Associative Access
in Persistent Object Stores

A thesis presented in partial fulfilment
of the requirements for the degree of

Master of Information Sciences
in Information Systems

at Massey University
Palmerston North, New Zealand

Weena Nusdin
2004

ABSTRACT

The overall aim of the thesis is to study associative access in a Persistent Object Store (POS) providing necessary object storage and retrieval capabilities to an Object Oriented Database System (OODBS) (Delis, Kanitkar & Kollios, 1998 cited in Kirchberg & Tretiakov, 2002).

Associative access in an OODBS often includes navigational access to referenced or referencing objects of the object being accessed (Kim, Kim, & Dale, 1989). The thesis reviews several existing approaches proposed to support associative and navigational access in an OODBS. It was found that the existing approaches proposed for associative access could not perform well when queries involve multiple paths or inheritance hierarchies.

The thesis studies how associative access can be supported in a POS regardless of paths or inheritance hierarchies involved with a query. The thesis proposes extensions to a model of a POS such that approaches that are proposed for navigational access can be used to support associative access in the extended POS. The extensions include (1) approaches to cluster storage objects in a POS on their storage classes or values of attributes, and (2) approaches to distinguish references between storage objects in a POS based on criteria such as reference types – inheritance and association, storage classes of referenced storage objects or referencing storage objects, and reference names.

The thesis implements Matrix-Index Coding (MIC) approach with the extended POS by several coding techniques. The implementation demonstrates that (1) a model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS and (2) the MIC implemented with the extended POS can support a query that requires associative access in an OODBS and involves multiple paths or inheritance hierarchies. The implementation also provides proof of the concepts suggested by Kirchberg & Tretiakov (2002) that (1) the MIC can be made independent from a coding technique, and (2) data compression techniques should be considered as appropriate alternatives to implement the MIC because they could reduce the storage size required.

ACKNOWLEDGEMENTS

I would like to thank Alexei Tretiakov, my supervisor, for his patience, guidance and suggestions during this thesis. I am also thankful to Markus Kirchberg for his guidance and suggestions through the early stages of work. Thanks to Klaus-Dieter Schewe for his kindness in allowing me to attend his paper, 157.794 Object Oriented Databases. Thanks to Roland Kaschek for his comments during the thesis.

Thanks to staff of International Student Support at Massey University, Palmerston North for their assistance. Special thanks go to Susan Flynn for her assistance and the arrangement of financial support during my thesis.

While I was undertaking the thesis, a number of friends I met them in New Zealand have given me lots of helps and encouragement, which has helped me pass through difficult times. Special thanks go to the following friends.

- Marie Hau for being my good, helpful friend and flatmate,
- Yuen Xie for our discussions about data access approaches used in relational database systems.
- Jayson Speer for his suggestions and encouragement during the beginning of learning a C++ programming language,
- Lin Shi for his suggestions during the thesis,
- Rebecca Freeman for her assistance in proof-reading my draft thesis,
- Nanthaporn Chitchai for always coming to visit and listen to my troubles,
- Weerawate Utto for helping me out of depressing times and especially for smiling tomatoes from his experiment,
- Angkana Noisuwan, Duljira Sukboonyasatit and Chanapha Sawatdeenaruenat for the laughs, encouragement and assistance they have given,

- Somsaowanuch Chamusri and Kuephan Klankaradi for their kindness for taking turns to accompany me and giving me a ride from my office to my flat during the night for the last three months of my thesis, and
- Thiengtham's family, Kanittha Watanakeeree, Wanwadee Wongmongkol, Piyarat Piyaket and Duangrat Thongphak for good times, assistance and encouragement.

Thanks to my friends from Kasetsart University in Thailand for their encouragement and understanding. Special thanks go to Nuanjan Suntornkiti, without whom this year would have been much harder and longer.

I also would like to thank my colleagues, especially Thra Boondechanun, at the Bureau of Flight Safety Standards, Department of Civil Aviation of Thailand for their invaluable assistance in taking care of my jobs and responsibilities during my study leave. Special thanks go to Jutharat Nakhsewi for the kindness, patience, and wonderful support he has always given me throughout my study.

I wish to express my deep gratitude to my parents for their endless patience and love, and wonderful support they have given me. Without them, I would never have completed this work.

Finally, I would like to express my sincere thanks to the New Zealand Government for their financial support during my stay in New Zealand, and to the Thai Government for granting my study leave.

Weena Nusdin
February 2004

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Database Systems

For many years, database systems that are supported by computer technologies have played important roles in businesses (Elmasri & Navathe, 2000; Garcia-Molina, Ullman, & Widom, 2002; Ramakrishnan & Gehrke, 2003). They became essential tools for storing and managing data of many kinds of organisations, for example universities, hospitals, airlines, etc.

A database system can be thought of as a mediator between users who want to use data and physical storage devices that store it (Abiteboul, Hull, & Vianu, 1995). A database system is commonly defined by a database and its management system (Abiteboul et al., 1995; Elmasri & Navathe, 2000; Garcia-Molina et al., 2002; Ramakrishnan & Gehrke, 2003). A database is a collection of data such as facts, activities, events, etc. stored in storage devices and a Database Management System (DBMS) is software that supports the management of data in a database by providing the means for users to maintain and utilise it.

In the early stages, database systems were developed based on file systems and tailored applications (Abiteboul et al., 1995; Elmasri & Navathe, 2000; Ramakrishnan & Gehrke, 2003). Subsequently, principles that insulate users from the way that data is structured and actually stored in storage devices were developed. In the late 1960s, the three-level architecture, which separates functions of a database system into physical, logical and external levels, was introduced as presented in Figure 1.1.

A physical level specifies data storage details, for example techniques that are used to organise data in storage devices and approaches that are used to speed up data access (Abiteboul et al., 1995; Elmasri & Navathe, 2000; Ramakrishnan & Gehrke, 2003).

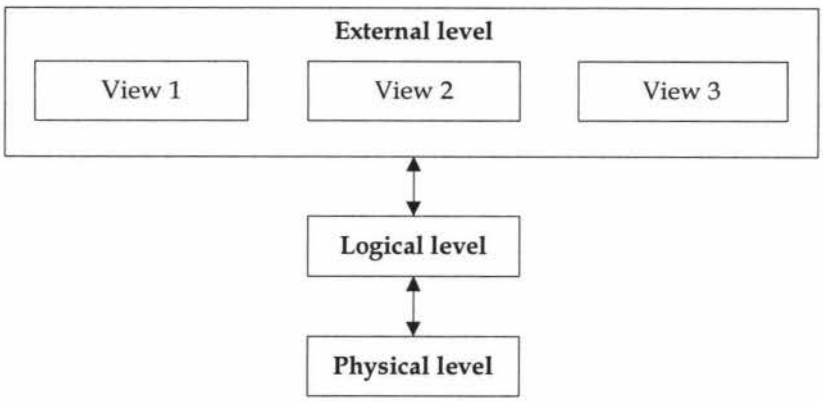


Figure 1.1 Three-level architecture of database systems
(based on Abiteboul et al., 1995, p.5)

A logical level describes data in terms of a data model (Abiteboul et al., 1995; Elmasri & Navathe, 2000; Ramakrishnan & Gehrke, 2003). Ramakrishnan & Gehrke (2003, p.10) define that, "a data model is a collection of high-level data description constructs that hide many low-level storage details". Abiteboul et al. (1995, p.28) describes that, "a database model provides the means for specifying particular data structures, for constraining the data sets associated with these structures, and for manipulating the data". A data model typically includes a Data Definition Language (DDL) that enables users to specify structure and constraints of data in a database, a Data Manipulation Language (DML) that enables users to manipulate data in a database, and a Data Query Language (DQL) that enables users to pose questions about data in a database. Often, a DQL is referred to as part of a DML (Abiteboul et al., 1995; Ramakrishnan & Gehrke, 2003; Silberschatz, Korth, & Sudarshan, 1997).

The separation of the logical level from the physical level is called physical data independence (Abiteboul et al., 1995; Elmasri & Navathe, 2000; Ramakrishnan & Gehrke, 2003). It insulates users from the way that data is stored in physical storage devices, which reduces the necessity of altering existing application programs according to the changes of data storage in a physical level.

A view, in an external level, can be thought of as a subset of a logical concept, which is tailored for the needs of specific users (Abiteboul et al., 1995; Elmasri & Navathe, 2000; Ramakrishnan & Gehrke, 2003). A view conceals unrelated information about data structure and constraint, and restructures data that is retained. The separation of the external level from the logical level, which is called logical data independence, allows several different views in an external level. Abiteboul et al. (1995) give examples of views that they can be as simple as an Automotive Teller Machine (ATM), or as complicated as a Computer-Aided Design (CAD) system.

1.2 The Functionality of a DBMS

A DBMS provides a broad range of functions to manage data in a database (Abiteboul et al., 1995; Elmasri & Navathe, 2000; Garcia-Molina et al., 2002; Ramakrishnan & Gehrke, 2003; Zezulan & Rabitti, 1993). The followings are fundamental functions of a DBMS.

- **Secondary storage management:** As mentioned, a DBMS is software designed to provide the means for users to maintain and utilise data in a database. Data in a database should be persistent, i.e. data should outlive database applications such that it may be reused later. Typically, the size of data stored in a database is too large to fit in the main memory. Hence, non-primary storage devices such as magnetic disks are used to store data instead of main memory because of their cheap cost, large capacity, and non-volatility. However, magnetic disks do not allow a Central Processing Unit (CPU) to directly manipulate data stored in them like main memory does. Moreover, magnetic disks provide slower data access speed compared to main memory. While data access in main memory is in a speed range between 10^{-8} - 10^{-7} seconds, data access in magnetic disks is at speed of 10^{-3} seconds. Thus, one necessary function of a DBMS is the management of data stored in magnetic disks such that the processes of fetching disk pages[1] where requested data resides in magnetic disks into main memory for manipulation, and the processes of writing the (modified) disk pages back to magnetic disks are carried

[1] A disk page is a unit transferred between main memory and magnetic disks.

out with efficiency. The secondary storage management is relevant to several issues such as the following.

- Data storage techniques:
  - Clustering techniques that cluster data that are always simultaneously accessed to stay as close as possible in a magnetic disk to reduce the number of I/O operations[2] and time needed when fetching disk pages that store requested data into main memory.
  - Partitioning techniques that partition large data into several units stored in a magnetic disk for the ease of transferring data between main memory and magnetic disks.
- Data access approaches: Typically, a DBMS employs auxiliary data structures called indices to speed up the search for requested data. These indices can be compared with indices of books, which provide ways to find topics rapidly (Folk, Zoellick, & Riccardi, 1998). An efficient data access approach reduces the number of I/O operations needed and time spent during the search.
- Buffer replacement policies: As mentioned, the capacity of main memory is usually too small to store all data in a database. Hence, a DBMS needs to bring disk pages where requested data resides into main memory for manipulation. Buffer replacement policies assist a DBMS to decide which disk page to be brought back from main memory to a magnetic disk in order to acquire space in main memory for the new disk page. An efficient buffer replacement policy reduces the number of I/O operations required during transferring disk pages between main memory and magnetic disks.

- **Concurrent access and Crash recovery:** In general, several users simultaneously share data in a database. Hence, a DBMS should be able to control concurrent access such that no access conflict occurs. In addition, a DBMS should be able to protect data from damages caused by a system crash that may occur. In relation to the control of concurrent access and crash recovery, the term transaction involves. Ramakrishnan & Gehrke (2003, p.521) define that, "a transaction is an execution of

---

[2] An I/O (Input/Output) operation is also referred to as a read or write operation (Ramakrishnan & Gehrke, 2003). The term 'input' refers to input from a magnetic disk to main memory. The term 'output' refers to an output from main memory to a magnetic disk.

a user program, seen by the DBMS as a series of read and write operations". A transaction has four significant properties as follows.

- A transaction must be atomic, i.e. either all operations are carried out or none are.
- A transaction must preserve the consistency of data in a database, i.e. a database must be left in a consistent state after executing a transaction.
- Interleaved transactions, which are transactions that require read or write operations on the same data, may occur because users concurrently share data in a database. Therefore, a transaction must be isolated and protected from the effects of other transactions.
- A transaction must be durable, i.e. after a transaction has been committed, its effects must persist although the system failure occurs before they are written in magnetic disks.

- **Data integrity and security:** A DBMS should be able to guarantee the consistency of data in a database to ensure data integrity in a database. For instance, a DBMS should be able to check, before new data is inserted into a database, whether the data is in a correct format and whether it breaks any semantic rules of a database. Furthermore, a DBMS should provide data security such as preventing data from being retrieved or updated by unauthorised users.
- **Distribution:** Nowadays, in many applications, data resides in different locations. A DBMS should also be able to provide users with transparent access to data stored in different locations.
- **Query optimisation:** Another important function of a DBMS is to translate user requests in external or logical levels into executable transactions. Such requests include queries that are questions that users ask about data in a database. Optimisation techniques should be provided in order to reduce the cost of executing these transactions.

1.3 An Object Oriented Database System (OODBS)

An OODBS is developed based on an Object Oriented Data Model (OODM) that provides various aspects that meet the needs of modelling complex data (Atkinson et al.,

1989; Elmasri & Navathe, 2000; Kim, 1990a; Maier, 1986; Schewe & Thalheim, 1993; Zand, Collins, & Caviness, 1995).

Because of a powerful OODM, it was claimed that an OODBS could overcome shortcomings of a RDBS that has problems in supporting complex data manipulated by certain types of applications such as Computer-Aided Design (CAD) systems, Computer-Aided Software Engineering (CASE), and multimedia applications (Atkinson et al., 1989; Leavitt, 2000; Schewe & Thalheim, 1993). It was also predicted that an OODBS could replace a RDBS that was broadly used in database markets. However, RDBSs are still the most widely used database systems. It is stated by Leavitt (2000) that in 2000 the sales of RDBSs were 50 times larger than the sales of OODBSs.

One important reason that makes OODBSs not successful in database markets, although an OODM is more capable compared with a Relational Database Model (RDM), is possibly that there is still no concrete agreement on a formal concept of an OODM (Atkinson et al., 1989; Schewe & Thalheim, 1993). In addition, several criticisms and concerns in an OO paradigm still remain open, which suggests the need for further investigation and research (Schewe & Thalheim, 1993). Moreover, the powerful concept of an OODM results in complexities in the implementation of an OODBS (Kuckelberg, 1998; Zezulan & Rabitti, 1993).

Although there is still no agreement on the formal concept of an OODM, it is broadly accepted that an OODM is based on the following fundamental concept (Bertino, 1994; Elmasri & Navathe, 2000; Kim, 1990a, 1990b). Any real world entity is represented by an object. An object has a unique Object Identifier (OID) that encodes its identity. An object has methods that represent its behaviours. Several type constructors, such as a set, a list, a union of type constructors, etc., can be used to construct object values. Objects reference each other via their OIDs. Objects with similar structures and behaviours are categorised into the same class. A class (called a subclass) may inherit structures and behaviours from other classes (called superclasses), and it may specify its owns local structure and behaviour. This means that an object may belong to many classes, i.e. an object that belongs to a subclass must belong to a superclass.

1.4 Providing Data Access in an OODBS

The area of providing access to stored data is one of important research topics of database systems (Abiteboul et al., 1995). It relates to several functions of a DBMS such as query languages (DQLs) and secondary storage management.

Abiteboul et al. (1995, p.736) define that a DQL is, "a notation for expressing queries, coupled with a mechanism for assigning meaning to the expression". In other words, a DQL enables users to pose questions about data in a database (Abiteboul et al., 1995; Elmasri & Navathe, 2000; Ramakrishnan & Gehrke, 2003; Silberschatz et al., 1997). For an OODM, there have been a number of various query languages proposed, for example algebraic, calculus, logic-programming oriented and SQL-extensions (Abiteboul & Catriel, 1995).

In relation to the secondary storage management of an OODBS, which affects the efficiency of retrieval of requested data from a database, there are several issues[3] concerned (Ramakrishnan & Gehrke, 2003; Zezulan & Rabitti, 1993). Such issues include data access approaches that are the focus of the thesis.

As mentioned, data access approaches are ideas to build auxiliary data structures called indices that speed up the search for requested data (Ramakrishnan & Gehrke, 2003). The efficient data access approach reduces the number of I/O operations needed and the time spent during the search for requested data in magnetic disks. The criteria used to evaluate the performance of indices are for example the cost to search for requested data with the indices and the storage size required for the indices (Bertino & Kim, 1989; Kim, Kim et al., 1989; Kuckelberg, 1998; Zezulan & Rabitti, 1993). The cost performance of the indices can be estimated in terms of the number of I/O operations required or the time spent during the lookup with the indices. However, if the size of indices can fit into the main memory, there is no I/O operation and time required in order to bring the indices from magnetic disks into main memory. Only time to execute the search with the indices in the main memory is counted.

---

[3] See section 1.2

As many researchers agreed, queries in an OODBS are supported by three types of data access including direct, navigational and associative access (Kirchberg & Tretiakov, 2002; Kuckelberg, 1998; Zezulan & Rabitti, 1993).

- Direct access is retrieval of a set of objects via their own OIDs.
- Navigational access is retrieval of a set of OIDs based on navigation along references between objects beginning from any given object. Navigation from an object to its referenced objects is called forward navigation. In contrast, navigation from an object to its referencing objects is called backward navigation.
- Associative access is retrieval of a set of OIDs that meet conditions specified on objects. Often, associative access in an OODBS includes navigational access to referenced or referencing objects of the objects being accessed (Kim, Kim et al., 1989).

There have been a number of data access approaches proposed for an OODBS. For direct access that is the simplest type of data access, Zezulan & Rabitti (1993) suggest that it may be basically implemented by associative access approaches used in a RDBS such as B-tree or hashing indices. This is because the direct access can be compared with associative access to data records in a RDBS via search key values.

Examples of the approaches that have been proposed to support navigational access in an OODBS are navigation index (Zezulan & Rabitti, 1993), ring and spider data structures (Subieta, 1994a, 1994b), object skeletons (Hua & Tripathy, 1994), join index hierarchies (Han, Xie, & Fu, 1999; Xie & Han, 1994), triple-node hierarchies (Luk & Fu, 1998), and Matrix-Index Coding (MIC) (Kuckelberg, 1998). The join index hierarchies and triple-node hierarchies are modifications of join indices used in a RDBS. The object skeletons maintain networks of OIDs as structures to support navigation among objects. In the ring and spider data structures, an object stores reference pointers that indicate referenced or referencing objects of the object. The navigation index and MIC employ in-memory calculation techniques by appending codes with OIDs of objects. The code of an object can be expanded to a set of OIDs of referenced or referencing objects of the object.

Among the approaches proposed to support navigational access, MIC employs relatively simple and efficient concepts compared with others. The MIC works like join indices but it eliminates the needs to simultaneously access to obtain OIDs of referenced or referencing objects of the object being accessed (Kuckelberg, 1998). For each object, the MIC transforms sequences of OIDs of objects, which are referenced by or reference the object, to two codes by a coding technique called Simple Continued Fractions (SICF). The two SICF codes are appended with an OID of the object. One SICF code is for forward navigation and another one is for backward navigation. The SICF code of the object can be expanded back to sequences of OIDs of referenced or referencing objects.

Kirchberg & Tretiakov (2002) suggest that the original concept of the navigation index and MIC can be generalised by making the navigation index and MIC independent from a coding technique used, i.e. any appropriate coding technique can be selected to guarantee optimal performance in different circumstances. Especially, data compression techniques should be considered as alternative coding techniques because they can reduce the storage size of the navigation index and MIC.

The approaches that have been proposed to support associative access in an OODBS are for example Single-Class (SC) and Class-Hierarchy (CH) indices (Kim, Kim et al., 1989), H-tree (Low, Ooi, & Lu, 1992), hierarchy class Chain (hcC) tree (Sreenath & Seshadri, 1994), Class-Division (CD) index (Ramaswamy & Kanellakis, 1995), nested, path and multi indices (Bertino & Kim, 1989), access support relations (Kemper & Moerkotte, 1990), and Nested-Inherited index (NIX) (Bertino, 1991). These approaches are modifications of approaches used in a RDBS such as B+ tree and join indices. In these approaches, a class is regarded as a relation and a logical object, which is an object defined in a data model, is regarded as a tuple over a relation. A UID, which is a system-defined unique identifier for a tuple, is regarded as an OID of a logical object. Unfortunately, these approaches cannot perform well when associative access includes navigational access involved with multiple paths or inheritance hierarchies (Hua & Tripathy, 1994). A path is a subset of a class-attribute hierarchy, which is a directed graph presenting how attributes of classes references other classes (Bertino & Kim, 1989). An inheritance hierarchy is a directed graph, which presents a class and its

subclasses (Kim, Kim et al., 1989). The greater the number of paths or inheritance hierarchies involved, the greater the number of indices required. This means that storage redundancy may occur. Moreover, the number of indices involved may lead to an increasing number of I/O operations needed and time spent during lookups for requested objects by the indices.

1.5 A Persistent Object Store (POS)

A POS is a component of an OODBS that provides necessary object storage and retrieval capabilities (Delis, Kanitkar & Kollios, 1998 cited in Kirchberg & Tretiakov, 2002). A POS is also referred to as a storage level. A POS handles storage objects that are objects mapped from logical objects for the purpose of efficient data retrieval (Kirchberg, Schewe, & Tretiakov, 2003; Zezulan & Rabitti, 1993). The mapping of objects between levels can be done by clustering or partitioning techniques (Zezulan & Rabitti, 1993).

Zezulan & Rabitti (1993) state that an efficient POS should be able to support all three types of data access – direct, navigational and associative, in an OODBS. However, to the best of our knowledge, so far only navigational access in a POS has been studied such as ring and spider data structures (Subieta, 1994a, 1994b), navigation index (Zezulan & Rabitti, 1993) and MIC (Kuckelberg, 1998). The existing data access approaches proposed for associative access in an OODBS seem to be constructed based on the organisation of physical objects, i.e. data records that are stored in magnetic disks, because they are modifications of approaches used in a RDBS.

In Subieta (1994a; 1994b), ring and spider data structures are proposed to support navigational access and to be implemented in a POS of LOQIS. A model of a POS of LOQIS consists of atoms (storage objects), where each atom is regarded as a physical independent unit. Each atom has an OID. A model of a POS of LOQIS is informative, i.e. details of the database schema defined in a data model such as inheritances between classes and membership of classes are known. Moreover, rules that are used to map logical objects to atoms in a POS are rigid.

In Kuckelberg (1998) and Zezulan & Rabitti (1993), navigation index and MIC are proposed to support navigational access based on the structure of storage objects in a POS. A model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) are similar. It consists of a set of storage objects and a set of references between storage objects (Zezulan & Rabitti, 1993). A POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) is less informative compared with a POS employed by Subieta (1994a; 1994b). References are not distinguished by any criteria. Storage objects are not clustered by any criteria, i.e. classes, values of attributes, etc. Furthermore, there is no restriction on how logical objects are mapped to storage objects. Hence, in our opinion, a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) is considered more simple and flexible compared with a POS employed by Subieta (1994a; 1994b).

1.6 Thesis Motivations

Within the area of providing access to stored data in database systems, query languages (DQLs), data storage in magnetic disks, and buffer replacement policies are interesting research topics in their own rights, but beyond the study of this thesis. The focus of this thesis is to study data access approaches for associative access in a POS of an OODBS.

In terms of the motivation,

- The thesis aims to study associative access in an OODBS, especially how associative access can be supported in a POS.
- As magnetic disks are in the order of 10^5 times slower than main memory, the thesis anticipates an in-memory data structure (index) that allows executing associative in-memory without accessing magnetic disks and supports associative access in a POS regardless of paths or inheritance hierarchies involved. In that connection, we will make an effort to estimate the memory requirements of the data structure. On the other hand, for as long as the execution can be performed in-memory, we do not focus on estimating the execution time for the data structure, and leave it as a topic for further research.

1.7 Thesis Objectives

The thesis aims to accomplish the four following objectives:

1. The first objective is to review the existing data access approaches that support associative access in an OODBS. In addition, the thesis aims to review data access approaches that support navigational access in an OODBS because associative access in an OODBS often includes navigational access to referenced or referencing objects of the object being accessed (Kim, Kim et al., 1989).

2. The second objective is to study how associative access can be supported in a POS of an OODBS, i.e. to extend a model of a POS such that data access approaches that support navigational access can support associative access in the extended POS regardless of paths or inheritance hierarchies involved. To accomplish this objective,
   - The thesis selects to extend a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) because it is more simple and flexible compared with a model of a POS employed by Subieta (1994a; 1994b).
   - The thesis takes into account conditions which can be specified in queries that require associative access while considering extending a model of a POS. For example, structures of objects or values of attributes would be typical conditions. In addition, criteria that can be used to distinguish references, i.e. types of references, structures of referenced or referencing objects, and reference names, are also considered.

3. The third objective is to provide proof of the concepts that
   - A model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS, and
   - A data access approach, which is originally proposed to support navigational access, implemented with the extended POS can support a query that requires associative access in an OODBS and involves paths or inheritance hierarchies.

To accomplish this objective, the thesis implements MIC, which is originally proposed to support navigational access in an OODBS, with the extended POS to support a query requiring associative access in an OODBS and involving paths and inheritance hierarchies. This is because the MIC employs an in-memory calculation technique that implies the fast speed to support the access. In addition, the MIC uses relatively simple and efficient concepts compared with other approaches such as a navigation index, and ring and spider structures.

4. As mentioned, Kirchberg & Tretiakov (2002) suggest that the original concepts of MIC can be generalised by making the MIC independent from a coding technique used, i.e. any appropriate coding technique can be selected to guarantee optimal performance in different circumstances. Especially, data compression techniques should be considered as alternative coding techniques because they can reduce the storage size of the MIC.

   Therefore, the fourth objective is to provide proof of the concepts that (1) the MIC can be made independent from a coding technique and (2) data compression techniques should be considered as appropriate alternatives to implement the MIC because they can reduce the storage size of the MIC.

   To accomplish this objective, the MIC is implemented with the extended POS by the following coding techniques.

   - The first technique is to represent sequences of OIDs as they are,
   - The second technique is to transform sequences of OIDs to codes by using a SICF coding technique which is originally used, and
   - The third technique is to individually transform each OID in sequences of OIDs to a code by a Start/Stop data compression technique (Pigeon, 2001).

   The storage sizes required of the MIC implemented with these coding techniques are compared.

1.8 Structure of the Thesis

The thesis comprises nine chapters. This introductory chapter describes in outline the basic and important issues regarding database systems, especially an OODBS. Then the chapter addresses the thesis motivations and objectives in which associative access in a POS of an OODBS is the focus of the thesis.

In the thesis, a literature review is separated into four chapters beginning from chapters two to five.

Chapter two describes concepts of RDM and OODM in order to develop understanding on how data in RDBS and OODBS is structured in a logical level. At the end of chapter, a brief conclusion of the differences between RDM and OODM concepts is established, which shows advantages of an OODM over a RDM in modelling complex data.

Even though the powerful concept of an OODM results in difficulties in the implementation of an OODBS and there is still no agreement on the formal concept of an OODM, there have been a number of OODBSs developed or researched for both commercial and academic purposes because an OODM has shown its advantages in modelling complex data (Atkinson et al., 1989; Elmasri & Navathe, 2000; Kim, 1990a; Maier, 1986; Schewe & Thalheim, 1993; Zand et al., 1995). These OODBSs are implemented in different ways, i.e. they have different OODMs and components, and employ different approaches to support data access in storage devices. Thus, chapter three briefly describes components and their functions of several proposed OODBSs including ORION (Kim, Ballou, Chou, Garza, & Woelk, 1989), O_2 (Banchilhon, Delobel, & Kanellakis, 1992), LOQIS (Subieta, 1994a, 1994b), and multi-level architecture for distributed object bases (Kirchberg et al., 2003).

Chapter four reviews the existing data access approaches proposed for associative access in an OODBS. In addition, the chapter reviews data access approaches that support navigational access in an OODBS because associative access in an OODBS

often includes navigational access (Kim, Kim et al., 1989). Advantages and disadvantages of the reviewed approaches are also discussed at the end of the chapter.

The thesis objectives include the implementation of MIC with the extended POS by several coding techniques in order to provide proof of the concepts that data compression techniques should be considered as appropriate alternatives to implement the MIC because they can reduce the storage size. Hence, chapter five reviews data compression techniques. The chapter begins with the introduction of concepts and types of data compression techniques. Examples of data compression techniques of each type are also given. Finally, the chapter briefly describes a Start/Stop data compression technique (Pigeon, 2001) which is one of coding techniques used to implement the MIC in the thesis.

Chapter six introduces the extensions proposed to a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) such that data access approaches that support navigational access can support associative access in the extended POS. The extensions include (1) approaches to cluster storage objects in a POS on their structures or values of attributes, and (2) approaches to distinguish references between storage objects in a POS based on some criteria such as reference types, structures of referenced storage objects or referencing storage objects, and reference names.

Chapter seven discusses the performance of MIC implemented with the extended POS in terms of the cost performance and the storage size required.

Chapter eight describes the implementation of MIC with the extended POS by several coding techniques to support a query requiring associative access in an OODBS and involving paths and inheritance hierarchies. The chapter demonstrates that:

- A model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS,
- MIC that is implemented with the extended POS can support a query that requires associative access and involves paths and inheritance hierarchies, and
- MIC can be made independent from a data coding technique used.

In addition, the results of the implementation show that MIC implemented with a Start/Stop data compression technique (Pigeon, 2001) requires the smallest storage size compared with MIC implemented with no coding technique and SICF coding technique.

Chapter nine describes in outline the conclusion of the thesis. Suggestions for future research are addressed.

CHAPTER 2

RELATIONAL AND OBJECT ORIENTED DATA MODELS

A RDM is accepted as a traditional data model that succeeds in modelling data for business applications that manipulate simple data such as numbers, texts, dates, etc. (Maier, 1986; *Third-Generation Database System Manifesto*, 1990). However, a RDM has run into difficulties with modelling complex data such as images, voices, icons and animations, that are manipulated by certain types of applications, for instance Computer-Aided Design (CAD) systems, Computer-Aided Software Engineering (CASE), and multimedia applications (Kim, 1990b; Maier, 1986; *Third-Generation Database System Manifesto*, 1990). There have been attempts made to overcome the limitations of a RDM. These include an extension of a RDM and an introduction of an OODM that has powerful concept to support modelling complex data (Kim, 1990b).

This chapter describes and compares concepts of RDM and OODM in order to develop understanding on how they structure data in a database. The chapter is separated into three sections. Concepts of RDM and OODM are described in the first and second sections, respectively. The differences between RDM and OODM concepts are compared and a conclusion is established in the last section.

2.1 The Concept of a RDM

A RDM was first introduced by Codd in 1970, and immediately drew attention because of its simplicity and mathematics foundations (Abiteboul et al., 1995; Elmasri &

Navathe, 2000; Garcia-Molina et al., 2002; Ramakrishnan & Gehrke, 2003). A RDM provides significant improvements on earlier data models by offering three influential features (Codd, 1970, 1982). Firstly, a RDM allows a high degree of physical data independency that is, a RDM provides the means of describing the structure of data without superimposing any additional structure for representation of data in a physical level. Secondly, the structure of data defined in a RDM is simple therefore it is easy for all types of users and programmers to understand. Thirdly, a RDM introduces high-level language concepts that allow users to express operations on large sets of data at a time. These three features simplify the development of applications and the expressions of data queries and manipulation.

For simplicity of understanding, this section describes the concept of a RDM along with parts of a sample RDM of a database system of an air operator company, which is fully presented in Appendix A.

A RDM structures data in a database as a collection of relations (Codd, 1970; Elmasri & Navathe, 2000). Hence, the database schema that describes structure of data in a RDM is defined as a set of relations. A relation is defined as a subset of the Cartesian product $S_1 \times S_2 \times \ldots \times S_n$, where S is a domain that is a set of constants such as a set of Strings, a set of Integers, a set of Booleans, etc. As defined, a relation is said to have degree n. A relation can also be defined as a set of tuples. Each tuple is an ordered list of n values, where the first value is an element of S_1, the second value is an element of S_2, and so on. A relation schema describes a relation. It is made up of a relation name, and a list of attributes and their domains. An attribute is a name whose values are elements of its domain specified in the relation schema.

Consider the example of the RDM presented in Appendix A. It consists of ten relations including Person, Airplane, Engine, Part, MarriedPerson, Pilot, Mechanic, InstalledEngine, InstalledPart and JourneyLog. A relation called Person is defined as,

Person = {person ID: Integer, name: String, date of birth: Date, address: String}
Primary key: {person ID}

A Person is a relation name. Person ID, name, date of birth and address are attributes of the relation Person. A set of Integers is a domain of an attribute person ID; a set of Strings is a domain of an attribute name; a set of Dates is a domain of an attribute date of birth; a set of Strings is a domain of an attribute address.

A minimum subset of attributes whose values uniquely identify tuples over a relation is called a primary key of that relation (Codd, 1970). For example, in an example RDM presented in Appendix A, an attribute person ID is defined as a primary key of the relation Person.

In a RDM, tuples reference each other by values of a foreign key that is a subset of attributes of a relation (Codd, 1970). The value of an element of a foreign key of a relation must also exist as a value of a primary key of some relations in a database. Consider the relation Pilot of an example RDM presented in Appendix A, which is defined as follows.

Pilot = {pilot person ID: Integer, pilot license No: String}
Primary key: {pilot person ID, pilot license No}
Foreign key: [pilot person ID] $\subseteq$ Person [person ID]

Tuples over the relation Pilot reference tuples over the relation Person by an attribute pilot person ID that is defined as a foreign key. A set of values of an attribute pilot person ID of the relation Pilot must be a subset of a set of values of an attribute person ID of the relation Person.

Constraints are restrictions that can be specified in a relation schema (Elmasri & Navathe, 2000; Ramakrishnan & Gehrke, 2003). They can be thought of as conditions that need to be met by all elements of a relation. Constraints help by preventing the inconsistencies of data in a database such as a wrong update inquiry. In addition, constraints add semantics to a RDM. In a RDM, there are various types of constraints for instance domain, key, foreign key and general.

- A domain that specifies values of an attribute is also considered as one type of constraints. A domain constraint is a condition that attribute values must satisfy.

Consider an example RDM presented in Appendix A. A domain of an attribute person ID of the relation Person is a set of Integers therefore values of an attribute person ID must be elements of a set of Integers only.

- A key constraint does not allow two tuples over a relation to have the same primary key value because values of a primary key must uniquely identify tuples over a relation. In addition, a key constraint does not allow a primary key to be null values, i.e. unknown or non-existent. In an example RDM presented in Appendix A, values of an attribute person ID, which is defined as a primary key of the relation Person, cannot be duplicated between any two tuples over the relation Person. In addition, they cannot be null values.
- A foreign key constraint allows a tuple to reference only existing tuples in a database. In an example RDM presented in Appendix A, tuples over the relations Airplane and Engine must have already existed in a database before they are referenced by tuples over the relation InstalledEngine.
- A general constraint is a constraint that is additionally specified according to the semantics of a database. In an example RDM presented in Appendix A, a general constraint of the relation MarriedPerson is defined to protect users from inserting the same values for attributes married person ID and spouse person ID of the same tuple over the relation MarriedPerson.

MarriedPerson = {married person ID: Integer, spouse person ID: Integer}
Primary key: {married person ID}
Foreign key: [married person ID] $\subseteq$ Person [person ID]
[spouse person ID] $\subseteq$ Person [person ID]
Constraint: married person ID $\neq$ spouse person ID

This simply implies the semantics of a database that is, a person cannot marry himself.

As presented, a RDM describes structure of data in a database by using the concepts of a mathematics relation (Abiteboul et al., 1995; Codd, 1970, 1982; Elmasri & Navathe, 2000; Ramakrishnan & Gehrke, 2003). A RDM is simple and easy to understand for all kinds of users and programmers. Various constraints can also be defined to ensure the

consistency of data in a database. However, in a RDM, data can only be constructed as tuples over relation, and domains of attributes can only be sets of constants. These mean that a RDM would not work well for modelling complex data.

2.2 The concept of an OODM

An OODM provides various aspects that meet the needs to model complex data (Atkinson et al., 1989; Elmasri & Navathe, 2000; Kim, 1990a; Maier, 1986; Schewe & Thalheim, 1993; Zand et al., 1995). It allows users to arbitrarily define complex data, which is an important advantage of an OODM over traditional data models. Hence, in the early stage of an OODM, an OODBS, which is built based on an OODM, has been thought to be able to overcome the limitations of a RDBS that is built based on a RDM (Atkinson et al., 1989; Leavitt, 2000; Schewe & Thalheim, 1993). The limitations of a RDBS result from the problems a RDM has in supporting the modelling of complex data that is manipulated by certain types of applications such as CAD systems, CASE, and multimedia applications. However, RDBSs are still the most widely used database system today. It is stated by Leavitt (2000) that in 2000 the sales of RDBSs were 50 times larger than the sales of OODBSs.

One important reason that makes OODBSs unsuccessful in database markets, although an OODM is more capable compared with a RDM, is that there is still no concrete agreement on the formal concept of an OODM (Atkinson et al., 1989; Schewe & Thalheim, 1993). Furthermore, the powerful concept of an OODM results in complexities in the implementation of an OODBS (Kuckelberg, 1998; Zezulan & Rabitti, 1993). Moreover, several criticisms and concerns in an OO paradigm still remain open and need further investigation and research (Schewe & Thalheim, 1993).

Although there is still no concrete agreement on the formal concept of an OODM, it is broadly accepted that an OODM is based on the following fundamental concept (Bertino, 1994; Elmasri & Navathe, 2000; Kim, 1990a, 1990b; Schewe & Thalheim, 1993). Any real world entity is represented by an object. An object has a unique OID that encodes its identity. An object has methods that represent its behaviours. Several

type constructors, such as a set, a list, a union of type constructors, etc., can be used to construct object values. Objects reference each other via their OIDs. Objects with similar structures and behaviours are categorised into the same class. A class (called a subclass) may inherit structures and behaviours from other classes (called superclasses), and it may specify its owns local structure and behaviour. This results in that an object may belong to many classes, i.e. an object that belongs to a subclass must belong to a superclass.

Because there have been a number of OODMs developed, note that the concept of an OODM presented in this section is based on an OODM presented by Schewe & Thalheim (1993). For simplicity of understanding, the section describes the concept of an OODM along with parts of a sample OODM of a database system of an air operator company, which is fully presented in Appendix B.

In Schewe & Thalheim (1993), an object is an abstraction of a real world object. It consists of a unique object identifier (OID), a set of (type-, value-) pairs, a set of (reference-, object-) pairs and a set of methods.

An OID encodes an object identity, which makes an object distinct from a value because a value has no identity (Schewe & Thalheim, 1993). A value identifies itself. An OID is immutable, abstract (has no meaning) and hidden from end users because it serves as an internal mechanism for a database system to uniquely identify an object. In addition, objects reference each other via their OIDs. Using an OID to identify an object also results in the independence between object identity and values, which supports the ease of sharing and updating of values.

Types define values (Cardelli & Wegner, 1985 cited in Schewe & Thalheim, 1993). A type can be thought of as an unchangeable set of uniform structure values with their defined operations. Types can be categorised into a base type, a constructed type and a generalised constructed type. A base type is either Boolean, Natural, Integer, Float, String, ID (an abstract identifier type), Date, or a trivial type. A constructed type is constructed by type constructors such as, a record, a finite set, a union of type constructors, etc. A base type and a constructed type are called a proper type.

A generalised constructed type may be defined as a recursive type constructed by substituting a type or a constructed type into a type constructor. A type expression that does not contain a base type ID is called a value type expression. The following is a type expression of a type PERSON defined in an example OODBM presented in Appendix B. It is constructed by a constructed type record (.).

```
Type  PERSON
      = (PersonID: Integer, Name: String, DateOfBirth: Date, Address: String)
End   PERSON
```

Subtyping is a partial order ($\leq$) relation on types (Cardelli & Wegner, 1985). A function $t' \rightarrow t$ is called a subtype function, where t' and t is a subtype and a supertype, respectively. A type is a set. Therefore, a subtype can be defined as a subset of a supertype. $t' \rightarrow t$ can also be written as $t' \subseteq t$. A set of values defined by a subtype is a subset of a set of values defined by a supertype. Consider an example OODM presented in Appendix B. Types PERSON and PILOT are defined as follows. Both of them are constructed by a constructed type record (.).

```
Type  PERSON
      = (PersonID: Integer, Name: String, DateOfBirth: Date, Address: String)
End   PERSON

Type  PILOT
      = (PersonID: Integer, Name: String, DateOfBirth: Date, Address: String,
        PilotLicenseNo: String)
End   PILOT
```

A type PILOT is a subtype of a type PERSON because a set of records of a type PILOT is a subset of records of a type PERSON. (A type PERSON is a type of all records that consist of at least values of attributes PersonID, Name, DateOfBirth and Address while a type PILOT is a type of all records that consists of at least values of attributes PersonID, Name, DateOfBirth, Address and PilotLicenseNo.)

A class defines structure and behaviours of objects (Schewe & Thalheim, 1993). In Schewe & Thalheim (1993), structure combines aspects of object values and references. A definition of a class is given in both static and dynamic views. In a static view, a class serves as a structural primitive of objects. It is defined by a class name, a structure expression, and a set of superclasses. Each object in a class in a static view has an OID, a collection of values, and references to objects in other classes. An OID is represented by a value of a base type ID. In a dynamic view, a class is defined by a class name, a structure expression, a set of superclasses, and a set of methods. In this thesis, associative access in an OODBS, which is mainly related to object retrieval based on conditions specified on objects, is the focus. Thus, methods are not taken into account, and a definition of a class in only a static view is explained in the thesis.

A structure expression of a class describes structure of objects in the class. It includes types of object values, and references to objects in other classes (Schewe & Thalheim, 1993). A structure expression in which a base type ID replaces a description of references is called a representation type (T_C) type of the class. Hence, object values and references can be described by a representation type (T_C), where each occurrence of ID denotes a reference of the object to another object.

A class is called a subclass if it inherits structure of objects from another class that is called a superclass (Schewe & Thalheim, 1993). A subclass may define additional types and references in its structure expression other than types and references that are defined in a superclass. Subclass and superclass relations correspond to subtype and supertype relations. A type expressed in a structure expression of a subclass is a subtype of a type expressed in a structure expression of a superclass. Objects of a subclass must also belong to a superclass.

Consider classes PERSONC, PILOTC, PARTC and ENGINEC in an example OODM in Appendix B, which are defined as follows. A constructed type record is represented by (.). $\circ$ is used to link types record. A constructed type set is represented by {.}.

```
Class  PERSONC
       Structure  PERSON
End    PERSONC

Class  PILOTC
       IsA PERSONC
       Structure  PILOT
End    PILOTC

Class  PARTC
       Structure  PART
End    PARTC

Class  ENGINEC
       Structure  ENGINE ° (Part: {PARTC})
End    ENGINEC
```

The expressions of classes PERSONC, PILOTC, PARTC and ENGINEC presented above can be explained as follows.

- Classes PERSONC and PARTC are defined by class names PERSONC and PARTC, respectively. Their class structure expressions describe types of values of objects of classes PERSONC and PARTC, which are types PERSON and PART, respectively. Objects of classes PERSONC and PARTC do not reference any object. Both classes PERSONC and PARTC do not have superclasses.
- A class PILOTC is defined as a subclass of a class PERSONC by an IsA relationship. Its class structure expression describes a type of values of objects of the class, which is a type PILOT. Objects of a class PILOTC do not reference any object.
- A structure expression of a class ENGINEC describes a type of values of objects of the class, which is a type ENGINE, and references to a set of objects of a class PARTC.

The database schema that describes structure of data in an OODM is defined as a collection of classes (Schewe & Thalheim, 1993). The instance of a database is the content of a database at a given timepoint. It is defined as a finite set of pairs of values of type (OID: ID, value: T_C) that satisfy the following constraints.

- Uniqueness of OIDs: OIDs are unique within a class,
- Inclusion Integrity: An object of a subclass always belongs to the superclass, i.e. an OID of an object of a subclass must occur as an OID of the same object of a superclass.
- Referential Integrity: An object can reference only existing objects in a database, i.e. a value of a base type ID, which occurs in an object value of a representation type (T_C), must occur as an OID of an object in a referenced class.

Consider the following element of the instance of an example OODM in Appendix B. It is an object of a class ENGINEC.

$$(i_{22}, (\text{“CFM56-3C1”}, \text{“E001”}, 1999, \text{“installed”}, \{i_8, i_9\}))$$

i_{22} is a value of a base type ID, which represents an OID of the object of a class ENGINEC. i_8 and i_9 are also values of a base type ID but they represent references from an object i_{22} to objects i_8 and i_9 of a class PARTC.

As presented, an OODM constructs structure of data in a database by combining aspects of values and references into structure of objects (Schewe & Thalheim, 1993). Types are used to define values while classes are used to define objects. In an OODM, various types can be used to define values including base types, constructed types and generalised constructed types. An object has an OID, which encodes its identity. An OID makes an object distinct from a value because a value identifies itself. An object can reference other objects via their OIDs. In addition, an object can be captured by a collection of possible classes, i.e. an object of a subclass is also an object of a superclass. These make an OODM can support better for modelling complex data that require more complex structure compared with a RDM.

2.3 Differences between RDM and OODM concepts

Concepts of RDM and OODM presented in the last two sections show that an OODM offers more various aspects (i.e. OID, type constructors, superclass and subclass relations, and references, etc.) to provide ease of data modelling. A brief summary of differences between RDM and OODM concepts is presented in Table 2.1.

Table 2.1 Differences between RDM and OODB concepts

RDM	OODM
1. The database schema that describes structure of data in a database is defined as a collection of relations.	1. The database schema that describes structure of data in a database is defined as a collection of classes.
2. A relation is defined as a subset of the Cartesian product $S_1 \times S_2 \times \ldots \times S_n$, where S is a domain that is a set of constants such as a set of Strings, a set of Integers, a set of Booleans, etc. As defined, a relation is said to have degree n. A relation can also be defined as a set of tuples. A relation schema describes a relation. It is made up of a relation name, and a list of attributes and their domains. An attribute is a name whose values are elements of its domain specified in the relation schema.	2. A class that serves as a structural primitive of objects is defined by a class name, a structure expression, and a set of superclasses. A structure expression of a class describes structure of objects in the class. It includes types of object values, and references to objects in other classes. A structure expression in which the base type ID replaces a description of references is called a representation type (T_C) type of the class. Hence, object values and references can be described by a representation type (T_C), where each occurrence of ID denotes a reference of the object to another object.

RDM	OODM
3. Each tuple over a relation is an ordered list of n values, where the first value is an element of S_1, the second value is an element of S_2, and so on.	3. Each object in a class that serves as a structural primitive of objects consists of an OID, a collection of values and references to objects in other classes.
4. Values of attributes of a relation are defined by domains that can only be sets of constants such as a set of Strings, a set of Integers, a set of Booleans, etc.	4. Values of objects can be defined by various types including base types, constructed types, and generalised constructed types.
5. A primary key of a relation is a minimum subset of attributes whose values uniquely identify tuples over that relation. Hence, a primary key depends on attribute values.	5. An object has an OID that encodes its object identity. This makes an object distinct from a value because a value has no identity (Schewe & Thalheim, 1993). A value identifies itself. An OID is immutable, abstract (has no meaning) and hidden from end users because it serves as an internal mechanism for a database system to uniquely identify an object. In addition, objects reference each other via their OIDs. Using an OID to identify an object results in the independence between object identity and values, which supports the ease of sharing and updating of values. Thus, updating object values does not affect an OID of the object.

RDM	OODM
6. Tuples reference each other by a foreign key that is a subset of attributes of a relation. A value of an element of a foreign key of a relation must also exist as a value of a primary key of some relations in a database. Hence, updating a value of a primary key of a relation, which is part of a foreign key of another relation, also causes the necessity to update values of the foreign key too. This also makes difficulties to model complex data. For example, in a RDM presented in Appendix A, it is troublesome to model a real world engine that consists of parts where each part is also needed to be maintained separately, and can be moved to install with another engine. In an example RDM in Appendix A, we represent data of engines and parts by three relations that are Engine, Part and InstalledEngine. Updating values of primary keys of the relations Part and Engine causes updating values of foreign keys of the relation InstalledEngine too.	6. Objects reference each other via their OIDs. An OID is immutable and independent from object values. Hence, updating values of objects that reference each other do not affect their OIDs. Being able to reference other objects also simplifies modelling complex data. Consider an OODM presented in Appendix B. Being able to define references from a class ENGINEC to a class PARTC makes it easy to model a real world engine that consists of parts where each part is also needed to be maintained separately and can be moved to install with another engine. An object of a class ENGINEC represents an engine and its parts. At the same time, an object of a class PARTC represents an engine part. Updating values of an object of a class PARTC does not affect an object of a class ENGINEC that references it. Likewise, updating values of an object in a class ENGINEC does not affect objects of a class PARTC that are referenced by it.

