposed, attempting to capture more semantic meaning in the relational data model, introducing the important concepts of surrogate.

The reason for the creation of of OODBs is to provide database support for applications that are not well served by conventional record-oriented systems. OODBs provide for direct representation of complex data structures and for the creation of new database types by the database programmer. Extended relational systems and extensible database systems are two other approaches which seek to support non-traditional applications.

Extended relational systems support higher-level modeling features, such as are found in semantic data models, plus some capa-
bilities found in OODBs: adding new scalar types, set- or array-valued attributes to DML stored in the database.

The GEM data language [Zani83] can be viewed as providing syntactic extensions to the relational model. In fact, the initial implementation of GEM mapped GEM constructs into the Quel language. GEM took common relational programming idioms and gave them language support. GEM has a generalization hierarchy of relation schemas (called entities, presumable from similarity to entities in the ER model [Chen76]). This hierarchy supports the addition of attributes in subschemes, and of disjoint subschemes. GEM provides reference attributes, which serve much as object identifiers, except references from any one attribute are restricted to tuples from a single relation (and its subscheme relations). Thus, GEM could support a Road Segments relation with attributes START and END that reference tuples in an Intersection relation. A path notation is provided to traverse reference attributes. So, if \( r \) is a tuple variable ranging over RoadSegments, and POSITION is an attribute of Intersection, then \( r\.START.POSITION \) is a legal path description, referring to the position of the start intersection referenced by a RoadSegments tuple. Path notation is a shorthand form for a join operation, and probably in excess of 80 percent of the joins in relational queries can be handled by paths. GEM also provides for set-valued attributes and for set-comparison operators, which are useful for queries involving quantification.

The Postgres data model [Rowe87] also incorporates a hierarchy of relations, where subrelations can add attributes. However, the semantics of this hierarchy differ from those of the GEM hierarchy.
In GEM, an entity belonging to a relation is viewed as also being an element of all the relations above it in the hierarchy. In Postgres, the relations in the hierarchy are hold entities. The language for Postgres, Postquel, gives the choice of querying just a single relation, or a relation and all its subrelations.

Postgres allows users to add new base types to the system, by providing information on storage requirements, literal formats, comparison operators, and evaluation costs. Postgres also allows attributes that are arrays of base types. Perhaps the most powerful extension in Postgres is allowing a field of a tuple to contain a Postquel statement. A Postquel file can contain a different statement for every tuple in a relation, or the statement can be given along with the relation scheme, the tuples then providing parameters to that statement. Postquel fields can be used to represent attributes whose values are heterogeneous sets (sets composed of tuples from several relations) and also to support reference attributes similar to those in GEM. Postquel provides a path syntax similar to that of GEM. Postgres also incorporates database procedures, which are defined on tuples of a relation and can be inherited by tuples in subrelations, in much the way methods are inherited in OODBs.

**B.4 Extensible DBMSs**

The goal of extensible database systems is to provide a ready means for constructing database systems tuned or biased toward a particular domain of applications. Much of the work in this field
can be characterized as software engineering: trying to produce clean specifications of database components and their interfaces so that these components can be easily interchanged or modified. Many extensible systems, such as EXODUS, GENESIS, Starburst, PROBE, DMC (Data Model Compiler) etc. have been developed. We introduce some features of EXODUS and GENESIS in the following.

EXODUS provides a kernel of DBMS functions, a language (E) for writing database systems, and a variety of tools for implementing a full database system. EXODUS provides a storage manager, a library of access methods and operator algorithms, a query optimizer generator, and a tool for query-language construction. It has been used to implement a relational database system, as well as an object-oriented data model and language, call EXTRA and EXCESS, respectively.

GENESIS emphasizes a library of components that have standard interfaces and that can be layered on one another to produce a database system with little writing of new code. The components include a library of transformers, which provide mappings from abstract file and link structures to more concrete forms of these structures. These structures can in turn be transformed into even more concrete structures, and so on, eventually ending up at basic access methods. Other components include expanders, which express operations on abstract structures as one or more operations on their concrete implementations; the Grand Central component, which dispatches the correct expander for an operation, and a file system, Jupiter. GENESIS project posits a new role: the database
B.4. EXTENSIBLE DBMSS

implementor (DBI). The DBI lies somewhere between a database administrator and the programmers of the database system.
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


tems, 2nd Int’l. Workshop on Object-Oriented Database Sys-
tems, LNCS#334, Springer-Verlag, Bad Munster am Stein-

[Horn87] Mark F. Hornick and Stanley B. Zdonik: A Shared, Seg-
mented Memory System for an Object-Oriented Database. 

[Hevn79] Hevner. A. R & Yao. S. B.: Query Processing in Dis-
tributed Database System, IEEE Trans. Softw. Eng., SE-

[Hugh91] J. G. Hughes (Univ. of Ulster): Object Oriented 

[Inag84a] Y. Inagaki, and T. Sakabe, Many-Sorted Algebra and 
Equational Logic (1), Journal of Information Processing So-

[Inag84b] Y. Inagaki, and T. Sakabe, Many-Sorted Algebra and 
Equational Logic (2), Journal of Information Processing So-

[Inag84c] Y. Inagaki, and T. Sakabe, Many-Sorted Algebra and 
Equational Logic (3), Journal of Information Processing So-

[Inag84d] Y. Inagaki, and T. Sakabe: Many-Sorted Algebra and 
Equational Logic (4), Journal of Information Processing So-
REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


Acknowledgements

I am deeply indebted to Professor Y. Fujiwara, my supervisor, for his invaluable guidance, help, and moral support over the past five years. Very special thanks go to my academic adviser, Associate Professor N. Ohbo, who has directed my research since I joined the project, and who has had a great influence on what I have accomplished in these years and what I know today. His thoroughness and sincerity in doing this will provide a model for me in the years to come. I owe special thanks to Associate Professor H. Kitagawa and Associate Professor K. Yamaguchi, for their constant encouragement, numerous suggestions and many inspiring discussions, especially their helpful comments at the completion of this paper. A special mention must be made of Associate Professor Jeferry X. Yu, who has always been there when I needed his help, over the past five years. Without the able guidances of these men, this thesis would simply not have been completed. I am indebted to the referees Professor Y. Fujiwara, Professor I. Suzuki, Associate Professor N. Ohbo, and Associate Professor Y. Kiyoki for many helpful comments on the paper.

I would like to thank all the members of the Database Laboratory to which I was attached, especially my co-workers Mr. Xiaodong Zhang, Mr. Hong Shen, Mr. Haitao Ma, and the other members in this project. I benefited particularly from lengthy discussions with – and suggestions made by – the members in the project. It is a pleasure to acknowledge the helpful suggestions of Mr. Ronald Craig in proof-reading the paper.