RDM	OODM
7. To model objects that have multiple structures, for example a person who is a pilot, a foreign key is employed. A similar problem that results from that primary and foreign keys are dependent to values also occurs – that is updating values of a primary key of a relation that is a foreign key of another relation causes the necessity to update values of the foreign key too.	7. An object can be captured by a collection of possible classes. A class (called a subclass) can inherit structure from other classes (called superclasses). An object of a subclass is also an object of a superclass. The inclusion integrity defines that an OID of an object of a subclass must occur as an OID of the same object of a superclass. These simplify modelling objects that have multiple structures, for example, a real world person who is a pilot. Updating values of an object defined in a subclass does not affect values of an object defined in a superclass. Similarly, updating values of an object defined in a superclass does not affect values of an object defined in a subclass.

2.4 Summary

In this chapter, concepts of RDM and OODM are presented, followed by a comparison between them, in order to develop understanding on how data in RDBS and OODBS is structured in a logical level.

A RDM describes structure of data in a database by using the concepts of a mathematics relation, which is simple and easy to understand for all kinds of users and programmers

(Codd, 1970, 1982). Various constraints can also be defined to ensure the consistency of data in a database. A RDM provides significant improvements on earlier data models by offering a high degree of physical data independency and high-level language concepts that allow users to express operations on large sets of data at a time. However, these aspects are not enough to model complex data because in a RDM data can only be constructed as tuples over relations, and domains of attributes can only be sets of constants.

In contrast, an OODM provides various aspects to support modelling complex data. Although there is still no concrete agreement on the formal concept of an OODM, it is widely accepted that an OODM is based on the following fundamental concept (Bertino, 1994; Elmasri & Navathe, 2000; Kim, 1990a, 1990b; Schewe & Thalheim, 1993). An object represents any real world entity. Each object has an OID, a collection of values and references to objects in other classes. Various types can be used to define values. Classes are used to define objects. Objects with multiple structures can be captured by a collection of possible classes. A class (called a subclass) may inherit structure from other classes (called superclasses), and it may specify its owns local structure. An object that belongs to a subclass must also belong to a superclass.

CHAPTER 3

OBJECT ORIENTED DATABASE SYSTEMS (OODBSs)

Although there is still no agreement on the formal concept of an OODM, there have been a number of OODBSs developed and researched for both commercial and academic purposes because the concept of an OODM has shown its advantages in modelling complex data (Atkinson et al., 1989; Elmasri & Navathe, 2000; Kim, 1990a; Maier, 1986; Schewe & Thalheim, 1993; Zand et al., 1995). These OODBSs are implemented in different ways, i.e. they have different OODMs and components, and employ different approaches to support data access in storage devices.

This chapter briefly describes components and their functions of several proposed OODBSs including ORION (Kim, Ballou et al., 1989), O_2 (Banchilhon et al., 1992), Link Objects in a Query Integrated System (LOQIS) (Subieta, 1994a, 1994b), and multi-level architecture for distributed object bases (Kirchberg et al., 2003).

3.1 ORION

An ORION is an OODBS prototype developed by the Advanced Computer Architecture Program at Microelectronics and Computer Technology Corporation (MCC) (Kim, Ballou et al., 1989; Zand et al., 1995). An ORION is a single-user database system running in a workstation environment. It consists of four subsystems including message handler, object, transaction and storage subsystems as presented in Figure 3.1.

The message handler receives all messages and passes them to the system (Kim, Ballou et al., 1989). Such messages are for example user-defined methods, requests to access or update data, and system-defined functions.

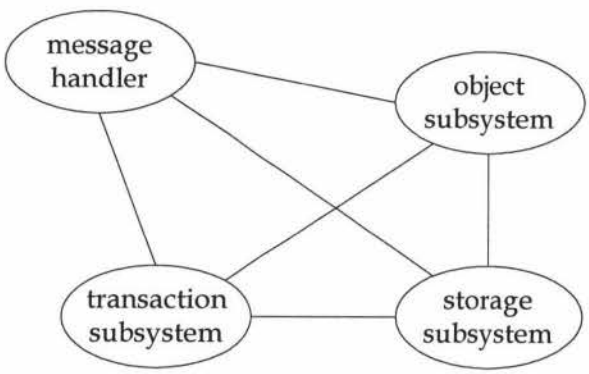


Figure 3.1 ORION Architecture (Kim, Ballou et al., 1989, p.253)

The object subsystem provides functions in a high-level (a logical level) such as schema evolution, version control, query optimisation, and multimedia information management (Kim, Ballou et al., 1989).

The storage subsystem organises data in magnetic disks (a physical level) (Kim, Ballou et al., 1989). Basically, the storage subsystem allocates a segment in a magnetic disk for a class. Objects of the same class are automatically located in the same segment. However, users can specify clustering objects, which are always simultaneously accessed, in the same segment. A data access capability of the system is implemented in the storage subsystem. A SC[4] index proposed by Kim, Kim et al. (1989) is used to support data access in the ORION (Kim, Kim et al., 1989).

The transaction subsystem provides concurrency control and crash recovery (Kim, Ballou et al., 1989). A locking protocol is used to handle interleaved transactions. A logging mechanism is used for crash recovery.

---

[4] See details of a SC index in chapter four, section 4.2.2.2

3.2 O_2

An O_2 system is an OODBS with a complete development environment and a set of user interface tools developed by Altaïr research consortium (Banchilhon et al., 1992; Zand et al., 1995). An O_2 system is designed to support applications like Computer Aided Design and Manufacturing (CAD/CAM), geographic and urban systems, editorial information systems, and office automation. However, it is also able to support traditional applications such as business and transaction applications. An O_2 system has a server/workstation system architecture. Both server and workstations have similar interface. The main difference is that a server is multi-user and disk based while a workstation is single user and memory based.

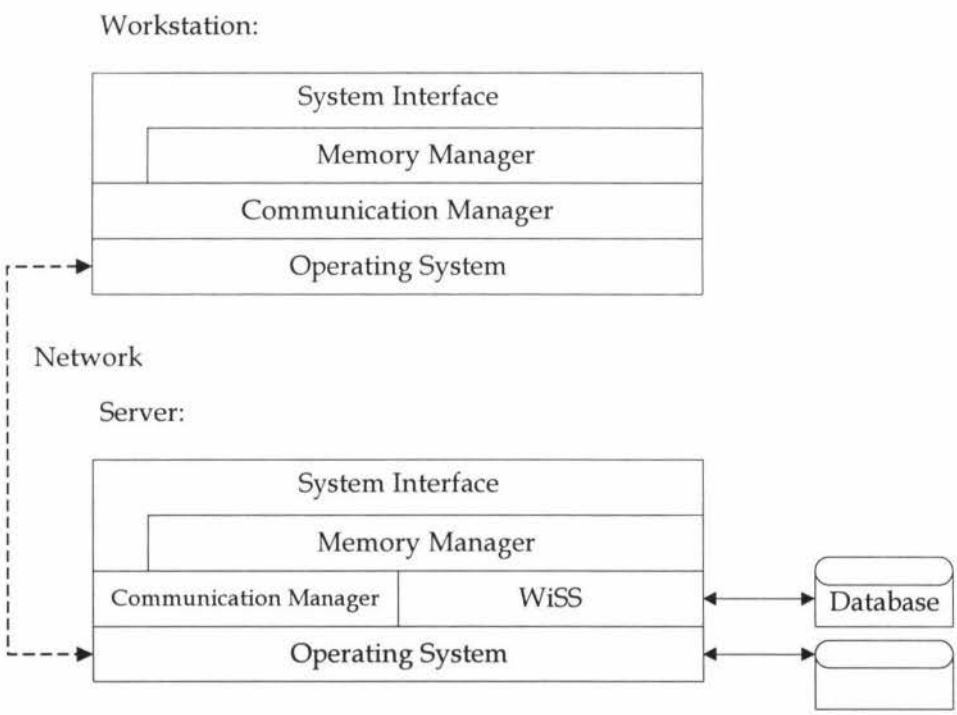


Figure 3.2 The O_2 Object Manager (Banchilhon et al., 1992, p. 356)

An O_2 system consists of two main components that are the schema manager and the O_2 object manager (Banchilhon et al., 1992). The schema manager is an upper layer (a logical level) functioning to create, retrieve, update and delete classes, methods and

global names. It also enforces the semantics of inheritance and checks the consistency of a schema.

The O_2 object manager is considered as a major component of an O_2 system (Banchilhon et al., 1992). It consists of layers as presented in Figure 3.2. The O_2 object manager deals with the representation and organisation of data in magnetic disks, data transfer between main memory and magnetic disks, the implementation of method calls, the creation and deletion of objects, the distribution in a server/workstation environment, concurrent access, crash recovery, query optimisation, indexing and versions.

Generally, an object defined in a data model level is stored as a data record handled by a lower layer of the O_2 object manager called a Wisconsin Storage System (WiSS) (Banchilhon et al., 1992). A data record identifier, which consists of a volume identifier, a page identifier and a slot number locating where a data record resides in a magnetic disk, is used as an OID. A WiSS takes full control of the organisation of locations of data records in magnetic disk pages. A clustering technique called placement trees is employed to cluster objects that are always retrieved together to stay as close as possible in magnetic disks. A WiSS also implements data access approaches to support access to data in magnetic disks. For associative access to objects in an inheritance hierarchy, a WiSS implements SC or CH[5] indices proposed by Kim, Kim et al. (1989). Locking and Write Ahead Log (WAL) techniques are used for concurrency control, rollbacks and recovery.

3.3 Link Objects in a Query Integrated System (LOQIS)

Link Objects in a Query Integrated System (LOQIS) is a high-level programming system (Subieta, 1994a, 1994b). It is intended to be used as a tool to develop complex applications that require non-traditional databases such as an OO database. LOQIS consists of two subsystems called dynamic memory and POS.

---

[5] See details of a CH index in chapter four, subsection 4.2.2.3

Logical objects

PERSONC class
{(i_1, (1, "Supa Rattana", 30-09-1970, "123 Sakorn Road, Bangkok")),
(i_4, (4, "Ram Pongsa", 04-06-1975, "567 Lamklong Road, Phuket")),
(i_6, (6, "Suay Rattana", 26/11/1972, "123 Sakorn Road, Bangkok"))}

MARRIEDPERSONC class
{(i_1, (1, "Supa Rattana", 30-09-1970, "123 Sakorn Road, Bangkok", i6),
(i_6, (6, "Suay Rattana", 26/11/1972, "123 Sakorn Road, Bangkok" , i1)}

PILOTC class
{(i_1, (1, "Supa Rattana", 30-09-1970, "123 Sakorn Road, Bangkok", "P001"))}

POS (storage objects)

Objects
<O_1, PERSON, {<O_2, PersonID, 4>, <O_3, Name, "Ram Pongsa">, <O_4, DateOfBirth, 04-06-1975 >,
<O_5, Address, "567 Lamklong Road, Phuket">}>
<O_6, PERSONMARRIEDPERSONPILOT, {<O_7, PersonID, 1>, <O_8, Name, "Supa Rattana">,
<O_9, DateOfBirth, 30-09-1970>,
<O_{10}, Address, "123 Sakorn Road, Bangkok">,
<O_{11}, Spouse, O13>, <O_{12}, PilotLicenseNo, "P001">}>
<O_{13}, PERSONMARRIEDPERSON, {<O_{14}, PersonID, 6>, <O_{15}, Name, "Suay Rattana">,
<O_{16}, DateOfBirth, 26-11-1972>,
<O_{17}, Address, "123 Sakorn Road, Bangkok">, <O_{18}, Spouse, O_6>}>

Classes
<O_{19},PERSONC, {<O_{20}, PersonID, Integer>, <O_{21}, Name, String>, <O_{22}, DateOfBirth, Date>,
<O_{23}, Address, String>}>
<O_{24},MARRIEDPERSONC, {<O_{25}, Spouse, MARRIEDPERSONC>}>
<O_{26},PILOTC, {<O_{27}, PilotLicenseNo, String>}>
<O_{28},PERSONMARRIEDPERSON, {}>
<O_{29},PERSONMARRIEDPERSONPILOTC, {} >

Inheritance
<O_{24}, O_{19}>, <O_{26}, O_{19}>, <O_{28}, O_{19}>, <O_{28}, O_{24}>, <O_{29}, O_{19}>, <O_{29}, O_{24}>, <O_{29}, O_{26}>

Object in a class
<O_1, O_{19}>, <O_6, O_{29}>, <O_{13}, O_{28}>

Figure 3.3 An Approach to Map Logical Objects to Atoms in LOQIS Proposed by Subieta (1994a; 1994b)

A dynamic memory subsystem is similar to a heap of C or Pascal however objects can be arbitrarily extended or shortened (Subieta, 1994a, 1994b). In addition, objects that are not accessed for a period of time are automatically stored in a secondary file.

A POS subsystem handles persistent objects in magnetic disks (Subieta, 1994a, 1994b). A POS subsystem allows objects to be organised into complex hierarchies. In a POS subsystem, objects are mapped to units called atoms (storage objects). Each atom has a persistent OID. An atom may store a record, an attribute of an entity, text, number,

graphic screen, a reference (pointer) to another atom, etc. A structure expression of a class is stored in an atom, too.

Logical objects

PERSONC class
{(i_1, (1, "Supa Rattana", 30-09-1970, "123 Sakorn Road, Bangkok")),
(i_4, (4, "Ram Pongsa", 04-06-1975, "567 Lamklong Road, Phuket"))
(i_6, (6, "Suay Rattana", 26/11/1972, "123 Sakorn Road, Bangkok"))}

MARRIEDPERSONC class
{(i_1, (1, "Supa Rattana", 30-09-1970, "123 Sakorn Road, Bangkok", i_6),
(i_6, (6, "Suay Rattana", 26/11/1972, "123 Sakorn Road, Bangkok" , i_1)}

PILOTC class
{(i_1, (1, "Supa Rattana", 30-09-1970, "123 Sakorn Road, Bangkok", "P001"))}

POS (storage objects)

Objects
<O_1, PERSON, {<O_2, PersonID, 1>, <O_3, Name, "Supa Rattana">, <O_4, DateOfBirth, 30-09-1970>,
<O_5, Address, "123 Sakorn Road, Bangkok">}>
<O_6, MARRIEDPERSON, {<O_7, Spouse, O_{10}>}>
<O_8, PILOT, {<O_9, PilotLicenseNo, "P001">}>
<O_{10}, PERSON, {<O_{11}, PersonID, 6>, <O_{12}, Name, "Suay Rattana">, <O_{13}, DateOfBirth, 26-11-1972>,
<O_{14}, Address, "123 Sakorn Road, Bangkok">
<O_{15}, MARRIEDPERSON, <O_{16}, Spouse, O_1>}>

Classes
<O_{17},PERSONC, {<O_{18}, PersonID, Integer>, <O_{19}, Name, String>, <O_{20}, DateOfBirth, Date>,
<O_{21}, Address, String>}>
<O_{22},MARRIEDPERSONC, {<O_{23}, Spouse, MARRIEDPERSONC>}>
<O_{24},PILOTC, {<O_{25}, PilotLicenseNo, String>}>

Inheritance
<O_6, O_1>, <O_8, O_1>, <O_{15}, O_{10}>

Objects in a class
<O_1, O_{17}>, <O_{10}, O_{17}>, <O_6, O_{22}>, <O_{15}, O_{22}>, <O_8, O_{24}>

Figure 3.4 An Approach to Map Logical Objects to Atoms in LOQIS Proposed by Jodlowski (2002)

In a POS of LOQIS, a value of an attribute is held by an atom (Subieta, 1994a, 1994b). Several atoms that hold values of attributes form an atom that represents a logical object. Membership of classes is maintained by atoms where each holds a pair of OIDs of an atom representing a logical object and an atom representing a structure expression of a class of the logical object. A reference between logical objects is maintained in a POS by an atom that holds a pair of OIDs of atoms that represent the corresponding

logical objects. An inheritance is maintained in a class level by an atom that holds a pair of OIDs of an atom reflecting a structure expression of a subclass and an atom reflecting a structure expression of a superclass.

However, an original approach to map logical objects to atoms of LOQIS cannot work well for multiple inheritances (Jodlowski, 2002). Therefore, Jodlowski (2002) proposes a new approach to maintain an inheritance in an object level by storing pairs of OIDs of an atom reflecting a logical object of a subclass and an atom reflecting a logical object of a superclass.

Figure 3.3 presents an example of mapping logical objects defined in an OODM in Appendix B to atoms in a POS by an approach originally proposed by Subieta (1994a; 1994b). And, Figure 3.4 presents an example of mapping logical objects defined in an OODM in Appendix B to atoms by an approach proposed by Jodlowski (2002).

In a POS of LOQIS, two data structures called ring and spider[6] are used to support navigational access (Subieta, 1994a, 1994b).

3.4 A Multi-Level Architecture for Distributed Object Bases

A multi-level architecture for distributed object bases is proposed to support the needs of data distribution and efficient complex data storage and retrieval of an OODBS (Kirchberg et al., 2003). It is now being researched.

A system architecture of a multi-level architecture for distributed object bases for each site of a network comprises levels as presented in Figure 3.5. Each level of a system serves its higher levels.

At the top level, data is described in terms of an OODM (Kirchberg et al., 2003). An OODM proposed by Schewe & Thalheim (1993) is employed. In this level, data is

---

[6] See details of ring and spider data structures in chapter four, section 4.3.3

regarded as global logical objects. As for the needs of data distribution, global logical objects are split into local logical objects stored at each site by fragmentation techniques. For the efficient data storage and retrieval, local logical objects are then mapped to storage objects in a POS level. A POS provides facilities to retrieve storage objects in order to reconstruct local logical objects as requested from the upper levels. A POS supports three types of data access – direct, navigational and associative access, to storage objects. Approaches supporting the data access in a POS of a multi-level architecture for distributed object bases are to be researched and developed. Storage objects are subsequently mapped to physical objects stored in magnetic disks. Physical objects are data records stored in disk pages handled by a System Buffer and Record Administration Server (SyBRAS).

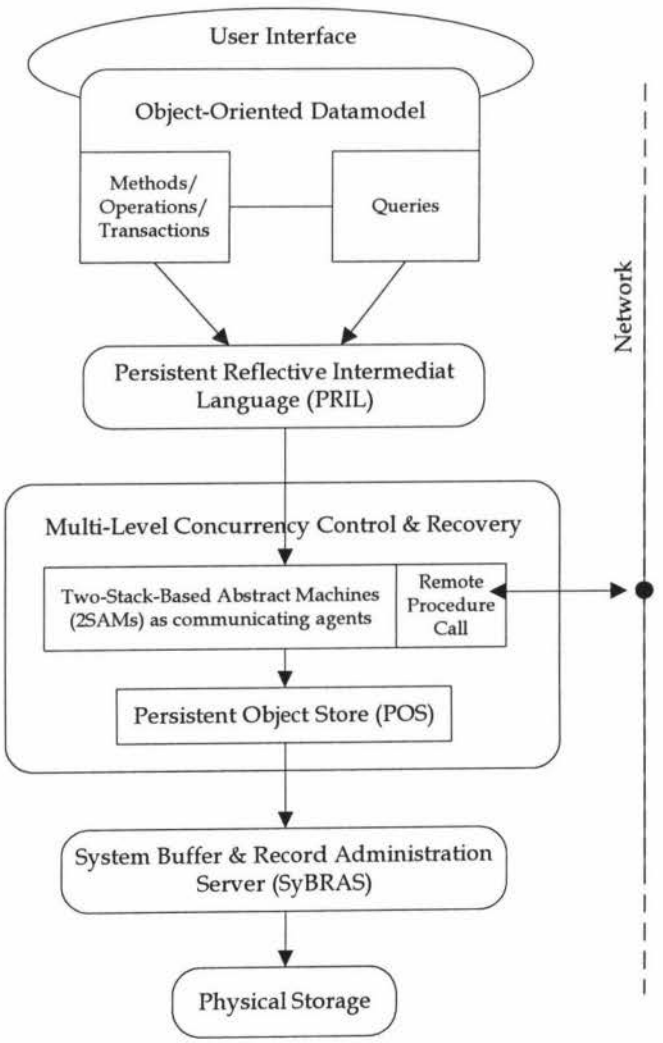


Figure 3.5 Architecture for Distributed Object Bases (Kirchberg et al., 2003, p.2)

The communication agents implement the functionality of the whole system (Kirchberg et al., 2003). They are comprised of two levels. The upper level functions as a co-ordinator for transactions. It deals with global logical objects constructed from local logical objects. The lower level connects to a POS and deals with local logical objects, which can be local logical objects stored in local magnetic disks or local logical object stored at other sites.

As global logical objects are not directly mapped to local logical objects processed by the communication agents, a Persistent Reflective Intermediate Language (PRIL) functions to translate methods, operations, transaction, queries, etc. into codes that can be interpreted by communication agents (Kirchberg et al., 2003).

3.5 Summary

This chapter introduces components and their functions of several proposed OODBSs including ORION (Kim, Ballou et al., 1989), O_2 (Banchilhon et al., 1992), LOQIS (Subieta, 1994a, 1994b) and multi-level architecture for distributed object bases (Kirchberg et al., 2003). Basically, these OODBSs have similar architecture as the three-level architecture presented in Figure 1.1 in chapter one. These OODBSs have a component regarded as a logical level that describes logical objects stored in a database in terms of a data model, and a component regarded as a physical level that handles physical objects (data records) that are stored in magnetic disks. In addition, the LOQIS (Subieta, 1994a, 1994b) and multi-level architecture for distributed object bases (Kirchberg et al., 2003) have a POS or a storage level that handles storage objects which are mapped from logical objects.

In the ORION and O_2, SC or CH indices that are approaches proposed by Kim, Kim et al. (1989) for associative access in an OODBS are implemented in a physical level of the systems (Banchilhon et al., 1992; Kim, Ballou et al., 1989).

Subieta (1994a; 1994b) does not mention about how associative access is supported in the LOQIS. However, navigational access is supported by ring and spider data structures in a POS of the system.

Data access approaches in the multi-level architecture for distributed object bases are to be researched and developed, and they are intended to be implemented in a POS level of the system (Kirchberg et al., 2003).

CHAPTER 4

EXISTING DATA ACCESS APPROACHES IN AN OODBS

This chapter reviews the existing data access approaches that support associative access in an OODBS. Moreover, the chapter reviews data access approaches that support navigational access in an OODBS because associative access in an OODBS often includes navigational access (Kim, Kim et al., 1989).

This chapter is separated into three sections.

- The first section illustrates examples of queries that require associative access in an OODBS in different circumstances including the following.
  - o Example queries that require only associative access,
  - o Example queries that are made against logical objects of classes in an inheritance hierarchy, i.e. queries that are made against logical objects that belong to many classes,
  - o Example queries that are made against logical objects of classes in a class-attribute hierarchy, i.e. queries that are made against logical objects and their referenced or referencing logical objects, and
  - o Example queries that are made against logical objects of classes in inheritance or class-attribute hierarchies.
- The second section describes approaches that are proposed for associative access in an OODBS, and
- The third section describes approaches that are proposed for navigational access in an OODBS.

Advantages and disadvantages of the reviewed approaches are then discussed at the end of the chapter.

4.1 Queries Requiring Associative Access in an OODBS

Associative access is retrieval of a set of OIDs of objects that meet specified conditions (Kirchberg & Tretiakov, 2002; Kuckelberg, 1998; Zezulan & Rabitti, 1993). Often, associative access in an OODBS includes navigational access (Kim, Kim et al., 1989). Example 4.1 presents example queries that require associative access in an OODBS in different circumstances.

Example 4.1 Example queries that require associative access in an OODBS in different circumstances. These queries are made against logical objects of a sample OODM presented in Appendix B.

Note that data access approaches for associative access in an OODBS reviewed in this chapter are all modifications of approaches used in a RDBS. In these approaches, a class is regarded as a relation and a logical object is regarded as a tuple over a relation. A UID, which is a system-defined unique identifier for a tuple, is regarded as an OID of a logical object. Details of mapping between logical objects to storage objects or physical objects are not given. However, because these approaches are modifications of approaches used in a RDBS, we assume that they are constructed based on the organisation of physical objects, data records stored in a magnetic disk.

Therefore, we explain how these queries are supported by assuming that a data access capability is provided in a physical level of a database system. However, these explanations are similar to what occurs when a data access capability is provided in a POS.

- Example queries that require only associative access

  Q1. Retrieve all parts manufactured in 1999.

  Q1 (Retrieve all parts manufactured in 1999.) retrieves logical objects of a class PARTC whose value of an attribute ManufacturedYear is 1999. Consider an OODM presented in Appendix B. A class PARTC does not reference any class. It also has no superclass. Assume that a logical object of a class PARTC is mapped to a physical object that holds values of attributes defined in a class PARTC. Q1 can be supported by associative access to physical objects corresponding to logical objects of a class PARTC whose value of an attribute ManufacturedYear is 1999.

- Example queries that are made against logical objects of classes in an inheritance hierarchy, i.e. queries that are made against logical objects belonging to many classes

  Q2. Retrieve all people who were born in 1975.

  Q3. Retrieve mechanics and pilots who were born in 1975.

  - Q2 (Retrieve all people who were born in 1975.) retrieves logical objects of a class PERSONC whose value of year of birth of an attribute DateOfBirth is 1975. Consider an example OODM in Appendix B. A class PERSONC has classes PILOTC, MECHANICC and MARRIEDPERSONC as its subclasses. Hence, logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC are also logical objects of a class PERSONC. Assume that each logical object is mapped to a physical object as follows:
    - A logical object of only a class PERSONC is mapped to a physical object that holds values of attributes defined in a class PERSONC.
    - A logical object of a class PILOTC is mapped to a physical object that holds values of attributes inherited from a class PERSONC and values of attributes defined in a class PILOTC.
    - A logical object of a class MECHANICC is mapped to a physical object that holds values of attributes inherited from a class PERSONC and values of attributes defined in a class MECHANICC.

- A logical object of only a class MARRIEDPERSONC is mapped to a physical object that holds values of attributes inherited from a class PERSONC and values of attributes defined in a class MARRIEDPERSONC.

Assume that Q2 is meant to retrieve logical objects of a class PERSONC whose value of year of birth of an attribute DateOfBirth is 1975 (including logical objects that also belong to classes PILOTC, MECHANICC and MARRIEDPERSONC). Q2 can be supported by associative access to physical objects corresponding to logical objects of classes PERSONC, PILOTC, MECHANICC and MARRIEDPERSONC whose value of year of birth of an attribute DateOfBirth is 1975.

Assume that Q2 is meant to retrieve logical objects of only a class PERSONC (not including logical objects that also belong to classes PILOTC, MECHANICC and MARRIEDPERSONC). Q2 can be supported by associative access to physical objects corresponding to logical objects of only a class PERSONC whose value of year of birth of an attribute DateOfBirth is 1975.

° Q3 (Retrieve mechanic and pilots who were born in 1975.) retrieves logical objects of classes MECHANICC and PILOTC whose value of year of birth of an attribute DateOfBirth is 1975. In an example OODM in Appendix B, classes MECHANICC and PILOTC are defined as subclasses of a class PERSONC. An attribute DateOfBirth is inherited from a class PERSONC to classes MECHANICC and PILOTC. A class MECHANICC references a class AIRPLANEC. Assume that Q3 does not retrieve objects of a class AIRPLANEC and objects of a class that is referenced by a class AIRPLANEC and so on.

Assume that a logical object of a class MECHANICC is mapped to two physical objects in a magnetic disk, where one physical object holds values of attributes inherited from a class PERSONC and another physical object holds values of attributes defined in a class MECHANICC. Similarly, a logical object

of a class PILOTC is mapped to two physical objects in a magnetic disk, where one physical object holds values of attributes inherited from a class PERSONC and another physical object holds values of attributes defined in a class PILOTC. In this case, to support Q3, we may first perform associative access to physical objects that hold values of attributes inherited from a class PERSONC and match a value of year of birth of an attribute DateOfBirth, 1975. Then we navigate from the matched physical objects that hold values of attributes inherited from a class PERSONC to physical objects that hold values of attributes defined in a class MECHANICC and physical objects that hold values of attributes defined in a class PILOTC.

- Example queries that are made against logical objects of classes in a class-attribute hierarchy, i.e. queries made against logical objects and their referenced or referencing logical objects

  Q4. Retrieve details of all journeys of an airplane that is installed with an engine part whose serial number is Part001.

  Consider an example OODM in Appendix B. Logical objects of a class JOURNEYLOGC reference logical objects of a class AIRPLANEC; logical objects of a class AIRPLANEC reference logical objects of a class ENGINEC; and logical objects of a class ENGINEC reference logical objects of a class PARTC. Q4 retrieves logical objects of a class JOURNEYLOGC that reference a logical object of a class AIRPLANEC that references a logical object of a class ENGINEC that references a logical object of a class PARTC whose value of an attribute SerialNumber is Part001.

  - If corresponding logical objects of classes JOURNEYLOGC, AIRPLANEC, ENGINEC, and PARTC are clustered into the same physical object, Q4 falls into a type of query that requires only associative access. Thus, Q4 can be supported by associative access to physical objects that hold a value of an attribute SerialNumber, Part001.
  - In contrast, assume that each logical object is mapped into one physical object as follows:

- A logical object of a class JOURNEYLOGC is mapped to only one physical object that holds values of attributes defined in a class JOURNEYLOGC.
- A logical object of a class AIRPLANEC is mapped to only one physical object that holds values of attributes defined in a class AIRPLANEC.
- A logical object of a class ENGINEC is mapped to only one physical object that holds values of attributes defined in a class ENGINEC.
- A logical object of a class PARTC is mapped to only one physical object that holds values of attributes defined in a class PARTC.

References between these logical objects are mapped to references between the corresponding physical objects. One solution to support Q4 is to first perform associative access to a physical object corresponding to a logical object of a class PARTC whose value of an attribute SerialNumber is Part001. Then navigational access is made from the physical object corresponding to a logical object of a class PARTC whose value of an attribute SerialNumber is Part001 to a physical object corresponding to a logical object of a class ENGINEC, then to a physical object corresponding to a logical object of a class AIRPLANEC, and then to physical objects corresponding to logical objects of a class JOURNEYLOGC.

- Example queries that are made against logical objects of classes in inheritance or class-attribute hierarchies.

  Q5. Retrieve mechanics born in 1975 and maintaining a HS-AAA airplane.

Q5 (Retrieve mechanics born in 1975 and maintaining a HS-AAA airplane.) retrieves logical objects of a class MECHANICC whose value of year of birth of an attribute DateOfBirth is 1975, and that reference logical objects of a class AIRPLANEC whose value of an attribute RegistrationName is HS-AAA. Consider an example OODM in Appendix B. A class MECHANICC is a subclass of a class PERSONC and references a class AIRPLANEC. A class AIRPLANEC references a class ENGINEC. A class ENGINEC references a class PARTC. Assume that Q5 does not retrieve logical objects of classes ENGINEC and PARTC.

- In case that corresponding logical objects of classes PERSON, MECHANICC and AIRPLANEC are mapped to only one physical object, Q5 falls into a type of query that requires only associative access. Thus, Q5 can be supported by associative access to physical objects corresponding to logical objects of classes PERSON, MECHANICC and AIRPLANEC that hold value of an attribute RegistrationName, HS-AAA and value of year of birth of an attribute DateOfBirth, 1975.
- In contrast, assume that a logical object of only a class PERSONC is mapped to a physical object that holds values of attributes defined in a class PERSONC. A logical object of a class MECHANICC is mapped to two physical objects where one physical object holds values of attributes inherited from a class PERSONC and another physical object holds values of attributes defined in a class MECHANICC. A logical object of a class AIRPLANEC is mapped to a physical object that holds values of attributes defined in a class AIRPLANEC. To support Q5, we may perform
  - Forward navigation from physical objects corresponding to logical objects of a class MECHANICC to physical objects corresponding to logical objects of a class PERSONC whose value of year of birth of an attribute DateOfBirth is 1975, and
  - Forward navigation from the matched physical objects corresponding to logical objects of a class MECHANICC to a physical object corresponding to a logical object of a class AIRPLANEC whose value of an attribute RegistrationName is HS-AAA.

  Or, we may perform
  - Backward navigation from physical objects corresponding to logical objects of a class PERSONC whose value of year of birth of an attribute DateOfBirth is 1975 to physical objects corresponding to logical objects of a class MECHANICC, and
  - Backward navigation from a physical object corresponding to a logical object of a class AIRPLANEC whose value of an attribute RegistrationName is HS-AAA to physical objects corresponding to logical objects of a class MECHANICC.

This section presents example queries that require associative access in an OODBS in different circumstances. The section also shows how queries, which involve associative access in an OODBS, often involve navigational access. Furthermore, the section gives examples of how logical objects are mapped to physical objects by clustering or partitioning techniques (Kirchberg et al., 2003; Zezulan & Rabitti, 1993). It is seen that retrieval of a logical object may actually be retrieval of several storage or physical objects that result from partitioning the logical object. Or, retrieval of several logical objects may actually be retrieval of just one storage or physical object that results from clustering several logical objects together.

In the next two sections, approaches proposed to support associative access and navigational access in an OODBS are described, respectively.

4.2 Approaches Supporting Queries Requiring Associative Access in an OODBS

This section reviews data access approaches that support queries requiring associative access in an OODBS.

As mentioned, the approaches reviewed in this section are modifications of approaches used in a RDBS such as B+ tree and join indices. In these approaches, a class is regarded as a relation and a logical object is regarded as a tuple over a relation. A UID, which is a system-defined unique identifier for a tuple, is regarded as an OID of a logical object. Unfortunately, these approaches do not express details of mapping between logical objects to storage objects or physical objects. However, because these approaches are modifications of approaches used in a RDBS, we assume that these approaches are constructed based on the organisation of physical objects, data records stored in a magnetic disk.

We may categorise data access approaches that support queries requiring associative access into the following.

- Approaches that support queries that require only associative access,

- Approaches that support queries that are made against logical objects of classes in an inheritance hierarchy, i.e. queries that are made against logical objects that belong to many classes,
- Approaches that support queries that are made against logical objects of classes in a class-attribute hierarchy, i.e. queries that are made against logical objects and their referenced or referencing logical objects, and
- Approaches that support queries that are made against logical objects of classes in inheritance or class-attribute hierarchies.

This section is separated into four subsections according to the categorised data access approaches mentioned above.

4.2.1 Approaches Supporting Queries that require only Associative Access

Queries that require only associative access in an OODBS can be compared with queries that require associative access to data records of a relation in a RDBS. Hence, simple associative access approaches used in a RDBS such as B-tree or hashing indices can be employed. Q1 (Retrieve all parts manufactured in 1999.) of Example 4.1 can be taken as an example query that requires only associative access to objects, if a logical object of a class PARTC is mapped to a physical object that hold values of attributes defined in a class PARTC. Q1 can be supported by a B+ tree that clusters physical objects, which correspond to logical objects of a class PARTC, on values of the attribute ManufacturedYear defined in the class PARTC.

4.2.2 Approaches Supporting Queries Made against Logical Objects of Classes in an Inheritance Hierarchy

This subsection reviews data access approaches that support queries made against logical objects of classes in an inheritance hierarchy, i.e. queries that are made against logical objects that belong to many classes. An inheritance hierarchy is a directed graph, which presents a class and its subclasses (Kim, Kim et al., 1989).

Q2 (Retrieve all people who were born in 1975) of Example 4.1 can be taken as an example query that requires associative access to physical objects, which correspond to logical objects of classes in an inheritance hierarchy, for both of the following interpretations:

- Q2 is meant to retrieve logical objects of a class PERSONC whose year of birth is 1975 (including logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC), and
- Q2 is meant to retrieve logical objects of only a class PERSONC whose year of birth is 1975 (not including logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC).

Figure 4.1 presents an inheritance hierarchy of classes of an OODM presented in Appendix B used for Q2 in Example 4.1.

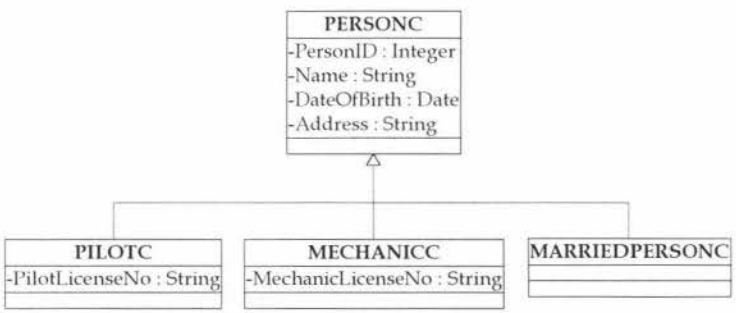


Figure 4.1 An Example Inheritance Hierarchy

The approaches reviewed in this subsection include SC and CH indices (Kim, Kim et al., 1989), H-tree (Low et al., 1992), hcC-tree (Sreenath & Seshadri, 1994), and CD index (Ramaswamy & Kanellakis, 1995). They all are modifications of B tree family indices.

Because the data access approaches reviewed in this section are modifications of B tree family indices, this subsection will first introduce B tree family indices in order to establish basic understanding. Subsequently, SC and CH indices (Kim, Kim et al., 1989), H-tree (Low et al., 1992), hcC-tree (Sreenath & Seshadri, 1994), and CD index (Ramaswamy & Kanellakis, 1995) will be described.

4.2.2.1 B Tree Family Indices

B tree family indices are introduced here as basic knowledge to aid understanding of the data access approaches reviewed in subsection 4.2.2.

B tree family indices include B, B* and B+ trees (Ooi & Tan, 2001). They are powerful dynamic tree data structures supporting associative access to data records stored in a RDBS. B-tree family indices capably adjust when data records are inserted or deleted. They efficiently support both equality retrieval, i.e. retrieve a data record of a Person relation whose Person ID is 1, and range retrieval, i.e. retrieve data records of a Person relation whose Person ID is less than 6.

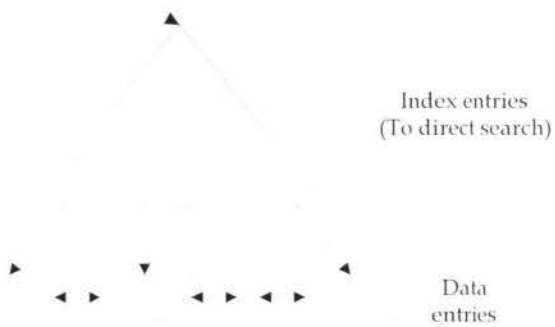


Figure 4.2 Structure of a B+ tree (Ramakrishnan & Gehrke, 2003, p.345)

A B tree was first developed, and both B* and B+ trees are modification of a B tree (Ooi & Tan, 2001). Here, a B+ tree is first described because it is the most widely used index among indices in a B tree family. Then differences between a B+ tree and the other two are concluded.

In a B+ tree, every node represents a disk page containing entries that are used to search for data records that match search key values (Kirchberg, 2003; Ramakrishnan & Gehrke, 2003). Hence, each time a node is visited refers to one I/O operation.

In a B+ tree, every node except for the root node contains m search keys, where $d \leq m \leq 2d$. The root node of a B+ tree contains m search keys, where $1 \leq m \leq 2d$. d is an order

of a B+ tree. It is used to measure the capacity of tree nodes. Figure 4.2 presents structure of a B+ tree.

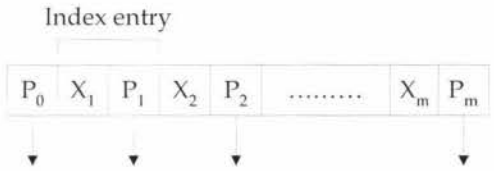


Figure 4.3 Structure of a Non-Leaf Node of a B+ Tree (Kirchberg, 2003, p.32)

Non-leaf nodes of a B+ tree are used for directing the search (Kirchberg, 2003; Ramakrishnan & Gehrke, 2003). Each contains index entries consisting of m search keys (X_i where $1 \leq i \leq m$) and m + 1 index pointers (P_j where $0 \leq j \leq m$) in a form of (P_0, X_1, P_1, X_2, ..., P_{m-1}, X_m, P_m). Structure of a non-leaf node of a B+ tree is presented in Figure 4.3. Each index pointer (P_i) in the non-leaf node, except for the left-most (P_0) and the right-most (P_m) index pointers, points to a node in the next level containing search keys (K) where $X_i \leq K < X_{i+1}$. The left-most index pointer (P_0) points to a node in the next level containing search keys $K < X_1$. The right-most pointer (P_m) points to a node in the next level containing search keys $K \geq X_m$.

Leaf nodes of a B+ tree locate disk pages where requested data records reside (Kirchberg, 2003; Ramakrishnan & Gehrke, 2003). Each of them contains data entries consisting of m search keys (K_i where $1 \leq i \leq m$), two neighbour node pointers (P_0 and P_{m+1}) and m data pointers (P_i where $1 \leq i \leq m$) in a form of (P, K_1, P_1, K_2, P_2, ..., P_{m-1}, K_m, P_m, N). Two adjacent leaf nodes of a B+ tree are doubly linked by pointers (P and N) in order to support sequential traversing between leaf nodes. Figure 4.4 shows the structure of a leaf node of a B+ tree.

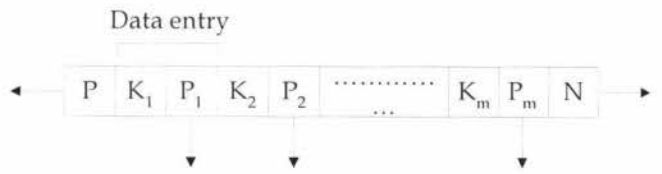


Figure 4.4 Structure of a Leaf Node of a B+ Tree (Kirchberg, 2003, p.33)

Searching in a B+ tree begins from the root of the tree and goes down to a leaf node that indicates a disk page where requested data records reside (Kirchberg, 2003; Ramakrishnan & Gehrke, 2003). Hence, for equality retrieval, the number of I/O operations is guaranteed to be equal to the height of a B+ tree. Figure 4.5 shows a sample B+ tree and demonstrates a search in a B+ tree for data records matching the search key value, 11.

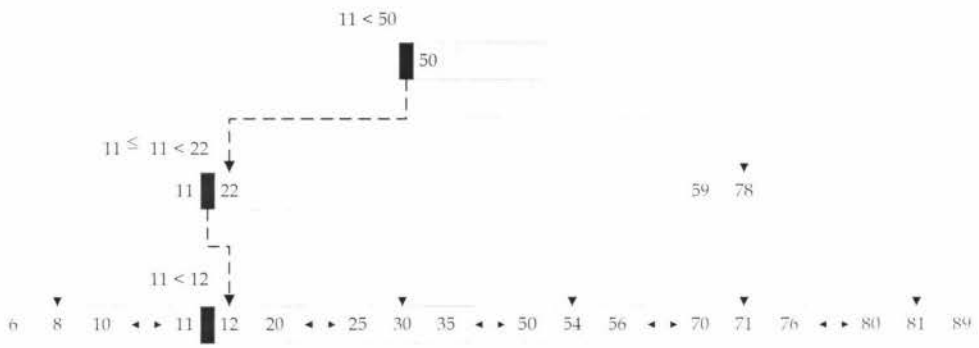


Figure 4.5 An Example B+ Tree (Ooi & Tan, 2001, p.14)

Table 4.1 Differences between B, B+ and B* trees

B tree	B* tree	B+ tree
1. Every node, except for the root node, contains m search keys, where $d \leq m \leq 2d$.)	1. Every node, except for the root node, contains m search keys, where $\frac{4}{3}d \leq m \leq 2d$.)	1. Every node, except for the root node, contains m search keys, where $d \leq m \leq 2d$.)
2. Search keys in non-leaf nodes are a subset of search keys appearing in leaf nodes.	2. Search keys in non-leaf nodes are a subset of search keys appearing in leaf nodes.	2. Search keys of non-leaf nodes are not necessary a subset of search keys appearing in leaf nodes.
3. Traversing along leaf nodes are not allowed.	3. Traversing along leaf nodes are not allowed.	3. Traversing along leaf nodes are allowed.

A B tree differs from a B+ tree in the following ways (Ooi & Tan, 2001). In a B tree, leaf nodes are not doubly linked therefore sequential traversing between the leaf nodes are not allowed. In addition, in a B tree, only values of search keys are allowed to appear in nodes of the tree. In contrast, in a B+ tree, values of search keys appear in leaf nodes of the tree while in non-leaf nodes any values are allowed. These differences result in that a B+ tree performs better for range retrieval because with a B+ tree we can do sequential traversing between the leaf nodes, which eliminates the necessity to begin the search for the next value of a search key in a range at the root of a tree. In addition, allowing values that are not search keys to appear in non-leaf nodes of a B+ tree simplifies the adjustment of the tree when new data records are inserted.

A B* tree differs from a B+ tree, in that for a B* tree all nodes except the root node are required to be at least two-thirds full, i.e. every node, except for the root node in a B* tree must contains m search keys, where $\frac{4}{3}d \leq m \leq 2d$ (Ooi & Tan, 2001). This is to reduce the number of nodes and the height of a B* tree in order to decrease the number of I/O operations required while searching in the tree. In addition, similar to a B tree, leaf nodes of a B* tree are not doubly linked therefore sequential traversing between the leaf nodes are not allowed. Also, only values of search keys are allowed to appear in nodes of a B* tree.

Differences between B, B+ and B* trees presented by Ooi & Tan (2001) are concluded as presented in Table 4.1.

SC and CH indices (Kim, Kim et al., 1989), H-tree (Low et al., 1992), hcC-tree (Sreenath & Seshadri, 1994), and CD index (Ramaswamy & Kanellakis, 1995) are approaches which are proposed to support queries made against logical objects of classes in an inheritance hierarchy. Next, each of them will be described.

4.2.2.2 A Single-Class (SC) Index

A SC index is proposed by Kim, Kim et al. (1989). It is a modification of a B tree. A SC index is constructed to support associative access to physical objects that correspond to

logical objects of a class in an inheritance hierarchy. In other words, a SC index clusters physical objects that correspond to logical objects of a class in an inheritance hierarchy on values of a search key.

Structure of a non-leaf node of a SC index is the same as structure of a non-leaf node of a B tree (Kim, Kim et al., 1989). It consists of m index entries where $d \leq m \leq 2d$. Each index entry consists of a pair of (a search key, an index pointer that points to a corresponding node at the lower level). A search key is a pair of (key-length, key-value). Structure of a non-leaf node of a SC index is presented in Figure 4.6.

Figure 4.6 A Non-Leaf Node of a SC Index (Kim, Kim et al., 1989, p.377)

A leaf node of a SC index consists of data entries where each consists of the length of a data entry, a search key, a pointer to an overflow page, the number of UIDs of data records that match a search key, and a collection of UIDs (Kim, Kim et al., 1989). Figure 4.7 shows the structure of a leaf node of a SC index.

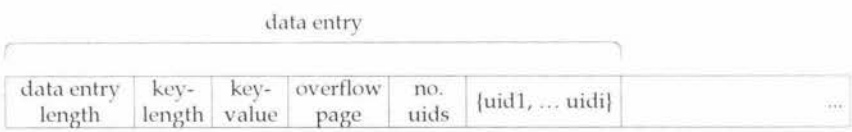


Figure 4.7 A Leaf Node of a SC Index (Kim, Kim et al., 1989, p.377)

The number of data entries contained in leaf nodes of a SC index depends on the length of search keys and the number of records that match each search key (Kim, Kim et al., 1989). If a small data entry grows out of a leaf node where it resides, but the size of a data entry is still smaller than the size of a leaf node, the leaf node is split. If a small data entry grows until its size is bigger than the size of a leaf node, an overflow disk

page is employed to store the rest of the data entry that cannot be placed in its original leaf node.

Consider Q2 (Retrieve all people who were born in 1975) of Example 4.1. If Q2 is meant to retrieve logical objects of a class PERSONC whose value of year of birth of an attribute DateOfBirth is 1975 (including logical objects that also belong to classes PILOTC, MECHANICC and MARRIEDPERSONC), two possible choices can be implemented.

- Four SC indices are needed to support Q2 as follows.
  - o A SC index clusters physical objects corresponding to logical objects of only a class PERSONC on values of year of birth of an attribute DateOfBirth,
  - o A SC index clusters physical objects corresponding to logical objects of a class PILOTC on values of year of birth of an attribute DateOfBirth,
  - o A SC index clusters physical objects corresponding to logical objects of a class MECHANICC on values of year of birth of an attribute DateOfBirth, and
  - o A SC index clusters physical objects corresponding to logical objects of a class MARRIEDPERSONC on values of year of birth of an attribute DateOfBirth.
- Only one SC index that clusters physical objects corresponding to all logical objects of a class PERSONC on values of year of birth of an attribute DateOfBirth is needed to support Q2.

If Q2 is meant to retrieve logical objects of only a class PERSONC whose year of birth is 1975 (not including logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC), only one SC index that is maintained for physical objects corresponding to logical objects of only a class PERSONC is required.

Compared with a CH index that is presented next, a SC index performs better to support queries made against logical objects of at most two classes in an inheritance hierarchy (Kim, Kim et al., 1989).

4.2.2.3 A Class-Hierarchy (CH) index

A CH index is proposed by Kim, Kim et al. (1989). It is a modification of a B tree. A CH index is constructed to support associative access to physical objects that

correspond to logical objects of all classes in an inheritance hierarchy. In other words, a CH index clusters physical objects that correspond to logical objects of all classes in an inheritance hierarchy on values of a search key.

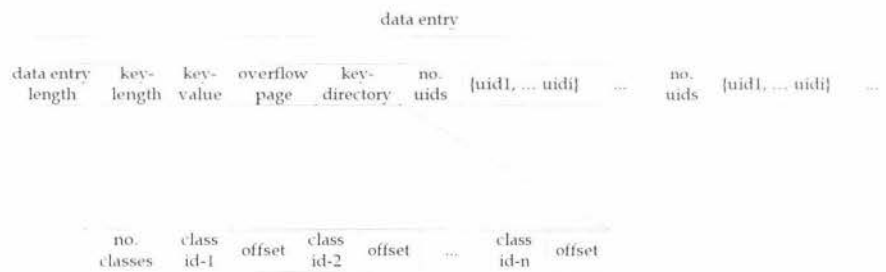


Figure 4.8 A Leaf Node of a CH Index (Kim, Kim et al., 1989, p.377)

Structure of a non-leaf node of a CH index is the same as structure of a non-leaf node of a B tree, and thus is also the same as a non-leaf node of a SC index presented in Figure 4.6 (Kim, Kim et al., 1989).

A leaf node of a CH index consists of data entries where each of them consists of the length of a data entry, a search key, a pointer to an overflow page, a key directory, and pairs of (the number of UIDs of data records that match a search key, and a collection of the UIDs) (Kim, Kim et al., 1989). A key directory consists of the number of classes whose objects matching a search key, pairs of (a class identifier, an offset). Each offset locates a corresponding pair of (the number of UID, a collection of UIDs). Figure 4.8 shows structure of a leaf node of a CH index.

The number of data entries contained in leaf nodes of a CH index also depends on the length of a search key and the number of records that match a search key (Kim, Kim et al., 1989). To adjust a CH index, the same approach used with a SC index is employed.

Consider Q2 (Retrieve all people who were born in 1975) of Example 4.1. Only one CH index that is maintained for physical objects that correspond to logical objects of classes PERSONC, PILOTC, MECHANICC and MARRIEPERSONS can support Q2 for both of the following interpretations:

- If Q2 is meant to retrieve logical objects of a class PERSONC whose year of birth is 1975 (including logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC), a CH index is accessed and UIDs of physical objects corresponding to logical objects, whose value of year of birth of an attribute DateOfBirth is 1975, of all classes in an inheritance hierarchy are retrieved.
- If Q2 is meant to retrieve logical objects of only a class PERSONC whose year of birth is 1975 (not including logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC), a CH index is accessed and UIDs of physical objects corresponding to logical objects, whose value of year of birth of an attribute DateOfBirth is 1975, of only a class PERSONC in an inheritance hierarchy are retrieved.

It is obviously seen that a leaf node of a CH index carries UIDs of physical objects that correspond to logical objects of all classes in an inheritance hierarchy. Hence, compared with a SC index that is presented in the last subsection, a CH index performs better to support queries made against logical objects of at least two classes in an inheritance hierarchy (Kim, Kim et al., 1989).

4.2.2.3 H-trees

H-trees are proposed by Low et al. (1992). An H tree is a modification of a SC index. The only difference between H-tree and SC index is that a non-leaf node of an H-tree contains an additional pointer called a linkage. In this approach, H-trees (modified SC indices) of a class and its subclasses are linked together by extra linkages in their non-leaf nodes. This is to facilitate simultaneous access between H-trees of a class and its subclasses. Figure 4.9 shows example H-trees of a superclass and a subclass.

To support queries that require associative access to logical objects of classes in an inheritance hierarchy, only full access is required for a superclass H-tree (Low et al., 1992). Linkages between non-leaf nodes of H-trees allow searching to jump from a superclass H-tree to subclass H-trees without beginning the search from the root node of subclass H-trees again. At the same time, one can also choose to perform independent search in each H-tree. Low et al. (1992) compare the performance of H-trees with SC

and CH indices. The result shows that H-trees achieve better retrieval speed compared with SC and CH indices.

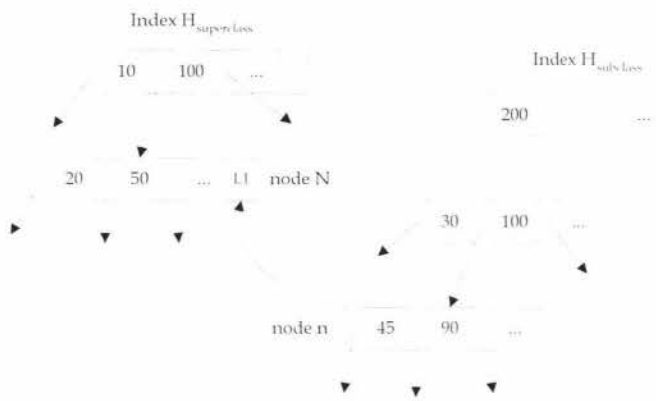


Figure 4.9 H-trees (Low et al., 1992, p.136)

Consider Q2 (Retrieve all people who were born in 1975) of Example 4.1.

- If Q2 is meant to retrieve logical objects of a class PERSONC whose value of year of birth of an attribute DateOfBirth is 1975 (including logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC), first an H tree that clusters physical objects corresponding to logical objects of only a class PERSONC is accessed. Once a non-leaf node that stores a value of year of birth that is 1975 is visited then the search is continued down the H tree of a class PERSONC to a leaf node that stores a value of year of birth that is 1975. In addition, the search is also continued via the linkages from the H tree of a class PERSONC to:
  - o An H tree that clusters physical objects corresponding to logical objects of a class PILOTC,
  - o An H tree that clusters physical objects corresponding to logical objects of a class MECHANICC, and
  - o An H tree that clusters physical objects corresponding to logical objects of a class MARRIEDPERSONC.
- If Q2 is meant to retrieve logical objects of only a class PERSONC whose year of birth is 1975 (not including logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC), only H tree that clusters physical objects corresponding to logical objects of only a class PERSONC is accessed.

4.2.2.4 A hierarchy class Chain (hcC) tree

An hcC tree is proposed by Sreenath & Seshadri (1994). It is an integration of SC and CH indices. An hcC-tree can also be viewed as a B+ tree with an additional level of nodes, called OID nodes, below a level of leaf nodes.

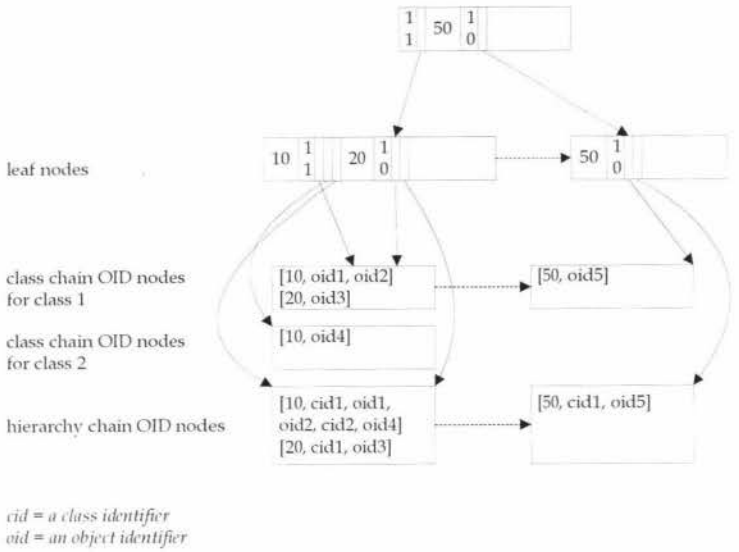


Figure 4.10 A hcC-tree (Sreenath & Seshadri, 1994, p.206)

The OID nodes of an hcC-tree are categorised into two types (Sreenath & Seshadri, 1994). The first type is called a class-chain OID node. It clusters OIDs of physical objects that correspond to logical objects of a class in an inheritance hierarchy on search key values. The function of a class-chain OID node is similar to a leaf node of a SC index, which is to support associative access to physical objects that correspond to logical objects of a class in an inheritance hierarchy. The second type of OID nodes is called a hierarchy-chain OID node. It clusters OIDs of physical objects that correspond to logical objects of all classes in an inheritance hierarchy on search key values. The function of a hierarchy-chain OID is similar to a leaf node of a CH index, which is to support associative access to physical objects that correspond to logical objects of all classes in a class-hierarchy. An example of an hcC-tree is presented in Figure 4.10.

Consider Q2 (Retrieve all people who were born in 1975) of Example 4.1. If Q2 is meant to retrieve logical objects of a class PERSONC whose value of year of birth of an

attribute DateOfBirth is 1975 (including logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC), a hierarchy-chain OID node, which contains a value of year of birth that is 1975 in an hcC-tree, is accessed to retrieve OIDs of physical objects corresponding to logical objects of classes PERSONC, PILOTC, MECHANICC and MARRIEDPERSONC. If Q2 is meant to retrieve logical objects of only a class PERSONC whose year of birth is 1975 (not including logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC), a class-chain OID node, which contains a value of year of birth that is 1975 in an hcC-tree, is accessed to retrieve OIDs of physical objects corresponding to logical objects of only a class PERSONC.

4.2.2.5 A Class-Division (CD) Index

A CD index is proposed by Ramaswamy & Kanellakis (1995). It consists of a CH index for physical objects that correspond to logical objects of all classes in an inheritance hierarchy, and a set of CH indices maintained for physical objects that correspond to logical objects of some classes in an inheritance hierarchy. The set of CH indices is intended to support access to physical objects corresponding to logical objects of some classes in an inheritance hierarchy, and speed up data access for range queries.

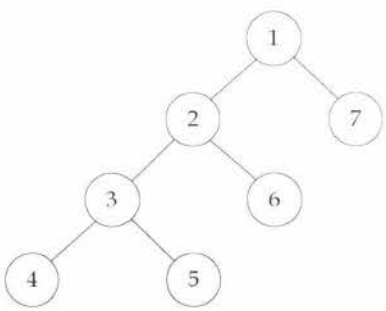


Figure 4.11 An Inheritance Hierarchy (Ramaswamy & Kanellakis, 1995, p.144)

Figure 4.11 presents a sample inheritance hierarchy. A circle represents a class in which its identifier is presented in the circle. A circle in a lower level is a subclass of circles in higher levels. Figure 4.12 presents a CD index created based on the inheritance

hierarchy in Figure 4.11. A circle represents a CH index. Bold circles represent created CH indices. Similar to Figure 4.11, identifiers of classes are presented in the circle.

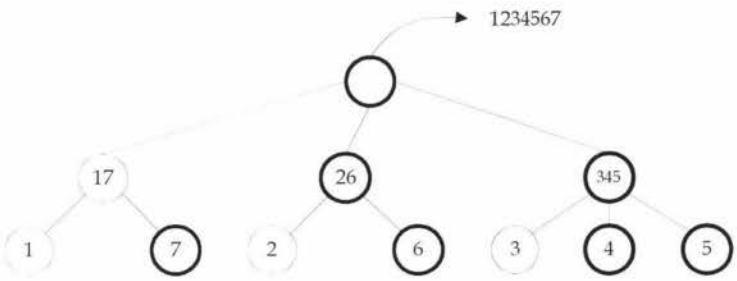


Figure 4.12 An Example CD Index (Ramaswamy & Kanellakis, 1995, p.144)

We can use a CD index to simultaneously support Q2 (Retrieve all people who were born in 1975) and Q3 (Retrieve mechanics and pilots who were born in 1975) of Example 4.1. The CD index that supports Q2 and Q3 consists of a CH index for physical objects corresponding to logical objects of all classes in an inheritance hierarchy, a CH index for physical objects corresponding to logical objects of only a class PERSONC (A CH index that is maintained for physical objects that correspond to logical objects of only one class is actually a SC index.), and a CH index for physical objects corresponding to logical objects of classes MECHANICC and PILOTC.

Consider Q2 (Retrieve all people who were born in 1975). If Q2 is meant to retrieve logical objects of a class PERSONC whose value of year of birth of an attribute DateOfBirth is 1975 (including logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC), a CH index maintained for physical objects corresponding to logical objects of all classes in an inheritance hierarchy is accessed. If Q2 is meant to retrieve logical objects of only a class PERSONC whose value of year of birth of an attribute DateOfBirth is 1975 (not including logical objects of classes PILOTC, MECHANICC and MARRIEDPERSONC), a CH index maintained for physical objects corresponding to logical objects of only a class PERSONC is accessed.

Consider Q3 (Retrieve mechanics and pilots who were born in 1975). If Q3 is meant to retrieve logical objects of classes MECHANICC and PILOTC whose value of year of

birth of an attribute DateOfBirth is 1975, only CH index for physical objects corresponding to logical objects of classes MECHANICC and PILOTC is employed.

Next, data access approaches that are proposed to support queries made against logical objects in a class-attribute hierarchy are described.

4.2.3 Approaches Supporting Queries Made against Logical Objects of Classes in a Class-Attribute Hierarchy

This subsection reviews data access approaches that support queries made against logical objects of classes in a class-attribute hierarchy, i.e. associative access to logical objects and navigational access to referenced or referencing logical objects of the logical objects being accessed. A class-attribute hierarchy is a directed graph presenting how attributes of classes reference other classes (Bertino & Kim, 1989). Figure 4.13 presents a class-attribute hierarchy of classes in an OODM presented in Appendix B.

Q4 (Retrieve details of all journeys of an airplane that is installed with an engine part whose serial number is Part001) of Example 4.1 can be taken as an example query that requires associative access to objects in a class-attribute hierarchy, if each involved logical object is mapped into one physical object as follows:

- A logical object of a class JOURNEYLOGC is mapped to one physical object that holds values of attributes defined in a class JOURNEYLOGC.
- A logical object of a class AIRPLANEC is mapped to one physical object that holds values of attributes defined in a class AIRPLANEC.
- A logical object of a class ENGINEC is mapped to one physical object that holds values of attributes defined in a class ENGINEC.
- A logical object of a class PARTC is mapped to one physical object that holds values of attributes defined in a class PARTC.

In addition, if a logical object of a class PARTC is mapped to several physical objects, i.e. each physical object holds a value of each attribute of the logical object, data access approaches reviewed in this section can be employed to support Q1 (Retrieve all parts manufactured in 1999.) of Example 4.1. This is because, in this case, navigational

access among the physical objects composed to be the corresponding logical objects of a class PARTC is required.

The approaches in this group are for example nested, path, multi indices (Bertino & Kim, 1989), and access support relations (Kemper & Moerkotte, 1990). A path index and access support relations are modifications of join indices.

The approaches in this group are built based on a path in a class-attribute hierarchy. A path is a subset of a class-attribute hierarchy. It is defined as C(1). A(1) . A(2) ... A (n) where n 1. n is the length of a path. C(1) is a class in a class-attribute hierarchy. A (1) is an attribute of C(1). A(i) is an attribute of C(i), where 1< i n and C(i) is referenced by A(i-1). Figure 4.14 shows example paths from a class-attribute hierarchy presented in Figure 4.13. In Figure 4.14, P1 supports Q4 of Example 4.1. Note that a path can also be presented in a reverse direction of references, for example paths P5 and P6 in Figure 4.14.

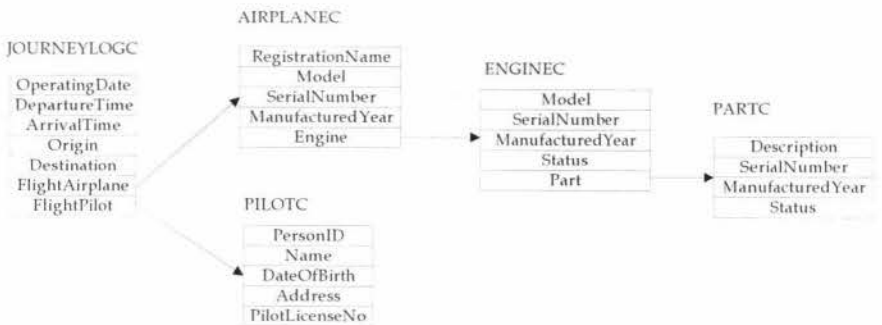


Figure 4.13 A Class-Attribute Hierarchy

P1: JOURNEYLOGC.FlightAirplane.Engine.SerialNumber
P2: JOURNEYLOGC.FlightAirplane.RegistrationName
P3: JOURNEYLOG.FlightPilot.PilotLicenceNumber
P4: JOURNEYLOGC.Origin
P5: ENGINEC.Engine.RegistrationName
P6: PARTC.Part.Engine.RegistrationName

Figure 4.14 Example Paths

Because a path index (Bertino & Kim, 1989) and access support relations (Kemper & Moerkotte, 1990) are modifications of join indices, this subsection will first introduce the join indices in order to establish basic knowledge for understanding. Subsequently, nested, path, multi indices (Bertino & Kim, 1989), and access support relations (Kemper & Moerkotte, 1990) are described.

4.2.3.1 Join Indices

A join index is introduced here as basic knowledge for understanding the data access approaches reviewed in subsection 4.2.3.

A join index is proposed by Valduriez (1987) to support associative access that involves a join operation of relations in a RDBS. The main concept is to create a set of join indices where each Join Index (JI) maintains pairs of surrogates[7] of data records of two relations that relate to each other.

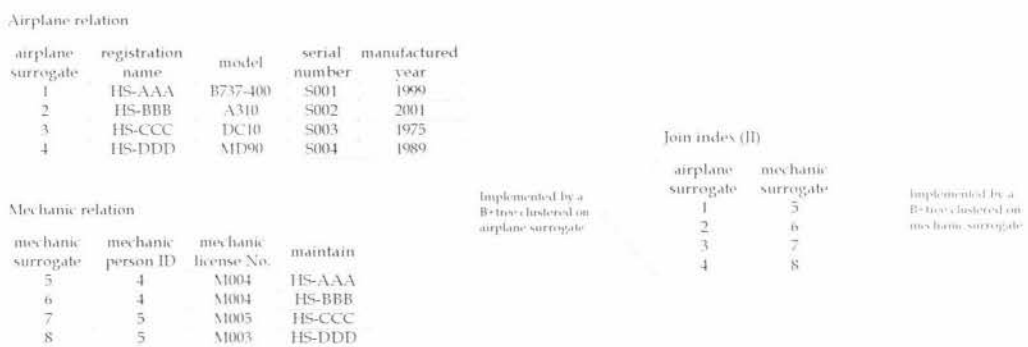


Figure 4.15 An Example of a Join Index (JI)

To support fast accessing to a join index via surrogates of data records of both relations, two copies of the same join index may be maintained where each join index is implemented by a B+ tree clustered on surrogates of data records of each relation (Valduriez, 1987). Figure 4.15 presents sample join indices supporting a join operation of two relations Airplane and Mechanic of a RDM, which is presented in Appendix A,

---

7 A surrogate is an immutable record identifier created by a RDBS.

on an attribute registration name. Two join indices are implemented by a B+ tree. One is clustered on surrogates of data records of a relation Airplane and the other is clustered on surrogates of data records of a relation Mechanic.

Next, nested, path, multi indices (Bertino & Kim, 1989), and access support relations (Kemper & Moerkotte, 1990) that are approaches proposed to support queries made against logical objects of classes in a class-attribute hierarchy are described.

4.2.3.2 A Nested Index

A nested index is proposed by Bertino & Kim (1989). It is a modification of a B tree. Non-leaf node and leaf node of a nested index are the same as those of a SC index presented in Figure 4.6 and 4.7, respectively. A nested index maintains an association between C(1) and A(n) in a path. In other words, a nested index clusters physical objects corresponding to logical objects of a class C(1) of a path on values of A(n).

A nested index that maintains an association between physical objects that correspond to logical objects of a class JOURNEYLOGC and values of an attribute SerialNumber defined in a class PARTC of a path P1 in Figure 4.14 can support Q4 (Retrieve details of all journeys of an airplane that is installed with an engine part whose serial number is Part001) in Example 4.1, if Q4 is meant to retrieve only logical objects of a class JOURNEYLOGS (not including their referenced logical objects in classes AIRPLANEC, ENGINEC and PARTC).

4.2.3.3 Multi Indices

Multi indices are also proposed by Bertino & Kim (1989). Multi indices consist of a set of nested indices where each nested index maintains an association between C(i) and A (i) in a path. In other words, each nested index in multi indices clusters physical objects that correspond to logical objects of a class C(i) in a path on values of an attribute A(i).

Consider Q4 (Retrieve details of all journeys of an airplane that is installed with an engine part whose serial number is Part001) in Example 4.1. Assume that Q4 is meant

to retrieve logical objects of a class JOURNEYLOGC, and also corresponding logical objects of class AIRPLANEC, ENGINEC and PARTC. Hence, the four following nested indices that are built based on a path P1 in Figure 4.14 are maintained to support Q4.

- A nested index that clusters physical objects corresponding to logical objects of a class JOUNEYLOGC on UIDs of physical objects corresponding to logical objects of a class AIRPLANEC,
- A nested index that clusters physical objects corresponding to logical objects of a class AIRPLANEC on UIDs of physical objects corresponding to logical objects of a class ENGINEC,
- A nested index that clusters physical objects corresponding to logical objects of a class ENGINEC on UIDs of physical objects corresponding to logical objects of a class PARTC, and
- A nested index that clusters physical objects corresponding to logical objects of a class PARTC on values of an attribute SerialNumber.

4.2.3.4 A Path Index

A path index is also proposed by Bertino & Kim (1989). Its non-leaf node is the same as that of a SC index presented in Figure 4.6.

A path index is a modification of a join index. It clusters physical objects that correspond to logical objects of all classes in a path on values of A(n). The difference between a path index and a join index is that in a path index UIDs of physical objects that correspond to logical objects of all classes in a path are maintained and clustered on values of A(n) while in a join index surrogates of related records of only two relations are maintained and clustered on one of the surrogates. Structure of a leaf node of a path index is presented in Figure 4.16.

Consider Q4 (Retrieve details of all journeys of an airplane that is installed with an engine part whose serial number is Part001) in Example 4.1. Assume that Q4 is meant to retrieve objects of a class JOURNEYLOGC, and also objects of classes AIRPLANEC, ENGINEC and PARTC. A path index that is built based on a path P1 in

Figure 4.14, and that clusters physical objects that correspond to logical objects of classes JOUNEYLOGC, AIRPLANEC, ENGINEC and PARTC on values of an attribute SerialNumber is maintained to support Q4.

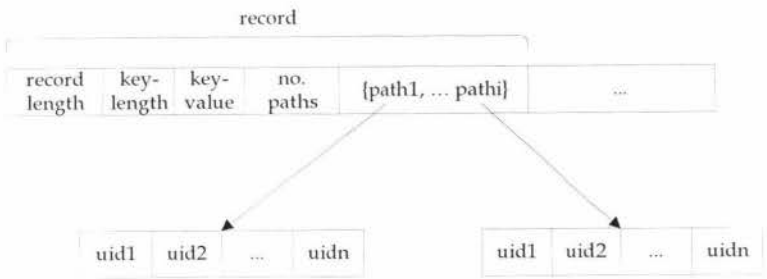


Figure 4.16 Structure of a Leaf Node of a Path Index

4.2.3.5 Access Support Relations

Access support relations are proposed by Kemper & Moerkotte (1990). They employ similar concepts as a path index proposed by Bertino & Kim (1989).

Figure 4.17 An Example Access Support Relation

In access support relations, a relation that structures tuples, which contain OIDs of all physical objects corresponding to logical objects of classes in a path starting from C(1) to C(n) and values of an attribute A(n) is created (Kemper & Moerkotte, 1990). Then two B+ trees are created, where one B+ tree clusters tuples of the relation on OIDs of physical objects that correspond to logical objects of a class C(1) and another B+ tree clusters tuples of the relation on values of an attribute A(n). Figure 4.17 presents an example of an access support relation created for Q4 (Retrieve details of all journeys of an airplane that is installed with an engine part whose serial number is Part001) in Example 4.1.

4.2.4 Approaches Supporting Queries Made against Logical Objects of Classes in Inheritance or Class-Attribute Hierarchies

Indices reviewed in sections 4.2.2 and 4.2.3 are constructed based on either a path or an inheritance hierarchy. However, a query may involve classes in several paths or inheritance hierarchies. To the best of our knowledge, the only approach that supports queries made against logical objects of classes that involves both path and inheritance hierarchies is a Nested-Inherited index[8] (NIX) proposed by Bertino (1991). However, a NIX is constructed based on only one path. Q5 (Retrieve mechanics were born in 1975 and maintaining a HS-AAA airplane) in Example 4.1 can be taken as an example query that is supported by a NIX because a class MECHANICC is a subclass of a class PERSONC and references a class AIRPLANEC.

A NIX consists of two indices called a primary index and an auxiliary index. Both of them are modifications of a B+ tree. Structure of a NIX is presented in Figure 4.18. A primary index of a NIX is built based on a path defined in a class-attribute hierarchy. It clusters physical objects that correspond to logical objects of all classes in a class-attribute hierarchy on search key values. Hence, a primary index of a NIX is similar to a CH index. A primary index of a NIX differs from a CH index in that a leaf node of the primary index contains extra pointers that point to leaf nodes of auxiliary indices. Each extra pointer is for physical objects that correspond to logical objects of a class in a path.

An auxiliary index of a NIX clusters 4-tuples on OID_i. OID_i is an OID of a physical object that occurs in a leaf node of the primary index. A 4-tuple is in a form (OID_i, a pointer that points to the corresponding leaf nodes of the primary index, the number of OIDs of objects that have an inheritance relationship with the OID_i, a list of OIDs of physical objects that have an inheritance reference with OID_i).

---

[8] Although a NIX is modified in Bertino & Foscoli (1995) to support for the case of multi value attribute, its core concept is still the same.

To retrieve a set of physical objects that match the search key, a primary index is first employed. If the physical objects being retrieved involve an inheritance hierarchy, an auxiliary index is employed.

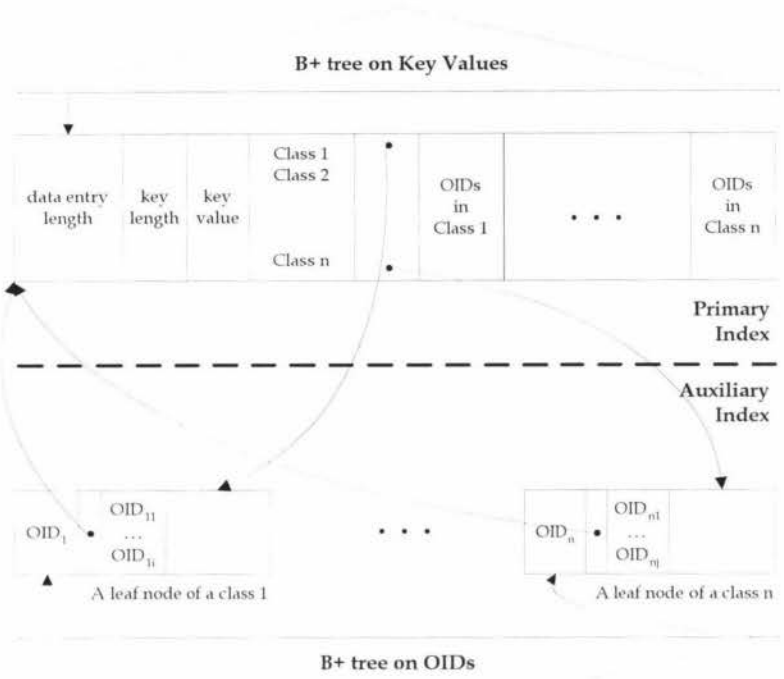


Figure 4.18 Structure of a NIX (Hua & Tripathy, 1994, p.510)

Section 4.2 reviews data access approaches that have been proposed to support associative access in an OODBS. The approaches reviewed include SC and CH indices (Kim, Kim et al., 1989), H-tree (Low et al., 1992), hcC-tree (Sreenath & Seshadri, 1994), CD index (Ramaswamy & Kanellakis, 1995), nested, path and multi indices (Bertino & Kim, 1989), access support relations (Kemper & Moerkotte, 1990), and a NIX (Bertino, 1991).

In the next section, approaches proposed to support navigational access in an OODBS are described.

4.3 Approaches Supporting Navigational Access in an OODBS

Navigational access is retrieval of a set of OIDs based on navigation along references between objects (Kirchberg & Tretiakov, 2002; Kuckelberg, 1998; Zezulan & Rabitti, 1993). Often, associative access in a OODBS involves navigational access as described in section 4.1. Hence, this section reviews data access approaches proposed to support navigational access in an OODBS. The approaches can be categorised, based on techniques used, into four groups as follows:

- Modifications of join indices,
- Object skeletons,
- Reference pointer approaches, and
- In-memory calculation approaches.

This section is separated into four subsections according to the categorised navigational data access approaches.

4.3.1 Modifications of Join Indices

This subsection reviews data access approaches that are modifications of join indices to support navigational access in an OODBS. Join indices[9] are proposed by Valduriez (1987) to support associative access that involves a join operation of two relations in a RDBS. The main concept is to create a set of join indices where each join index maintains pairs of surrogates of data records of two relations that relate to each other.

Modifications of join indices to support navigational access in an OODBS which are reviewed in this subsection are join index hierarchies (Han et al., 1999; Xie & Han, 1994) and triple-node hierarchies (Luk & Fu, 1998). Han et al. (1999), Xie & Han (1994) and Luk & Fu (1998) do not mention which level, i.e. physical or storage level, that join index hierarchies and triple-node hierarchies are based. However, as they both

---

[9] See details of join indices in subsection 4.2.3.1

are modifications of join indices (Valduriez, 1987) that support associative access in a RDBS, we make the same assumptions as when we review data access approaches that support associative access in an OODBS as the following.

- We assume that join index hierarchies (Han et al., 1999; Xie & Han, 1994) and triple-node hierarchies (Luk & Fu, 1998) are constructed based on the organisation of physical objects, data records stored in a magnetic disk.
- We also assume further that a logical object of a class is mapped to a physical object that hold values of attributes defined in the class.

In fact, MIC (Kuckelberg, 1998) is also considered a modification of join indices to support navigational access. However, because the MIC has a prominent characteristic that is it supports navigational access by in-memory calculation, it is categorised as an approach that employs an in-memory calculation technique and thus will be explained in subsection 4.3.4.

4.3.1.1 Join Index Hierarchies

Join index hierarchies are proposed by Han et al. (1999) and Xie & Han (1994) for the construction of an appropriate set of join indices to support navigational access, where each join index supports navigation between two classes.

In Han et al. (1999) and Xie & Han (1994), the database schema of a database is represented by a directed graph where each node represents a class and an edge between nodes represents a reference. An edge starts from a referencing class to a referenced class. References are categorised as direct and indirect references. A reference between two classes, in which an attribute of a class references another class, is regarded as a direct reference. Consider an example OODM presented in Appendix B, references between logical objects of classes MECHANICC and AIRPLANEC are direct references because a logical object of a class MECHANICC references logical objects of a class AIRPLANEC via an attribute Maintain. An indirect reference is a reference that can be derived from direct references. In an example OODM presented in Appendix B, references between logical objects of classes MECHANICC and ENGINEC are indirect references because these references are derived from direct references between

logical objects of classes MECHANICC and AIRPLANEC, and direct references between logical objects of classes AIRPLANEC and ENGINEC.

In Han et al. (1999) and Xie & Han (1994), a schema path is a directed graph, which is a subset of the database schema, consisting of n classes (C) and n-1 references. One class is regarded as the root of a schema path. A reference edge starts from the root to the next class, and so on. In a schema path, there is no reference edge from the last class to the root. A schema path can be expressed by $P = <(C_1), A_2(C_2), A_3(C_3), \ldots, A_n(C_n)>$. P is a schema path. C_1 is the root of a schema path. C_i is a class in a schema path where $1 < i \leq n$. A_i is an attribute of C_{i-1} that references C_i. Figure 4.19 a)-b) present examples of schema paths of the database schema of an example OODM presented in Appendix B.

As mentioned, join index hierarchies suggest ideas for building an appropriate set of join indices based on a schema path (Han et al., 1999; Xie & Han, 1994). Instead of maintaining a set of pairs of surrogates of data records of two relations like original join indices proposed by Valduriez (1987), each join index in join index hierarchies maintains a set of (OID_i, OID_j, m), where OID_i is an OID of a physical object corresponding to a referencing logical object, OID_j is an OID of a physical object corresponding to a referenced logical object, and m is the number of paths that link OID_i and OID_j together. To support forward navigation, a join index is implemented by a B+ tree clustered on OID_i. Similarly, to support backward navigation, a join index is implemented by a B+ tree clustered on OID_j.

A set of join indices developed in join index hierarchies can be categorised into base join index, complete join index, and partial join index hierarchies (Han et al., 1999; Xie & Han, 1994).

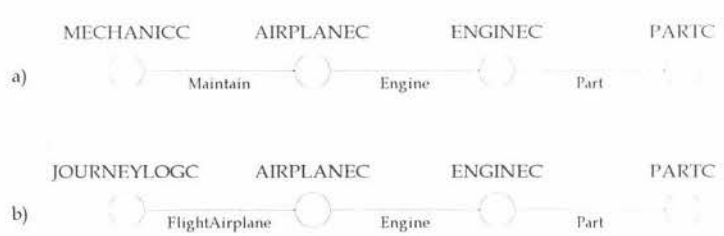


Figure 4.19 a)-b) Schema Paths

Figure 4.20 A Complete Join Index Hierarchy

In a given schema path, a base join index hierarchy consists of all possible join indices that are built from direct references (Han et al., 1999; Xie & Han, 1994). For instance, for a schema path in Figure 4.19a), a base join index hierarchy consists of three join indices including JI[10](MECHANICC, AIRPLANEC), JI(AIRPLANEC, ENGINEC) and JI(ENGINEC, PARTC).

A complete join index hierarchy consists of all possible join indices that are built from both direct and indirect references (Han et al., 1999; Xie & Han, 1994). Figure 4.20 presents an example of a complete join index hierarchy, based on a schema path in Figure 4.19a). A base join index hierarchy actually consists of join indices in a Level 1 in a complete join index hierarchy. Join indices in higher levels (starts from level 2) can be constructed by join indices in lower levels. Join indices that are used to construct join indices in a higher level are called auxiliary join indices.

Partial join indices consists of join indices in Level 1 (a base join index hierarchy) and necessary auxiliary join indices, which lead to target join indices (Han et al., 1999; Xie & Han, 1994). A target join index is a join index that supports required navigational access of end users. Figure 4.21 a) – b) show two possible partial join index hierarchies based on a schema part in Figure 4.19 a), where a target join index is assumed to be a join index maintained for classes MECHANICC and PARTC.

Most of the time, a partial join index hierarchy is implemented because normally users only want to retrieve objects based on references of some classes in a schema path (Han

---

[10] JI stands for Join Index.

et al., 1999; Xie & Han, 1994). In addition, it is expensive to maintain join indices for every pair of classes in a schema path. As for one target join index, there are various possible choices of partial index hierarchies, Han et al. (1999) and Xie & Han (1994) suggest the following to choose the most appropriate partial index hierarchy,

- The most appropriate partial index should contain the minimum number of auxiliary join indices.
- The most appropriate partial index should offer the most inexpensive cost in updating the target join index.

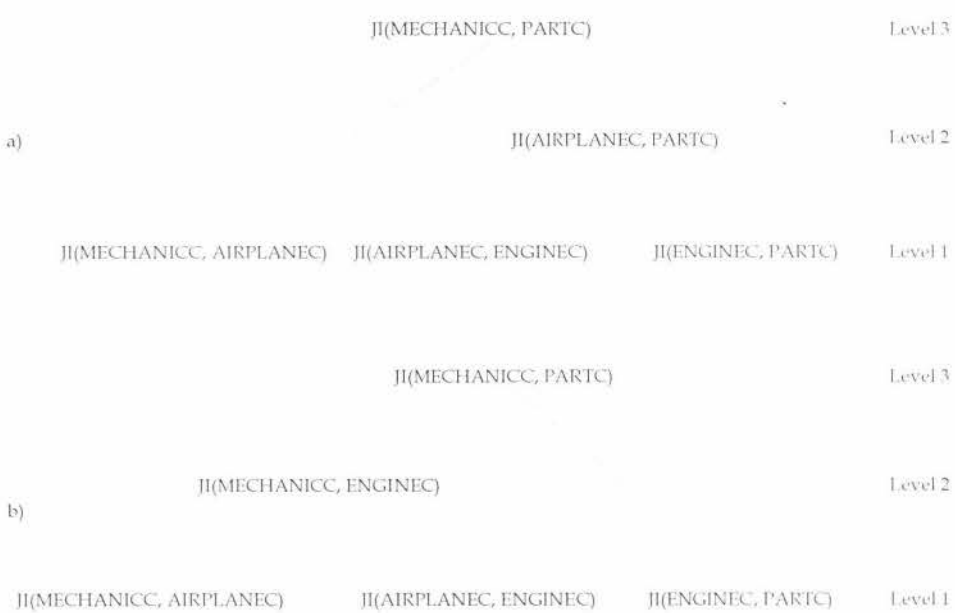


Figure 4.21 a) – b) Partial Join Index Hierarchies

4.3.1.2 Triple-Node Hierarchies

Triple-node hierarchies are proposed by Luk & Fu (1998) as improvement of join index hierarchies proposed by (Han et al., 1999; Xie & Han, 1994) in terms of update cost.

The core concepts of triple-node hierarchies and join index hierarchies are the same (Luk & Fu, 1998). The only difference is that, in triple-node hierarchies, a triple-node that maintains references between three classes is employed to reduce the cost of an update operation.

Given a schema path P = <(C_1), A_2(C_2), A_3(C_3), …, A_n(C_n)>, a triple-node is denoted by Tri(i, j, k) where in a path ($1 \leq i < j < k \leq n$) (Luk & Fu, 1998). Tri(i, j, k) is implemented by a B+ tree index that maintains a set of tuples <O_j, O_t, TYPE, m> clustered on O_i. O_i is a physical object corresponding to a logical object that belongs to a class C_i. O_t is a physical object corresponding to a logical object that belongs to either C_j or C_k distinguishing by a flag TYPE. Similar to join index hierarchies, m indicates the number of paths that link O_j and O_t together.

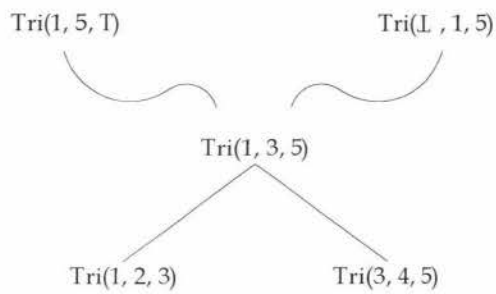


Figure 4.22 A Triple-Node Hierarchy Supporting Tri (1, 5, T) and Tri (⊥, 1, 5) (Luk & Fu, 1998, p.8)

Tri (⊥, j, k) is a triple node that maintains a set of tuples <O_j, O_k, m> clustered on O_j. O_j and O_k are objects that belong to classes C_j and C_k, respectively (Luk & Fu, 1998). Tri (⊥, j, k) supports forward navigation from C_j to C_k. Tri (j, k, T) is a triple node that maintains a set of tuples <O_j, O_k, m> clustered on O_k. ⊥ and T represents a non-existent class in a triple-node. Tri (j, k, T) supports backward navigation from a class C_k to a class C_j. Tri(i, j, k) is used for the purpose of an update operation. Figure 4.22 presents an example triple-node hierarchy supporting forward and backward navigation between classes C_1 and C_5.

4.3.2 Object Skeletons

Object skeletons are proposed by Hua & Tripathy (1994) to support navigational access by a network of OIDs.

In this approach, object skeletons, which are networks of OIDs used to support navigational access, are stored in magnetic disks (Hua & Tripathy, 1994). An existing indexing method, for example a CH index is needed to first obtain a starting object. It is shown that object skeletons are proposed to be implemented in a physical level. However, mapping logical objects to physical object is not explicitly described. Hence, we assume that a logical object of a class is mapped to a physical object that holds values of attributes defined in the class.

Figure 4.23 presents the framework of storage and navigation organisation in an object skeletons approach.

Figure 4.23 The Framework of Object Skeletons (Hua & Tripathy, 1994, p. 511)

For example, to support Q5 (Retrieve mechanics born in 1975 and maintaining a HS-AAA airplane) of Example 4.1, object skeletons need an index to first obtain OIDs of physical objects that correspond to logical objects of a class MECHANICC whose value of year of birth of an attribute DateOfBirth is 1975. Then an object skeleton involved with physical objects corresponding to logical objects of a class MECHANICC whose value of year of birth of an attribute DateOfBirth is 1975 is loaded into main memory to

aid navigation from physical objects that correspond to logical objects of a class MECHANICC whose value of year of birth of an attribute DateOfBirth is 1975 to a physical object that corresponds to a logical object of a class AIRPLANEC whose value of an attribute RegistrationName is HS-AAA.

Or, object skeletons need an index to first obtain an OID of a physical object that corresponds to a logical object of a class AIRPLANEC whose registration name is HS-AAA. Then an object skeleton involved with a physical object that corresponds to a logical object of a class AIRPLANEC whose value of an attribute RegistrationName is HS-AAA is loaded into main memory to aid navigation from a physical object that corresponds to a logical object of a class AIRPLANEC whose value of an attribute RegistrationName is HS-AAA to physical objects that correspond to logical objects of a class MECHANICC whose value of year of birth of an attribute DateOfBirth is 1975.

4.3.3 Reference Pointer Approaches

This section reviews data access approaches that support navigational access by using reference pointers.

The simplest approach to support forward navigational access is to append an object with a reference pointer indicating its referenced objects (Kuckelberg, 1998). Similarly, if backward navigation is required, a reference pointer that indicates a referencing object is needed to be appended with the object too. This approach requires first fetching an object into main memory in order to obtain a reference pointer that indicates referenced or referencing object of the object. In other words, in this approach, each step of navigation requires at least one read operation, if the object is not already in the main memory.

In LOQIS, a reference pointer approach is employed in a POS level to support navigational access (Subieta, 1994a, 1994b).

As mentioned in section 3.3 of chapter three, the model of a POS of LOQIS consists of atoms where each atom is a physical independent unit. An atom may store a record, an

attribute of an entity, text, number, graphic screen, a reference (pointer) to another atom, etc. Each atom has an OID. The reference pointer approach is adopted in two basic data structures called ring and spider. The ring and spider consist of a number of atoms where one atom is regarded as an owner and the rests are members.

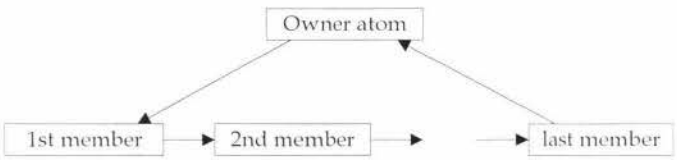


Figure 4.24 Ring Structure (Subieta, 1994b, p.12)

In the ring, an owner atom contains a reference pointer that points to the first member atom, and the first member atom contains a reference pointer that points to the next member, and so on (Subieta, 1994a, 1994b). The last member atom contains a reference pointer that points to the owner atom. The ring is used to support forward navigation between atoms in the ring. Figure 4.24 presents structure of the ring.

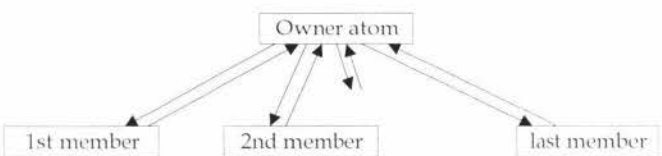


Figure 4.25 Spider Structure (Subieta, 1994b, p.12)

In the spider, an owner atom contains reference pointers that point to each member atom (Subieta, 1994a, 1994b). Every member atom of the spider contains a reference pointer that points to an owner atom. The spider is used to support both forward and backward navigation between an owner atom and member atoms. Figure 4.25 presents the structure of the spider.

An owner atom of the ring or spider may simultaneously be a member of other rings or spiders (Subieta, 1994a, 1994b). If an atom in the ring or spider is referenced by atoms in other rings or spiders, a link atom is inserted in the ring or spider in which a referencing atom is a member. A link atom is an atom that contains a reference from an

atom in the ring or spider to an atom in another ring or spider. In addition, a backward atom is inserted in the ring or spider after an atom referenced by atoms in other rings or spiders. A backward atom is an atom that contains a reference pointer that points to its corresponding link atoms. Figure 4.26 presents an example of link and backward atoms in the ring. Link and backward atoms are used to support both forward and backward navigation along many to many references between atoms in different rings or spiders.

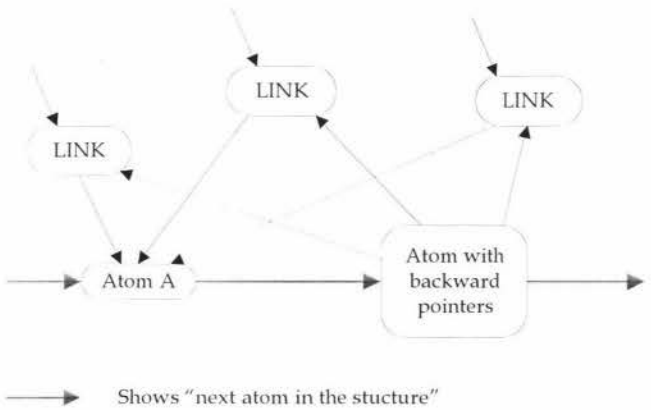


Figure 4.26 Organisation of Backward Pointers (Subieta, 1994b, p.12)

4.3.4 In-memory Calculation Approaches

This subsection reviews data access approaches that use in-memory calculation techniques to support navigational access in an OODBS. The approaches in this group are navigation index (Zezulan & Rabitti, 1993) and MIC (Kuckelberg, 1998). These two approaches employ similar ideas in appending an OID with a code that can be expanded to a set of referenced or referencing OIDs of the object being accessed. They are constructed based on the organisation of storage objects in a POS. Models of a POS in Zezulan & Rabitti (1993) and Kuckelberg (1998) are similar but they are different from a model of a POS in Subieta (1994a; 1994b).

For simplicity of understanding, this subsection first briefly describes the model of a POS that is employed by Zezulan & Rabitti (1993) and Kuckelberg (1998). Then navigation index and MIC are described.

4.3.4.1 A model of a POS employed by Navigation Index and MIC

This subsection describes a model of a POS that is employed by navigation index and MIC approaches to establish basic knowledge.

It is defined that a POS consists of a set of storage objects and a set of references between storage objects (Zezulan & Rabitti, 1993). A storage object is in a form of (OID, OC, OS, OV), where OID is an object identifier of a storage object, OC is a storage object characteristic, i.e. type of persistent, object dependent, sharability, etc., OS is an object structure, and OV is an object value. A reference between storage object is in a form of (RN, O_p, O_q), where O_p is a referencing storage object, O_q is a referenced storage object, and RN is a reference name. RN needs not be unique, however two references from O_p to O_q cannot have the same name.

A storage object is called an entry storage object if it references storage objects but is not referenced by any storage object (Zezulan & Rabitti, 1993). An end storage object is a storage object that does not reference any storage object but is referenced by storage objects. A shared storage object is a storage object that is referenced by more than one storage object. A branching storage object is a storage object that references more than one storage object. A storage object is an isolated storage object if it does not reference any storage object and is not referenced by any storage object.

In a POS, storage objects are not clustered by any criteria, i.e. classes, values of attributes, etc. References are not distinguished by any criteria i.e. types of references, reference names etc. (Zezulan & Rabitti, 1993). In addition, there is no restriction on how logical objects are mapped to storage objects.
Next, navigation index and MIC are described.

4.3.4.2 A Navigation Index

A navigation index is proposed to support navigational access, in particular for backward navigation, in a POS of an OODBS (Zezulan & Rabitti, 1993). However, with the same principles, a navigation index can also support forward navigation.

A navigation index employs two data structures called expander and linker (Zezulan & Rabitti, 1993). An expander is a set of (OID, SICF code) pairs. A SICF code is a code calculated by a fixed-length coding technique called Simple Continued Fractions (SICF), based on a reference graph. A SICF code of an object can be expanded to a set of codes of objects referencing it. A reference graph is a directed graph that presents objects and references between objects. In a reference graph, a node represents an OID of a storage object. An edge between nodes represents a reference between storage objects. An edge begins from a referencing storage object to a referenced storage object. A reference graph can be decomposed into trees. An isolated, entry or shared object is the root of a tree. A tree does not contain other trees. Figure 4.27 presents an example reference graph. Dot circles represent trees in a reference graph. A linker is a set of pairs of SICF codes of the root object of a tree and objects referencing it.

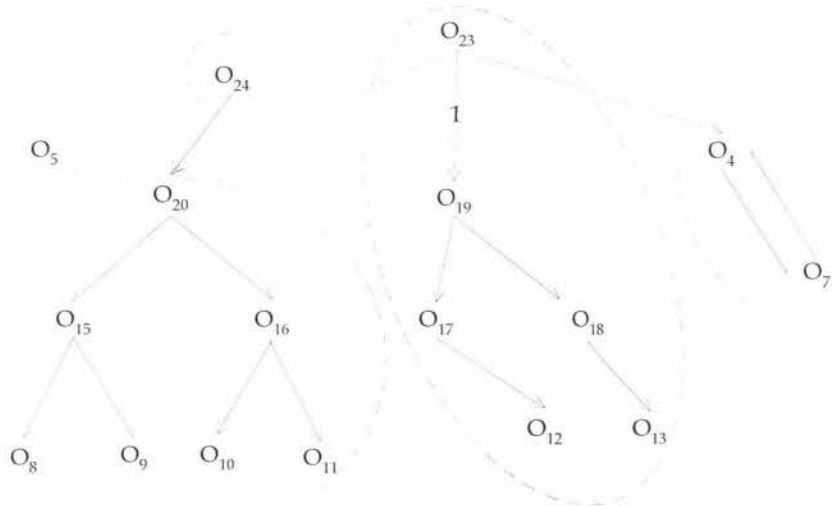


Figure 4.27 An Example Reference Graph

A SICF code is expressed in the form of $1/\{q_1 + 1/ [q_2 +\ldots+ 1/(q_{m-1} + 1/q_m) \ldots]\}$, where m is a finite number and q_i ($1 \quad i \quad m$) is an Integer that is greater than zero and q_m is an Integer that is greater than one. A SICF code can also be represented by $[q_1, q_2, \ldots, q_{m-1}, q_m]$ which is an expansion of exactly one rational number (N/D) (Zezulan & Rabitti, 1993).

$$N_i/D_i = [q_1, q_2, \ldots, q_{m-1}, q_m]$$

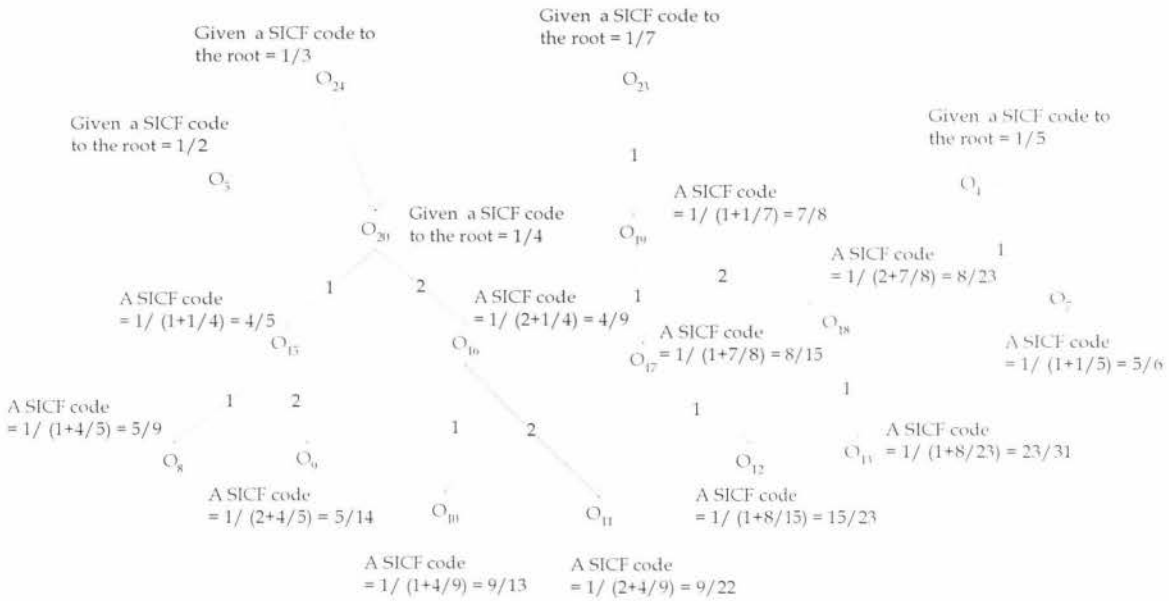


Figure 4.28 Calculation of SICF Codes

The following equations hold for a SICF code.

$$N_1/D_1 = N/D,$$

$$N_m/D_m = 1/q_m,$$

$$N_i/D_i = 1/ (q_i + N_{i+1}/D_{i+1}),$$

$$D_{i+1} = N_i,$$

$$q_i = D_i \text{ div } N_i,$$

$$N_{i+1} = D_i \text{ mod } N_i$$

The following are processes to calculate a SICF code of an object.

- Assign a SICF code to the root of a tree = 1/S, where S > 1. A SICF code of the root must be unique among trees.
- Assign a number to a reference branch beginning from left to right. A reference that points to the root of a tree is not assigned a branch number.
- Calculate a SICF code of an object by the following equation.

$$\text{A SICF code of an object } (C_i) = \frac{1}{\text{branch sequence number} + \text{a SCIF code of a referencing object}}$$

Figure 4.28 demonstrates the calculation of SICF codes for storage objects in a reference graph presented in Figure 4.27. Figure 4.29 presents expander and linker tables for a reference graph presented in Figure 4.27.

Example 4.2 demonstrates using a navigation index presented in Figure 4.29 to support backward navigation from O_8.

Example 4.2 Retrieve OIDs of all storage objects that reference a storage object O_8.

1. Access an expander table and retrieve a SICF code of a storage object O_8 which is = 5/9.
2. Expand a SICF code of a storage object O_8

$$N_1/D_1 = \text{a SICF code of a storage object } O_8 = 5/9$$

$$D_2 = N_1 = 5$$

$$N_2 = D_1 \bmod N_1 = 9 \bmod 5 = 4$$

$$N_2/D_2 = 4/5$$

$$D_3 = N_2 = 4$$

$$N_3 = D_2 \bmod N_2 = 5 \bmod 4 = 1$$

   We stop when N = 1 because it implies that we reach the root of a tree. We get Expand 5/9 = {4/5, 1/4}
3. Access a linker table to check if there is any storage object that references a storage object whose SICF code = 1/4. We found that two storage objects whose SICF codes are 1/3 and 1/7 reference a storage object whose SICF code = 1/4.
4. Make a union of found codes. We get {5/9, 4/5, 1/4, 1/3, 1/7}.
5. Access Expander to find OIDs of storage objects from SICF codes. We get {O_8, O_{15}, O_{20}, O_{24}, O_{23}}.

Example 4.2 shows that it is necessary to access an expander table twice, one (step 1) is to find SICF codes for given OIDs of storage objects and another one (step 5) is to find OIDs of storage objects for SICF codes. To facilitate this need, two copies of an expander table may be maintained. One is clustered on OIDs of storage objects and another one is clustered on SICF codes.

As mentioned, a navigation index can also be used to support forward navigation (Zezulan & Rabitti, 1993). To do that, we need to first invert directions of references in a reference graph and then employ the same principles as for the backward navigation.

Expander

OID	SICF Codes
O_4	1/5
O_5	1/6
O_7	5/6
O_8	5/14
O_9	9/13
O_{10}	9/13
O_{11}	9/22
O_{12}	15/23
O_{13}	23/31
O_{15}	4/5
O_{16}	4/9
O_{17}	8/15
O_{18}	8/23
O_{19}	7/8
O_{20}	1/4
O_{23}	1/7
O_{24}	1/3

Linker

SICF Code	SICF code
1/4	1/3
1/4	1/7
1/5	5/6
1/5	1/7
1/7	1/5

Figure 4.29 Example Expander and Linker Tables

4.3.4.3 Matrix-Index Coding (MIC)

Matrix-Index Coding (MIC) is proposed by Kuckelberg (1998) to support navigational access in a POS. MIC works like a join index but it eliminates the need to simultaneously access to obtain OIDs of referenced or referencing objects of the object being accessed. MIC employs a similar idea as a navigation index (Zezulan & Rabitti, 1993) in appending an OID with SICF codes that can be expanded to OIDs of referenced or referencing storage objects. However, the concepts of MIC are much simpler than a navigation index. MIC differs from a navigation index in the following ways:

- MIC appends two SICF codes with an OID of a storage object, one for forward navigation and one for backward navigation.
- In MIC, a SICF code is calculated straightaway from OIDs of storage objects while in a navigation index, a SICF code is calculated from SICF codes of other storage objects.
- MIC uses a matrix to represent references between storage objects while a navigation index uses a reference graph.

In a reference matrix, only two values, which are 0 and 1 appear. Each row and column of a matrix is assigned to a storage object. A row represents referenced storage objects of its assigned storage object. For example, in the reference matrix presented in Figure 4.30, in an O_4 row, all values are 0 except at a column O_7. This means that a storage object O_4 references only a storage object O_7. A column presents referencing storage objects of its assigned storage object. For example, in an O_4 column all values are 0, except at row O_7 and O_{23}. This means that a storage object O_4 is referenced by storage objects O_7 and O_{23}.

	O_4	O_5	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}	O_{13}	O_{15}	O_{16}	O_{17}	O_{18}	O_{19}	O_{20}	O_{23}	O_{24}
O_4	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O_5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O_7	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O_8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O_9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O_{10}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O_{11}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O_{12}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O_{13}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O_{15}	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
O_{16}	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
O_{17}	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
O_{18}	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
O_{19}	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
O_{20}	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
O_{23}	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
O_{24}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Figure 4.30 A Reference Matrix

The following are rules to compute SICF codes for a storage object by MIC, which also shows how MIC overcomes the join indices by eliminating simultaneous access to obtain OIDs of referenced or referencing objects of the object being accessed.

- If a storage object does not reference any storage object, a SICF code for forward navigation is 0.
- If a storage object references only one storage object (single outgoing reference) and that storage object also references only one storage object and so on, a SICF code for forward navigation is calculated from OIDs of storage objects that can be reached by single outgoing references only. A numerator and a denominator are then exchanged to indicate that this code is calculated from single outgoing references. Each time a code for forward navigation of a storage object, which is calculated from several steps of

single outgoing references, is expanded, we get all OIDs of storage objects that are referenced by single outgoing references beginning from the storage object. For example, in Figure 4.30, a storage object O_4 references a storage object O_7 and a storage object O_7 references a storage object O_4. A SICF code for forward navigation of a storage object O_4 is 29/4. When a SICF code for forward navigation of a storage object O_4 is expanded, we get O_7 and O_4 which are OIDs of storage objects that are referenced by single outgoing references beginning from the storage object O_4.

- If a storage object references many storage objects (multiple outgoing reference), a SICF code for forward navigation is calculated from OIDs of storage objects being referenced. Each time a code for forward navigation of a storage object, which is calculated from multiple outgoing references, is expanded, we get all OIDs of storage objects that are referenced by the storage object. For example, in Figure 4.29, a storage object O_{19} references storage objects O_{17} and O_{18}. A SICF code for forward navigation of a storage object O_{19} is 18/307. When a SICF code for forward navigation of a storage object O_{19} is expanded, we get O_{17} and O_{18}which are OIDs of storage objects that are referenced by the storage object O_{19}.

OID	Forward SICF codes	Backward SICF codes
O_4	29/4	29/4
O_5	0	0
O_7	29/7	29/7
O_8	0	301/20
O_9	0	301/20
O_{10}	0	321/20
O_{11}	0	321/20
O_{12}	0	7469/438
O_{13}	0	7907/438
O_{15}	9/73	1/20
O_{16}	11/111	1/20
O_{17}	1/12	438/23
O_{18}	1/13	438/23
O_{19}	18/307	1/23
O_{20}	16/241	24/553
O_{23}	20/381	0
O_{24}	1/20	0

Figure 4.31 SICF Codes Calculated by MIC

The algorithm of calculation SICF code for backward navigation is the same as forward navigation. Figure 4.31 presents OIDs of storage objects and their SICF codes calculated from references presented in a reference matrix in Figure 4.30.

4.4 Summary

This chapter reviews data access approaches that support associative and navigational access in an OODBS.

There have been a number of data access approaches proposed to support associative access in an OODBS. The data access approaches supporting associative access in an OODBS reviewed include SC and CH indices (Kim, Kim et al., 1989), H-tree (Low et al., 1992), hcC-tree (Sreenath & Seshadri, 1994), CD index (Ramaswamy & Kanellakis, 1995), nested, path and multi indices (Bertino & Kim, 1989), access support relations (Kemper & Moerkotte, 1990), and a NIX (Bertino, 1991).

- The approaches can be categorised into the following.
  - o Approaches that support queries that require only associative access,
  - o Approaches that support queries that are made against logical objects of classes in an inheritance hierarchy, i.e. queries that are made against logical objects that belong to many classes,
  - o Approaches that support queries that are made against logical objects of classes in a class-attribute hierarchy, i.e. queries that are made against logical objects and their referenced or referencing logical objects, and
  - o Approaches that support queries that are made against logical objects of classes in inheritance or class-attribute hierarchies.
- These approaches are modifications of approaches used in a RDBS, where data is regarded as related tuples, such as B tree family indices and join indices.
- In these approaches, a class is regarded as a relation and a logical object is regarded as a tuple over a relation. A UID, which is a system-defined unique identifier for a tuple, is regarded as an OID of a logical object. Details of mapping between logical objects to storage objects or physical objects are not given. However, because these approaches are modifications of approaches used in a RDBS, we assume that these approaches are constructed based on the organisation of physical objects, data records stored in a magnetic disk.
- These approaches are built based on either a path or an inheritance hierarchy. However, a query may involve classes in several paths or inheritance hierarchies. To the best of

our knowledge, a NIX proposed by Bertino (1991) is an only approach that is proposed to support queries made against logical objects of classes in both path and inheritance hierarchies. Unfortunately, a NIX can only support well when a query is made against logical objects of classes that simultaneously occur in a path and inheritance hierarchies because a NIX is constructed based on only one path (Hua & Tripathy, 1994).

To support queries that require associative access and involve multiple paths or inheritance hierarchies by these existing data access approaches, a number of indices are required (Hua & Tripathy, 1994). This implies storage redundancy that may occur, which is important especially if these indices are to be stored in main memory. In addition, a number of indices involved may lead to an increasing number of I/O operations needed and the time spent during lookups for requested objects by the indices.

The data access approaches proposed to support navigational access in an OODBS reviewed in this chapter could be categorised based on types of techniques used as follows.

- Modification of join indices

  For data access approaches that are modifications of join indices, their disadvantages are similar to those of data access approaches that are proposed to support associative access to objects in a class-attribute hierarchy because they all are built based on paths. In the database schema, a class may reference or be referenced by many classes while one join index in data access approaches that are modifications of join indices can maintain references between only two or three classes. As a result, these data access approaches may cause storage redundancy, and lead to the increasing number of I/O operations needed and the time spent during lookups for requested objects by the indices too.

- Object skeletons

  For object skeletons, an index that supports associative access in an OODBS is also required before loading the corresponding skeleton into main memory. This approach may result in the same problems as of data access approaches that are modifications of join indices.

- Reference pointer approaches

  For data access approaches that employ a reference pointer technique, an important disadvantage is that every step of navigation requires at least one read operation to

bring an object into main memory (in case that the object is not already in main memory) to obtain a reference pointer that indicates a referenced or referencing object of the object being accessed.

- In-memory calculation approaches.

  Data access approaches that employ in-memory calculation techniques, like navigation index and MIC, have shown their prominent advantage. They efficiently obtain a set of OIDs from navigational access by just a calculation in main-memory regardless of classes or types of references involved. This implies the less time spent during a data retrieval operation. In addition, Kirchberg & Tretiakov (2002) suggest that the original concepts of the navigation index and MIC can be generalised by making them independent from coding techniques used, i.e. any appropriate coding technique can be selected to guarantee optimal performance in different circumstances. Especially, data compression techniques should be considered as alternative coding techniques because they can reduce the storage size of the navigation index and MIC.

As mentioned, the MIC employs simpler concepts compared with the navigation index to support navigational access by an in-memory calculation in a POS of an OODBS. Hence, the MIC is an appropriate approach to be implemented in the extended POS in order to (1) demonstrate that a model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS and (2) show that the MIC that is implemented with the extended POS can support associative access in an OODBS regardless of paths or inheritance hierarchies involved.

In addition, the thesis aims to provide proof of the concepts that (1) the MIC can be made independent from a coding technique and (2) data compression techniques should be considered appropriate alternatives to implement the MIC because they can reduce the storage size. Therefore, in the next chapter, data compression techniques are reviewed. A Start/Stop data compression technique (Pigeon, 2001) which is one of coding techniques used to implement the MIC in the thesis is also described.

CHAPTER 5

DATA COMPRESSION TECHNIQUES

One of the thesis objectives is to provide proof of the concepts that (1) MIC can be made independent from a coding technique and (2) data compression techniques should be considered appropriate alternatives to implement the MIC because they can reduce the storage size of the MIC. Therefore, this chapter reviews data compression techniques.

The chapter is separated into four sections.

- The first section describes and outlines important fundamentals of data compression techniques.
- The second section describes some common measures of data compression.
- The third section presents types of data compression techniques. Also, some example techniques for each type are briefly explained.
- The last section briefly describes a Start/Stop data compression technique (Pigeon, 2001) that is used to implement the MIC in thesis.

5.1 Fundamentals of Data Compression

This section outlines important fundamentals of data compression. Data compression is transformation of source messages that are some representations of data into codewords that are another representations of data, where the codewords contain the same information as the source messages but have smaller size (Lelewer & Hirschberg, 1987).

Data compression plays important roles in areas of data transmission and data storage (Lelewer & Hirschberg, 1987). It reduces transmission cost and increases the amount of data that can be stored in magnetic disks. In an area of database systems, data compression may also help by reducing the number of read or write operations required during transferring data between main memory and magnetic disks.

Lelewer & Hirschberg (1987) describe data compression as a branch of information theory where the primary objective is to minimise the size of data to be transmitted. Information theory is defined as, "the study of efficient coding and its consequences in probability of error" (Ingels, 1971 cited in Lelewer & Hirschberg, 1987, p.261) where coding is defined as, "a general term encompassing any special representation of data that satisfies a given need"(Lelewer & Hirschberg, 1987, p.261).

The term code refers to a mapping between source messages and codewords (Lelewer & Hirschberg, 1987). Codes can be categorised into block-block, block-variable, variable-block and variable-variable mappings. The term block indicates that the size of source messages or codewords is fixed. In contrast, the term variable indicates that the size of source messages or codewords is changeable. For example, a block-block code maps fixed-size source messages into fixed-size codewords and a block-variable code maps fixed-size source messages into variable-size codewords.

Codes can also be categorised into static, dynamic (adaptive) and hybrid (Lelewer & Hirschberg, 1987). Static codes means that a mapping between source messages and codewords is previously defined and fixed. The same source message is always transformed into the same codeword. In contrast, dynamic codes means that a mapping between messages and codewords is changeable based on some criteria such as the probabilities of occurrences of the same source messages. Hybrid codes refer to codes that are neither entirely static nor entirely dynamic.

The term coding or encoding refers to transformation of source messages into codewords (Lelewer & Hirschberg, 1987). The term decoding refers to transformation of codewords back to source messages.

5.2 Common Measures of Data Compression

This section describes some common measures of data compression including redundancy (Shannon & Weaver, 1949 cited in Lelewer & Hirschberg, 1987), average message length (Huffman, 1952 cited in Lelewer & Hirschberg, 1987), and compression ratio (Rubin, 1976 and Ruth & Kreutzer, 1972 cited in Lelewer & Hirschberg, 1987).

5.2.1 Redundancy

The purpose of data compression is to minimise redundancy, and retain only the information content (Lelewer & Hirschberg, 1987). Shanon & Weaver (1949 cited in Lelewer & Hirschberg, 1987) define that entropy (H), which is the average information over the content of each alphabet (a_i) in the source messages, can be calculated by $\sum_{i=1}^{n}[-p(a_i)\log_2 p(a_i)]$, where $p(a_i)$ is the probability of the occurrence of a_i in the source messages. It is also defined that the minimum total number of bits required for the codewords, which adequately carry the information content of the source messages, is the product of entropy (H) and the length of the source messages.

The redundancy of data compression is defined as the difference between average codeword length and the average information content of the source message (H) (Lelewer & Hirschberg, 1987). Thus, the redundancy of data compression is computed by $\sum p(a_i)l_i - \sum_{i=1}^{n}[-p(a_i)\log_2 p(a_i)]$, where l_i is the length of a codeword representing an alphabet a_i. The codes are said to be optimal if they have minimum redundancy.

5.2.2 Average Message Length

As presented, the redundancy of data compression is the average codeword length minus the entropy (H) (Lelewer & Hirschberg, 1987). For a given probability distribution, while the entropy (H) is constant, the average message length of codewords can be also used to measure data compression (Huffman, 1952 cited in Lelewer & Hirschberg, 1987). The small average codeword length means small redundancy.

The code is called asymptotically optimal if, for a given probability distribution, the ratio of average codeword length to entropy (H) approaches 1 when the entropy (H) tends to infinity (Lelewer & Hirschberg, 1987). The asymptotically optimal code guarantees that the average codeword length approaches theoretical minimum because, as described, the minimum total number of bits required for the codewords is the product of the entropy (H) and the length of the source messages.

5.2.3 Compression Ratio

There are a number of ways to define compression ratio (C). Cappellini (1985 cited in Lelewer & Hirschberg, 1987) defines that a compression ratio (C) can be computed by (average source message length)/(average codeword length).

Rubin (1976 cited in Lelewer & Hirschberg, 1987) gives a definition of compression ratio (C) that it can be computed by $\frac{(S-O-OR)}{S}$, where S is the length of source messages, O is the length of codewords, and OR is the number of bits required for the encoder to transmit the code mapping to the decoder.

5.3 Data Compression Techniques

This section reviews types of data compression techniques. Examples of data compression techniques of each type are given.

Broadly, data compression techniques can be categorised into semantic-dependent and general-purpose techniques (Lelewer & Hirschberg, 1987). They are described in the following two subsections, respectively.

5.3.1 Semantic-Dependent Data Compression Techniques

Semantic-dependent data compression techniques make use of the context and semantics of data to achieve redundancy reduction (Lelewer & Hirschberg, 1987). They are designed for specific types of local redundancies taking place in particular applications.

Semantic-dependent data compression techniques are essential in areas of image representation and processing (Lelewer & Hirschberg, 1987). This is because images contain a large amount of redundancy. Also, images usually occupy a large number of bits. Examples of semantic-dependent data compression techniques used for compression of image data are run-length encoding (Gonzalez & Wintz, 1977 cited in Lelewer & Hirschberg, 1987) and difference mapping. A run-length encoding technique maps the sequence of image elements along a scan line (row) $x_1, x_2, \ldots, x_n$ to a sequence of pairs $(c_1, l_1), (c_2, l_2), \ldots, (c_k, l_k)$, where c_i is an intensity or colour and l_i is the length of the i^{th} run. A difference mapping technique maps an image to an array of differences in brightness (or colour) between adjacent pixels.

Semantic-dependent data compression techniques are also used in areas of business data processing (Lelewer & Hirschberg, 1987). Redundancies of data manipulated in business applications are for instance sequences of zeros in numeric fields and sequences of blanks in alphanumeric fields. Sequences of zeros or blanks can be compressed by a run-length encoding technique. Dictionary substitution is also another example of a semantic-dependent data compression technique used in business applications (Reghbati, 1981 cited in Lelewer & Hirschberg, 1987).

5.3.2 General-Purpose Data Compression Techniques

General-purpose data compression techniques are opposite to semantic-dependent data compression techniques (Lelewer & Hirschberg, 1987). They do not exploit the context and semantics of data for the purpose of data compression. General-purpose data compression techniques can be categorised into static defined-words schemes and dynamic defined-words schemes.

As described, static codes means that a mapping between source messages and codewords is previously defined and fixed while dynamic codes means that a mapping between messages and codewords is variable based on some criteria (Lelewer & Hirschberg, 1987).

5.3.2.1 Static Defined-Words Schemes

This subsection reviews some data compression techniques that are static defined-words schemes including static Huffman coding, Shannon-Fano, universal codes, and arithmetic coding techniques. Huffman and Shannon-Fano coding techniques are variable-variable codes and they include block-variable codes as their special case (Lelewer & Hirschberg, 1987).

- A static Huffman coding technique, which was presented in Huffman's paper (1952) on minimum redundancy coding, is considered the classic defined-words scheme (Lelewer & Hirschberg, 1987). Under the constraints that each source message is mapped into a unique codeword and that the output of data compression is the concatenation of codewords for the source messages, a static Huffman coding technique compresses the source messages with the best possible compression ratio (C). A static Huffman coding technique first determines the weights (w_i), which are the probabilities of occurrences of letters or alphabets in source messages. Then a binary tree is constructed. Each leaf of a binary tree represents a letter or an alphabet (a_i) in source messages, and is labelled with the corresponding weight. A leaf labelled with a higher weight stays closer to the root of the tree. The codeword for each letter is the sequence of labels of leaves along a path beginning from the root of the binary tree to a leaf that represents that letter.
- A Shannon-Fano technique employs simple concepts (Lelewer & Hirschberg, 1987). It first lists the source messages, a_i, with their probabilities of occurrences, p (a_i) in order of non-increasing probability. Then the list is divided into two groups in such a way that the sums of the probabilities of these two groups are almost equal as possible. The codewords of source messages in the first group are assigned to begin with zero and the codewords of source messages in the second group are assigned to begin with one. Each of these groups is subsequently divided into two

groups with the same criterion, and so on. The process ends when there is only one source message left in a group.

- Universal Codes are codes that map source messages to codewords whose average length bound by c_1H+c_2, where c_1 and c_2 are constants (Lelewer & Hirschberg, 1987). If $c_1 = 1$, the universal code is asymptotically optimal. Universal codes do not require prior knowledge about the probabilities of occurrences of source messages (Lelewer & Hirschberg, 1987; Verdú, 1998), which is an advantage of universal codes over the Huffman and Shannon-Fano coding techniques. Examples of universal codes are an Elias code (Elias, 1975 cited in Lelewer & Hirschberg, 1987) and a Fibonacci code (Apostolico & Fraenkel, 1985 cited in Lelewer & Hirschberg, 1987).
  - o There are two Elias coding techniques (Elias, 1975 cited in Lelewer & Hirschberg, 1987). The first Elias code maps an Integer, x to codewords consisting two parts. The first part contains $\lfloor \log_2 x \rfloor$ zeros. The second part is the bit value of x, which is expressed in as few bits as possible, and begins with one to delimit the prefix. The second Elias code also maps an Integer x to a codeword consisting two parts. The first part results from using the first Elias code to map $(\lfloor \log_2 x \rfloor + 1)$. The second part is the bit value of x, which is expressed in as few bits as possible, with the first bit is deleted. The second Elias code is asymptotically optimal while the first one is not.
  - o A Fibonacci code (Apostolico & Fraenkel, 1985 cited in Lelewer & Hirschberg, 1987) is based on Fibonacci numbers of order $m \geq 2$. A Fibonacci coding technique maps an Integer x to a codeword that is the reversed binary value of x, which is expressed in as few bits as possible, appended with one.
- An arithmetic coding technique maps sources messages to codewords that are represented by intervals between 0 and 1 on the real number line (Lelewer & Hirschberg, 1987). An arithmetic coding technique begins by creating an unordered list of source messages with their probabilities of occurrences. The probabilities specify the sizes of intervals. The source message with higher probability is mapped to the narrower interval. This results in that the source message with high probability is mapped to the smaller codeword.

5.3.2.2 Dynamic Defined-Words Schemes

This subsection reviews some data compression techniques that are dynamic defined-words schemes including dynamic Huffman coding, and Lempel-Ziv coding techniques.

- A dynamic Huffman coding technique was first formed separately by Faller (1973 cited in Lelewer & Hirschberg, 1987) and Gallager (1978 cited in Lelewer & Hirschberg, 1987). And, it was improved by Knuth (1985 cited in Lelewer & Hirschberg, 1987), which results in a dynamic Huffman coding called FGK algorithm. Later, Vitter (1987) invented another version of a dynamic Huffman coding called V algorithm.
    - An important concept of a FGK algorithm is the sibling property of nodes in a binary tree ((Faller, 1973, Gallager, 1978 and Knuth, 1985 cited in Lelewer & Hirschberg, 1987). In a FGK algorithm, a binary tree that consists of a single leaf called the 0-node is primarily constructed. The 0-node represents (n-k) unused source messages at any particular time, where n is the number of possible source messages, and k is the number of source messages occurring at a particular time. If a source message occurring has never been transmitted before, the 0-node is split to a pair of leaves. One represents the occurred source message and another one represents a sibling node which is the new 0-node. Then a binary tree is rearranged such that leaves that are labelled with higher weights are moved to be closer to the root of the tree. If a source message occurring is already represented by a leaf in a binary tree, i.e. it has been transmitted before, the weights of a leaf that represents that source message and the 0-node are incremented. Then the binary tree is rearranged again with the same criteria. At the point of time after source messages have been transmitted, the binary tree consists of k+1 leaves, one for each of k messages and one for the 0-node.
    - A V algorithm improves a dynamic Huffman coding technique by incorporating two improvements with a FGK algorithm (Vitter, 1987 cited in Lelewer & Hirschberg, 1987) . First, during the rearrangement of a binary tree, the number of interchanges that a leaf is moved upward in the tree is limited to only one. In a FGK algorithm, this number is limited to l/2 where l is the length of a codeword for a_{t+1}, where t is the number source messages that have been

transmitted at the time that the rearrangement of the tree begins. Second, a V algorithm minimises the value of Σl_i and $\max\{l_i\}$ according to the requirements of minimising $\Sigma w_i l_i$. These two improvements are achieved by a method used to number the leaves in the tree. By a V algorithm, leaves in the tree are always a level ordering while by a FGK algorithm they are not. At the same level of the tree, blocks of leaves are ordered by increasing weight. The advantage of a V algorithm over a FGK algorithm is that a V algorithm guarantees the minimum height of a binary tree and the minimum value of Σl_i.

- A Lempel-Ziv coding technique is a variable-block coding technique. It defines a mapping between source messages and codewords as algorithm executes (Lelewer & Hirschberg, 1987). It first parses strings of symbols into substrings whose lengths do not exceed a given Integer L_1. Then substrings are mapped to unique decipherable codewords with fixed length L_2. By a Lempel-Ziv coding technique, the strings of symbols are chosen so that they have almost equal probabilities of occurrences. The symbols with high probabilities of occurrences are grouped into long strings while the symbols with low probabilities of occurrences are grouped into short strings.

5.4 A Start/Stop Data Compression Technique

This section describes a Start/Stop data compression technique used to implement MIC in the thesis.

A Start/Stop data compression technique is proposed by Pigeon (2001). It is a generalisation of a (Start, Step, Stop) data compression technique proposed by Fiala (1989 cited in Pigeon, 2001). Both Start/Stop and (Start, Step, Stop) data compression techniques can be used as universal codes to encode Integers.

A Start/Stop data compression technique has a simple concept. It produces smaller average codeword length compared with Huffman coding technique (Pigeon, 2001). A Start/Stop coding technique is based on a sequence of Integers $\{m_1, m_2, \ldots, m_k\}$, where $m_i \in \mathbf{Z}^*$.

A Start/Stop data compression technique maps an Integer x to a codeword whose length is $\min(k'_{max}, k+1) + \sum_{j=0}^{k} m_j$ (Pigeon, 2001). It is able to code arbitrary long Integers by specifying $m_i = c$ where c is constant and k = infinity. In addition, a Start/Stop data compression technique can be optimised with regardless of the number of m_i.

An algorithm of a Start/Stop coding technique is presented in Table 5.1. An example codewords encoded by a Start/Sop coding technique when $\{m_1, m_2, \ldots, m_k\} = \{0, 3, 2, 0\}$ is presented in Table 5.2

Table 5.1 Algorithm of a Start/Stop data compression technique ($\langle m_i \rangle$ representing a string of m_i bits) (Pigeon, 2001)

Source Messages	Codewords
$0 \le x < 2^{m_0}$	$0\langle m_0 \rangle$
$2^{m_0} \le x < 2^{m_0} + 2^{m_0+m_1}$	$10\langle m_0 \rangle\langle m_1 \rangle$
. . .	. . .
$\sum_{j=0}^{i-1} 2^{\sum_{h=0}^{j} m_h} \le x < \sum_{j=0}^{i} 2^{\sum_{h=0}^{j} m_h}$	$11\ldots0\langle m_0 \rangle\langle m_1 \rangle\ldots\langle m_i \rangle$
. . .	. . .
$\sum_{j=0}^{k-2} 2^{\sum_{h=0}^{j} m_h} \le x < \sum_{j=0}^{k-1} 2^{\sum_{h=0}^{j} m_h}$	$11\ldots0\langle m_0 \rangle\langle m_1 \rangle\ldots\langle m_{k-1} \rangle$
$\sum_{j=0}^{k-1} 2^{\sum_{h=0}^{j} m_h} \le x < \sum_{j=0}^{k} 2^{\sum_{h=0}^{j} m_h}$	$11\ldots1\langle m_0 \rangle\langle m_1 \rangle\ldots\langle m_{k-1} \rangle\langle m_k \rangle$

Table 5.2 Example codewords encoded by a Start/Sop coding technique when $\{m_1, m_2, \ldots, m_k\} = \{0, 3, 2, 0\}$(Pigeon, 2001)

Steps	Ranges	Source Messages	Codewords
1	$0 \le x < 2^0$	0	0
2	$2^0 \le x < 2^0 + 2^{0+3}$	1	10000
		2	10001
		3	10010
		4	10011
		5	10100
		6	10101
		7	10110
		8	10111
3	$2^0 + 2^{0+3} \le x < 2^0 + 2^{0+3} + 2^{0+3+2}$	9	11000000
		10	11000001
		11	11000010
		12	11000011
		13	11000100
		. . .	. . .
		39	11011110
		40	11011111
4	$2^0 + 2^{0+3} 2^{0+3+2} \le x < 2^0 + 2^{0+3} + 2^{0+3+2} + 2^{0+3+2+0}$	41	11100000
		42	11100001
		43	11100010
		. . .	
		71	11111110
		72	11111111

5.5 Summary

This chapter gives an overview of the fundamentals of data compression. Some common measures of data compression are described. Also, types of data compression techniques with example techniques are presented. Finally, a Start/Stop data compression technique (Pigeon, 2001), which is used to implement MIC in the thesis, is described in the last section of the chapter.

CHAPTER 6

THE EXTENDED POS

An efficient POS should be able to support all three types of data access – direct, navigational and associative, for an OODBS (Zezulan & Rabitti, 1993). However, to the best of our knowledge, only navigational access in a POS has been studied so far (Kirchberg & Tretiakov, 2002; Kuckelberg, 1998; Subieta, 1994a, 1994b; Zezulan & Rabitti, 1993). The existing approaches that have been proposed for associative access are more likely based on the organisation of physical objects, i.e. data records, because they are modifications of approaches used in a RDBS. Unfortunately, these existing approaches cannot efficiently support queries that involve multiple paths or inheritance hierarchies (Hua & Tripathy, 1994).

The objectives of the thesis include studying how associative access can be supported in a POS of an OODBS, i.e. to extend a model of a POS such that data access approaches that support navigational access can support associative access in the extended POS regardless of paths or inheritance hierarchies involved.

This chapter proposes extensions made to a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) such that the model of a POS can support associative access in an OODBS. The extensions include

- Approaches to cluster storage objects in a POS on their structures or values of attributes, and
- Approaches to distinguish references between storage objects in a POS based on criteria such as reference types – inheritance and association, structures of referenced or referencing storage objects and reference names.

The chapter is divided into three sections as follows.

- The first section expresses the reasons for choosing to extend a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) to be able to support associative access. Also, the section describes issues that are taken into account while extending a model of a POS.
- The second section presents approaches proposed to extend the model of a POS by clustering storage objects in the POS on their structures or values of attributes.
- Lastly, the third section describes approaches proposed to extend the model of a POS by distinguishing references in a POS based on criteria such as reference types – inheritance and association, structures of referenced or referencing storage objects, and reference names.

6.1 Extending a Model of a POS

This section expresses the reasons for choosing to extend a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) to be able to support associative access. Also, the section describes issues that are taken into account while extending a model of a POS.

A POS is a component of an OODBS that provides necessary object storage and retrieval capabilities (Delis, Kanitkar & Kollios, 1998 cited in Kirchberg & Tretiakov, 2002). An efficient POS should be able to support all three types of data access – direct, navigational and associative (Zezulan & Rabitti, 1993). However, to the best of our knowledge, only navigational access in a POS has been studied so far, for example ring and spider data structures by Subieta (1994a; 1994b), a navigation index by Zezulan & Rabitti (1993) and MIC by Kuckelberg (1998).

Please refer to details of a model of a POS employed by Subieta (1994a; 1994b) presented in section 3.3 and subsection 4.3.3, and details of a model of a POS employed by Zezulan & Rabitti (1993) and Kuckelberg (1998) presented in subsection 4.3.4.1.

The thesis elects to extend a model of a POS used by Kuckelberg (1998) and Zezulan & Rabitti (1993) because it is more uncomplicated and flexible compared with a model of a POS used by Subieta (1994a; 1994b). While a model of a POS used by Subieta (1994a; 1994b) has rigid rules for mapping logical objects to storage objects, a model of a POS used by Kuckelberg (1998) and Zezulan & Rabitti (1993) has no such rules. In addition, a model of a POS used by Subieta (1994a; 1994b) is quite informative, i.e. details of the database schema defined in a data model such as inheritances between classes and membership of classes are known in the POS. In contrast, a model of a POS used by Kuckelberg (1998) and Zezulan & Rabitti (1993) is less informative. Storage objects are not clustered and references are not distinguished by any criteria.

Associative access is retrieval of a set of OIDs of objects that match specified conditions (Kirchberg & Tretiakov, 2002; Kuckelberg, 1998; Zezulan & Rabitti, 1993). Hence, while extending the model of a POS to support associative access in an OODBS, conditions that can be specified such as structures or values of attributes of objects are taken into account. Thus, the thesis proposes approaches to extend the model of a POS by clustering storage objects based on their structures or values of attributes.

Moreover, according to the extensions proposed to a model of a POS by clustering storage objects on their structures or values of attributes, associative access always includes navigational access. Hence, it should be beneficial if one could indicate which reference to navigate along. Thus, the thesis takes account of criteria that can be used to distinguish references including reference types – association or inheritance, structures of referenced or referencing storage objects, and reference names. Therefore, the thesis also proposes approaches to extending a model of a POS by distinguishing references based on criteria such as reference types, structures of referenced or referencing storage objects, and reference names.

In the next two sections, details of extensions proposed to a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) in order to support associative access in an OODBS are described.

6.2 Clustering Storage Objects

This section describes approaches proposed to extend a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) to be able to support associative access in an OODBS by clustering storage objects on their structures or values of attributes.

However, before we begin describing approaches proposed to cluster storage objects in a POS on their structures, it is first necessary to define a storage class, which is a class that groups storage objects with the same structure together. In other words, clustering storage objects on their structures is clustering storage objects on their storage classes.

Therefore, this section is divided into four subsections.

- In the first subsection, a definition of a storage class is given.
- The second subsection illustrates an example of mapping logical objects to storage objects, which is used as a running example to explain the approaches proposed to extend a model of a POS by clustering storage objects on their storage classes or values of attributes.
- The third subsection describes approaches proposed to cluster storage objects in a POS on their storage classes.
- The last subsection describes approaches proposed to cluster storage objects in a POS on their values of attributes.

6.2.1 A Storage Class

In an OODM proposed by Schewe & Thalheim (1993), a class in a static view serves as a primitive structure of logical objects and is defined by a class name, a structure expression, and a set of superclasses. A structure expression describes the structure of objects, which includes types of values and references to objects in other classes.

Storage objects in a POS are mapped from logical objects for the purpose of efficient retrieval (Kirchberg et al., 2003). Mapping objects between levels can be done by

clustering or partitioning objects (Zezulan & Rabitti, 1993). In case of clustering several logical objects into a storage object, references between these logical objects are included in a storage object. In case of partitioning logical objects into several storage objects, references between these storage objects are needed to be maintained in order to support recomposition of the logical object when it is requested. In case that a logical object is mapped to a storage object, references between the logical objects are maintained as references between the corresponding storage objects. Hence, we may conclude that references between storage objects in a POS reflect either references between the corresponding logical objects or references between storage objects that result from partitioning a logical object.

In a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993), a storage object is in a form of (OID, OC, OS, OV), where OID is an object identifier of a storage object, OC is a storage object characteristic, i.e. type of persistent, object dependent, sharability, etc., OS is an object structure, and OV is an object value. A reference between storage objects is in a form of (RN, O_p, O_q), where O_p is a referencing storage object, O_q is a referenced storage object, and RN is a reference name.

As described above, references in a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) are maintained separately from storage objects. We may conclude that a storage object consists of an OID and values of attributes. In addition, in order to simplify the structure of storage objects in a POS, we define that a storage class has no superclass.

Thus a storage class is defined by a class name and a structure expression that describes only types of values.

Note that approaches that are used to map logical objects to storage objects are not restricted as long as a storage object consists only of its OID and values of attributes, and references between storage objects are maintained as described above. For example, a logical object may be mapped to several storage objects, several logical

objects may be mapped to only one storage object, or a logical object may be mapped to a storage object.

In the next subsection, an example of mapping logical objects to storage objects is illustrated.

6.2.2 Mapping Logical Objects to Storage Objects

This section illustrates an example of mapping logical objects to storage objects, which is used as a running example to explain approaches proposed to extend a model of a POS by clustering storage objects on their storage classes or values of attributes.

In the example, the following approaches are used for mapping logical objects to storage objects.

- A logical object that belongs to only a class that has no superclass is mapped to a storage object that holds values of attributes defined in the class.
    - o References between these logical objects are maintained as references between corresponding storage objects in the extended POS. These types of references are called association references.
    - o For example, consider an OODB in Appendix B. A logical object that belongs to a class AIRCRAFTC is mapped to a storage object that holds values of attributes defined in a class AIRCRAFTC. A logical object that belongs to a class ENGINEC is mapped to a storage object that holds values of attributes defined in a class ENGINEC. References from logical objects of a class AIRCRAFTC to logical objects of a class ENGINEC are mapped to references between storage objects that hold values of attributes defined in a class AIRCRAFTC and storage objects that hold values of attributes defined in a class ENGINEC.
- A logical object that belongs to many classes is mapped (partitioned) to several storage objects where each storage object holds values of attributes defined in each class the logical object belongs to.

- References between these storage objects are maintained in order to support recomposition of the logical object when it is requested. These types of references are called inheritance references.
- References between a logical object that belongs to many classes and other logical objects are mapped to references between the corresponding storage object that results from the partition and the corresponding storage object that is mapped from another logical object.
- For example, consider an OODM in Appendix B. A logical object that belongs to a class MECHANICC is mapped to two storage objects where one holds values of attributes defined in a class MECHANICC and another one holds values of attributes inherited from a class PERSONC. References between storage objects that hold values of attributes defined in a class MECHANICC and storage objects that hold values of attributes inherited from a class PERSONC are called inheritance references. Also, a logical object that belongs to a class AIRPLANEC is mapped to a storage object that holds values of all attributes defined in a class AIRPLANEC. Hence, references between a logical object of a class MECHANICC and logical objects of a class AIRPLANEC are mapped to references between a storage object that holds values of attributes defined in a class MECHANICC and storage objects that hold values of all attributes defined in a class AIRPLANEC. These types of references are called association references.

Figure 6.1 presents an example of mapping logical objects of an air operator OODBS whose OODM presented in Appendix B to storage objects, where i represents OIDs of logical objects and O represents OIDs of storage objects.

Consider Figure 6.1, where a logical object i_5 is shown as belonging to both classes PERSONC and MECHANICC. A logical object i_5 references logical objects i_{33} and i_{36} of a class AIRCRAFTC; and logical objects i_{33} and i_{36} of a class AIRCRAFTC reference logical objects i_{22}, i_{23}, i_{29} and i_{30} of a class ENGINEC; and logical objects i_{22}, i_{23}, i_{29} and i_{30} of a class ENGINEC reference logical objects i_8, i_9, i_{10}, i_{11}, i_{17} and i_{18} of a class PARTC. These logical objects are mapped into storage objects where details of storage objects are as follows.

- O_1 consists of values of attributes of i_5 defined in a class PERSONC,
- O_2 consists of values of attributes of i_5 defined in a class MECHANICC,
- O_{13} and O_{14} consist of values of attributes of logical objects i_{33} and i_{36} defined in a class AIRCRAFTC, respectively,
- O_9, O_{10}, O_{11} and O_{12} consist of values of attributes of logical objects i_{23}, i_{22}, i_{30} and i_{29} defined in a class ENGINEC, respectively, and
- O_3, O_4, O_5, O_6, O_7 and O_8 consist of values of attributes of logical objects i_{11}, i_{10}, i_9, i_8, i_{18} and i_{17} defined in a class PARTC, respectively.

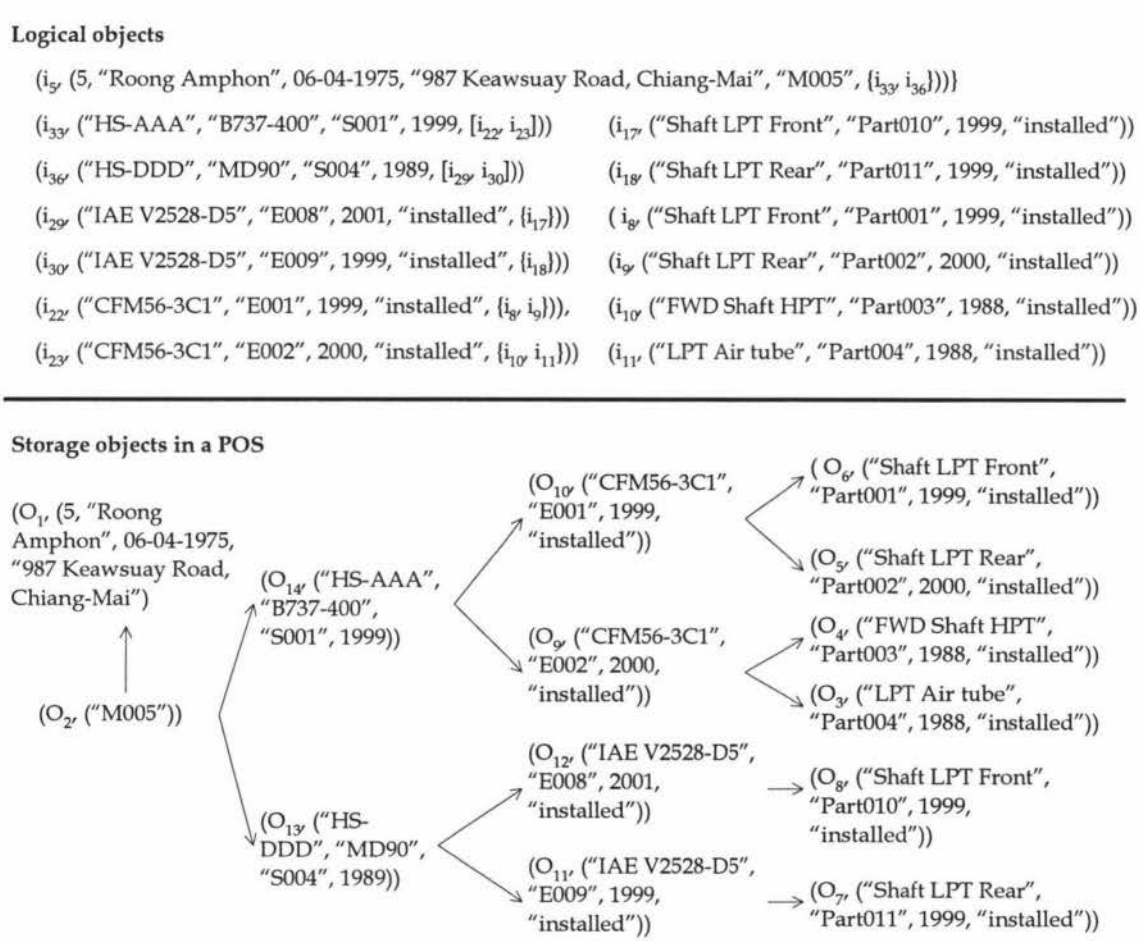


Figure 6.1 An Example of Mapping Logical Objects to Storage Objects

In Figure 6.1, references between these storage objects are presented as edges between two storage objects. A reference between storage objects O_1 and O_2 is an example of an inheritance reference because it results from partitioning i_5 according to an inheritance. The rest are association references that are mapped from references between logical

objects. For instance, a reference between storage objects O_2 and O_{14} reflects a reference between logical objects i_5 and i_{33}.

According to the definition of a storage class given in the last subsection, storage classes for storage objects in Figure 6.1 are defined as follows.

- A storage object O_1, which is mapped from a logical object i_5, consists of values of attributes of the logical object i_5 defined in a class PERSONC. Hence, a storage object O_1 is categorised into a storage class that is defined as follows.

  Storage Class PERSONC
  Structure (PersonID: Integer, Name: String, DateOfBirth: Date,
  Address: String)
  End PERSONC

- A storage object O_2, which is mapped from logical objects i_5, consists of values of attributes of the logical object i_5 defined in a class MECHANICC (not including values of types that are inherited from a class PERSONC). Hence, a storage object O_2 is categorised into a storage class that is defined as follows.

  Storage Class MECHANICC
  Structure (MechanicLicenseNo: String)
  End MECHANICC

- Storage objects O_{13} and O_{14}, which are mapped from logical objects i_{33} and i_{36}, consist of values of attributes of the logical objects i_{33} and i_{36} defined in a class AIRCRAFTC. Hence, Storage objects O_{13} and O_{14} are categorised into a storage class that is defined as follows.

  Storage Class AIRPLANEC
  Structure (RegistrationName: String, Model: String, SerialNumber: String,
  ManufacturedYear: Date)
  End AIRPLANEC

- Storage objects O_9, O_{10}, O_{11} and O_{12}, which are mapped from logical objects i_{23}, i_{22}, i_{30} and i_{29}, consist of values of attributes of the logical objects i_{23}, i_{22}, i_{30} and i_{29} defined in a class ENGINEC. Hence, Storage objects O_9, O_{10}, O_{11} and O_{12} are categorised into a storage class that is defined as follows.

Storage Class ENGINEC
    Structure (Model: String, SerialNumber: String, ManufacturedYear: Date, Status: String)
End ENGINEC

- Storage objects O_3, O_4, O_5, O_6, O_7 and O_8, which are mapped from logical objects i_{11}, i_{10}, i_9, i_8, i_{18} and i_{17}, consist of values of attributes of the logical objects i_{11}, i_{10}, i_9, i_8, i_{18} and i_{17} defined in a class PARTC. Hence, Storage objects O_3, O_4, O_5, O_6, O_7 and O_8 are categorised into a storage class that is defined as follows.

Storage Class PARTC
    Structure (Description: String, SerialNumber:String, ManufacturedYear: Date, Status: String)
End PARTC

6.2.3 Approaches to Cluster Storage Objects on Storage Classes

This subsection proposes two approaches to extend a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) to support associative access in an OODBS by clustering storage objects on their storage classes. The main purpose of this extension is to support retrieval of all storage objects of a storage class or to indicate a storage class that a storage object belongs to.

- The first approach proposed to cluster storage objects on their storage classes is to use a storage object called a Storage Class Object (SCO) to represent a storage class. Then references between storage objects and their corresponding SCOs are maintained. Not every storage class in a POS needs to be represented by a SCO because its purpose is to support a query that requires retrieving all storage objects of a storage class. Hence, only storage classes that involve such queries may require

this approach. In using this approach, we are able to retrieve storage objects of the same storage class by navigating from the corresponding SCO. Figure 6.2 illustrates SCOs for all storage objects in Figure 6.1.

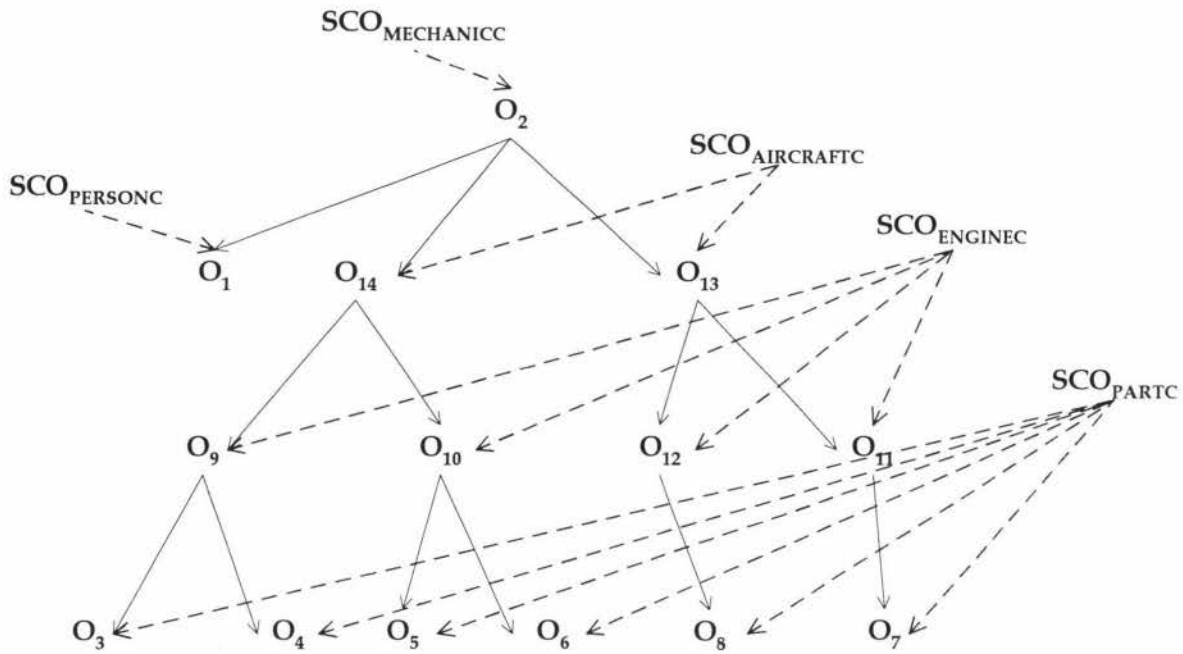


Figure 6.2 Storage Class Objects (SCOs)

- For example, we consider Q1 (Retrieve all parts manufactured in 1999.) of Example 4.1 of chapter four. According to the mapping of logical objects to storage objects presented in Figure 6.1, we may use a SCO approach to support retrieving all storage objects of a storage class PARTC before we select only storage objects whose value of an attribute ManufacturedYear is 1999. This is done by specifying a SCO for a storage class PARTC and maintaining references between storage objects of a storage class PARTC and the SCO.
- The additional storage size required for this approach depends on factors such as the number of storage classes that need to be represented by SCOs, the number of storage objects in each storage class, and the storage sizes of OIDs of storage objects in each storage class. Note that each factor's effect on the additional storage size also varies in different data access approaches used to navigate in the extended POS.

- The speed of this approach also depends on the performance of data access approaches used to navigate in the extended POS.

- The second approach proposed to cluster storage objects on their storage classes is to append Storage Class Flags (SCFs) with OIDs of storage objects to indicate their storage classes. Not every OID of a storage object needs to be appended with a SCF because its main purpose is to indicate which storage class a storage object belongs to while navigating along references between storage objects. Figure 6.3 illustrates SCFs appended with OIDs of all storage objects in Figure 6.1.
  - For example, we consider Q1 (Retrieve all parts manufactured in 1999.) of Example 4.1 of chapter four. According to the mapping of logical objects to storage objects presented in Figure 6.1, we may use a SCF approach to support retrieving all storage objects of a storage class PARTC before we select only storage objects whose value of an attribute ManufacturedYear is 1999. This is done by appending OIDs of all storage objects of a storage class PARTC by a SCF that indicates the storage class PARTC.

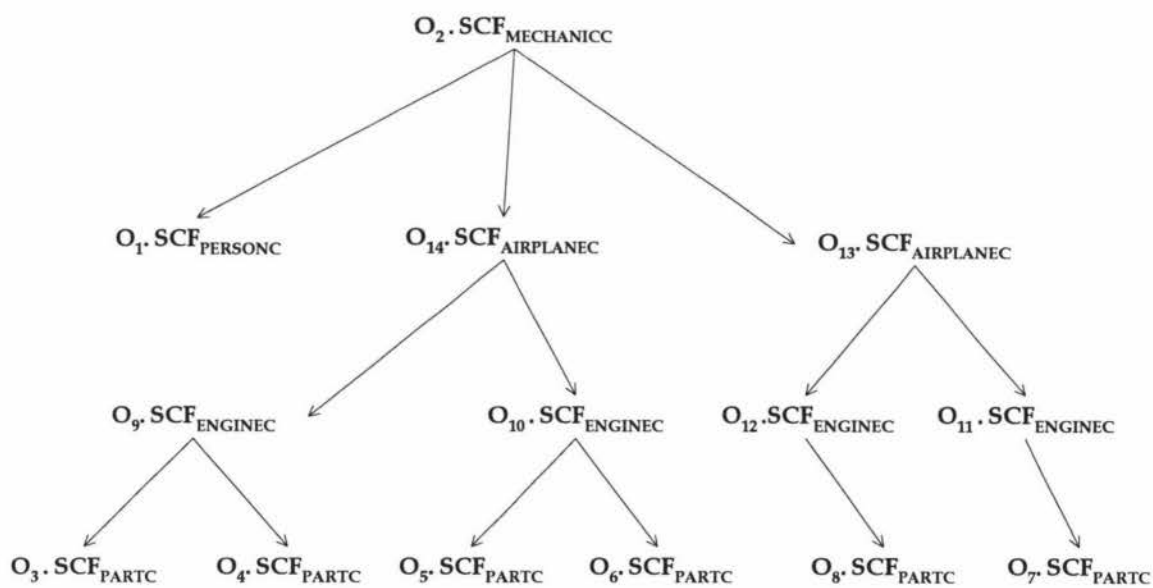


Figure 6.3 Storage Class Flags (SCFs)

  - The additional storage size required for this approach depends on factors such as the number of storage objects that need to append SCFs, the storage sizes of

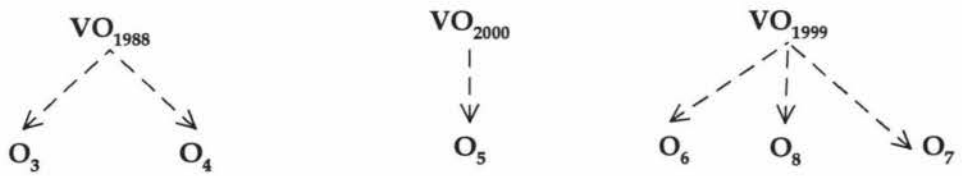
OIDs of storage objects that need to append SCFs and the storage sizes of SCFs. Note that each factor's effect on the additional storage size also varies in different data access approaches used to navigate in the extended POS.

- The speed of this approach also depends on the performance of data access approaches used to navigate in the extended POS.

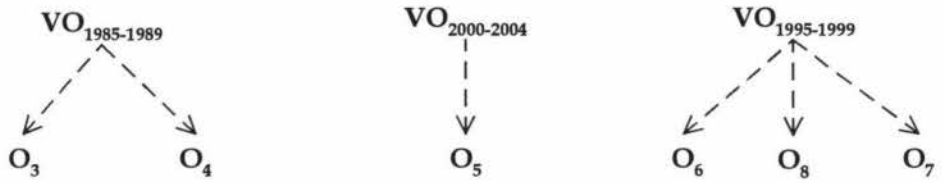
6.2.4 Approaches to Cluster Storage Objects on Values of Attributes

This subsection proposes two approaches to extending a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) to support associative access in an OODBS by clustering storage objects on their values of attributes. The main purpose of this extension is to support retrieval of storage objects based on their values of attributes.

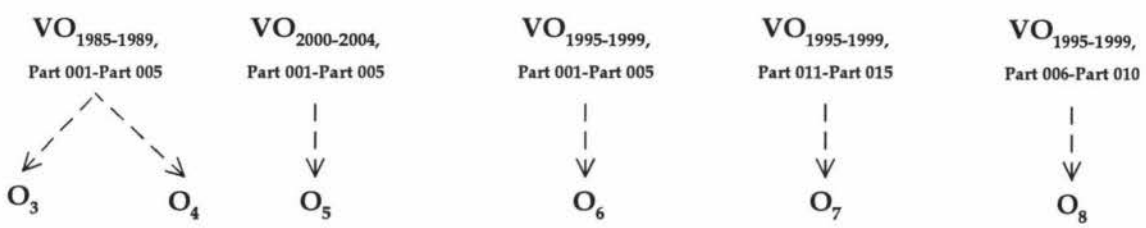
- The first proposed approach to clustering storage objects on their values of attributes is to use a storage object called a Value Object (VO) to represent a value of a storage object. Then references between storage objects and their corresponding VOs are maintained. Not every value of a storage object needs to be represented by a VO because its purpose is to support retrieval of storage objects that match conditions specified on values of attributes. Hence, only values of attributes that are specified in queries may require to be represented by VOs. In addition, a VO can represent values of more than one attribute of a storage object. A VO can also represent ranges of values of attributes. By this approach, we are able to retrieve storage objects that match a search key value by navigating from the corresponding VO. Figure 6.4 a)-c) illustrates various example VOs for storage objects of a storage class PARTC in Figure 6.1.
  - For example, consider Q1 (Retrieve all parts manufactured in 1999) of Example 4.1 of chapter four. According to an example of mapping logical objects to storage objects presented in Figure 6.1, we may specify VOs for values of an attribute ManufacturedYear. References between storage objects of a storage class PARTC and corresponding VOs are maintained. Then to support retrieval of details of all parts manufactured in 1999, we can navigate from a VO that indicates a value of 1999 to storage objects of a storage class PARTC.



a) VOs for storage objects of a storage class PARTC in Figure 6.1 to support queries that retrieve logical objects of a class PARTC on values of an attribute ManufacturedYear (equality retrieval)



b) VOs for storage objects of a storage class PARTC in Figure 6.1 to support queries that retrieve logical objects of a class PARTC on values of an attribute ManufacturedYear (range retrieval)



c) VOs for storage objects of a storage class PARTC in Figure 6.1 to support queries that retrieve logical objects of a class PARTC on values of attributes ManufacturedYear and SerialNumber (range retrieval)

Figure 6.4 a)-c) Value Objects (VOs)

- The additional storage size required for this approach depends on factors such as the number of values of attributes that need to be represented by VOs, the number of storage objects that match each VO, and the storage sizes of OIDs of storage objects that match each VO. Note that each factor's effect on the additional storage size also varies in different data access approaches used to navigate in the extended POS.
- The speed of this approach also depends on the performance of data access approaches used to navigate in the extended POS.

- The second approach proposed to cluster storage object on their values of attributes is to append Value Flags (VFs) with OIDs of storage objects to indicate their values

of attributes. Not every OID of a storage object needs to be appended with a VF. In addition, a VF can represent values of more than one attribute of a storage object. A VF can also represent ranges of values of attributes. Its main purpose is to support retrieval of a storage object that matches specified values of attributes while navigating along references between storage objects. By this approach, while navigating along references between storage objects, we can indicate values of attributes of storage objects. Figure 6.5 a)-c) illustrates various example VFs for storage objects of a storage class PARTC in Figure 6.1.

$O_3.VF_{1988}$ $O_4.VF_{1988}$ $O_5.VF_{2000}$

$O_6.VF_{1999}$ $O_8.VF_{1999}$ $O_7.VF_{1999}$

a) VFs for storage objects of a storage class PARTC in Figure 6.1 to support queries that retrieve logical objects of a class PARTC on values of an attribute ManufacturedYear (equality retrieval)

$O_3.VF_{1985\text{-}1989}$ $O_4.VF_{1985\text{-}1989}$ $O_5.VF_{2000\text{-}2004}$

$O_6.VF_{1995\text{-}1999}$ $O_8.VF_{1995\text{-}1999}$ $O_7.VF_{1995\text{-}1999}$

b) VFs for storage objects of a storage class PARTC in Figure 6.1 to support queries that retrieve logical objects of a class PARTC on values of an attribute ManufacturedYear (range retrieval)

$O_3.VF_{1985\text{-}1989,\ Part\ 001\text{-}Part\ 005}$ $O_4.VF_{1985\text{-}1989,\ Part\ 001\text{-}Part\ 005}$ $O_5.VF_{2000\text{-}2004,\ Part\ 001\text{-}Part\ 005}$

$O_6.VF_{1995\text{-}1999,\ Part\ 001\text{-}Part\ 005}$ $O_7.VF_{1995\text{-}1999,\ Part\ 011\text{-}Part\ 015}$ $O_8.VF_{1995\text{-}1999,\ Part\ 006\text{-}Part\ 010}$

c) VFs for storage objects of a storage class PARTC in Figure 6.1 to support queries that retrieve logical objects of a class PARTC on values of attributes ManufacturedYear and SerialNumber (range retrieval)

Figure 6.5 a)-c) Value Flags (VFs)

- For example, consider Q1 (Retrieve all parts manufactured in 1999) of Example 4.1 of chapter four. According to an example mapping of logical

objects to storage objects presented in Figure 6.1, we may first cluster storage objects of a storage class PARTC by a SCO approach. Then we append VFs that represent corresponding values of an attribute ManufacturedYear with OIDs of storage objects of a storage class PARTC. Hence, to support retrieval of details of all parts based on their manufactured years, we can navigate from the SCO to all storage objects of a storage class PARTC and retrieve only storage objects whose OIDs appended with a VF that indicates a value of 1999.

- The additional storage size required for this approach depends on factors such as the number of storage objects that need to append VFs, the storage sizes of OIDs of storage objects that need to append VFs and the storage sizes of VFs. Note that, each factor's effect on the additional storage size also varies in different data access approaches used to navigate in the extended POS.
- The speed of this approach also depends on the performance of data access approaches used to navigate in the extended POS.

6.3 Distinguishing References in a POS

According to our extensions proposed to cluster storage objects on storage classes or values of attributes presented in the last section, associative access always includes navigational access. Thus, it is beneficial to take account of criteria that can be used to distinguish references when working on associative access in an OODBS because during navigation one can choose which reference to navigate along. The criteria that can be used to distinguish references are for example reference types – i.e. inheritance and association, storage classes of referenced or referencing objects, and reference names.

This section is divided into two subsections. The first subsection describes example circumstances that one should distinguish references based on criteria including reference types – i.e. inheritance and association, storage classes of referenced or referencing objects, and reference names. Then the second subsection proposes two approaches to distinguish references between storage objects.

6.3.1 Circumstances to Distinguish References in a POS

This subsection describes example circumstances that one should distinguish references based on criteria including reference types – i.e. inheritance and association, storage classes of referenced or referencing objects, and reference names.

Figure 6.6 a) – c) illustrate example circumstances appropriate to each criterion used to distinguish references.

- Figure 6.6 a) illustrates a circumstance where criteria such as reference types, storage classes of referenced storage objects, or reference names can be used to distinguish references.
- Figure 6.6 b) illustrates a circumstance where only reference names can be used to distinguish references.
- Figure 6.6 c) illustrates a circumstance where criteria such as reference types or reference names can be used to distinguish references.

In Figure 6.6 a), reference types, storage classes of referenced storage objects, or reference names can be used to distinguish references in order to support the following two queries that are made against logical objects of an OODM presented in Appendix B.

Q1. Retrieve airplanes which are maintained by a mechanic whose licence number is M003.

Q2. Retrieve all details of a mechanic whose licence number is M003.

To support these two queries, one solution is to first obtain a storage object of a storage class MECHANICC whose value of an attribute MechanicLicenceNo is M003. Then,

- For Q1, navigation is made from the storage object of a storage class MECHANICC to storage objects of a storage class AIRPLANEC via association references, and
- For Q2, navigation is made from the storage object of a storage class MECHANICC to a storage object of a storage class PERSONC via an inheritance reference.

Hence, it would be beneficial if we knew which reference was inheritance and which was association. In addition, a class MECHANICC only references two classes, PERSONC and AIRPLANEC, where names of references to these two classes differ

from each other. Thus references from a class MECHANICC can also be distinguished by storage classes of referenced storage objects, and reference names.

In Figure 6.6 b), only reference names can be used to distinguish references. Consider an example OODM in Appendix B. Assume that logical objects of a class PILOTC recursively reference logical objects of a class PILOTC via references named Co-worker and Supervise. The following two queries are example of queries that can be supported by distinguishing references based on reference names.

Q1. Retrieve pilots supervised by a pilot whose pilot licence number is P001.

Q2. Retrieve co-workers of a pilot whose pilot licence number is P001.

To support these two queries, one solution is to first obtain a storage object of a storage class PILOTC whose value of an attribute PilotLicenceNo is P001. Then,

- For Q1, navigation is made from the storage object of a storage class PILOTC to storage objects of a storage class PILOTC via a reference named Supervise, and
- For Q2, navigation is made from the storage object of a storage class PILOTC to storage objects of a storage class PILOTC via a reference named Co-worker.

In this case reference types and classes of referenced storage objects cannot be used as criteria to distinguish references because they are the same.

In Figure 6.6 c), reference types or reference names can be used to distinguish references. Consider an example OODM in Appendix B. Assume that a logical object of a class PILOTC also references logical objects of a class PERSONC via a reference named Passenger. The following two queries are example of queries that can be supported by distinguishing references based on reference types or reference names.

Q1. Retrieve all passengers of a pilot whose pilot licence number is P001.

Q2. Retrieve all details of a pilot whose pilot licence number is P001.

To support these two queries, one solution is to first obtain a storage object of a storage class PILOTC whose value of an attribute PilotLicenceNo is P001. Then,

- for Q1, navigation is made from the storage object of a storage class PILOTC to storage objects of a storage class PERSONC via a reference named passenger, and
- for Q2, navigation is made from the storage object of a storage class PILOTC to storage objects of a storage class PERSONC via an inheritance reference.

In this case reference classes of referenced storage objects cannot be used as criteria to distinguish references because we assume that passengers and persons are stored in the same storage class PERSONC.

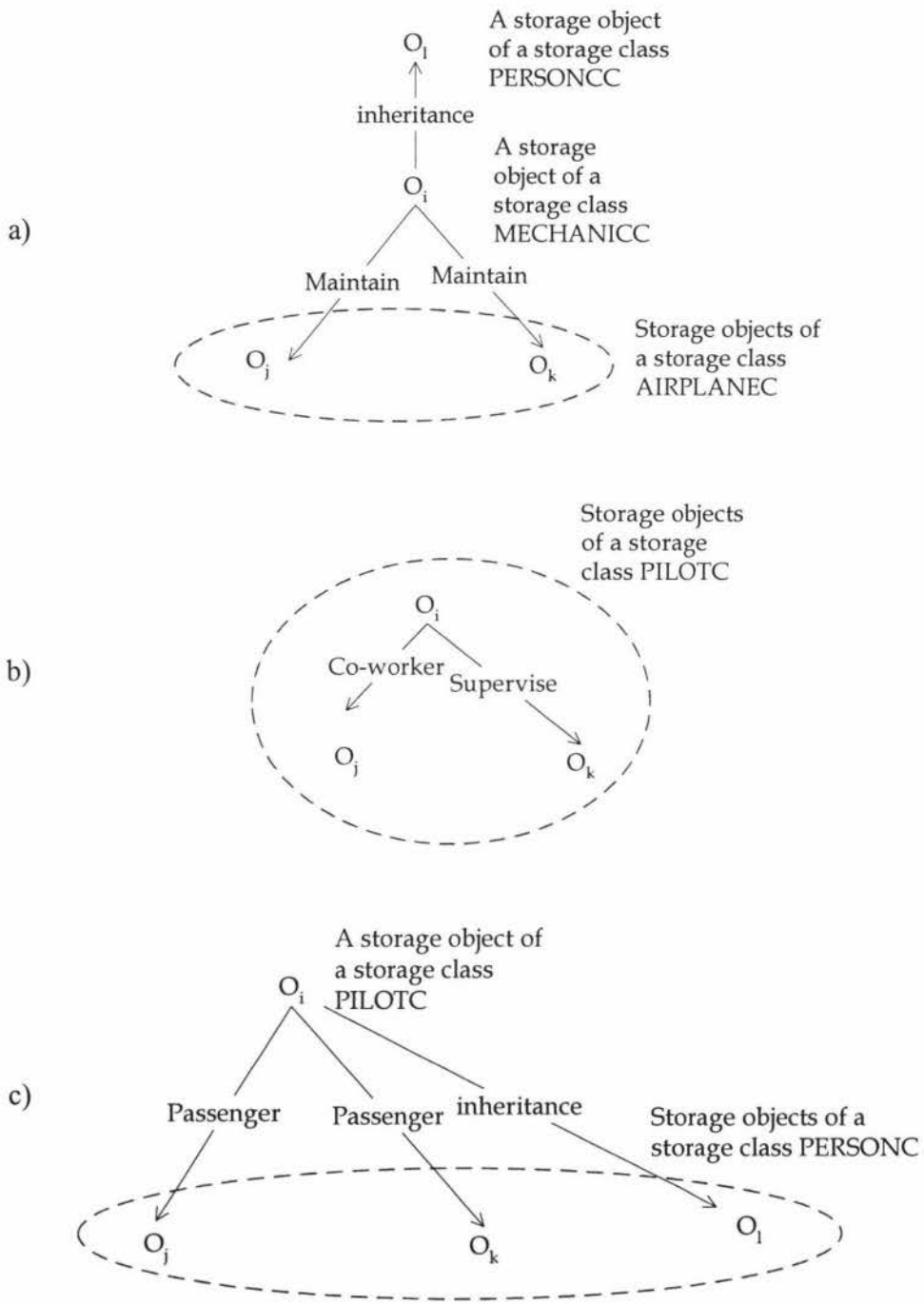


Figure 6.6 a)-c) Example Circumstances to Distinguish References

This subsection illustrates example circumstances when one should distinguish references in a POS. Also, the subsection shows benefits of distinguishing references between storage objects in a POS while working on associative access in an OODBS.

In the next subsection, approaches proposed to distinguish references between storage objects in a POS are described.

6.3.2 Approaches to Distinguish References in a POS

This subsection proposes two approaches to extending a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) to support associative access in an OODBS by distinguishing references in a POS by criteria described in the last subsection.

- The first approach proposed to distinguish references in a POS is to store references separately based on a criterion used, i.e. to store several sets of references. Figure 6.7 illustrates references between storage objects in Figure 6.1, separated by reference types – inheritance and association.

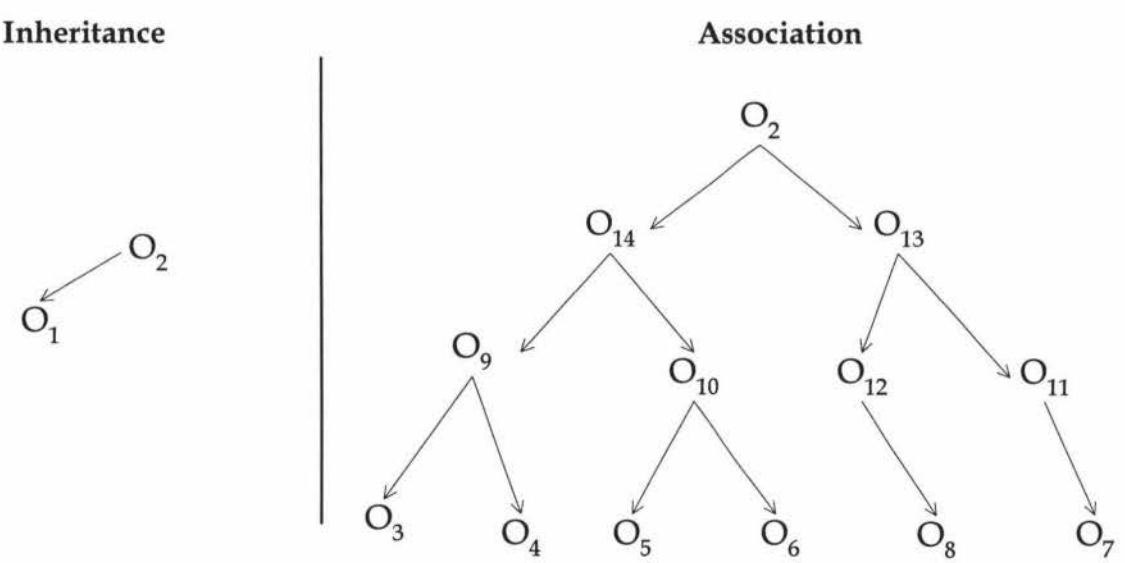


Figure 6.7 Storing References and Storage Objects Separately

- The additional storage size required for this approach may depend on factors such as the number and the storage sizes of OIDs of storage objects that repetitively occur in several sets. For example, in Figure 6.7, O_2 occurs in both sets – inheritance and association. Note that each factor's effect on the additional storage size also varies in different data access approaches used to navigate in the extended POS. This approach may be not appropriate to support queries that simultaneously involve several sets of references because a number of data structures to support navigation may be required, and this may result in storage redundancy.
- The speed of this approach also depends on the performance of data access approaches used to navigate in the extended POS.

- The second approach proposed to distinguish references in a POS is to append a Reference Flag (RF) with references based on criteria used. Not every reference needs to be appended with RF. Its main purpose is to help choosing which reference one wishes to navigate along. Figure 6.8 illustrates RFs to distinguish references in Figure 6.1 based on reference types – inheritance and association.

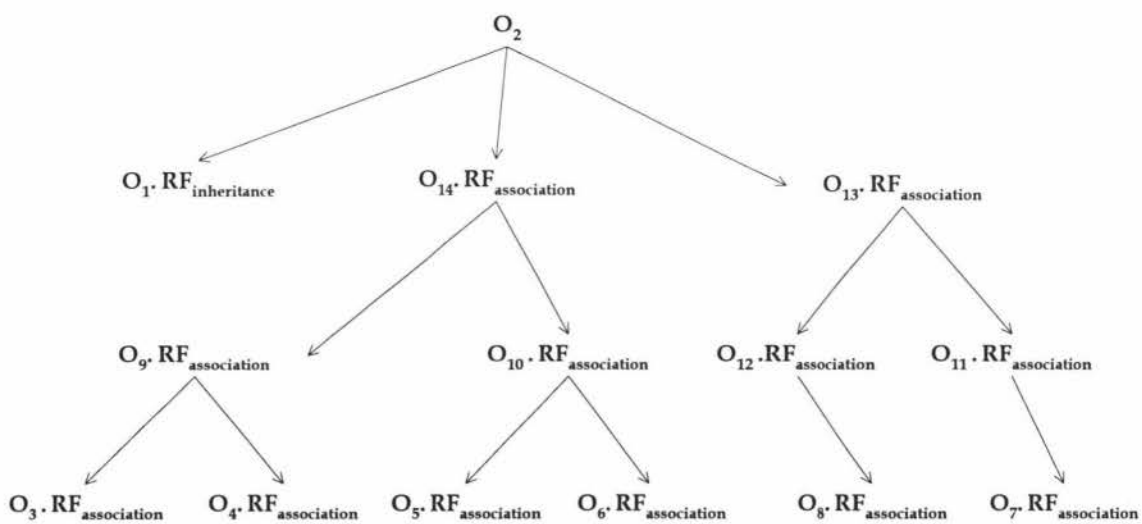


Figure 6.8 Reference Flags (RFs)

- The additional storage size required for this approach depends on factors such as the number of references that need to append RFs, the storage sizes of

referencing or referenced OIDs that are members of references that need to append RFs, and the storage sizes of RFs. Note that each factor's effect on the additional storage size also varies in different data access approaches used to navigate in the extended POS.

- The speed of this approach also depends on the performance of data access approaches used to navigate in the extended POS.

6.4 Summary

This chapter describes extensions proposed to a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) such that the extended POS can support associative access in an OODBS. The extensions include approaches to cluster storage objects in a POS on their storage classes and values of attributes and approaches to distinguish references between storage objects in a POS based on criteria such as reference types – inheritance and association, storage classes of referenced or referencing storage objects, and reference names.

For clustering storage objects in a POS on their storage classes to support retrieval of all storage objects in a storage class or to indicate a storage class of a storage object, the thesis proposes two approaches:

- The first approach is to use storage objects called SCO to represent storage classes. References between storage objects and their corresponding SCOs are maintained.
- The second approach is to append flags called SCF with OIDs of storage objects to indicate their storage classes.

For clustering storage objects in a POS on their values of attributes to support retrieval of storage objects that match conditions specified on values of attributes, the thesis proposes two approaches similar to the approaches proposed to cluster storage objects in a POS on their storage classes:

- The first approach is to use storage objects called VO to represent values of attributes of storage objects. References between storage objects and their corresponding VOs are maintained.

- The second approach is to append flags called VF with OIDs of storage objects to indicate their values of attributes.

For distinguishing references between storage objects in a POS, the thesis proposes two approaches:

- The first approach is to store references separately based on a criterion used.
- The second approach is to append flags called RF with references based on a criterion used.

These proposed approaches behave like tools that can be chosen to support queries requiring associative access in a POS, as appropriate.

For example, SCO, VF and RF approaches may be selected to be used together to support Q4 (Retrieve details of all journeys of an airplane that is installed with an engine part whose serial number is Part001) of Example 4.1 of chapter four.

- According to the mapping approach used in Figure 6.1, to support Q1, a SCO may be used to cluster storage objects of a storage class PARTC. Then VFs are appended to OIDs of storage objects of a storage class PARTC to indicate their values of an attribute SerialNumber.
- Then, RFs are appended to references that reference to storage objects of a storage class AIRPLANEC to distinguish between references from a storage class JOURNEYLOGC to a storage class AIRPLANEC, and references from a storage class MECHANICC to a storage class AIRPLANEC. In this case, reference names are used as a criterion to distinguish the references.
- To support Q4,
    - o We first navigate from a SCO to storage objects of a storage class PARTC and select only a storage object whose OID appended with a VF that indicates a value of an attribute SerialNumber, Part001.
    - o Then, backward navigation is made from the matched storage objects of a storage class PARTC along an association reference to a storage object of a storage class ENGINEC.

- Then, backward navigation is made from the matched storage objects of a storage class ENGINEC along an association reference to a storage object of a storage class AIRPLANEC.
- Finally, backward navigation is made from the matched storage object of a storage class AIRPLANEC along a reference appended with a RF indicating a reference name of FlightAirplane to storage objects of a storage class JOURNEYLOGC.

To illustrate another example, SF and VO approaches may be chosen to support Q3 (Retrieve mechanics and pilots who were born in 1975) of Example 4.1 of chapter four.

- Consider an OODM presented in Appendix B, assuming that a logical object of only a class PERSONC is mapped to a storage object that holds values of attributes defined in a class PERSONC. We categorise this storage object into a storage class PERSONC. A logical object of a class MECHANICC is mapped to a storage object that holds values of attributes inherited from a class PERSONC and values of attributes defined in a class MECHANICC. We categorise this storage object into a storage class MECHANICC. Similarly, a logical object of a class PILOTC is mapped to a storage object that hold values of attributes inherited from a class PERSONC and values of attributes defined in a class PILOTC. We categorise this storage object into a storage class PILOTC.
- We may use VOs to represent values of years of births of an attribute DateOfBirth. Then references between VOs and corresponding storage objects of storage classes PERSONC, MECHANICC and PILOTC are maintained.
- After that, SCFs are appended with OIDs of storage objects of classes PERSONC, MECHANICC and PILOTC to indicate their storage classes.
- To support Q3, we first obtain a VO that indicates a value of years of births of an attribute DateOfBirth, 1975. Then forward navigation is made from a VO to storage objects of storage classes PERSONC, MECHANICC and PILOTC that match a value of years of births of an attribute DateOfBirth, 1975. Then we select only storage objects whose OIDs appended with SCFs indicating storage classes MECHANICC and PILOTC.

Note that these proposed extensions result in that associative access always includes navigational access. In other words, we propose to support associative access in a POS of an OODBS by extending a model of a POS such that data access approaches that are proposed for navigational access can be used to support associative access in the extended POS. Hence, the additional storage size required and the speed of each proposed extension basically depend on the performance of data access approaches used to navigate in the extended POS.

The objectives of the thesis also include providing proof of the concepts that

- A model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS, and
- A data access approach, which is originally proposed to support navigational access, implemented with the extended POS can support a query that requires associative access in an OODBS and involves paths or inheritance hierarchies.

To accomplish this objective, MIC is chosen to be implement with the extended POS. Therefore, the next chapter is devoted to the performance evaluation of the MIC implemented with the extended POS to support associative access in an OODBS.

CHAPTER 7

PERFORMANCE OF MIC WITH THE EXTENDED POS

In the last chapter, we proposed six approaches to extend a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) to support associative access in an OODBS. The approaches are concluded as presented in Table 7.1.

Table 7.1 Extensions proposed to a model of a POS

Extensions	Approaches
Clustering storage objects in a POS on their storage classes	SCO
	SCF
Clustering storage objects in a POS on their values of attributes	VO
	VF
Distinguishing references based on criteria	Store several sets of references
	RF

As mentioned in the last chapter, the additional storage size required and the speed performance of the extensions proposed to a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) to support associative access in an OODBS basically depend on the performance of data access approaches used to navigate in the extended POS.

The thesis selects to implement MIC with the extended POS in order to provide proof of the concepts that

- A model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS, and
- A data access approach, which is originally proposed to support navigational access, implemented with the extended POS can support a query that requires associative access in an OODBS and involves paths or inheritance hierarchies.

This is because MIC employs an in-memory calculation technique, which implies fast speed to support access. In addition, MIC uses relatively simple and efficient concepts compared with other approaches such as a navigation index and ring and spider structures.

Therefore, this chapter is devoted to the performance evaluation of MIC implemented with the extended POS to support associative access in an OODBS.

The criteria that can be used to evaluate the performance of data access approaches include:

- The cost of accessing a data structure of the data access approach to search for requested data, which can also be described in terms of the number of I/O operations required or time spent (speed) during accessing the data structure, and
- Storage size required (Bertino & Kim, 1989; Kim, Kim et al., 1989; Kuckelberg, 1998; Zezulan & Rabitti, 1993).

Hence, the chapter is divided into two sections. The first section discusses the performance of MIC implemented with the extended POS in terms of the cost to support associative access in an OODBS. The second section discusses the performance of MIC implemented with the extended POS in terms of the additional storage size required for the MIC.

7.1 The Cost Performance of MIC to Support Associative Access in the Extended POS

This section discusses the performance MIC implemented with the extended POS in terms of the cost to support associative access in an OODBS. The cost of accessing a data structure of the data access approach to search for requested data can be calculated in terms of the number of I/O operations required or time spent (speed) during accessing the data structure (Bertino & Kim, 1989; Kim, Kim et al., 1989; Kuckelberg, 1998; Zezulan & Rabitti, 1993).

The following equation is formulated by Kuckelberg (1998) to estimate the cost of using MIC to support navigational access in a POS.

$$t_{MIC} = OAC \cdot NAO + IAC \cdot \left(\frac{AR \cdot BN}{AAR} + \frac{BR \cdot FN}{ABR} \right)$$

- t_{MIC}, the cost of MIC to support navigational access in a POS,
- OAC, the cost to access a storage object (depends on the performance of hardwares used),
- NAO, the number of accessed storage objects,
- IAC, the cost to access a commonly smaller index entry in additional navigation structures (depends on the performance of hardwares used),
- $AR \in [0,1]$, the assembling (sharing) rate, i.e. the probability of a storage object to be a sharing object,
- BN, the backward navigation length,
- AAR, the average sharing rate,
- $BR \in [0,1]$, the branching rate, i.e. the probability of a storage object to be a branching object,
- FN, the forward navigation length, and
- ABR, the average branching rate.

According to Table 7.1, two similar approaches are proposed to extend a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) by clustering storage objects in a POS on their storage classes or values of attributes. First, storage objects are used to represent storage classes (SCO) or values of attributes (VO) of storage objects. Second, flags are appended with OIDs of storage objects to indicate their storage classes (SCF) or values of attributes (VF). In addition, two approaches are proposed to distinguish references in a POS. The first approach is to store several sets of references based on criteria used. The second approach is to append references with flags based on criteria used.

- Extending a model of a POS by clustering storage objects using storage objects to represent storage classes (SCO) or values of attributes (VO) of storage objects should not effect the formulation of an equation to estimate the cost of MIC presented above. In other words, an equation formulated by Kuckelberg (1998) to estimate the cost of MIC to support navigational access in a POS should also be valid for estimation of the cost of using the MIC to support associative access in the extended POS. This is because storage objects that represent storage classes or values of attributes of storage objects can be compared with other typical storage objects. However, the introduction of storage objects that represent storage classes or values of attributes implies the increasing number of sharing storage objects after extending a model of a POS as presented in Figure 7.1. In addition, the number of branching storage objects in the extended POS is increased according to the number of storage objects that represent storage classes or values of attributes. The increasing number of sharing and branching storage objects are factors that may cause differences in the cost of the MIC between before and after extending a model of a POS by using SCO or VO approaches to cluster storage objects. If every storage object becomes a sharing and branching storage object, i.e. outgoing and incoming references of a storage object are always multiple, codes calculated for the storage object can be expanded for only one step of navigation. Hence, this may increase the number of I/O operations and time to access the MIC, if several navigational steps are required to support a query.
- Extending a model of a POS by clustering storage objects using flags to represent storage classes (SCF) or values of attributes (VF) of storage objects or distinguishing references by flags (RF) representing some criteria should not also

effect the formulation of an equation to estimate the cost of the MIC presented above. This is because the flags added into the extended POS do not cause the changes to any parameters in the equation. However, to support associative access by using SCF, VF or RF approaches, one needs to scan all OIDs of storage objects and retrieve only storage objects whose OIDs appended with the corresponding flags. Using flag approaches requires the additional number of I/O operations and time to scan storage objects in the extended POS.

- Extending a model of a POS by distinguishing references in a POS by storing them separately according to criteria used should not also effect the formulation of an equation to estimate the cost of the MIC presented above. However, storing references in a POS separately based on criteria may cause the decreasing number of sharing or branching storage objects after extending a model of a POS as presented in Figure 7.2. The decreasing number of sharing or branching storage objects are factors that may cause differences in the cost of the MIC between before and after extending a model of a POS by storing several sets of references. In addition, the number of MICs may be required to support several sets of references. This implies the additional number of I/O operations required and time to access several MICs to support a query that involves multiple sets of references.

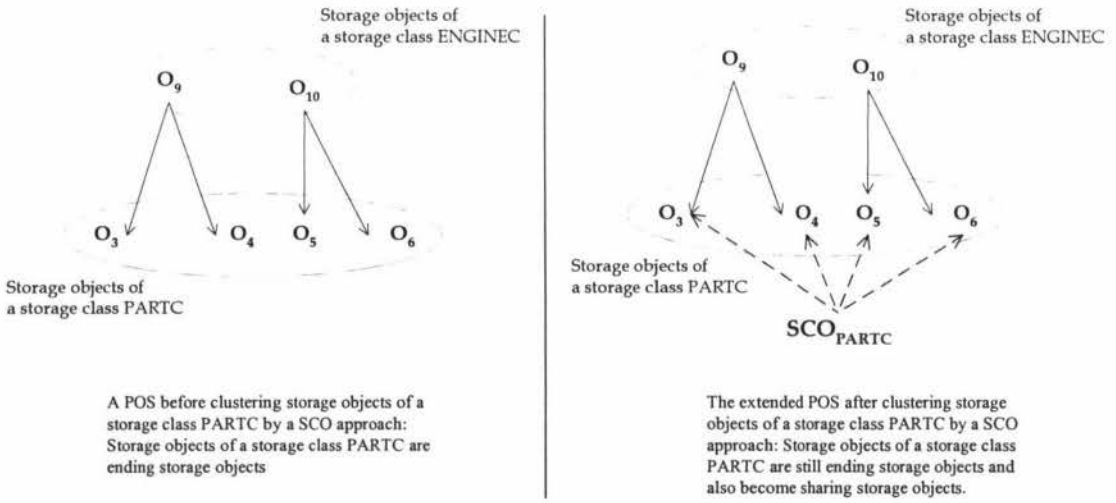


Figure 7.1 The Increasing Number of Sharing Storage Objects in the Extended POS

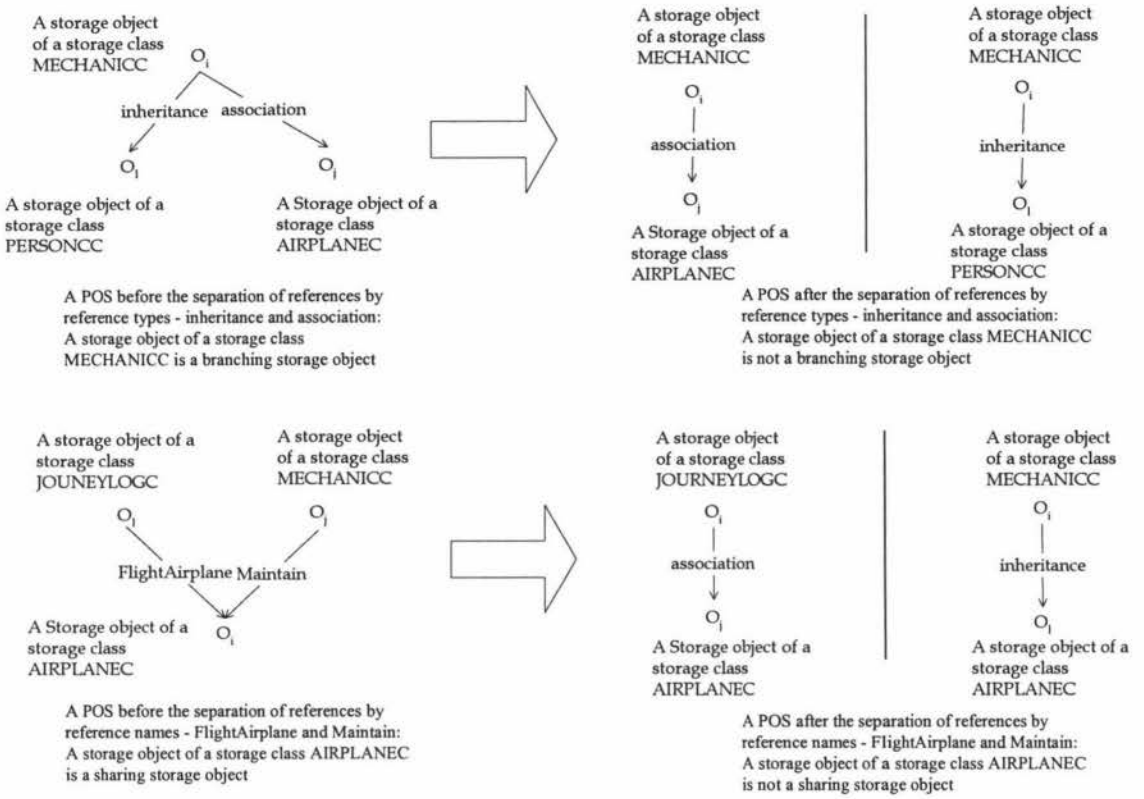


Figure 7.2 The Decreasing Number of Sharing or Branching Storage Objects in the Extended POS

Section 7.1 has discussed the cost performance of MIC to support associative access in the extended POS. It can be summarised as follows.

- For SCOs and SCFs that are proposed to support retrieval of all storage objects of a storage class or to indicate a storage class whom a storage object belongs to,
  - o In terms of the speed performance, from a practical of view, a SCO approach seems to make better use of the MIC because storage objects of a class can be retrieved by executing an expansion of a code for forward navigation of the corresponding SCO. However, the introduction of SCOs results in the increasing number of sharing and branching of storage objects in the extended POS. If every storage object becomes a sharing and branching storage object, i.e. outgoing and incoming references of a storage object always be multiple, codes calculated for the storage object can be expanded for only one step of navigation. Hence, this may increase the number of I/O operations and time to access the MIC, if several navigational steps are required to support a query.

  - To retrieve all storage objects of a class by using a SCF approach, one needs to scan all OIDs of storage objects and retrieve only storage objects whose OIDs appended with the corresponding flags. Hence, using a SICF approach requires the additional number of I/O operations and time to scan all storage objects in the extended POS. However, SCFs do not cause any changes to the number of sharing and branching of storage objects in the extended POS.
- For VOs and VFs that are proposed to support retrieval of storage objects based on conditions specified on their values of attributes results in similar effects to the cost performance of the MIC as SCOs and SCFs do. They may increase the number of I/O operations and time to access the MIC, if several navigational steps are required to support a query.
- Using a RF approach to distinguish references in the extended POS results in similar effects to the cost performance of the MIC as SCFs and VFs do. It requires the additional number of I/O operations and time to scan all storage objects in the extended POS. However, a RF approach does not cause any changes to the number of sharing and branching of storage objects in the extended POS.
- To distinguish references in the extended POS by storing them separately only works well for a query that does not involve several sets of references because the several sets of references involved implies the additional number of I/O operations and time to access several MICs.

Based on the cost performance of the MIC with the extended POS that has been discussed, to balance advantages and disadvantages of SCO, VO, SCF, VF, RF and storing references separately approaches, the following are suggested:

- One may choose to use SCO and VF approaches or VO and SCF approaches to support a query that requires retrieval of storage objects of a storage class based on conditions on values of attributes. This is because
  - Using SCO and VO approaches together may cause the large increasing number of sharing and branching storage objects in the extended POS. And thus it will result the more number of accessing the MIC if a query requires several steps of navigation.
  - Using SCF and VF approaches together do not make out of use of the MIC which employs an in memory technique. One always needs to scan storage

objects to find the storage objects that are appended with the corresponding flags.

- To support a query that requires only retrieval of all storage objects of a storage class, it is more beneficial to use a SCO approach than a SCF approach because OIDs of all storage objects of the storage class can be obtained by just expanding a code for forward navigation of the corresponding SCO.
- To support a query that requires only retrieval of storage objects of a storage class based on some conditions specified on values of attributes, it is more beneficial to use a VO approach than a VF approach because OIDs of storage objects that match specified conditions can be obtained by just expanding a code for forward navigation of the corresponding VO.
- A RF approach should be used with a query that needs distinguish references based on criteria.
- Distinguishing references by storing them separately should be used to support a query that does not involve several sets of references.

In the next section, the performance of MIC implemented with the extended POS is discussed in terms of the additional storage size of the MIC.

7.2 The Additional Storage Size of the MIC to Support Associative Access in the Extended POS

This section discusses the performance of MIC implemented with the extended POS in terms of the additional storage size of the MIC for extensions made to a model of a POS.

The following equation is formulated by Kuckelberg (1998) to estimate the storage size of MIC to support navigational access in a POS. In this equation, it is assumed that all codes calculated by a SICF coding technique require the same storage size.

$$S_{MIC} = NO \cdot (OS + 2 \cdot SS)$$

- S_{MIC}, the storage size required for the MIC to support navigational access in a POS,
- NO, the number of storage objects in a POS (Assume that the MIC is constructed for all storage objects in a POS (a reference graph)),
- OS, the storage size required for an OID of a storage object, and
- SS, the storage size required for each code calculated by a SICF coding technique.

In the thesis, MIC is implemented with the extended POS by several coding techniques. Codes calculated by each data coding technique may require different sizes of storage. In addition, for no coding technique and a Start/Stop data compression techniques (Pigeon, 2001), we have to store types of fanouts of outgoing and incoming references of a storage object, i.e. single, multiple or none. Hence, we reformulate the above equation to use in the thesis as follows.

$$S_{MIC} = \sum_{i=1}^{NO} \left(OS_i + \text{SizeOf}\left(\text{Code}_{\text{forward}_i}\right) + \text{SizeOf}\left(\text{Code}_{\text{backward}_i}\right) + \text{SizeOf}\left(\text{RFT}_{\text{forward}_i}\right) + \text{SizeOf}\left(\text{RFT}_{\text{backward}_i}\right) \right)$$

- S_{MIC}, the storage size required for the MIC to support navigational access in a POS,
- NO, the number of storage objects in a POS (Assume that the MIC is constructed for all storage objects in a POS),
- OS_i, the storage size required for an OID of a storage object O_i,
- $\text{SizeOf}(\text{Code}_{\text{forward}_i})$, the storage size required for a code calculated by a data coding technique for forward navigation from a storage object O_i,
- $\text{SizeOf}(\text{Code}_{\text{backward}_i})$, the storage size required for a code calculated by a data coding technique for backward navigation from a storage object O_i,
- $\text{SizeOf}(\text{RFT}_{\text{forward}_i})$, the storage size required for a type of reference fanout for forward navigation from a storage object O_i (Not applicable for a SICF coding technique), and

– $\text{SizeOf}(\text{RFT}_{\text{backward}_i})$, the storage size required for a type of reference fanout for backward navigation from a storage object O_i (Not applicable for a SICF coding technique)

According to Table 7.1, two similar approaches are proposed to extend a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) by clustering storage objects in a POS on their storage classes or values of attributes. First, storage objects are used to represent storage classes (SCO) or values of attributes (VO) of storage objects. Second, flag are appended with OIDs of storage objects to indicate their storage classes (SCF) or values of attributes (VF). In addition, two approaches are proposed to distinguish references in a POS. The first approach is to store several sets of references based on criteria used. The second approach is to append references with flags based on criteria used.

- Extending a model of a POS by clustering storage objects using storage objects to represent storage classes (SCO) or values of attributes (VO) of storage objects results in that MIC requires additional storage for OIDs of SCOs or VOs and codes for forward navigation from SCOs or VOs. In addition, as mentioned in the last subsection, the introduction of storage objects that represent storage classes or values of attributes implies the increasing number of sharing storage objects. This may also cause the changes of coding to occur in the extended POS as presented in Figure 7.3. The changes of coding also depend on the database schema of a database system, conditions specified in queries represented by SCOs or VOs and an approach used to map logical objects to storage objects.

  Therefore, we may formulate the two following equations for the estimation of additional storage required for the MIC that is implemented with a model of a POS that is extended by clustering storage objects on storage classes or values of attributes by SCO and VO approaches.

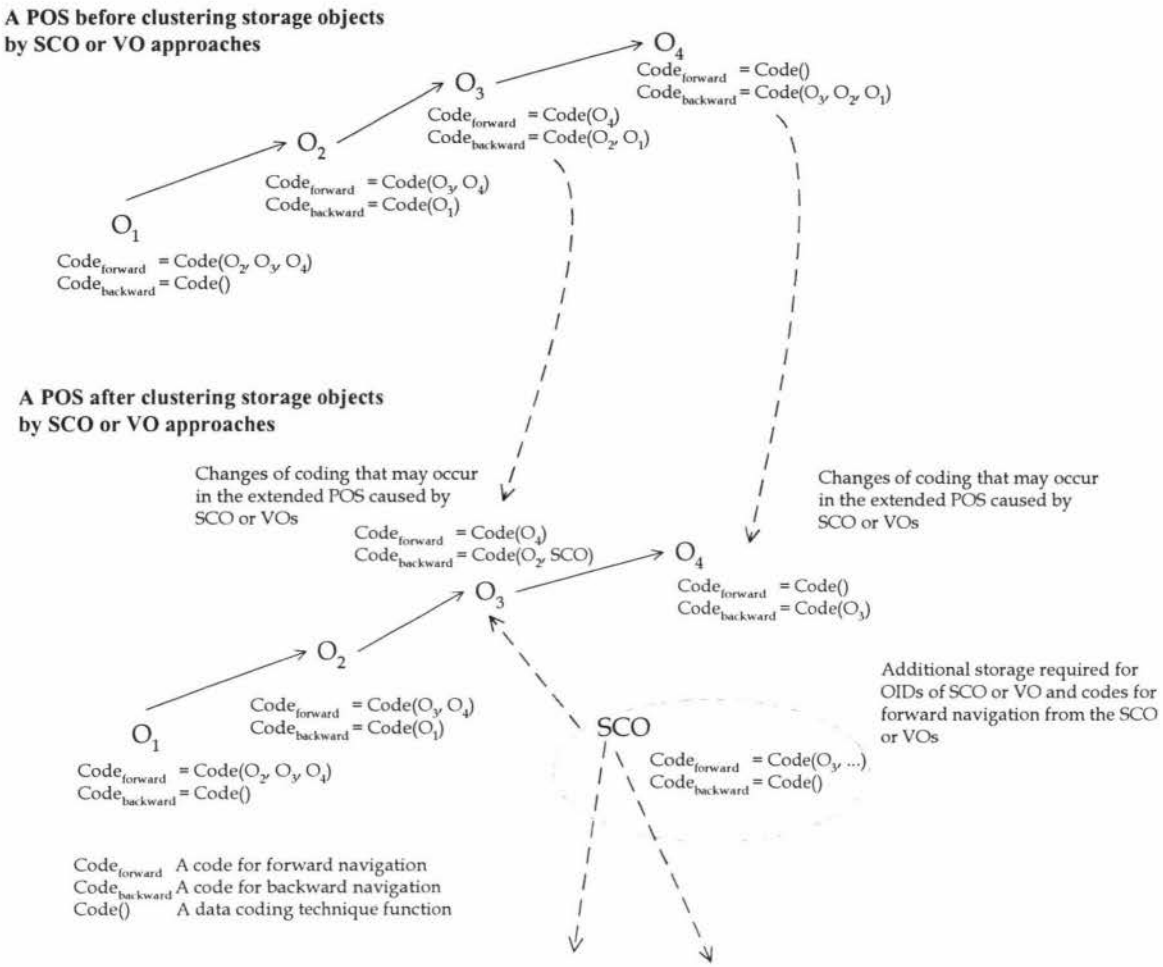


Figure 7.3 Changes in a Model of a POS extended by Clustering Storage Objects by SCO or VO Approaches

- The equation for the estimation of additional storage required for the MIC implemented with a model of a POS extended by a SCO approach is as follows.

$$AS_{SCO} = \sum_{i=1}^{m} \left(OS_{SCO_i} + SizeOf\left(Code_{forward_i} \left(O_{1_i}, \ldots, O_{o_i} \right) \right) \right) + SizeOf(\Delta_1)$$

  - AS_{SCO}, the additional storage size required for the MIC implemented with a model of a POS extended by a SCO approach,
  - m, the number of SCOs,
  - OS_{SCO_i}, the storage size required for an OID of SCO_i,

- o, the number of storage objects that are referenced by SCO_i,
- O_{o_i}, an OID of a storage object that is referenced by SCO_i,
- $SizeOf(Code_{forward_i}(O_{1_i},...,O_{o_i}))$, the storage size required for a code for forward navigation from a SCO_i (The code is calculated by a data coding technique from a sequence of OIDs of storage objects that are referenced by a SCO_i), and
- $SizeOf(\Delta_1)$, the storage size of the changes of coding in the extended POS caused by SCOs.

o The equation for the estimation of additional storage required for the MIC implemented with a model of a POS extended by a VO approach is as follows.

$$AS_{VO} = \sum_{i=1}^{n}\left(OS_{VO_i} + SizeOf\left(Code_{forward_i}(O_{1_i},...,O_{p_i})\right)\right) + SizeOf(\Delta_2)$$

- AS_{VO}, the additional storage size required for the MIC implemented with a model of a POS extended by a VO approach,
- n, the number of VOs,
- OS_{VO_i}, the storage size required for an OID of VO_i,
- p_i, the number of storage objects that are referenced by VO_i,
- O_{p_i}, an OID of a storage object that is referenced by VO_i,
- $SizeOf(Code_{forward_i}(O_{1_i},...,O_{p_i}))$, the storage size required for a code for forward navigation from VO_i (The code is calculated by a data coding technique from a sequence of OIDs of storage objects that are referenced by VO_i), and
- $SizeOf(\Delta_2)$, the storage size of the changes of coding in the extended POS caused by VOs.

- Extending a model of a POS by clustering storage objects using flags (SCF or VF) to represent storage classes or values of attributes of storage objects or

distinguishing references by flags (RF) representing some criteria results in that MIC requires more storage for flags. The additional storage size is caused by flags appended with OIDs that become parts of a sequence of OIDs of storage objects that are transformed to codes in the MIC as presented in Figure 7.4.

Therefore, we may formulate the three following equations for the estimation of additional storage required for the MIC that is implemented with a model of a POS that is extended by clustering storage objects on storage classes or values of attributes by SCF or VF approaches, and distinguishing references by flags (a RF approach).

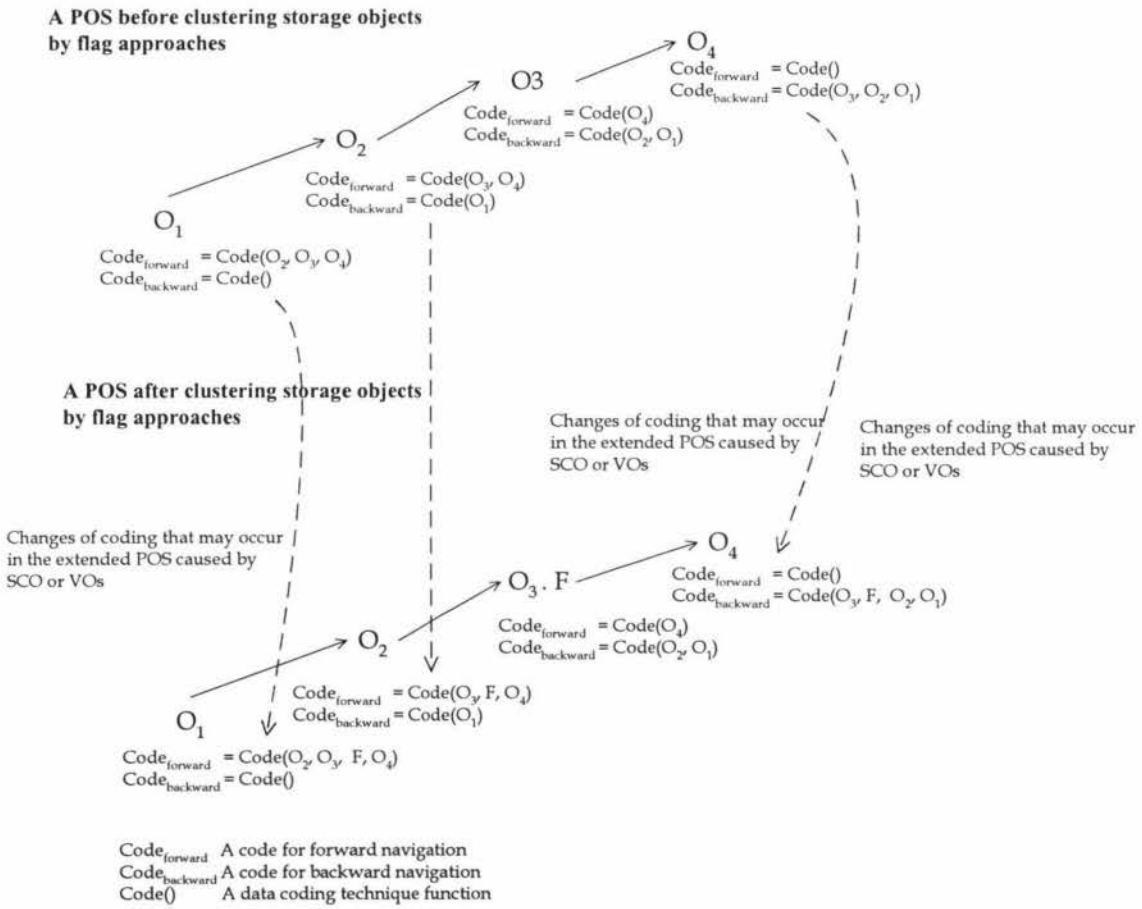


Figure 7.4 Additional Storage in the Extended POS caused by Flag Approaches

- The equation for the estimation of additional storage required for the MIC implemented with a model of a POS extended by a SCF approach is as follows.

$$AS_{SCF} = \sum_{i=1}^{q} (SizeOf(SCF_i) \times r_i) + SizeOf(\Delta_3)$$

- AS_{SCF}, the additional storage size required for the MIC implemented with a model of a POS extended by a SCF approach,
- q, the number of SCFs,
- SCF_i, a flag representing a storage class, and
- r_i, the number of storage objects whose OID appended with SCF_i,
- $SizeOf(\Delta_3)$, the storage size of the changes of coding in the extended POS caused by SCFs.

o The equation for the estimation of additional storage required for the MIC implemented with a model of a POS extended by a VF approach is as follows.

$$AS_{VF} = \sum_{i=1}^{s} (SizeOf(VF_i) \times t_i) + SizeOf(\Delta_4)$$

- AS_{VF}, the additional storage size required for the MIC implemented with a model of a POS extended by a VF approach,
- s, the number of VFs,
- VF_i, a flag representing values of attributes, and
- t_i, the number of storage objects whose OID appended with VF_i,
- $SizeOf(\Delta_4)$, the storage size of the changes of coding in the extended POS caused by VFs.

o The equation for the estimation of additional storage required for the MIC implemented with a model of a POS extended by a RF approach is as follows.

$$AS_{RF} = SizeOf(\Delta_5)$$

- AS_{RF}, the additional storage size required for the MIC implemented with a model of a POS extended by a RF approach,
- $SizeOf(\Delta_5)$, the storage size of the changes of coding in the extended POS caused by RFs.

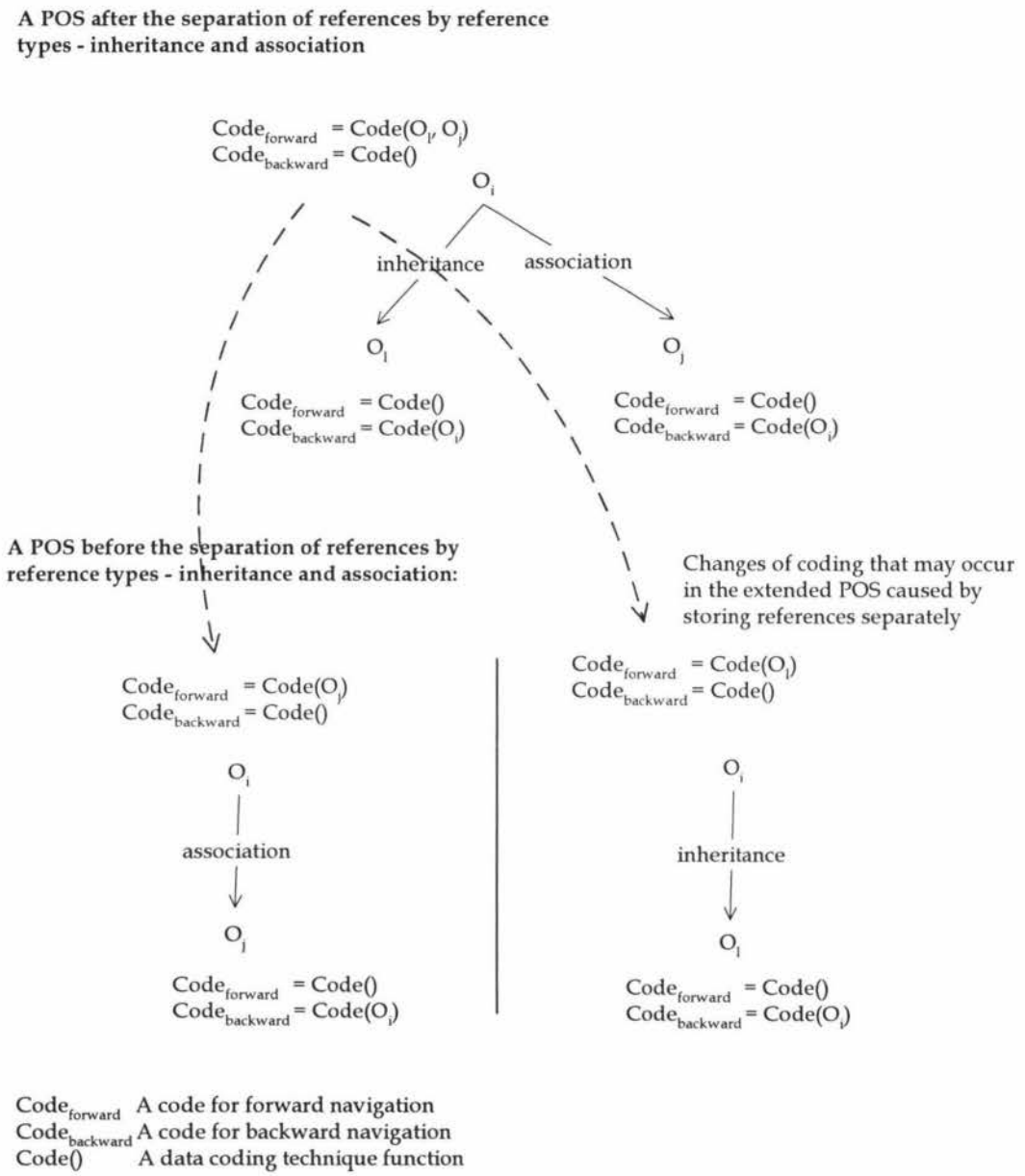


Figure 7.5 Additional Storage in the Extended POS caused by Storing Several Sets of References

- Extending a model of a POS by distinguishing references in a POS by storing them separately according to criteria used causes the MIC to require additional storage for OIDs of storage objects that may occur in several sets of references. In addition, storing references separately results in the changes of coding in the extended POS as presented in Figure 7.5.

  Thus, the equation for the estimation of additional storage required for the MIC implemented with a model of a POS extended by storing references separately is as follows.

$$AS_{SR} = OS_R + SizeOf(\Delta_6)$$

  - AS_{SR}, the additional storage size required for the MIC implemented with a model of a POS storing references separately,
  - OS_R, the storage size required for an OID of storage objects that occur in several sets of references, and
  - $SizeOf(\Delta_6)$, the storage size of the changes of coding in the extended POS caused by storing references separately.

Section 7.2 has discussed the performance of the MIC with the extended POS in terms of the cost and additional storage sizes required. It can be summarised as follows.

- For SCO and SCF approaches that are proposed to support retrieval of all storage objects of a storage class or to indicate a storage class whom a storage object belongs to, additional storage required can be estimated by the two following equations presented in Table 7.2.

  Using SCO or SCF approach to support the same query results in that the number of SCOs is equal to the number of SCFs (m = q) and the number of storage objects that are referenced by each SCO is equal to the number of storage objects that are appended with each SCF (o = r).

Table 7.2 Additional storage required for the MIC implemented with a POS extended by SCOs and SCFs approaches

Approaches	Equations to estimate additional storage required
SCO	$AS_{SCO} = \sum_{i=1}^{m}\left(OS_{SCO_i} + SizeOf\left(Code_{forward_i}\left(O_{1_i},\ldots,O_{o_i}\right)\right)\right) + SizeOf(\Delta_1)$
SCF	$AS_{SCF} = \sum_{i=1}^{q}\left(SizeOf(SCF_i)\times r_i\right) + SizeOf(\Delta_3)$

- For VO and VF approaches that are proposed to support retrieval of storage objects based on conditions specified on their values of attributes, additional storage required can be estimated by the two following equations presented in Table 7.2.

  Using VO or VF to support the same query results in that the number of VOs is equal to the number of VFs (n = s) and the number of storage objects that are referenced by each VO is equal to the number of storage objects that are appended with each VF (p = t).

Table 7.3 Additional storage required for the MIC implemented with a POS extended by VO and VF approaches

Approaches	Equations to estimate additional storage required
VO	$AS_{VO} = \sum_{i=1}^{n}\left(OS_{VO_i} + SizeOf\left(Code_{forward_i}\left(O_{1_i},\ldots,O_{p_i}\right)\right)\right) + SizeOf(\Delta_2)$
VF	$AS_{VF} = \sum_{i=1}^{s}\left(SizeOf(VF_i)\times t_i\right) + SizeOf(\Delta_4)$

- To distinguish references in a POS, two approaches are proposed to distinguish references in a POS. The first approach is to store several sets of references based on a criterion used. The second approach is to append reference flags based on a criterion used. Additional storage required for these two approaches can be estimated by the two following equations presented in Table 7.4.

Table 7.4 Additional storage required for the MIC implemented with a POS extended by distinguishing references in a POS

Approaches	Equations to estimate additional storage required
RF	$AS_{RF} = SizeOf(\Delta_5)$
Storing references separately	$AS_{SR} = OS_R + SizeOf(\Delta_6)$

7.3 Summary

This chapter discusses the performance of MIC implemented with the extended POS to support associative access in an OODBS. The criteria used to evaluate the performance of the MIC include the following.

- The cost of accessing the MIC to search for requested data, which can also be described in terms of the number of I/O operations required or time spent (speed) during accessing the data structure, and
- Storage size required.

In the next chapter, the implementation of MIC with the extended POS to support a query that requires associative access and involves multiple paths and inheritance hierarchies in an OODBS is described.

CHAPTER 8

THE IMPLEMENTATION OF MIC WITH THE EXTENDED POS

In the last chapter, we discussed the performance of MIC implemented with the extended POS to support associative access in an OODBS in terms of the cost of accessing the MIC to search for requested data and the storage size required.

The thesis implements MIC with the extended POS in order to provide proof of the concepts that

- A model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS, and
- A data access approach, which is originally proposed to support navigational access, implemented with the extended POS can support a query that requires associative access in an OODBS and involves paths or inheritance hierarchies.

In addition, several coding techniques are used to implement MIC in order to demonstrate that (1) MIC can be made independent from a coding technique and (2) data compression techniques should be considered as appropriate alternatives to implement the MIC because they can reduce the storage size. The coding techniques used in the implementation include the following.

- The first technique is to represent sequences of OIDs as they are,
- The second technique is to transform sequences of OIDs to codes by using a SICF coding technique which is originally used, and
- The third technique is to individually transform each OID in sequences of OIDs to a code by using a Start/Stop data compression technique (Pigeon, 2001)

This chapter describes the implementation of MIC with the extended POS.

- C++ source codes for the implementation are presented in Appendix C.
- Input data of the implementation program is presented in Appendix D.
- The data structures of the implemented MIC are presented in Appendix E, together with the number of bits required for each coding technique.
- Results of using the implemented MIC to support a query that involves classes in multiple paths and inheritance hierarchies are presented in Appendix F.

The chapter is divided into three sections.

- The first section formulates a query used in the implementation.
- The second section gives details of the implementation, i.e. an approach used to map logical objects to storage objects, and the extensions of a model of a POS and access steps to support the query in the implementation.
- Finally, the third section analyses the results of the implemented MIC.

8.1 A Query Used in the Implementation

The thesis aims to demonstrate that a model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS and show that MIC that is implemented with the extended POS can support associative access in an OODBS regardless of paths or inheritance hierarchies involved.

Hence, the thesis implements MIC with the extended POS to support a query that can represent queries requiring associative access that includes navigational access to logical objects of classes in multiple paths and inheritance hierarchies. This section formulates a query used in the implementation of the MIC as follows.

Consider an example OODM presented in Appendix B, the classes ENGINEC and PARTC are modified as follows.

- A class PARTC has classes LIFECYCLEPARTC and ONCONDITIONPARTC as its subclasses, where classes LIFECYCLEPARTC and ONCONDITIONPARTC have following structures.

Class LIFECYCLEPARTC
IsA PARTC
Structure (HoursToMaintain: Integer)
End LIFECYCLEPARTC

Class ONCONDITIONPARTC
IsA PARTC
Structure (ConditionsToMaintain: String)
End ONCONDITIONPARTC

- A structure of a class ENGINEC is modified as follows.

Class ENGINEC
Structure ENGINE ° (LifeCyclePart: {LIFECYCLEPARTC},
OnConditionPart: {ONCONDITIONPARTC})
End ENGINEC

With the modified OODM explained above, the following query is chosen to be used in the implementation of MIC in the thesis. The query Q involves four paths and two inheritance hierarchies as presented in Figure 8.1 and 8.2, respectively.

Q: *Retrieve details of journeys of an airplane HS-AAA including pilots of the journeys, details of engines and parts installed on the airplane, and details of mechanics who maintain the airplane.*

P1:JOURNEYLOGC.FlightAirplane.RegistrationName
P2:PARTC.Part.Engine.RegistrationName
P3:PILOTC.FlightPilot.FlightAirplane
P4:MECHANICC.Maintain.RegistrationName

Figure 8.1 Paths involved with a Query Q

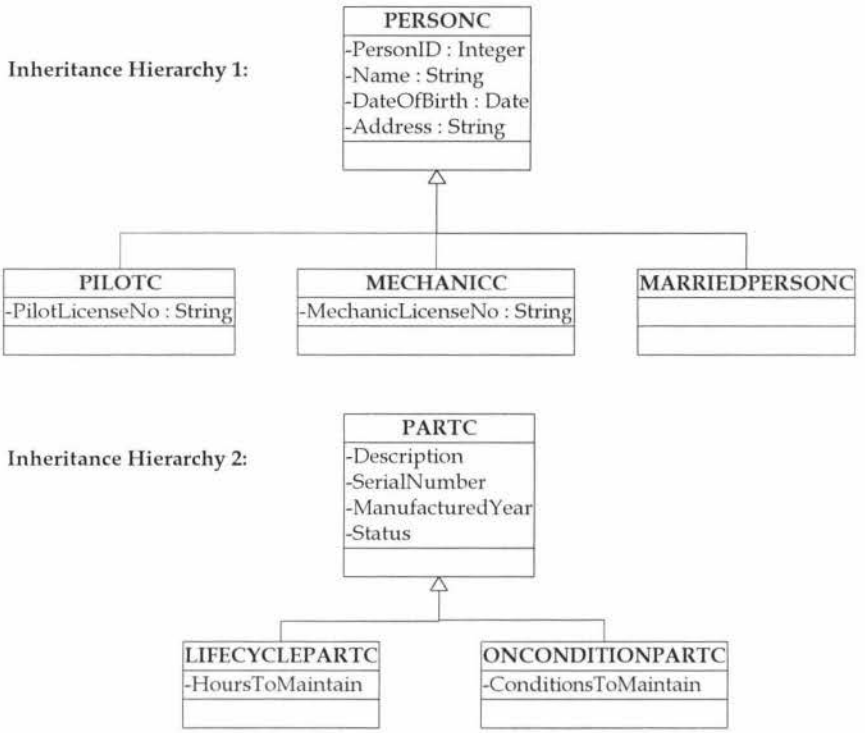


Figure 8.2 Inheritance Hierarchies involved with a Query Q

In the next section, details of the implementation of the MIC to support a query Q are explained. They include

- An approach used to map logical objects to storage objects,
- The extensions of a model of a POS to support a query Q in the implementation, and
- Access steps to support a query Q in the implementation.

8.2 Implementation Details

This section expresses details of the implementation of MIC with the extended POS to support a query Q including:

- An approach used to map logical objects to storage objects,
- The extensions of a model of a POS to support a query Q in the implementation, and
- Access steps to support a query Q in the implementation.

The section also demonstrates that the extended POS is capable of supporting associative access in an OODBS, and shows that MIC implemented with the extended POS can support associative access in an OODBS regardless of paths or inheritance hierarchies involved.

This section is separated into three subsections. The first subsection describes an approach used to map logical objects to storage objects used in the implementation. The second subsection describes the extensions made to a model of a POS in order to support a query Q. The last subsection then describes access steps to support a query Q in the implementation.

8.2.1 Mapping Logical Objects to Storage Objects in the Implementation

This subsection describes an approach to map logical objects to storage objects used in the implementation.

The approach used to map logical objects to storage objects used in the implementation of MIC is same as the approach used in subsection 6.2.2 of chapter six, which is as follows.

- A logical object that belongs to only a class that has no superclass is mapped to a storage object that holds values of attributes defined in the class.
    - References between these logical objects are maintained as references between corresponding storage objects in the extended POS. These types of references are called association references.
- A logical object that belongs to many classes is mapped (partitioned) to several storage objects where each storage object holds values of attributes defined in each class the logical object belongs to.
    - References between these storage objects are maintained in order to support recomposition of the logical object when it is requested. These types of references are called inheritance references.
    - References between a logical object that belongs to many classes and other logical objects are mapped to references between the corresponding storage

object that results from the partition and the corresponding storage object that is mapped from another logical object. These types of references are called association references.

Consider the modified OODM,

- A class JOURNEYLOGC references classes PILOTC and AIRPLANEC.
- A class AIRPLANEC references a class ENGINEC.
- A class ENGINEC references classes LIFECYCLEPARTC and ONCONDITIONPARTC.
- A class MECHANICC references a class AIRPLANEC.
- Classes MECHANICC and PILOTC are subclasses of a class PERSONC.
- Classes LIFECYCLEPARTC and ONCONDITIONPARTC are subclasses of a class PARTC.

In the implementation, details of mapping logical objects to storage objects are described as follows.

- A logical object of only a class PERSONC is mapped to a storage object of a storage class PERSONC that is defined as follows.

```
Storage Class PERSONC
        Structure   (PersonID: Integer, Name: String, DateOfBirth: Date,
                    Address: String)
End   PERSONC
```

- A logical object of a class MECHANICC is mapped to a storage object of a storage class MECHANICC and a storage object of a storage class PERSONC. The storage class MECHANICC is defined as follows.

```
Storage Class MECHANICC
        Structure   (MechanicLicenseNo: String)
End   MECHANICC
```

- A logical object of a class PILOTC is mapped to a storage object of a storage class PILOTC and a storage object of a storage class PERSONC. The storage class PILOTC is defined as follows.

```
Storage Class PILOTC
        Structure   (PilotLicenseNo: String)
End   PILOTC
```

- A logical object of only a class PARTC is mapped to a storage object of a storage class PARTC that is defined as follows.

```
Storage Class PARTC
        Structure   (Description: String, SerialNumber:String,
                    ManufacturedYear: Date, Status: String)
End   PARTC
```

- A logical object of a class LIFECYCLEPARTC is mapped to a storage object of a storage class LIFECYCLEPARTC and a storage object of a storage class PARTC. The storage class LIFECYCLEPARTC is defined as follows.

```
Storage Class LIFECYCLEPARTC
        Structure  (HoursToMaintain: Integer)
End    LIFECYCLEPARTC
```

- A logical object of a class ONCONDITIONPARTC is mapped to a storage object of a storage class ONCONDITIONPARTC and a storage object of a storage class PARTC. The storage class ONCONDITIONPARTC is defined as follows.

```
Storage Class ONCONDITIONPARTC
        Structure  (ConditionsToMaintain: String)
End    ONCONDITIONPARTC
```

- Logical objects of classes JOURNEYLOGC, AIRCRAFTC and ENGINEC are mapped to storage objects of storage classes JOURNEYLOGC, AIRCRAFTC and ENGINEC, respectively. The storage classes JOURNEYLOGC, AIRCRAFTC and ENGINEC are defined as follows.

  Storage Class JOURNEYLOGC
  Structure (OperatingDate: Date, DepartureTime: Date, ArrivalTime: Date, Origin: String, Destination: String)
  End JOURNEYLOGC

  Storage Class AIRCRAFTC
  Structure (RegistrationName: String, Model: String, SerialNumber: String, ManufacturedYear: Date)
  End AIRCRAFTC

  Storage Class ENGINEC
  Structure (Model: String, SerialNumber: String, ManufacturedYear: Date, Status: String)
  End ENGINEC

- References between storage objects of a storage class MECHANICC and the corresponding storage objects of a storage class PERSONC are maintained.
- References between storage objects of a storage class PILOTC and the corresponding storage objects of a storage class PERSONC are maintained.
- References between storage objects of a storage class LIFECYCLEPARTC and the corresponding storage objects of a storage class PARTC are maintained.
- References between storage objects of a storage class ONCONDITIONPARTC and the corresponding storage objects of a storage class PARTC are maintained.
- References between logical objects of classes JOURNEYLOGC, AIRCRAFTC, ENGINEC and PARTC defined in the modified OODM are maintained as references between corresponding storage objects of storage classes JOURNEYLOGC, AIRCRAFTC, ENGINEC and PARTC, respectively.

8.2.2 The Extensions of a Model of a POS in the Implementation

As mentioned in chapter six, the proposed extensions made to a model of a POS are tools that can be chosen to support queries requiring associative access in a POS, as appropriate.

Hence, to support a query Q (Retrieve details of journeys of an airplane HS-AAA including pilots of the journeys, details of engines and parts installed on the airplane, and details of mechanics who maintain the airplane), we consider extending a model of a POS by VO and RF approaches as follows.

- Storage objects of a storage class AIRPLANEC are clustered on values of an attribute RegistrationName by a VO approach. And, references between VOs and the corresponding storage objects of a storage class AIRPLANEC are maintained. We would not choose VFs to cluster storage objects of a storage class AIRPLANEC because to do that we would also need SOs to help supporting retrieval of all storage objects of a storage class AIRPLANEC before selecting the storage object whose OID appended with a VF indicating a value of an attribute RegistrationName, HS-AAA. If we did not use SOs with VFs we may then need to scan all storage objects and retrieve just a storage object whose OID appended with the corresponding VF.
- RFs are used to distinguish references. An approach to distinguish references by storing them separately was not chosen to support a query Q because the query involves multiple paths and inheritance hierarchies. Storing references separately may result in storage redundancy as described in chapter six. Details of RFs are as follows.
    - References from a storage class MECHANICC to a storage class AIRPLANEC and references from a storage class JOURNEYLOGC to a storage class AIRPLANEC are distinguished by RFs based on a criterion, reference names – Maintain and FlightAirplane.
    - References from a storage class JOURNEYLOGC to a storage class AIRPLANEC and references from a storage class JOURNEYLOGC to a storage class PILOTC are distinguished by RFs based on a criterion, reference names – FlightAirplane and FlightPilot.

- References from a storage class MECHANICC to a storage class AIRPLANEC and references from a storage class MECHANICC to a storage class PERSONC are distinguished by RFs based on criteria, reference names and reference types– Maintain and inheritance.
- References from a storage class LIFECYCLEPARTC to a storage class PARTC and references from a storage class ONCONDITIONPARTC to a storage class PARTC are distinguished by RFs based on a criterion, reference names – LifeCyclePart and OnConditionPart.

8.2.3 Access Steps in the Implementation

This subsection describes access steps to support a query Q in the implementation of MIC with a model of the extended POS.

According to an approach used to map logical objects to storage objects and approaches used to extend a model of a POS explained in the last two subsections, the following access steps are accomplished in the implementation to support a query Q:

1. The first step is to make forward navigation from a VO that represents a value of an attribute RegistrationName, HS-AAA, to the corresponding storage object of a storage class AIRPLANEC.
2. After that, the following navigational steps are made:
   - Forward navigation is made from a storage object of a storage class AIRPLANEC that holds a value HS-AAA for an attribute RegistrationName to storage objects of a storage class ENGINEC.
   - Backward navigation is made from a storage object of a storage class AIRPLANEC that holds a value HS-AAA for an attribute RegistrationName along a reference that is appended with a RF indicating a reference name, Maintain, to storage objects of a storage class MECHANICC.
   - Forward navigation is made from storage objects of a storage class MECHANICC along a reference that is appended with a RF indicating a reference type, inheritance, to storage objects of a storage class PERSONC.

- Forward navigation is made from storage objects of a storage class ENGINEC along a reference that is appended with a RF indicating a reference name, LifeCyclePart, to storage objects of a storage class LIFECYCLEPARTC.
- Forward navigation is made from storage objects of a storage class ENGINEC along a reference that is appended with a RF indicating a reference name, OnConditionPart, to storage objects of a storage class ONCONDITIONPARTC.
- Forward navigation is made from storage objects of storage classes LIFECYCLEPARTC and ONCONDITIONPARTC to storage objects of a storage class PARTC.
- Backward navigation is made from a storage object of a storage class AIRPLANEC that holds a value HS-AAA for an attribute registration name along a reference that is appended with a RF indicating a reference name, FilghtAirplane, to storage objects of a storage class JOURNEYLOGC.
- Forward navigation is made from storage objects of a storage class JOURNEYLOGC along a reference that is appended with a RF indicating a reference name, FilghtPilot, to storage objects of a storage class PILOTC.
- Forward navigation is made from storage objects of a storage class PILOTC to storage objects of a storage class PERSONC.

In the next section, the results of the implementation are described.

8.3 Implementation Results

This section describes results of the implementation of MIC with the extended POS to support a query Q.

- The implementation shows that:
  - o A model of a POS extended by VO and RF is capable of supporting a query that requires associative access and involves classes that occur in multiple paths and inheritance hierarchies.
  - o MIC that is implemented with a model of a POS extended by VO and RF approaches can support a query that requires associative access and involves classes that occur in multiple paths and inheritance hierarchies.

- The thesis implements MIC with the extended POS using several coding techniques including the following:
  - o Representing sequences of OIDs as they are,
  - o Transforming sequences of OIDs to codes by using a SICF coding technique which is originally used, and
  - o Individually transforming each OID in sequences of OIDs to a code by using a Start/Stop data compression technique (Pigeon, 2001)

  This demonstrates that the MIC can be made independent from a data coding technique used.
- The thesis compares the storage sizes required for MIC implemented with the three coding techniques. The result shows that a data compression technique should be considered as an appropriate data coding technique to implement the MIC because it can reduce the storage size required. Details of the comparison are as follows.

  In the implementation, VO and RF approaches are used to extend a model of a POS to support a query Q. According to section 7.2 of chapter seven, the additional storage size required when MIC is implemented with a model of a POS extended by VO and RF approaches can be expressed by the following equation.

$$AS_{VO+RF} = \sum_{i=1}^{n}\left(OS_{VO_i} + SizeOf\left(Code_{forward_i}\left(O_1,\ldots,O_p\right)\right)\right) + SizeOf(\Delta_2) + SizeOf(\Delta_4)$$

  Among the implementation of the MIC with the extended POS by no coding technique, a SICF coding technique, and a Start/Stop data compression technique (Pigeon, 2001), only storage sizes of parameters involved with codes calculated by the coding technique are varied.
  - o OS_{VO_i} are constant and
  - o $SizeOf\left(Code_{forward_i}\left(O_1,\ldots,O_p\right)\right)$, $SizeOf(\Delta_2)$ and $SizeOf(\Delta_4)$ are varied.

  Hence, we may conclude that the storage sizes of the implemented MIC by the three mentioned data coding techniques vary according to the performance of data coding technique used to encode the codes.

In the implementation, the thesis calculates the storage sizes required for the implemented MIC by the following equation, which was introduced in section 7.1 of chapter seven.

$$S_{MIC} = \sum_{i=1}^{NO} \left(OS_i + SizeOf\left(Code_{forward_i}\right) + SizeOf\left(Code_{backward_i}\right) + SizeOf\left(RFT_{forward_i}\right) + SizeOf\left(RFT_{backward_i}\right) \right)$$

In the calculation, we specify that data of type Char which indicates types of fanouts of references, i.e. s (single), m (multiple) and – (none) acquires 8 bits of storage size. For data of type Integer, we specify the following.

- If an Integer number is less than $(2^{16} - 1)$, it acquires 16 bits of storage size.
- If an Integer number is less than $(2^{32} - 1)$ and more than or equal to $(2^{16} - 1)$, it acquires 32 bits of storage size.
- If an Integer number is less than $(2^{64} - 1)$ and more than or equal to $(2^{32} - 1)$, it acquires 64 bits of storage size.

According to Appendix E, where data structures of the implemented MIC are presented with the number of bits required, it is shown that:

- MIC implemented with a Start/Stop data
- compression technique (Pigeon, 2001) requires the smallest space (13,332 bits) compared with MIC implemented with no coding technique (20,064 bits) and a SICF coding technique (17,936 bits).
- For MIC that is implemented with a SICF coding technique, two Integer numbers (N and D) that are components of a code can easily grow larger, which makes difficulties to handle if the Integer numbers become larger than 32 bits. If a system does not expand the number of bits to represent the Integer number, it implies that the Integer number that represents an OID of a storage objects cannot be very large since it may easily cause the SICF code to be larger than 32 bits.
- For MIC that is implemented with a Start/Stop data compression technique (Pigeon, 2001), each OID of a storage object is encoded, and thus requires a smaller number of bits than normal. Hence, it is easy to handle the MIC implemented with a Start/Stop data compression technique (Pigeon, 2001) in terms of the size of Integer numbers. In addition, because an OID of a storage

object is encoded individually, the maximum value of the OID can be much larger than $(2^{32}-1)$.

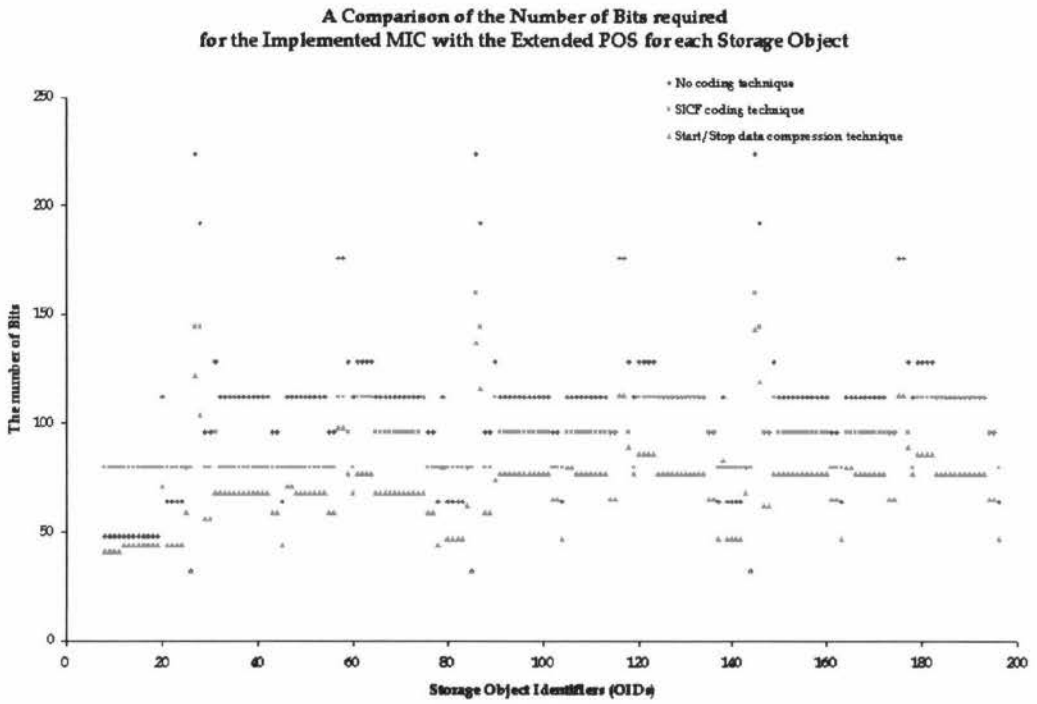


Figure 8.3 A Comparison of the Number of Bits required for the Implemented MIC with the Extended POS for each Storage Object

Consider Figure 8.3 and 8.4. Figure 8.3 presents the number of bits required for an OID and codes of each storage object. Figure 8.4 presents the number of bits required for an OID and codes of each storage object that is sorted by the number of bits required when the MIC is implemented with no coding technique.

- MIC that is implemented with a Start/Stop data compression technique (Pigeon, 2001) always requires the smallest storage size for each storage object compared with the MIC that is implemented with no coding technique and a SICF coding technique.
- In the case that a storage object does not reference any storage object, a code for forward navigation of the storage object in MIC implemented with a SICF coding technique requires more storage size than MIC implemented with no coding technique and a Start/Stop data compression technique (Pigeon, 2001). This is because the MIC implemented with a SICF coding technique always requires storage for two Integer numbers for each code. Even though a storage

object does not reference any storage object, the MIC implemented with a SICF coding technique always requires storage to store two 0 (zero), which requires 32 bits in total. In contrast, if a storage object does not reference any storage object, the MIC that is implemented with no coding technique and a Start/Stop data compression technique (Pigeon, 2001) only requires 8 bits to store data of type Char (-) that indicates that there is no outgoing reference from the storage object. This is also the same for a storage object that is not referenced by any storage object.

Figure 8.4 A Comparison of the Number of Bits required for the Implemented MIC with the Extended POS for each Storage Object (Sorted by the Number of Bits required When the MIC is Implemented with No Coding Technique)

8.4 Summary

This chapter describes the implementation of MIC with the extended POS with various coding techniques, i.e. no coding technique, a SICF coding technique and a Start/Stop data compression technique (Pigeon, 2001).

The results of the implementation show that:

- A model of a POS extended by proposed extensions is able to support associative access in an OODBS.
- MIC that is implemented with the extended POS can support a query requiring associative access in an OODBS and involving multiple paths and inheritance hierarchies. It is clear from the implementation that the MIC that is implemented with the extended POS can support a query that requires associative access and involves any number of paths or inheritance hierarchies.
- In the thesis, we demonstrate that MIC can be made independent from a coding technique used by implementing the MIC with several coding techniques including:
  - o no coding technique,
  - o A SICF coding technique, and
  - o A Start/Stop data compression technique (Pigeon, 2001).
- In the implementation, MIC that is implemented with a Start/Stop data compression technique (Pigeon, 2001) requires the smallest space compared with the other two techniques, i.e. no coding technique and a SICF coding technique. It also offers ease of handling in term of the size of Integer numbers. Hence, data compression techniques should be considered as appropriate choices to implement the MIC as they help by reducing the storage size required.
- In the case that a storage object does not reference any storage object, a code for forward navigation of the storage object in MIC that is implemented with a SICF coding technique requires more storage size than MIC that is implemented with no coding technique and a Start/Stop data compression technique (Pigeon, 2001).
- Similarly, in the case that a storage object is not referenced by any storage object, a code for backward navigation of the storage object in MIC that is implemented with a SICF coding technique requires more storage size than MIC that is implemented with no coding technique and a Start/Stop data compression technique (Pigeon, 2001).

CHAPTER 9

CONCLUSION

This final chapter summarises the core of this thesis and identifies future research to be conducted.

9.1 Conclusion

This thesis is carried out in order to accomplish four following objectives, which already addressed in chapter one.

1. The first objective is to review the existing data access approaches that support associative access in an OODBS. In addition, the thesis aims to review data access approaches that support navigational access in an OODBS because associative access in an OODBS often includes navigational access to referenced or referencing objects of the object being accessed (Kim, Kim et al., 1989).

2. The second objective is to study how associative access can be supported in a POS of an OODBS, i.e. to extend a model of a POS such that data access approaches that support navigational access can support associative access in the extended POS regardless of paths or inheritance hierarchies involved. To accomplish this objective,
   - The thesis selects to extend a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) because it is more simple and flexible compared with a model of a POS employed by Subieta (1994a; 1994b).
   - The thesis takes into account conditions which can be specified in queries that require associative access while considering extending a model of a POS. For

example, structures of objects or values of attributes would be typical conditions. In addition, criteria that can be used to distinguish references, i.e. types of references, structures of referenced or referencing objects, and reference names, are also considered.

3. The third objective is to provide proof of the concepts that
   - A model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS, and
   - A data access approach, which is originally proposed to support navigational access, implemented with the extended POS can support a query that requires associative access in an OODBS and involves paths or inheritance hierarchies.

   To accomplish this objective, the thesis implements MIC, which is originally proposed to support navigational access in an OODBS, with the extended POS to support a query requiring associative access in an OODBS and involving paths and inheritance hierarchies. This is because the MIC employs an in-memory calculation technique that implies the fast speed to support the access. In addition, the MIC uses relatively simple and efficient concepts compared with other approaches such as a navigation index, and ring and spider structures.

4. As mentioned, Kirchberg & Tretiakov (2002) suggest that the original concepts of MIC can be generalised by making the MIC independent from a coding technique used, i.e. any appropriate coding technique can be selected to guarantee optimal performance in different circumstances. Especially, data compression techniques should be considered as alternative coding techniques because they can reduce the storage size of the MIC.

   Therefore, the fourth objective is to provide proof of the concepts that (1) the MIC can be made independent from a coding technique and (2) data compression techniques should be considered as appropriate alternatives to implement the MIC because they can reduce the storage size of the MIC.

To accomplish this objective, the MIC is implemented with the extended POS by the following coding techniques.

- The first technique is to represent sequences of OIDs as they are,
- The second technique is to transform sequences of OIDs to codes by using a SICF coding technique which is originally used, and
- The third technique is to individually transform each OID in sequences of OIDs to a code by a Start/Stop data compression technique (Pigeon, 2001).

The storage sizes required of the MIC implemented with these coding techniques are compared.

This section concludes the thesis according to the objectives of the thesis presented above. The section is separated into three subsections. The first subsection summarises a review of existing data access approaches proposed for associative and navigational access in an OODBS. The second subsection concludes extensions proposed to a model of a POS to be able to support associative access in an OODBS. The last subsection summarises the implementation of MIC with the extended POS.

9.1.1 A Review of Existing Data Access Approaches in an OODBS

In the thesis, a number of existing data access approaches that have been proposed to support associative access in an OODBS are reviewed as presented in chapter four. In addition, because associative access often includes navigational access to referenced or referencing of the object being accessed (Kim, Kim et al., 1989), the thesis reviews existing data access approaches that have been proposed to support navigational access in OODBS as also presented in chapter four.

The existing data access approaches supporting associative access in an OODBS reviewed in the thesis are SC and CH indices (Kim, Kim et al., 1989), H-tree (Low et al., 1992), hcC-tree (Sreenath & Seshadri, 1994), CD index (Ramaswamy & Kanellakis, 1995), nested, path, multi indices (Bertino & Kim, 1989), access support relations (Kemper & Moerkotte, 1990), and a NIX (Bertino, 1991). These approaches are modifications of approaches used in a RDBS such as B+ tree and join indices. They are either based on a path or an inheritance hierarchy. Thus, they cannot perform well when

queries are made against logical objects of classes simultaneously occurring in multiple paths or inheritance hierarchies. To support queries involved with multiple paths or inheritance hierarchies by these existing data access approaches, a number of indices are required. This implies that storage redundancy may occur, which is important especially if these indices are to be stored in main memory. In addition, the number of indices involved may lead to an increase in the number of I/O operations needed and time spent during lookups for requested objects by the indices.

The existing data access approaches supporting navigational access in an OODBS reviewed in the thesis could be categorised based on types of techniques used as follows.

- Modifications of join indices
- Object skeletons
- Reference pointer approaches, and
- In-memory calculation approaches.

For data access approaches that are modifications of join indices and object skeletons, their disadvantages are similar to those of data access approaches proposed to support associative access in an OODBS, i.e. a number of indices may be required, which results in the storage redundancy, and an increase in the number of I/O operations needed and time spent during using the indices.

For data access approaches that employ a reference pointer technique, an important disadvantage is that, every step of navigation requires at least one read operation to bring an object into main memory (in case that object is not already in main memory) to obtain a reference pointer that indicates a referenced or referencing object of the object being accessed.

In contrast, data access approaches that employ in-memory calculation techniques, like navigation index and MIC approaches, have shown a prominent advantage. They efficiently obtain a set of OIDs from navigational access by just a calculation in main-memory regardless of classes or types of references involved. This implies that less time is spent and a smaller number of I/O operations is needed while using these approaches.

In addition, Kirchberg & Tretiakov (2002) suggest that the original concepts of the navigation index and MIC can be generalised by making the navigation index and MIC independent from a coding technique used, i.e. any appropriate coding technique can be selected to guarantee the optimal performance in different circumstances. Especially, data compression techniques should be considered as alternative coding techniques because they can reduce the storage size of the navigation index and MIC.

9.1.2 The Extended POS

An efficient POS should be able to support all three types of data access – direct, navigational and associative, for an OODBS (Zezulan & Rabitti, 1993). However, to the best of our knowledge, only navigational access in a POS has been studied so far (Kirchberg & Tretiakov, 2002; Kuckelberg, 1998; Subieta, 1994a, 1994b; Zezulan & Rabitti, 1993). The existing approaches that have been proposed for associative access are more likely based on the organisation of physical objects, i.e. data records, because they are modifications of approaches used in a RDBS. Unfortunately, as mentioned, these existing approaches cannot perform well when queries involve multiple paths or inheritance hierarchies (Hua & Tripathy, 1994).

Hence, the objectives of the thesis include studying how associative access can be supported in a POS of an OODBS, i.e. to extend a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) such that it can support associative access in an OODBS.

The thesis selects to extend a model of a POS used by Kuckelberg (1998) and Zezulan & Rabitti (1993) because it is uncomplicated and flexible, i.e. a model of a POS used by Kuckelberg (1998) and Zezulan & Rabitti (1993) has no rules for mapping logical objects to storage objects, and storage objects are not clustered and references are not distinguished by any criteria.

The extension proposed to a model of a POS employed by Kuckelberg (1998) and Zezulan & Rabitti (1993) to be able to support associative access in an OODBS include:

- Approaches to cluster storage objects in a POS on their storage classes or values of attributes, and
- Approaches to distinguish references between storage objects in a POS based on criteria such as reference types – inheritance and association, storage classes of referenced or referencing storage objects, and reference names.

For clustering storage objects in a POS on their storage classes to support retrieval of all storage objects in a storage class or to indicate a storage class of a storage object, the thesis proposes two approaches:

- The first approach is to use storage objects called SCO to represent storage classes. References between storage objects and their corresponding SCOs are maintained.
- The second approach is to append flags called SCF with OIDs of storage objects to indicate their storage classes.

For clustering storage objects in a POS on their values of attributes to support retrieval of storage objects that match conditions specified on values of attributes, the thesis proposes two approaches similar to the approaches proposed to cluster storage objects in a POS on their storage classes:

- The first approach is to use storage objects called VO to represent values of attributes of storage objects. References between storage objects and their corresponding VOs are maintained.
- The second approach is to append flags called VF with OIDs of storage objects to indicate their values of attributes.

For distinguishing references between storage objects in a POS, the thesis proposes two approaches:

- The first approach is to store references separately based on a criterion used.
- The second approach is to append flags called RF with references based on a criterion used.

These proposed approaches behave like tools that can be chosen to support queries requiring associative access in a POS, as appropriate. In addition, as these proposed approaches result in that associative access always includes navigational access, the additional storage size required and the speed of employing the proposed approaches

depend on the performance of data access approaches used to navigate in the extended POS.

9.1.3 The Implementation of the MIC

The thesis objectives include the implementation of MIC with the extending POS in order to provide proof of the concepts that:

- A model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS, and
- A data access approach, which is originally proposed to support navigational access, implemented with the extended POS can support a query that requires associative access in an OODBS and involves paths or inheritance hierarchies.

This is because the MIC employs an in-memory calculation technique that implies the fast speed to support the access. In addition, the MIC uses relatively simple and efficient concepts compared with other approaches such as a navigation index, and ring and spider structures.

Moreover, Kirchberg & Tretiakov (2002) suggest that the original concepts of the MIC can be generalised by making the MIC independent from a coding technique used and data compression techniques should be considered as alternative coding techniques because they can help by reducing the storage size of the MIC. Therefore, the following coding techniques are used to implement the MIC.

- The first technique is to represent sequences of OIDs as they are,
- The second technique is to transform sequences of OIDs to codes by using a SICF coding technique which is originally used, and
- The third technique is to individually transform each OID in sequences of OIDs to a code by a Start/Stop data compression technique (Pigeon, 2001).

This is to accomplish the last objective of the thesis which is to provide proof of the concepts that (1) the MIC can be made independent from a coding technique and (2) data compression techniques should be considered appropriate alternatives to implement the MIC because they can reduce the storage size.

The results of the implementation of MIC with the extended POS lead to the following conclusions.

- A model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS,
- MIC that is implemented with the extended POS can support a query that requires associative access and involves multiple paths and inheritance hierarchies,
- MIC can be made independent from a coding technique used, and
- Data compression techniques should be considered as appropriate choices to implement MIC as they help by reducing the storage size required.
- In the case that a storage object does not reference any storage object, a code for forward navigation of the storage object in MIC that is implemented with a SICF coding technique requires more storage size than MIC that is implemented with no coding technique and a Start/Stop data compression technique (Pigeon, 2001).
- Similarly, in the case that a storage object is not referenced by any storage object, a code for backward navigation of the storage object in MIC that is implemented with a SICF coding technique requires more storage size than MIC that is implemented with no coding technique and a Start/Stop data compression technique (Pigeon, 2001).

9.2 Future Research

- The complexity of an OODM has resulted in constraints on this thesis. In the thesis, an OODM proposed by Schewe & Thalheim (1993) is adopted. However, methods are not taken into account, and thus a definition of a class in only static view is employed. In order to advance a model of a POS to be able to support queries that involve the invocations of methods (Bertino & Foscoli, 1995), further research may be carried out with a dynamic definition of class, in which methods are included.
- In the implementation of MIC with the extended POS, only one data compression technique is used for the implementation of the MIC. However, different data compression techniques may result in advantages and disadvantages in different circumstances, for example the speed of encoding and decoding, and storage size.

Thus further research may be undertaken to study data compression techniques that are appropriate in different situations.

- As magnetic disks are in the order of 10^5 times slower than main memory, the thesis anticipates in-memory a data structure (index) that allows executing associative in-memory without accessing magnetic disks and supports associative access in a POS regardless of paths or inheritance hierarchies involved. In that connection, we only estimate the memory requirements of the data structure. On the other hand, for as long as the execution can be performed in-memory, we did not focus on estimating the execution time for the data structure, and have left it as a topic for further research.

REFERENCES

Abiteboul, S., & Catriel, B. (1995). The Power of Languages for the Manipulation of Complex Values. *VLDB Journals, 4*, 727-794.

Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of Databases*. USA: Addison-Wesley.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., & Zdonik, S. (1989). The Object Oriented Database System Manifesto. In W. Kim, J.-M. Nicolas & S. Nishio (Eds.), *Proceedings of the First International Conference on Deductive and Object-Oriented Databases* (pp. 40-57). Kyoto.

Banchilhon, F., Delobel, C., & Kanellakis, P. C. (Eds.). (1992). *Building an Object-Oriented Database System: The Story of O_2*. USA: Morgan Kaufmann Publishers, Inc.

Bertino, E. (1991). An Indexing Technique for Object-Oriented Databases. *Proceedings of the Seventh International Conference on Data Engineering*, 160-170.

Bertino, E. (1994). Index Configuration in Object-Oriented Databases. *VLDB Journal, 3* (3), 355-399.

Bertino, E., & Foscoli, P. (1995). Index Organizations for Object-Oriented Database Systems. *IEEE Transactions on Knowledge and Data Engineering, 7*(2), 193-209.

Bertino, E., & Kim, W. (1989). Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering, 1*(2), 196-214.

Bischofberger, W., & Pomberger, G. (1992). *Prototyping-Oriented Software Development: Concepts and Tools*. USA: Springer-Verlag.

Cardelli, L., & Wegner, P. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *Computer Surveys, 17*(4), 471-522.

Codd, E. (1970). A Relational Model for Large Shared Databanks. *Communications of the ACM, 13*(6), 377-387.

Codd, E. (1982). Relational Database: A Practical Foundation for Productivity. *Communications of the ACM, 25*(2), 109-117.

Elmasri, R., & Navathe, S. B. (2000). *Fundamentals of Database Systems* (3 ed.). USA: Addison-Wesley.

Folk, M. J., Zoellick, B., & Riccardi, G. (1998). *File Structures: An Object -Oriented Approach with C++*. USA: Addison Wesley Longman, Inc.

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2002). *Database Systems: The Complete Book*. USA: Prentice Hall.

Han, J., Xie, Z., & Fu, Y. (1999). Join Index Hierarchy: An Indexing Structure for Efficient Navigation in Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering, 11*(2), 321-337.

Hua, K. A., & Tripathy, C. (1994). Object Skeletons: An Efficient Navigation Structure for Object-Oriented Database Systems. *Proceedings of the 10th International Conference of Data Engineering*, 508-517.

Jodlowski, A. (2002). *Dynamic Object Roles in Conceptual Modeling and Databases.* The Polish Academy of Sciences, Poland.

Kemper, A., & Moerkotte, G. (1990). Access Support in Object Bases. *SIGMOD Conference 1990*, 364-374.

Kim, W. (1990a). Object-Oriented Databases: Definition and Research Directions. *IEEE Transactions on Knowledge and Data Engineering, 2*(3), 327-341.

Kim, W. (1990b). Research Directions in Object Oriented Databases Systems. *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems*, 1-15.

Kim, W., Ballou, N., Chou, K.-T., Garza, J. F., & Woelk, D. (1989). Features of the ORION Object-Oriented Database System. In W. Kim & F. Lochovsky (Eds.), *Object-Oriented Concepts, Databases, and Applications* (pp. 251-282). New York: Addison-Wesley Publishing Company.

Kim, W., Kim, K. C., & Dale, A. (1989). Indexing Techniques for Object-Oriented Database. In W. Kim & F. Lochovsky (Eds.), *Object-Oriented Concepts, Databases and Applications* (pp. 371-394). New York: Addison-Wesley Publishing Company.

Kirchberg, M. (2003). *57.737 Storage and Access Structures of Information Systems: Day 02 Lecture Notes.* Department of Information Systems, Massey University, Palmerston North, New Zealand.

Kirchberg, M., Schewe, K.-D., & Tretiakov, A. (2003). A Multi-Level Architecture for Distributed Object Bases. *Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS).*

Kirchberg, M., & Tretiakov, A. (2002). On Coding Navigation Paths for In-Memory Navigation of Persistent Object Stores. Palmerston North, New Zealand: Massey University.

Kuckelberg, A. (1998). The Matrix-Index Coding Approach to Efficient Navigation in Persistent Object Stores. *Workshop on Foundations of Models and Languages for Data and Objects*, 99-112.

Leavitt, N. (2000). *Whatever Happened to Object-Oriented Databases?* Retrieved 19 December 2003, from http://www.leavcom.com/db_08_00.htm

Lelewer, D. A., & Hirschberg, D. S. (1987). Data Compression. *ACM Computing Surveys, 19*(3), 261-296.

Low, C. C., Ooi, B. C., & Lu, H. (1992). H-trees: A Dynamic Associative Search Index for OODB. *Proceedings of ACM SIGMOD*, 134-143.

Luk, F. H.-W., & Fu, A. W. (1998). Triple-Node Hierarchies for Object-Oriented Database Indexing. *Proceedings of the 1998 ACM CIKM International Conference on Information and Knowledge Management*, 386-397.

Maier, D. (1986). Why Object-Oriented Databases Can Succeed Where Others Have Failed. *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, 227.

Ooi, B. C., & Tan, K.-L. (2001). B-trees: Bearing Fruits of All Kinds. *The Thirteenth Australasian Database Conference (ADC2002)*, 13-20.

Pigeon, S. (2001). *Start/Stop Coding*. Retrieved 1 Novemer 2003, from http://citeseer.nj.nec.com/600274.html

Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (3 ed.). USA: McGraw Hill Higher Education.

Ramaswamy, S., & Kanellakis, P. C. (1995). OODB Indexing by Class-Division. *SIGMOD Record, 24*, 139-150.

Schewe, K.-D., & Thalheim, B. (1993). Fundamental Concepts of Object Oriented Databases. *Acta Cybernetica, 11*(4), 49-84.

Silberschatz, A., Korth, H., & Sudarshan, S. (1997). *Database System Concepts*. Singapore: McGraw-Hill.

Sreenath, B., & Seshadri, S. (1994). The hcC- tree: An Efficient Index Structure for Object Oriented Databases. *Proceedings of the 20th VLDB Conference*, 203-213.

Subieta, K. (1994a). *The Object Manager of the LOQIS Programming Systems*. Retrieved 1 June 2003, from http://pjwstk.com/subieta/EngPapers/ObjectManager.ps.gz

Subieta, K. (1994b). A Persistent Object Store for the LOQIS Programming System. *International Journal on Microcomputer Applications, 13*(2), 50-61.

Third-Generation Database System Manifesto. (1990). The Committee for Advanced DBMS Function.

Valduriez, P. (1987). Join Indices. *ACM Transactions on Database Systems, 12*(2), 218-246.

Verdú, S. (1998). Fifty Years of Shannon Theory. *IEEE Transactions on Information Theory, 44*(6), 2057-2078.

Xie, Z., & Han, J. (1994). Join Index Hierarchies for Supporting Efficient Navigations in Object-Oriented Databases. *Proceedings of the 20th VLDB Conference*, 522-533.

Zand, M., Collins, V., & Caviness, D. (1995). A Survey of Current Object-Oriented Databases. *ACM SIGMIS Database, 26*(1), 14-29.

Zezulan, P., & Rabitti, F. (1993). Object Store with Navigation Accelerator. *Information Systems, 18*, 429-459.

APPENDIX A

AN EXAMPLE RELATIONAL DATA MODEL (RDM)

1. The database schema

AIR OPERATOR = {Person, Airplane, Engine, Part, MarriedPerson, Pilot, Mechanic, InstalledEngine, InstalledPart, JourneyLog}

2. Relation schemas

Person = {person ID: Integer, name: String, date of birth: Date, address: String}
Primary key: {person ID}

Engine = {model: String, serial number: String, manufactured year: Date, status: String}
Primary key: {serial number}

Part = {description: String, serial number: String, manufactured year: Date, status: String}
Primary key: {serial number}

MarriedPerson = {married person ID: Integer, spouse person ID: Integer}
Primary key: {married person ID}
Foreign key: [married person ID] $\subseteq$ Person [person ID]
[spouse person ID] $\subseteq$ Person [person ID]
Constraint: married person ID $\neq$ spouse person ID

Pilot = {pilot person ID: Integer, pilot license No: String}
Primary key: {pilot person ID, pilot license No}
Foreign key: [pilot person ID] ⊆ Person [person ID]

Airplane = {registration name: String, model: String, serial number: String,
manufactured year: Date}
Primary key: {registration name}
Foreign key: [maintained by] ⊆ Mechanic [mechanic license No]

Mechanic = {mechanic person ID: Integer, mechanic license No: String,
maintain: String}
Primary key: {mechanic person ID, mechanic license No, maintain}
Foreign key: [mechanic person ID] ⊆ Person [person ID]
[maintain] ⊆ Airplane [registration name]

InstalledEngine = {installed engine serial number: String, engine No: Integer,
with airplane: String}
Primary key = {installed engine serial number}
Foreign key: [with airplane] ⊆ Airplane [registration number]
[installed engine serial number] ⊆ Engine [serial number]

InstalledPart = {installed part serial number: String,
with engine serial number: String}
Primary key: {installed part serial number}
Foreign key: [installed part serial number] ⊆ Part [serial number]
[with engine serial number] ⊆ Engine [serial number]

JourneyLog = {operational date: Date, departure time: Date, arrival time: Date,
origin: String, destination: String, flight airplane: String,
flight pilot: String}
Primary key: {operational date, departure time, arrival time, flight airplane, flight pilot}
Foreign key: [flight airplane] ⊆ Airplane [registration name]
[flight pilot] ⊆ Pilot [pilot license No]

3. Relations

Person relation

{(1, "Supa Rattana", 30/09/1970, "123 Sakorn Road, Bangkok"),
(2, "Anong Sri-ngarm", 13/02/1956, "456 Sang-ngarm Road, Bangkok"),
(3, "Suk Jaidee", 20/07/1967, "234 Baansuk Road, Nontaburi"),
(4, "Ram Pongsa", 04/06/1975, "567 Lamklong Road, Phuket"),
(5, "Roong Amphon", 06/04/1975, "987 Keawsuay Road, Chiang-Mai"),
(6, "Suay Rattana", 26/11/1972, "123 Sakorn Road, Bangkok"),
(7, "Prasan Amphon", 15/03/1950, "987 Keawsuay Road, Chiang-Mai")}

Airplane relation

{("HS-AAA", "B737-400", "S001", 1999, "M004"),
("HS-BBB", "A310", "S002", 2001, "M004"),
("HS-CCC", "DC10", "S003", 1975, "M005"),
("HS-DDD", "MD90", "S004", 1989, "M003")}

Engine relation

{("CFM56-3C1", "E001", 1999, "installed"),
("CFM56-3C1", "E002", 2000, "installed"),
("PW4152", "E003", 1987, "installed"),
("PW4152", "E004", 1999, "installed"),
("CF6-50C2", "E005", 1998, "installed"),
("CF6-50C2", "E006", 1998, "installed"),
("CF6-50C2", "E007", 1998, "installed"),
("IAE V2528-D5", "E008", 2001, "installed"),
("IAE V2528-D5", "E009", 1999, "installed"),
("IAE V2528-D5", "E010", 2000, "preserved"),
("CF6-50C2", "E011", 1998, "preserved")}

Part relation

{("Shaft LPT Front", "Part001", 1999, "installed"),
("Shaft LPT Rear", "Part002", 2000, "installed"),
("FWD Shaft HPT", "Part003", 1988, "installed"),
("LPT Air tube", "Part004", 1988, "installed"),
("FWD Shaft HPT", "Part005", 1988, "installed"),
("FWD Shaft HPT", "Part006", 1988, "installed"),
("Shaft LPT Front", "Part007", 1999, "installed"),
("Shaft LPT Front", "Part008", 1999, "installed"),
("Shaft LPT Rear", "Part009", 1999, "installed"),
("Shaft LPT Front", "Part010", 1999, "installed"),
("Shaft LPT Rear", "Part011", 1999, "installed"),
("Shaft LPT Rear", "Part012", 2000, "installed"),
("LPT Air tube", "Part013", 1988, "installed"),
("LPT Air tube", "Part014", 1988, "stored")}

MarriedPerson relation

{(1, 6), (6, 1), (5, 7), (7, 5)}

Pilot relation

{(1, "P001"), (2, "P002")}

Mechanic relation

{(4, "M004", "HS-AAA"), (4, "M004", "HS-BBB"),
(5, "M005", "HS-CCC"), (3, "M003", "HS-DDD")}

InstalledEngine relation

{("E001", 1, "HS-AAA"), ("E002", 2, "HS-AAA"), ("E003", 1, "HS-BBB"),
("E004", 2, "HS-BBB"), ("E005", 1, "HS-CCC"), ("E006", 2, "HS-CCC"),
("E007", 3, "HS-CCC"), ("E008", 1, "HS-DDD"), ("E009", 2, "HS-DDD")}

InstalledPart relation

{("Part001", "E001"), ("Part002", "E001"), ("Part003", "E002"),
("Part004", "E002"), ("Part005", "E003"), ("Part006", "E004"),
("Part007", "E005"), ("Part008", "E006"), ("Part009", "E007"),
("Part010", "E008"), ("Part011", "E009"), ("Part012", "E010"),
("Part013", "E011")}

JourneyLog relation

{(21/9/1999, 8.30, 10.00, "Bangkok", "Chiang-Mai", "HS-AAA", "P001"),
(21/9/1999, 8.30, 10.00, "Bangkok", "Chiang-Mai", "HS-AAA", "P002"),
(17/10/1999, 11.00, 13.30, "Chiang-Mai", "Bangkok", "HS-AAA", "P001"),
(17/10/1999, 11.00, 13.30, "Chiang-Mai", "Bangkok", "HS-AAA", "P002"),
(18/10/1999, 10.00, 11.30, "Bangkok", "Phuket", "HS-BBB", "P001"),
(18/10/1999, 10.00, 11.30, "Bangkok", "Phuket", "HS-BBB", "P002"),
(20/11/1999, 13.00, 15.30, "Phuket", "Bangkok", "HS-BBB", "P001"),
(20/11/1999, 13.00, 15.30, "Phuket", "Bangkok", "HS-BBB", "P002")}

APPENDIX B

AN EXAMPLE OBJECT ORIENTED DATA MODEL (OODM)

In this example OODM,

- A constructed type record is represented by (.).
- ∘ is used to link types record.
- A constructed type set is represented by {.}.
- A constructed type list is represented by [.].

1. Type Expressions

```
Type  PERSON
      = (PersonID: Integer, Name: String, DateOfBirth: Date, Address: String)
End   PERSON

Type  ENGINE
      = (Model: String, SerialNumber: String, ManufacturedYear: Date,
        Status: String)
End   ENGINE

Type  PART
      = (Description: String, SerialNumber:String, ManufacturedYear: Date,
        Status: String)
End   PART
```

```
Type  PILOT
      = (PersonID: Integer, Name: String, DateOfBirth: Date, Address: String,
        PilotLicenseNo: String)
End   PILOT

Type  JOURNEYLOG
      = (OperatingDate: Date, DepartureTime: Date, ArrivalTime: Date,
        Origin: String, Destination: String)
End   JOURNEYLOG
```

2. Schema AIR OPERATOR

```
Class  PERSONC
   Structure  PERSON
End    PERSONC

Class  PARTC
   Structure  PART
End    PARTC

Class  ENGINEC
   Structure  ENGINE ° (Part: {PARTC})
End    ENGINEC

Class  AIRPLANEC
   Structure  (RegistrationName: String, Model: String, SerialNumber: String,
              ManufacturedYear: Date, Engine: [ENGINEC])
End    AIRPLANEC

Class  MARRIEDPERSONC
   IsA PERSONC
   Structure  (PersonID: Integer, Name: String, DateOfBirth: Date, Address: String,
              Spouse: MARRIEDPERSONC)
End    MARRIEDPERSONC
```

Class PILOTC
IsA PERSONC
Structure PILOT
End PILOTC

Class MECHANICC
IsA PERSONC
Structure (PersonID: Integer, Name: String, DateOfBirth: Date,
Address: String, MechanicLicenseNo: String,
Maintain: {AIRPLANEC})
End MECHANICC

Class JOURNEYLOGC
Structure JOURNEY ° (FlightAirplane: AIRPLANEC, FlightPilot: [PILOTC])
End JOURNEYLOGC

3. The instance of a database

PERSONC class

{(i_1, (1, "Supa Rattana", 30-09-1970, "123 Sakorn Road, Bangkok")),
(i_2, (2, "Anong Sri-ngarm", 13-02-1956, "456 Sang-ngarm Road, Bangkok")),
(i_3, (3, "Suk Jaidee", 20-07-1967, "234 Baansuk Road, Nontaburi")),
(i_4, (4, "Ram Pongsa", 04-06-1975, "567 Lamklong Road, Phuket")),
(i_5, (5, "Roong Amphon", 06-04-1975, "987 Keawsuay Road, Chiang-Mai")),
(i_6, (6, "Suay Rattana", 26/11/1972, "123 Sakorn Road, Bangkok")),
(i_7, (7, "Prasan Amphon", 15/03/1950, "987 Keawsuay Road, Chiang-Mai"))}

PARTC class

{(i_8, ("Shaft LPT Front", "Part001", 1999, "installed")),
(i_9, ("Shaft LPT Rear", "Part002", 2000, "installed")),
(i_{10}, ("FWD Shaft HPT", "Part003", 1988, "installed")),
(i_{11}, ("LPT Air tube", "Part004", 1988, "installed")),
(i_{12}, ("FWD Shaft HPT", "Part005", 1988, "installed")),

(i_{13}, ("FWD Shaft HPT", "Part006", 1988, "installed")),

(i_{14}, ("Shaft LPT Front", "Part007", 1999, "installed")),

(i_{15}, ("Shaft LPT Front", "Part008", 1999, "installed")),

(i_{16}, ("Shaft LPT Rear", "Part009", 1999, "installed")),

(i_{17}, ("Shaft LPT Front", "Part010", 1999, "installed")),

(i_{18}, ("Shaft LPT Rear", "Part011", 1999, "installed")),

(i_{19}, ("Shaft LPT Rear", "Part012", 2000, "installed")),

(i_{20}, ("LPT Air tube", "Part013", 1988, "installed")),

(i_{21}, ("LPT Air tube", "Part014", 1988, "stored"))}

ENGINEC class

{(i_{22}, ("CFM56-3C1", "E001", 1999, "installed", {i_8, i_9})),

(i_{23}, ("CFM56-3C1", "E002", 2000, "installed", {i_{10}, i_{11}})),

(i_{24}, ("PW4152", "E003", 1987, "installed" , {i_{12}})),

(i_{25}, ("PW4152", "E004", 1999, "installed", {i_{13}})),

(i_{26}, ("CF6-50C2", "E005", 1998, "installed", {i_{14}})),

(i_{27}, ("CF6-50C2", "E006", 1998, "installed", {i_{15}})),

(i_{28}, ("CF6-50C2", "E007", 1998, "installed", {i_{16}})),

(i_{29}, ("IAE V2528-D5", "E008", 2001, "installed", {i_{17}})),

(i_{30}, ("IAE V2528-D5", "E009", 1999, "installed", {i_{18}})),

(i_{31}, ("IAE V2528-D5", "E010", 2000, "preserved", {i_{19}})),

(i_{32}, ("CF6-50C2", "E011", 1998, "preserved", {i_{20}}))}

AIRPLANEC class

{(i_{33}, ("HS-AAA", "B737-400", "S001", 1999, [i_{22}, i_{23}])),

(i_{34}, ("HS-BBB", "A310", "S002", 2001, [i_{24}, i_{25}])),

(i_{35}, ("HS-CCC", "DC10", "S003", 1975, [i_{26}, i_{27}, i_{28}])),

(i_{36}, ("HS-DDD", "MD90", "S004", 1989, [i_{29}, i_{30}]))}

MARRIEDPERSONC class

{(i_1, (1, "Supa Rattana", 30-09-1970, "123 Sakorn Road, Bangkok", i_6),

(i_6, (6, "Suay Rattana", 26/11/1972, "123 Sakorn Road, Bangkok" , i_1)}

(i_5, (5, "Roong Amphon", 06-04-1975, "987 Keawsuay Road, Chiang-Mai", i_7)),

(i_7, (7, "Prasan Amphon", 15/03/1950, "987 Keawsuay Road, Chiang-Mai" , i_5))}

PILOTC class

{(i_1, (1, "Supa Rattana", 30-09-1970, "123 Sakorn Road, Bangkok", "P001")),

(i_2, (2, "Anong Sri-ngarm", 13-02-1956, "456 Sang-ngarm Road, Bangkok", "P002"))}

MECHANICC class

{(i_3, (3, "Suk Jaidee", 20-07-1967, "234 Baansuk Road, Nontaburi", "M003", {i_{35}})),

(i_4, (4, "Ram Pongsa", 04-06-1975, "567 Lamklong Road, Phuket", "M004", {i_{34}})),

(i_5, (5, "Roong Amphon", 06-04-1975, "987 Keawsuay Road, Chiang-Mai", "M005",

{i_{33}, i_{36}}))}

JOURNEYC class

{(i_{37}, (16/7/1999, 8.30, 10.00, "Bangkok", "Chiangmai", i_{33}, [i_1, i_2])),

(i_{38}, (17/10/1999, 11.00, 13.30, "Chiangmai", "Bangkok", i_{33}, [i_1, i_2])),

(i_{39}, (18/10/1999, 10.00, 11.30, "Bangkok", "Phuket", i_{34}, [i_1, i_2])),

(i_{40}, (20/11/1999, 13.00, 15.30, "Phuket", "Bangkok", i_{34}, [i_1, i_2]))}

APPENDIX C

C++ SOURCE CODES FOR THE IMPLEMENTATION OF MIC

This program is part of a thesis presented in partial fulfilment of the requirements for the degree of Master of Information Sciences in Infromation Systems, Massey University, Palmerston North, New Zealand. In this program, MIC (Kuckelberg, 1998) is implemented with a model of a POS extended by approaches (VO and RF) proposed in the thesis to support a query that could represent queries requiring associative access and involving multiple paths and inheritance hierarchies. In the program, MIC is implemented by the following coding techniques:

- No coding technique, i.e. sequences of OIDs are represented as they are,
- A SICF coding technique, i.e. sequences of OIDs are transformed to codes by a SICF coding technique, and
- A Start/Stop data coding technique, i.e. each OID in sequences of OIDs is dividually compressed by a Start/Stop data compression technique (Pigeon, 2001).

This program was implemented using a C++ programming language. The program is considered a prototype, which was developed with an exploratory prototyping approach (Bischofberger & Pomberger, 1992). The important factors considered in this approach are not the quality of the prototype implementation (e.g. reusability), but the functionality and the speed of development.

This is a proof of concept prototype. Neither the design nor the code in this prototype is intended to be the basis of the production system. We made a minimal use of OO features provided by the C++ programming language. The implementation relies

heavily on the Standard Template Library (STL) for implementing dynamically resizable collections.

As for the objectives of the thesis, this program

(1) Demonstrates that a model of a POS extended by proposed extensions is capable of supporting associative access in an OODBS
(2) Shows that data access approaches that are proposed to support navigational access can be used to support associative access in the extended POS regardless of paths or inheritance hierarchies involved with queries.
(3) Demonstrates that MIC can be made independent from a coding technique and a data compression technique helps by reducing the storage size required by MIC

The program is divided into fourteen modules as follows.

1. A "thesis" module: This module is the main part of the program. It displays all MICs implemented by three coding techniques. The module also displays the results of using MICs to support a query.
2. An "input" module: This module loads OIDs of storage objects and references between storage objects. This module also provides functions to display OIDs of storage objects and references.

   Classes:

   - class objectc: A class objectc for storage objects in the extended POS
   - class referencec: A class referencec for references in the extended POS

   Functions:

   - vector<objectc> load_object(): A function to create an array of OIDs of storage objects
     - Input: -
     - Output: A set of OIDs of storage objects and appended flags (SCF or VF)
   - void show_object(vector<objectc> pobject): A function to display a list of OIDs of storage objects
     - Input: A set of OIDs of storage objects and appended flags (SCF or VF)
     - Output: -

- vector<referencec> load_reference(): A function to create an array of references
  - Input: -
  - Output: A set of refereces between storage objects and appended flags (RF)
- void show_reference(vector<referencec> preference): A function to display a list of references
  - Input: A set of references between storage objects and appended flags (RF)
  - Output: -

3. A "count_ref" module: This module provides functions to count the number of storage objects that are referenced by or that reference the specific storage object. In other words, this module counts the number of outgoing and incoming references, i.e. multiple, single or none, of a storage object.

   Functions:

   - int count_fref(int poid): A function used to count the number of storage objects referenced by the specific storage object
     - Input: An OID of a storage object
     - Output: The number of storage objects that are referenced by the inputted storage object
   - int count_bref(int poid): A function used to count the number of storage objects referencing the specific storage object
     - Input: An OID of a storage object
     - Output: The number of storage objects that reference the inputted storage object

4. A "pigeon_code" module: This module creates codes compressed by a Start/Stop data compression technique (Pigeon, 2001). The module provides functions including a function to calculate an exponential function of 2, a function to encode by a Start/Stop data compression technique (Pigeon, 2001), and a function to decode by a Start/Stop data compression technique (Pigeon, 2001).

   Classes:

   - class pigeon_codec: A class pigeon_codec for codes caluculated by a Start/Stop data compression technique (Pigeon, 2001)

   Functions:

   - int cal_ex(int ppower): A function to calculate an exponential function of 2
     - Input: The number that 2 is to be multiplied by itself

- Output: The result of 2 to the power of the inputted number

- vector<int> encode_pigeon(int psource): A function to encode by a Start/Stop data compression technique (Pigeon, 2001)
  - Input: An OID of a storage object
  - Output: A sequence of bits that is encoded from the inputted OID by a Start/Stop data compression technique (Pigeon, 2001)
- int decode_pigeon(vector<int> pcode):A function to decode by a Start/Stop data compression technique (Pigeon, 2001)
  - Input: A sequence of bits
  - Output: An OID of a storage object that is decoded from the inputted sequence of bits by a Start/Stop data compression technique (Pigeon, 2001)

5. A "result" module: This module creates data access results. The module has a function to display data access results.

   Classes:

   - class access_resultc: A class access_resultc for data access results

   Functions:

   - void show_result(vector<access_resultc> paccess_result): A function to display data access results
     - Input: Data access results
     - Output: -

6. A "mic_no_coding" module: This module creates MIC implemented with no coding technique. The module also provides a function to display MIC implemented with no coding technique.

   Classes:

   - class mic_no_codingc: A class mic_no_codingc for MIC implemented with no coding technique

   Functions:

   - vector<mic_no_codingc> create_mic_no_coding(): A function to create MIC implemented with no coding technique
     - Input: -
     - Output: MIC implemented with no coding technique
   - void show_mic_no_coding(vector<mic_no_codingc> pmic_no_coding): A function to display MIC implemented with no coding technique

- Input: MIC implemented with no coding technique
- Output: -

7. A "mic_no_coding _navigation" module: This module provides functions for forward and backward navigation with MIC implemented with no coding technique.

   Functions:

   - vector<int> forward_mic_no_coding(int pbegin_oid, vector<mic_no_codingc> pmic_no_coding): A function for forward navigation with MIC implemented with no coding technique
     - Input: An OID of a storage object and MIC implemented with no coding technique
     - Output:

       By accessing MIC implemented with no coding technique:

       - If the inputted storage object does not reference any storage object, this function return an empty set.
       - If the inputted storage object references several storage objects, this function returns a set of OIDs of storage objects that are referenced by the inputted storage objects.
       - If the inputted storage object references only one storage object and that storage object references only one storage object and so on, this function returns a set of OIDs of storage objects that are referenced by single outgoing references beginning from the inputted storage objects.
   - vector<int> backward_mic_no_coding(int pbegin_oid, vector<mic_no_codingc> pmic_no_coding): A function for backward navigation with MIC implemented with no coding technique
     - Input: An OID of a storage object and MIC implemented with no coding technique
     - Output:

       By accessing MIC implemented with no coding technique:

       - If the inputted storage object is not referenced by any storage object, this function return an empty set.

- If the inputted storage object is referenced by several storage objects, this function returns a set of OIDs of storage objects that reference the inputted storage objects.
- If the inputted storage object is referenced by only one storage object and that storage object is referenced by only one storage object and so on, this function returns a set of OIDs of storage objects that have single outgoing references ending at the inputted storage objects.

8. A "mic_no_coding _support_Q" module: This module provides a function to support a query Q by MIC implemented with no coding technique.

   Functions:

   - vector<access_resultc> mic_no_coding_support_Q(vector<mic_no_codingc> pmic_no_coding): A function to support a query Q by MIC with no coding technique

     Q: Retreive details of journeys of an airplane HS-AAA including pilots of the journeys, details of engines and parts installed on the airplane, and details of mechanics who maintain the airplane.

     o Input: MIC implemented with no coding technique

     o Output: Access results for a query Q by using MIC implemented with no coding technique

9. A "mic_SICF" module: This module creates MIC implemented with a SICF coding technique. The module also provides a function to display MIC implemented with a SICF coding technique.

   Classes:

   - class mic_SICFc: A class mic_SICFc for MIC implemented with a SICF coding technique

   Functions:

   - vector<mic_SICFc> create_mic_SICF(): A function to create MIC implemented with a SICF coding technique

     o Input: -

     o Output: MIC implemented with a SICF coding technique

   - void show_mic_SICF(vector<mic_SICFc> pmic_SICF): A function to display MIC implemented with a SICF coding technique

     o Input: MIC implemented with a SICF coding technique

- Output: -

10. A "mic_SICF_navigation" module: This module provides functions for forward and backward navigation with MIC implemented with a SICF coding technique.

Functions:

- vector<int> forward_mic_SICF(int pbegin_oid, vector<mic_SICFc> pmic_SICF): A function for forward navigation with MIC implemented with a SICF coding technique
  - Input: An OID of a storage object and MIC implemented with a SICF coding technique
  - Output:

    By accessing MIC implemented with a SICF coding technique:
    - If the inputted storage object does not reference any storage object, this function return an empty set.
    - If the inputted storage object references several storage objects, this function returns a set of OIDs of storage objects that are referenced by the inputted storage objects.
    - If the inputted storage object references only one storage object and that storage object references only one storage object and so on, this function returns a set of OIDs of storage objects that are referenced by single outgoing references beginning from the inputted storage objects.
- vector<int> backward_mic_SICF(int pbegin_oid, vector<mic_SICFc> pmic_SICF): A function for backward navigation with MIC implemented with a SICF coding technique
  - Input: An OID of a storage object and MIC implemented with a SICF coding technique
  - Output:

    By accessing MIC implemented with a SICF coding technique:
    - If the inputted storage object is not referenced by any storage object, this function return an empty set.
    - If the inputted storage object is referenced by several storage objects, this function returns a set of OIDs of storage objects that reference the inputted storage objects.

- If the inputted storage object is referenced by only one storage object and that storage object is referenced by only one storage object and so on, this function returns a set of OIDs of storage objects that have single outgoing references ending at the inputted storage objects.

11. A "mic_SICF_support_Q" module: This module provides a function to support a query Q by MIC implemented with a SICF coding technique.

    Functions:

    - vector<access_resultc> mic_SICF_support_Q(vector<mic_SICFc> pmic_SICF);
      A function to support a query Q by MIC with a SICF coding technique
      Q: Retreive details of journeys of an airplane HS-AAA including pilots of the journeys, details of engines and parts installed on the airplane, and details of mechanics who maintain the airplane.
      - Input: MIC implemented with a SICF coding technique
      - Output: Access results for a query Q by using MIC implemented with a SICF coding technique

12. A "mic_start_stop" module: This module creates MIC implemented with a Start/Stop data compression technique (Pigeon, 2001). The module also provides a function to display MIC implemented with a Start/Stop data compression technique (Pigeon, 2001).

    Classes:

    - class mic_start_stopc: A class mic_start_stopc for MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)

    Functions:

    - vector<mic_start_stopc> create_mic_start_stop(): A function to create MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
      - Input: -
      - Output: MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
    - void show_mic_start_stop(vector<mic_start_stopc> pmic_start_stop): A function to display MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
      - Input: MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)

- Output: -

13. A "mic_start_stop_navigation" module: This module provides functions for forward and backward navigation with MIC implemented with a Start/Stop data compression technique (Pigeon, 2001).

    Functions:

    - vector<int> forward_mic_start_stop(int pbegin_oid, vector<mic_start_stopc> pmic_start_stop): A function for forward navigation by MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
        - Input: An OID of a storage object and MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
        - Output:

          By accessing MIC implemented with a Start/Stop data compression technique (Pigeon, 2001):
            - If the inputted storage object does not reference any storage object, this function return an empty set.
            - If the inputted storage object references several storage objects, this function returns a set of OIDs of storage objects that are referenced by the inputted storage objects.
            - If the inputted storage object references only one storage object and that storage object references only one storage object and so on, this function returns a set of OIDs of storage objects that are referenced by single outgoing references beginning from the inputted storage objects.
    - vector<int> backward_mic_start_stop(int pbegin_oid, vector<mic_start_stopc> pmic_start_stop): A function for backward navigation by MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
        - Input: An OID of a storage object and MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
        - Output:

          By accessing MIC implemented with a Start/Stop data compression technique (Pigeon, 2001):
            - If the inputted storage object is not referenced by any storage object, this function return an empty set.

- If the inputted storage object is referenced by several storage objects, this function returns a set of OIDs of storage objects that reference the inputted storage objects.
- If the inputted storage object is referenced by only one storage object and that storage object is referenced by only one storage object and so on, this function returns a set of OIDs of storage objects that have single outgoing references ending at the inputted storage objects.

14. A "mic_start_stop_support_Q" module: This module provides a function to support a query Q by MIC implemented with a Start/Sop data compression technique (Pigeon, 2001).

    Functions:

    - vector<access_resultc> mic_start_stop_support_Q(vector<mic_start_stopc> pmic_start_stop): A function to support a query Q by MIC with a Start/Sop data compression technique (Pigeon, 2001)

      Q: Retreive details of journeys of an airplane HS-AAA including pilots of the journeys, details of engines and parts installed on the airplane, and details of mechanics who maintain the airplane.

      o Input: MIC implemented with a Start/Sop data compression technique (Pigeon, 2001)

      o Output: Access results for a query Q by using MIC implemented with a Start/Sop data compression technique (Pigeon, 2001)

The following diagram illustrates how modules in the program are related to each other.

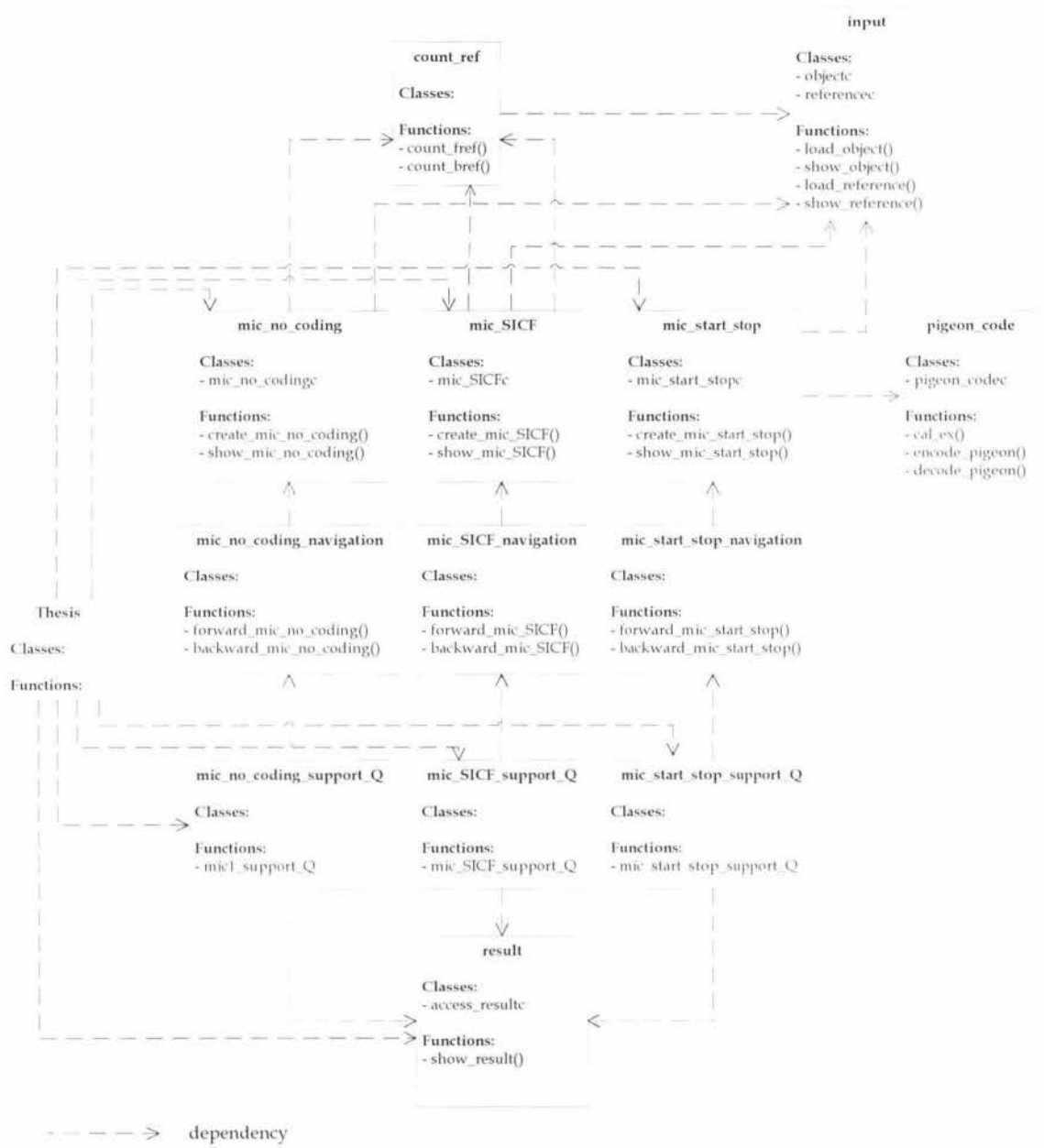


Figure C.1 Modules

```
/* This program is part of a thesis presented in partial fulfilment of the requirements for
 * the degree of Master of Information Sciences in Infromation Systems, Massey University,
 * Palmerston North,New Zealand.
 *
 * In this program MIC (Kuckelberg, 1998 ) is implemented with a model of a POS extended by
 * approaches (VO and RF) proposed in the thesis to support a query that could represent queries that
 * require associative access that includes navigational access to logical objects of classes
 * in multiple paths or inheritance hierarchies. In the program, MIC is implemented by
 * the following coding techniques.
 * - No coding technique, i.e. a list of OIDs of storage objects are represented as they are,
 * - A SICF coding technique, and
 * - A Start/Stop data coding technique (Pigeon, 2001)
 *
 * Programmer: Weena Nusdin
 * Last Modified: 6 February 2004
 */

#include<iostream>
#include<vector>
using namespace std;

#include "input.h"
#include "mic_no_coding.h"
#include "result.h"
#include "mic_no_coding_support_Q.h"
#include "mic_SICF.h"
#include "mic_SICF_support_Q.h"
#include "mic_start_stop.h"
#include "mic_start_stop_support_Q.h"

int main(){
        vector<objectc> object;                         //Create an array of storage objects
        object = load_object();                         //Load OIDs of storage objects
        show_object(object);                            //Display a list of storage objects

        vector<referencec> reference;                   //Create an array of referecnes
        reference = load_reference();                   //Load references
        show_reference(reference);                      //Display a list of references

        //Create arrays of access results
        vector<access_resultc> access_result_no_coding, access_result_SICF, access_result_start_stop;

        vector<mic_no_codingc> mic_no_coding;                   //Create MIC implemented by no coding
technique
        mic_no_coding = create_mic_no_coding();
        show_mic_no_coding(mic_no_coding);              //Display MIC implemented by no coding technique

       vector<mic_SICFc> mic_SICF;                      //Create MIC implemented by a SICF coding technique
       mic_SICF = create_mic_SICF();
       show_mic_SICF(mic_SICF);                         //Display MIC implemented by a SICF coding technique

       vector<mic_start_stopc> mic_start_stop;                  //Create MIC implemented by a Start/Stop
                                                        //data compression technique (Pigeon, 2001)
       mic_start_stop = create_mic_start_stop();
        show_mic_start_stop(mic_start_stop);                    //Display MIC implemented by a Start/Stop
                                                        //data compression technique (Pigeon, 2001)

        //Support a query Q by MIC implemented by no coding technique
        access_result_no_coding = mic_no_coding_support_Q(mic_no_coding);
        cout << "\nThe result for a query Q supported by MIC implemented with no coding technique\n";
        //Display results of supporting a query Q by MIC implemented by no coding technique
        show_result(access_result_no_coding);

        //Support a query Q by MIC implemented by a SICF coding technique
        access_result_SICF = mic_SICF_support_Q(mic_SICF);
        cout << "\nThe result for a query Q supported by MIC implemented with a SICF coding technique\n";
        //Display results of supporting a query Q by MIC implemented by a SICF coding technique
        show_result(access_result_SICF);

        //Support a query Q by MIC implemented by a Start/Stop data compression technique (Pigeon, 2001)
        access_result_start_stop = mic_start_stop_support_Q(mic_start_stop);
        cout << "\nThe result for a query Q supported by MIC implemented with a Start/Stop data
compression technique\n";
        //Display results of supporting a query Q by MIC implemented by a Start/Stop data compression
technique (Pigeon, 2001)
```

```
        show_result(access_result_start_stop);

        return 0;
}
```

```
/* ---input.h---
 * - Classes objectc and referencec for storage objects and referencec in the extended POS.
 * - Functions to load and display storage objects and referencec in the extended POS.
 *
 * Programmer: Weena Nusdin
 * Last Modified: 6 February 2004
 */

//A class objectc for storage objects in the extended POS
class objectc{
public:
        int oid;
        int condition_flag;

        objectc(int poid, int pcondition_flag);
};

//A class referencec for references in the extended POS
class referencec{
public:
        int begin_oid;
        int end_oid;
        int reference_flag;

        referencec(int pbegin_oid, int pend_oid, int preference_flag);
};

vector<objectc> load_object();              //A function to create an array of OIDs of storage objects
void show_object(vector<objectc> pobject);  //A function to display a list of OIDs of storage objects

vector<referencec> load_reference();                //A function to create an array of references
void show_reference(vector<referencec> preference); //A function to display a list of references
```

```
/* ---input.cpp---
 * This program loads OIDs of storage objects and references between storge objects.
 * The program also provides functions to display OIDs of storage objects and
 * references in the extended POS.
 *
 * Programmer: Weena Nusdin
 * Last Modified: 6 February 2004
 */

#include<iostream>
#include<fstream>
#include<vector>
using namespace std;

#include "input.h"

//A constructor fuction for objects of a class objectc for storage objects
objectc::objectc(int poid, int pcondition_flag){
        oid = poid;
        condition_flag = pcondition_flag;
}

//A function to load OIDs of storage objects
vector<objectc> load_object(){
        int poid, pcondition_flag;
        vector<objectc> object;
        ifstream object_file;
        object_file.open("/home/wnusdin/program_files/thesis_object1.txt");
        while(!object_file.eof()){
                object_file >> poid;
                object_file.ignore(1);
                object_file >> pcondition_flag;
                object_file.ignore(1);
                object.push_back(objectc(poid, pcondition_flag));
        }
        object_file.close();
        return object;
}

//A function to display a list of OIDs of storage objects
void show_object(vector<objectc> pobject){
        cout << "-- All objects created --\n";
        cout << "(OID, condition flag) \n";
        vector<objectc>::iterator s_o = pobject.begin();
        while (s_o != pobject.end()){
                cout << s_o->oid << "  " << s_o->condition_flag << "\n";
                s_o++;
        }//end while
        cout << "\n";
}

//A constructor fuction for objects of a class referencec for references in the extended POS
referencec::referencec(int pbegin_oid, int pend_oid, int preference_flag){
        begin_oid = pbegin_oid;
        end_oid = pend_oid;
        reference_flag = preference_flag;
}

//A function to load references
vector<referencec> load_reference(){
        int pbegin_oid, pend_oid, preference_flag;
        vector<referencec> reference;

        ifstream reference_file;
        reference_file.open("/home/wnusdin/program_files/thesis_reference1.txt");
        while(!reference_file.eof()){
                reference_file >> pbegin_oid;
                reference_file.ignore(1);
                reference_file >> pend_oid;
                reference_file.ignore(1);
                reference_file >> preference_flag;
                reference_file.ignore(1);
                reference.push_back(referencec(pbegin_oid, pend_oid, preference_flag));
        }
```

```
        reference_file.close();
        return reference;
}

//A function to display references
void show_reference(vector<referencec> preference){
        cout << "-- All references created -- \n";
        cout << "(begin OID, end OID, reference flag) \n";
        vector<referencec>::iterator s_r = preference.begin();
        while (s_r != preference.end()){
                cout << s_r->begin_oid << "  " << s_r->end_oid << "  " << s_r->reference_flag << "\n";
                s_r++;
        }//end while
        cout << "\n";
}
```

```
/* ---count_ref.h---
 * Functions used to count the number of storage objects that are referenced by or reference
 * to the specific storage object.
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

//A function used to count the number of storage objects referenced by the specific storage object
int count_fref(int poid);

//A function used to count the number of storage objects referencing to the specific storage object
int count_bref(int poid);
```

```
/* ---count_ref.cpp---
 * Functions used to count the number of storage objects that are referenced by or reference
 * to the specific storage object.
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

#include<iostream>
#include<vector>
using namespace std;

#include "count_ref.h"
#include "input.h"

//A function used to count the number of storage objects referenced by the specific storage object
int count_fref(int poid){
        int fref_count = 0;
        vector<referencec> reference;
        reference.clear();
        reference = load_reference();
        vector<referencec>::iterator c_fr = reference.begin();
        while (c_fr != reference.end()){
                if (c_fr->begin_oid == poid){
                        fref_count++;
                }
                c_fr++;
        }
        return fref_count;
}

//A function used to count the number of storage objects referencing to the specific storage object
int count_bref(int poid){
        int bref_count = 0;
       vector<referencec> reference;
       reference.clear();
       reference = load_reference();
        vector<referencec>::iterator c_br = reference.begin();
        while (c_br != reference.end()){
                if (c_br->end_oid == poid){
                        bref_count++;
                }
                c_br++;
        }
        return bref_count;
}
```

```
/* ---pigeon_code.h---
 * A class pigeon_codec for codes caluculated by a data compression technique (Pigeon, 2001)
 * - A function to calculate an exponential function of 2
 * - Functions to encode and encode by a Start/Stop data compression technique (Pigeon, 2001)
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

#include<iostream>
#include<vector>

//A class pigeon_codec for codes caluculated by a data compression technique (Pigeon, 2001)
class pigeon_codec{
public:
        vector<int> pigeon_code;

         pigeon_codec(vector<int> ppigeon_code);
};

// A function to calculate an exponential function of 2
int cal_ex(int ppower);

//A function to encode by a Start/Stop data compression technique (Pigeon, 2001)
vector<int> encode_pigeon(int psource);

//A function to decode by a Start/Stop data compression technique (Pigeon, 2001)
int decode_pigeon(vector<int> pcode);
```

```
/* ---pigeon_code.cpp---
 * This program creates codes compressed by a Start/Stop data compression technique (Pigeon, 2001)
 * The program provides the following functions
 * - A function to calculate an exponential function of 2
 * - A function to encode by a Start/Stop data compression technique (Pigeon, 2001)
 * - A function to decode by a Start/Stop data compression technique (Pigeon, 2001)
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

using namespace std;

#include "pigeon_code.h"

/*A constructor fuction for objects of a class pigeon_codec for a code compressed by a Start/Stop
 * data compression technique (Pigeon,2001)
 */
pigeon_codec::pigeon_codec(vector<int> ppigeon_code){
        pigeon_code = ppigeon_code;
}

//A function to calculate an exponential function of 2
int cal_ex(int ppower){
        int result = 1;

        for (int i = 1; i <= ppower; i++)
        { result = result*2;}
        return result;
}

//A function to encode an OID by a Start/Stop data compresstion technique (Pigeon, 2001)
vector<int> encode_pigeon(int psource){
        int step = 0;
        int m = 2;
        int trans, s;
        int lower = 0;
        int upper = cal_ex(m*(step+1));
        int prefix = 0;
        int diff;

        while (psource>=upper){
                step++;
                lower = upper;
                trans = cal_ex(m*(step+1));
                upper = lower + trans;
        }
        diff = psource - lower;
        s = 2*(step+1) + (step+1);
        vector<int> trans_pigeon(s, 5);
        for (int i=s-1; i>=s-(2*(step+1)); i--){
                trans_pigeon[i] = diff%2;
                diff = diff/2;
        }
        if (step > 0){
                for (int i = 1; i<step+1; i++){
                        prefix = prefix + cal_ex(i);
                }
        }
        for (int i = s-(2*(step+1))-1; i >= 0; i--){
                trans_pigeon[i] = (prefix%2);
                prefix = prefix/2;
        }
        return trans_pigeon;
}

//A function to decode a code compressed by a Start/Stop data compresstion technique (Pigeon, 2001)
int decode_pigeon(vector<int> pcode){
        vector<int> code;
        int step;
        int base_10 = 0;
        int lower = 0;

        code = pcode;
        for (unsigned int i=0; i<code.size(); i++){
```

```
                if (code[i] == 0){
                        step = i;
                        break;
                }
        }
        for (int i= 1; i<=step; i++){
                lower = lower+cal_ex(2*i);
        }
        for (int i=0; i<2*(step+1); i++){
                base_10 = base_10 + code[step+1+i]*cal_ex(2*(step+1)-1-i);
        }
        base_10 = base_10+lower;
        return base_10;
}
```

```
/* ---result.h---
 *  - A class access_resultc for data access results
 *  - A function to display data access results
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

//A class access_resultc for data access results
class access_resultc{
public:
        char begin_storage_class[50];
        char end_storage_class[50];
        int begin_oid;
        vector<int> end_oid;

        access_resultc(char pbegin_storage_class[50], char pend_storage_class[50], int pbegin_oid, vector
<int> pend_oid);
};

//A function to display data access results
void show_result(vector<access_resultc> paccess_result);
```

```
/* ---result.cpp---
 * This program creates data access results. The program has a function to display data access results.
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

#include<iostream>
#include<vector>
using namespace std;

#include "result.h"

//A constructor function of a class access_resultc for data access results
access_resultc::access_resultc(char pbegin_storage_class[50], char pend_storage_class[50], int pbegin_oid,
vector<int> pend_oid){
        strcpy(begin_storage_class, pbegin_storage_class);
        strcpy(end_storage_class, pend_storage_class);
        begin_oid = pbegin_oid;
        end_oid = pend_oid;
}

//A function to display data access results
void show_result(vector<access_resultc> paccess_result){
        vector<int> show_result;
        vector<access_resultc> access_result;
        access_result = paccess_result;
        vector<access_resultc>::iterator s_result = access_result.begin();
        while (s_result != access_result.end()){
                cout << s_result->begin_storage_class << ": " ;
                cout << s_result->begin_oid << " references "
                << s_result->end_storage_class << " " ;
                show_result = s_result->end_oid;
                vector<int>::iterator s_eoid = show_result.begin();
                cout << "[ ";
                while (s_eoid != show_result.end()){
                        cout << *s_eoid << " ";
                        s_eoid++;
                }//end while
                cout << "]\n";
                s_result++;
        }//end while
}
```

```
/* ---mic_no_coding.h---
 * - A class mic_no_codingc for MIC implemented with no coding technique
 * - A function to create MIC implemented with no coding technique
 * - A function to display MIC implemented with no coding technique
 *
 * Programmer: Weena Nusdin
 * Last Modified: 6 February 2004
 */

//A class mic_no_codingc for MIC implemented with no coding technique
class mic_no_codingc{
public:
        int mic_no_coding_oid;
        char ffanout_type;
        char bfanout_type;
        vector<int> for_oid;
        vector<int> bac_oid;
        int fcode_size, bcode_size;

        mic_no_codingc(int pmic_no_coding_oid, char pffanout_type, vector<int> pfor_oid, char
pbfanout_type, vector<int> pbac_oid, int pfcode_size, int pbcode_size);
};

//A function to create MIC implemented with no coding technique
vector<mic_no_codingc> create_mic_no_coding();

//A function to display MIC implemented with no coding technique
void show_mic_no_coding(vector<mic_no_codingc> pmic_no_coding);
```

```
/* ---mic_no_coding.cpp---
 * This program creates MIC implemented with no coding technique. The program also provides
 * functions to display MIC implemented with no coding technique.
 *
 * Programmer: Weena Nusdin
 * Last Modified: 6 February 2004
 */

#include<iostream>
#include<vector>
using namespace std;

#include "mic_no_coding.h"
#include "input.h"
#include "count_ref.h"

//A constructor function of a class mic_no_codingc for MIC implemented with no coding technique
mic_no_codingc::mic_no_codingc(int pmic_no_coding_oid, char pffanout_type, vector<int> pfor_oid, char
pbfanout_type, vector<int> pbac_oid, int pfcode_size, int pbcode_size){
                mic_no_coding_oid = pmic_no_coding_oid;
                ffanout_type = pffanout_type;
                for_oid = pfor_oid;
                bfanout_type = pbfanout_type;
                bac_oid = pbac_oid;
                fcode_size = pfcode_size;
                bcode_size = pbcode_size;
}

//A function to create MIC implemented with no coding technique
vector<mic_no_codingc> create_mic_no_coding(){
        vector<referencec> reference;
        reference.clear();
        reference = load_reference();
        vector<objectc> object;
        object.clear();
        object = load_object();
        vector<mic_no_codingc> mic_no_coding;
        int pmic_no_coding_oid, pfcode_size, pbcode_size;
        char pffanout_type, pbfanout_type;
        vector<int> pfor_oid, pbac_oid;
        vector<int> same_f, same_b;
        int r_4, r_10, o_run_f, o_run_b;

        vector<objectc>::iterator r_1 = object.begin();
        while (r_1 != object.end()){
                pfor_oid.clear();
                pbac_oid.clear();
                pmic_no_coding_oid = r_1->oid;
                if (count_fref(pmic_no_coding_oid) == 0){    //no forward reference
                        pffanout_type = '-';
                }//end if
                else if (count_fref(pmic_no_coding_oid) > 1){         //multiple forward references
                        pffanout_type = 'm';
                        vector<referencec>::iterator r_2 = reference.begin();
                        while (r_2 != reference.end()){
                                if (r_2->begin_oid == pmic_no_coding_oid && r_2->reference_flag != 0){
                                        pfor_oid.push_back(r_2->end_oid);
                                        pfor_oid.push_back(r_2->reference_flag);
                                        vector<objectc>::iterator r_3 = object.begin();
                                        while (r_3 != object.end()){
                                                if (r_3->oid == r_2->end_oid && r_3->condition_flag
!= 0){
                                                        pfor_oid.push_back(r_3->condition_flag);
                                                        break;
                                                }//end if
                                                r_3++;
                                        }//end while
                                }//end if
                                else if (r_2->begin_oid == pmic_no_coding_oid && r_2->reference_flag
== 0){
                                        pfor_oid.push_back(r_2->end_oid);
                                        vector<objectc>::iterator r_3 = object.begin();
                                        while (r_3 != object.end()){
                                                if (r_3->oid == r_2->end_oid && r_3->condition_flag
!= 0){
```

```
                                                pfor_oid.push_back(r_3->condition_flag);
                                                break;
                                        }//end if
                                        r_3++;
                                }//end while
                        }//end else if
                        r_2++;
                }//end while
        }//end else if
        else if (count_fref(pmic_no_coding_oid) == 1){          //single forward reference
                pffanout_type = 's';
                same_f.clear();
                o_run_f = pmic_no_coding_oid;
                r_4 = 0;
                while (r_4 == 0){
                        same_f.push_back(o_run_f);
                        vector<referencec>::iterator r_5 = reference.begin();
                        while (r_5 != reference.end()){
                                if (r_5->begin_oid == o_run_f && r_5->reference_flag != 0){
                                        o_run_f = r_5->end_oid;
                                        pfor_oid.push_back(o_run_f);
                                        pfor_oid.push_back(r_5->reference_flag);
                                        vector<objectc>::iterator r_6 = object.begin();
                                        while (r_6 != object.end()){
                                                if (r_6->oid == o_run_f && r_6->
condition_flag != 0){
                                                        pfor_oid.push_back(r_6->
condition_flag);
                                                        break;
                                                }//end if
                                                r_6++;
                                        }//end while
                                        break;
                                }//end if
                                else if (r_5->begin_oid == o_run_f && r_5->reference_flag ==
0){
                                        o_run_f = r_5->end_oid;
                                        pfor_oid.push_back(o_run_f);
                                        vector<objectc>::iterator r_6 = object.begin();
                                        while (r_6 != object.end()){
                                                if (r_6->oid == o_run_f && r_6->
condition_flag != 0){
                                                        pfor_oid.push_back(r_6->
condition_flag);
                                                        break;
                                                }//end if
                                                r_6++;
                                        }//end while
                                        break;
                                }//end else if
                                r_5++;
                        }//end while
                        if(count_fref(o_run_f) != 1){
                                r_4 = 1;
                                break;
                        }//end if
                        else {
                                vector<int>::iterator r_7 = same_f.begin();
                                while(r_7 != same_f.end()){
                                        if (*r_7 == o_run_f){
                                                r_4 = 1;
                                                break;
                                        }//end if
                                        r_7++;
                                }//end while
                        }//end else
                }//end while
        }//end else if
        if (count_bref(pmic_no_coding_oid) == 0){     //no backward reference
                pbfanout_type = '-';
        }//end if
        else if (count_bref(pmic_no_coding_oid) > 1){          //multiple backward reference
                pbfanout_type = 'm';
                vector<referencec>::iterator r_8 = reference.begin();
                while (r_8 != reference.end()){
```

```
                                if (r_8->end_oid == pmic_no_coding_oid && r_8->reference_flag !=0){
                                        pbac_oid.push_back(r_8->begin_oid);
                                        pbac_oid.push_back(r_8->reference_flag);
                                        vector<objectc>::iterator r_9 = object.begin();
                                        while (r_9 != object.end()){
                                                if (r_9->oid == r_8->begin_oid && r_9->
condition_flag != 0){
                                                        pbac_oid.push_back(r_9->condition_flag);
                                                        break;
                                                }//end if
                                                r_9++;
                                        }//end while
                                }//end if
                                else if (r_8->end_oid == pmic_no_coding_oid && r_8->reference_flag
==0){
                                        pbac_oid.push_back(r_8->begin_oid);
                                        vector<objectc>::iterator r_9 = object.begin();
                                        while (r_9 != object.end()){
                                                if (r_9->oid == r_8->begin_oid && r_9->
condition_flag != 0){
                                                        pbac_oid.push_back(r_9->condition_flag);
                                                        break;
                                                }//end if
                                                r_9++;
                                        }//end while
                                }//end else if
                                r_8++;
                        }//end while
                }//end else if
                else if (count_bref(pmic_no_coding_oid) == 1){        //single backward reference
                        pbfanout_type = 's';
                        same_b.clear();
                        o_run_b = pmic_no_coding_oid;
                        r_10 = 0;
                        while (r_10 == 0){
                                same_b.push_back(o_run_b);
                                vector<referencec>::iterator r_11 = reference.begin();
                                while (r_11 != reference.end()){
                                        if (r_11->end_oid == o_run_b && r_11->reference_flag !=0){
                                                o_run_b = r_11->begin_oid;
                                                pbac_oid.push_back(o_run_b);
                                                pbac_oid.push_back(r_11->reference_flag);
                                                vector<objectc>::iterator r_12 = object.begin();
                                                while (r_12 != object.end()){
                                                        if (r_12->oid == o_run_b && r_12->
condition_flag != 0){
                                                                pbac_oid.push_back(r_12->
condition_flag);
                                                                break;
                                                        }//end if
                                                        r_12++;
                                                }//end while
                                                break;
                                        }//end if
                                        else if (r_11->end_oid == o_run_b && r_11->reference_flag
==0){
                                                o_run_b = r_11->begin_oid;
                                                pbac_oid.push_back(o_run_b);
                                                vector<objectc>::iterator r_12 = object.begin();
                                                while (r_12 != object.end()){
                                                        if (r_12->oid == o_run_b && r_12->
condition_flag != 0){
                                                                pbac_oid.push_back(r_12->
condition_flag);
                                                                break;
                                                        }//end if
                                                        r_12++;
                                                }//end while
                                                break;
                                        }//end else if
                                        r_11++;
                                }//end while
                                if(count_bref(o_run_b) != 1){
                                        r_10 = 1;
                                        break;
```

```
                                        }//end if
                                        else {
                                                vector<int>::iterator r_13 = same_b.begin();
                                                while(r_13 != same_b.end()){
                                                        if (*r_13 == o_run_b){
                                                                r_10 = 1;
                                                                break;
                                                        }//end if
                                                        r_13++;
                                                }//end while
                                        }//end else
                                }//end while
                        }//end else if
                        pfcode_size = pfor_oid.size()*16;
                        pbcode_size = pbac_oid.size()*16;
                        mic_no_coding.push_back(mic_no_codingc(pmic_no_coding_oid, pffanout_type, pfor_oid,
pbfanout_type, pbac_oid, pfcode_size, pbcode_size));
                        r_1++;
                }//end while
                return mic_no_coding;
}

//A function to display MIC implemented with no coding technique
void show_mic_no_coding(vector<mic_no_codingc> pmic_no_coding){
                vector<mic_no_codingc> mic_no_coding;
                mic_no_coding.clear();
                mic_no_coding = pmic_no_coding;
                vector<int> show_f_oid, show_b_oid;
                cout << "-- MIC with no coding technique -- \n";
                cout << "(OID, forward fanout type, referenced OIDs, backward fanout type, referencing OIDs, bits
for forward code, bits for backward codes)\n";
                vector<mic_no_codingc>::iterator s_m1 = mic_no_coding.begin();
                while (s_m1 != mic_no_coding.end()){
                        cout << s_m1->mic_no_coding_oid << " " << s_m1->ffanout_type << " [ ";
                show_f_oid = s_m1->for_oid;
                        vector<int>::iterator s_foid = show_f_oid.begin();
                        while (s_foid != show_f_oid.end()){
                                cout << *s_foid << " ";
                                s_foid++;
                        }//end while
                        cout << "] ";
                        cout << s_m1->bfanout_type << " [ ";
                        show_b_oid = s_m1->bac_oid;
                        vector<int>::iterator s_boid = show_b_oid.begin();
                        while (s_boid != show_b_oid.end()){
                                cout << *s_boid << " ";
                                s_boid++;
                        }//end while
                        cout << "] ";
                        cout << s_m1->fcode_size << " ";
                        cout << s_m1->bcode_size << " \n";
                        s_m1++;
                }//end while
                cout << "\n";
}
```

```
/* ---mic_no_coding_navigation.h---
 * Functions to navigate with MIC implemented with no coding technique
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */


#include "mic_no_coding.h"
//A function for forward navigation with MIC implemented with no coding technique
vector<int> forward_mic_no_coding(int pbegin_oid, vector<mic_no_codingc> pmic_no_coding);

//A function for backward navigation with MIC implemented with no coding technique
vector<int> backward_mic_no_coding(int pbegin_oid, vector<mic_no_codingc> pmic_no_coding);
```

```
/* ---mic_no_coding_navigation.cpp---
 * Functions to navigate with MIC implemented with no coding technique
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

#include<iostream>
#include<vector>
using namespace std;

#include "mic_no_coding_navigation.h"

//A function for forward navigation with MIC implemented with no coding technique
vector<int> forward_mic_no_coding(int pbegin_oid, vector<mic_no_codingc> pmic_no_coding){
        vector<mic_no_codingc> mic_no_coding;
        mic_no_coding = pmic_no_coding;
        int begin_oid;
        vector<int> end_oid;

        begin_oid = pbegin_oid;
        vector<mic_no_codingc>::iterator i = mic_no_coding.begin();
        while (i != mic_no_coding.end()){
                if (begin_oid == i->mic_no_coding_oid){
                        end_oid = i->for_oid;
                }//end if
                i++;
        }//end while
        return end_oid;
}

//A function for backward navigation with MIC implemented with no coding technique
vector<int> backward_mic_no_coding(int pbegin_oid, vector<mic_no_codingc> pmic_no_coding){
       vector<mic_no_codingc> mic_no_coding;
       mic_no_coding = pmic_no_coding;
        int begin_oid;
        vector<int> end_oid;

        begin_oid = pbegin_oid;
        vector<mic_no_codingc>::iterator i = mic_no_coding.begin();
        while (i != mic_no_coding.end()){
                if (begin_oid == i->mic_no_coding_oid){
                        end_oid = i->bac_oid;
                }//end if
                i++;
        }//end while
        return end_oid;
}
```

```
/* ---mic_no_coding_support_Q.h---
 * A function to a query Q by MIC implemented with no coding technique
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

/* A function to support a query Q by MIC with no coding technique
 * Q: Retreive details of journeys of an airplane HS-AAA including pilots of the journeys, details
 *    of engines and parts installed on the airplane, and details of mechanics who maintain
 *    the airplane.
 */
vector<access_resultc> mic_no_coding_support_Q(vector<mic_no_codingc> pmic_no_coding);
```

```
/* ---mic_no_coding_support_Q.cpp---
 * A function to a query Q by MIC implemented with no coding technique
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

#include<iostream>
#include<list>
#include<vector>
using namespace std;

#include "result.h"
#include "mic_no_coding_navigation.h"
#include "mic_no_coding_support_Q.h"

/* A function to support a query Q by MIC with no coding technique
 * Q: Retreive details of journeys of an airplane HS-AAA including pilots of the journeys, details
 *    of engines and parts installed on the airplane, and details of mechanics who maintain
 *    the airplane.
 */
vector<access_resultc> mic_no_coding_support_Q(vector<mic_no_codingc> pmic_no_coding){
        vector<access_resultc> access_result;        //Create an array of access results
        access_result.clear();

        vector<mic_no_codingc> mic_no_coding;
        mic_no_coding = pmic_no_coding;

        int VO;                                      //Define a Value Object(VO)
        char Value_Object[50];
        vector<int> temp, tempv, airplane_OID, engine_OID, part_OID;
        vector<int> partlc_OID, partonc_OID, mechanic_OID, journey_OID, person_OID, pilot_OID;
        list<int> templ, templ, temp2;

        VO = 8;                                      //An OID of a VO representing a value of
                                                     //an attribute RegistrationName, HS-AAA
        strcpy(Value_Object, "Value Object (HS-AAA)");
        access_result.clear();

        /* Retreive a storage object of a storage class AIRPLANEC whose value of an attribute
         * RegistrationName is HS-AAA by forward navigation from the corresponding VO.
         */
        airplane_OID = forward_mic_no_coding(VO, mic_no_coding);
        access_result.push_back(access_resultc(Value_Object, "Airplane Storage OID", VO, airplane_OID));

       /* Retreive storage objects of a storage class ENGINEC by forward navigation from
         * the corresponding storage object of a storage class AIRPLANEC
         */
        engine_OID = forward_mic_no_coding(airplane_OID[0], mic_no_coding);

        access_result.push_back(access_resultc("Airplane Storage OID", "Engine Storage OID", airplane_OID
[0], engine_OID ));
        templ.clear();
        for (unsigned int i=0; i<engine_OID.size(); i++){
                templ.push_back(engine_OID[i]);
        }
        templ.sort();
        templ.unique();
        engine_OID.clear();
        list<int>::iterator t = templ.begin();
        while(t != templ.end()){
                engine_OID.push_back(*t);
                t++;
        }
        temp.clear();
        templ.clear();
        temp2.clear();

       /* Retreive storage objects of storage classes LIFECYCLEPARTC and ONCONDITIONPARTC
         * by forward navigation from the corresponding storage objects of a storage
         * class ENGINEC
         */
        for (unsigned int i=0; i<engine_OID.size(); i++){
                partlc_OID.clear();
                partonc_OID.clear();
```

```
                temp.clear();
                temp = forward_mic_no_coding(engine_OID[i], mic_no_coding);
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 6){
                                partlc_OID.push_back(temp[j-1]);
                                temp1.push_back(temp[j-1]);
                        }
                        else if (temp[j] == 7){
                                partonc_OID.push_back(temp[j-1]);
                                temp2.push_back(temp[j-1]);
                        }
                }
                access_result.push_back(access_resultc("Engine Storage OID", "Life Cycle Part Storage
OID", engine_OID[i], partlc_OID ));
                access_result.push_back(access_resultc("Engine Storage OID", "On Condition Part Storage
OID", engine_OID[i], partonc_OID ));
        }
        temp1.sort();
        temp1.unique();
        temp2.sort();
        temp2.unique();
        temp.clear();
        list<int>::iterator t1 = temp1.begin();
        while(t1 != temp1.end()){
                temp.push_back(*t1);
                t1++;
        }

       /* Retreive storage objects of a storage class PARTC by forward navigation from
        * the corresponding storage objects of a storage class LIFECYCLEPARTC
        */
        for (unsigned int i=0; i<temp.size(); i++){
                tempv.clear();
                tempv = forward_mic_no_coding(temp[i], mic_no_coding);
                part_OID.clear();
                for (unsigned int j=0; j<tempv.size(); j++){
                        if (tempv[j] == 2){
                                part_OID.push_back(tempv[j-1]);
                                access_result.push_back(access_resultc("Life Cycle Part Storage OID",
"Part Storage OID", temp[i], part_OID));
                        }
                }
        }

        temp.clear();
        list<int>::iterator t4 = temp2.begin();
        while(t4 != temp2.end()){
                temp.push_back(*t4);
                t4++;
        }
        /* Retreive storage objects of a storage class PARTC by forward navigation from
         * the corresponding storage objects of a storage class ONCONDITIONPARTC
         */
        for (unsigned int i=0; i<temp.size(); i++){
                tempv.clear();
                tempv = forward_mic_no_coding(temp[i], mic_no_coding);
                part_OID.clear();
                for (unsigned int j=0; j<tempv.size(); j++){
                        if (tempv[j] == 2){
                                part_OID.push_back(tempv[j-1]);
                                access_result.push_back(access_resultc("On Condition Part Storage
OID", "Part Storage OID", temp[i], part_OID));
                        }
                }
        }
        temp.clear();

        //Backward navigation from the corresponding storage object of a storage class AIRPLANEC
        temp = backward_mic_no_coding(airplane_OID[0], mic_no_coding);

        for (unsigned int i=0; i<temp.size(); i++){
                if (temp[i] == 3){
                        //Retreive storage objects of a storage class MECHANICC
                        mechanic_OID.push_back(temp[i-1]);
                }
```

```
                else if(temp[i] == 4){
                        //Retreive storage objects of a storage class JOURNEYLOGC
                        journey_OID.push_back(temp[i-1]);
                }
        }
        access_result.push_back(access_resultc("Airplane Storage OID", "Mechanic Storage OID",
airplane_OID[0], mechanic_OID ));
        if (journey_OID.size()!=0){
                access_result.push_back(access_resultc("Airplane Storage OID", "Journey Storage OID",
airplane_OID[0], journey_OID ));
        }

       /* Retreive storage objects of a storage class PERSONC by forward navigation from
         * the corresponding storage objects of a storage class MECHANICC
        */
        for (unsigned int i=0; i<mechanic_OID.size(); i++){
                temp.clear();
                temp = forward_mic_no_coding(mechanic_OID[i], mic_no_coding);
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 2){
                                person_OID.push_back(temp[j-1]);
                                access_result.push_back(access_resultc("Mechanic Storage OID", "Person
Storage OID", mechanic_OID[i], person_OID));
                        }
                }
                person_OID.clear();
        }

       /* Retreive storage objects of a storage class PILOTC by forward navigation from
         * the corresponding storage objects of a storage class JOURNEYLOGC
         */
        for (unsigned int i=0; i<journey_OID.size(); i++){
                temp.clear();
                temp = forward_mic_no_coding(journey_OID[i], mic_no_coding);
                pilot_OID.clear();
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 5){
                                pilot_OID.push_back(temp[j-1]);
                                templ.push_back(temp[j-1]);
                        }
                }
                access_result.push_back(access_resultc("Journey Storage OID", "Pilot Storage OID",
journey_OID[i], pilot_OID));
        }
        templ.sort();
        templ.unique();
        pilot_OID.clear();
        list<int>::iterator t3 = templ.begin();
        while(t3 != templ.end()){
                pilot_OID.push_back(*t3);
                t3++;
        }

        /* Retreive storage objects of a storage class PERSONC by forward navigation from
         * the corresponding storage objects of a storage class PILOTC
         */
        for (unsigned int i=0; i<pilot_OID.size(); i++){
                temp.clear();
                temp = forward_mic_no_coding(pilot_OID[i], mic_no_coding);
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 2){
                                person_OID.push_back(temp[j-1]);
                                access_result.push_back(access_resultc("Pilot Storage OID", "Person
Storage OID", pilot_OID[i], person_OID));
                        }
                }
                person_OID.clear();
        }
        return access_result;
}
```

```
/* ---mic_SICF.h---
 * - A class mic_SICFc for MIC implemented with a SICF coding technique
 * - A function to create MIC implemented with a SICF coding technique
 * - A function to display MIC implemented with a SICF coding technique
 *
 * Programmer: Weena Nusdin
 * Last Modified: 6 February 2004
 */

//A class mic_SICFc for MIC implemented with a SICF coding technique
class mic_SICFc{
public:
        int mic_SICF_oid;
        unsigned int a, b, c, d;
        mic_SICFc(int pmic_SICF_oid, unsigned int pa, unsigned int pb, unsigned int pc, unsigned int pd);
};

vector<mic_SICFc> create_mic_SICF();          //A function to create MIC implemented with
                                              //a SICF coding technique
void show_mic_SICF(vector<mic_SICFc> pmic_SICF);      //A function to display MIC implemented with
                                                      //a SICF coding technique
```

```
/* ---mic_SICF.cpp---
 * This program creates MIC implemented with a SICF coding technique. The program also provides
 * a function to display MIC implemented with a SICF coding technique.
 *
 * Programmer: Weena Nusdin
 * Last Modified: 6 February 2004
 */

#include<iostream>
#include<vector>
using namespace std;

#include "mic_SICF.h"
#include "input.h"
#include "count_ref.h"

//A constructor function of a class mic_SICFc for MIC implemented with a SICF coding technique
mic_SICFc::mic_SICFc(int pmic_SICF_oid, unsigned int pa, unsigned int pb, unsigned int pc, unsigned int
pd){
        mic_SICF_oid = pmic_SICF_oid;
        a = pa;
        b = pb;
        c = pc;
        d = pd;
}

//A function to create MIC implemented with a SICF coding technique
vector<mic_SICFc> create_mic_SICF(){
        vector<referencec> reference;
        reference.clear();
        reference = load_reference();
        vector<objectc> object;
        object.clear();
        object = load_object();
        vector<mic_SICFc> mic_SICF;
        mic_SICF.clear();

        int pmic_SICF_oid;
        unsigned int pa, pb, pc, pd;
        vector<int> pfor_oid, pbac_oid;
        vector<int> same_f, same_b;
        int r_4, r_10, o_run_f, o_run_b;

        vector<objectc>::iterator r_1 = object.begin();
        while (r_1 != object.end()){
                pfor_oid.clear();
                pbac_oid.clear();
                pmic_SICF_oid = r_1->oid;
                if (count_fref(pmic_SICF_oid) == 0){          //no forward reference
                        pa = 0;
                        pb = 0;
                }//end if
                else if (count_fref(pmic_SICF_oid) > 1){      //multiple forward reference
                        vector<referencec>::iterator r_2 = reference.begin();
                        while (r_2 != reference.end()){
                                if (r_2->begin_oid == pmic_SICF_oid && r_2->reference_flag != 0){
                                        pfor_oid.push_back(r_2->end_oid);
                                        pfor_oid.push_back(r_2->reference_flag);
                                        vector<objectc>::iterator r_3 = object.begin();
                                        while (r_3 != object.end()){
                                                if (r_3->oid == r_2->end_oid && r_3->condition_flag
!= 0){
                                                        pfor_oid.push_back(r_3->condition_flag);
                                                        break;
                                                }//end if
                                                r_3++;
                                        }//end while
                                }//end if
                                else if (r_2->begin_oid == pmic_SICF_oid && r_2->reference_flag == 0){
                                        pfor_oid.push_back(r_2->end_oid);
                                        vector<objectc>::iterator r_3 = object.begin();
                                        while (r_3 != object.end()){
                                                if (r_3->oid == r_2->end_oid && r_3->condition_flag
!= 0){
                                                        pfor_oid.push_back(r_3->condition_flag);
```

```
                                                        break;
                                                }//end if
                                                r_3++;
                                        }//end while
                                }//end else if
                                r_2++;
                        }//end while

                        unsigned int paold;
                        pa = pfor_oid[pfor_oid.size()-1];
                        pb = (pfor_oid[pfor_oid.size()-2]*pfor_oid[pfor_oid.size()-1])+1;
                        for (int i = pfor_oid.size()-2; i>0;  i--){
                                paold = pa;
                                pa = pb;
                                pb = (pa*pfor_oid[i-1])+ paold;
                        }//end for
                }//end else if

                else if (count_fref(pmic_SICF_oid) == 1){              //single forward reference
                        same_f.clear();
                        o_run_f = pmic_SICF_oid;
                        r_4 = 0;
                        while (r_4 == 0){
                                same_f.push_back(o_run_f);
                                vector<referencec>::iterator r_5 = reference.begin();
                                while (r_5 != reference.end()){
                                        if (r_5->begin_oid == o_run_f && r_5->reference_flag !=0){
                                                o_run_f = r_5->end_oid;
                                                pfor_oid.push_back(o_run_f);
                                                pfor_oid.push_back(r_5->reference_flag);
                                                vector<objectc>::iterator r_6 = object.begin();
                                                while (r_6 != object.end()){
                                                        if (r_6->oid == o_run_f && r_6->
condition_flag != 0){
                                                                pfor_oid.push_back(r_6->
condition_flag);
                                                                break;
                                                        }//end if
                                                        r_6++;
                                                }//end while
                                                break;
                                        }//end if
                                        else if (r_5->begin_oid == o_run_f && r_5->reference_flag ==
0){
                                                o_run_f = r_5->end_oid;
                                                pfor_oid.push_back(o_run_f);
                                                vector<objectc>::iterator r_6 = object.begin();
                                                while (r_6 != object.end()){
                                                        if (r_6->oid == o_run_f && r_6->
condition_flag != 0){
                                                                pfor_oid.push_back(r_6->
condition_flag);
                                                                break;
                                                        }//end if
                                                        r_6++;
                                                }//end while
                                                break;
                                        }//end else if
                                        r_5++;
                                }//end while
                                if(count_fref(o_run_f) != 1){
                                        r_4 = 1;
                                        break;
                                }//end if
                                else {
                                        vector<int>::iterator r_7 = same_f.begin();
                                        while(r_7 != same_f.end()){
                                                if (*r_7 == o_run_f){
                                                        r_4 = 1;
                                                        break;
                                                }//end if
                                                r_7++;
                                        }//end while
                                }//end else
                        }//end while
```

```
                        if (pfor_oid.size() == 1){
                                pa = 1;
                                pb = pfor_oid[0];
                        }
                        else {
                                unsigned int paold;
                                pa = pfor_oid[pfor_oid.size()-1];
                                pb = (pfor_oid[pfor_oid.size()-2]*pfor_oid[pfor_oid.size()-1])+1;
                                for (int i = pfor_oid.size()-2; i>0;  i--){
                                        paold = pa;
                                        pa = pb;
                                        pb = (pa*pfor_oid[i-1])+ paold;
                                }//end for
                                int tranf;
                                tranf = pa;
                                pa = pb;
                                pb = tranf;
                        }//end else

                }//end else if

                if (count_bref(pmic_SICF_oid) == 0){          //no backward reference
                        pc = 0;
                        pd = 0;
                }//end if
                else if (count_bref(pmic_SICF_oid) > 1){      //multiple backward reference
                        vector<referencec>::iterator r_8 = reference.begin();
                        while (r_8 != reference.end()){
                                if (r_8->end_oid == pmic_SICF_oid && r_8->reference_flag != 0){
                                        pbac_oid.push_back(r_8->begin_oid);
                                        pbac_oid.push_back(r_8->reference_flag);
                                        vector<objectc>::iterator r_9 = object.begin();
                                        while (r_9 != object.end()){
                                                if (r_9->oid == r_8->begin_oid && r_9->
condition_flag != 0){
                                                        pbac_oid.push_back(r_9->condition_flag);
                                                        break;
                                                }//end if
                                                r_9++;
                                        }//end while
                                }//end if
                                else if (r_8->end_oid == pmic_SICF_oid && r_8->reference_flag == 0){
                                        pbac_oid.push_back(r_8->begin_oid);
                                        vector<objectc>::iterator r_9 = object.begin();
                                        while (r_9 != object.end()){
                                                if (r_9->oid == r_8->begin_oid && r_9->
condition_flag != 0){
                                                        pbac_oid.push_back(r_9->condition_flag);
                                                        break;
                                                }//end if
                                                r_9++;
                                        }//end while
                                }//end else if
                                r_8++;
                        }//end while
                        unsigned int pcold;
                        pc = pbac_oid[pbac_oid.size()-1];
                        pd = (pbac_oid[pbac_oid.size()-2]*pbac_oid[pbac_oid.size()-1])+1;
                        for (int i = pbac_oid.size()-2; i>0;  i--){
                                pcold = pc;
                                pc = pd;
                                pd = (pc*pbac_oid[i-1])+ pcold;
                        }
                }//end else if
                else if (count_bref(pmic_SICF_oid) == 1){     //single backward reference
                        same_b.clear();
                        o_run_b = pmic_SICF_oid;
                        r_10 = 0;
                        while (r_10 == 0){
                                same_b.push_back(o_run_b);
                                vector<referencec>::iterator r_11 = reference.begin();
                                while (r_11 != reference.end()){
                                        if (r_11->end_oid == o_run_b && r_11->reference_flag != 0){
                                                o_run_b = r_11->begin_oid;
                                                pbac_oid.push_back(o_run_b);
```

```
                                                pbac_oid.push_back(r_11->reference_flag);
                                                vector<objectc>::iterator r_12 = object.begin();
                                                while (r_12 != object.end()){
                                                        if (r_12->oid == o_run_b && r_12->
condition_flag != 0){
                                                                pbac_oid.push_back(r_12->
condition_flag);
                                                                break;
                                                        }//end if
                                                        r_12++;
                                                }//end while
                                                break;
                                        }//end if
                                        else if (r_11->end_oid == o_run_b && r_11->reference_flag ==
0){
                                                o_run_b = r_11->begin_oid;
                                                pbac_oid.push_back(o_run_b);
                                                vector<objectc>::iterator r_12 = object.begin();
                                                while (r_12 != object.end()){
                                                        if (r_12->oid == o_run_b && r_12->
condition_flag != 0){
                                                                pbac_oid.push_back(r_12->
condition_flag);
                                                                break;
                                                        }//end if
                                                        r_12++;
                                                }//end while
                                                break;
                                        }//end else if

                                        r_11++;
                                }//end while
                                if(count_bref(o_run_b) != 1){
                                        r_10 = 1;
                                        break;
                                }//end if
                                else {
                                        vector<int>::iterator r_13 = same_b.begin();
                                        while(r_13 != same_b.end()){
                                                if (*r_13 == o_run_b){
                                                        r_10 = 1;
                                                        break;
                                                }//end if
                                                r_13++;
                                        }//end while
                                }//end else
                        }//end while

                        if (pbac_oid.size() == 1){
                                pc = 1;
                                pd = pbac_oid[0];
                        }
                        else {
                                unsigned int pcold;
                                pc = pbac_oid[pbac_oid.size()-1];
                                pd = (pbac_oid[pbac_oid.size()-2]*pbac_oid[pbac_oid.size()-1])+1;
                                for (int i = pbac_oid.size()-2; i>0;  i--){
                                        pcold = pc;
                                        pc = pd;
                                        pd = (pc*pbac_oid[i-1])+ pcold;
                                }//end for
                                int tranb;
                                tranb = pc;
                                pc = pd;
                                pd = tranb;
                        }//end else
                }//end else if
                mic_SICF.push_back(mic_SICFc(pmic_SICF_oid, pa, pb, pc, pd));
                r_1++;
        }//end while
        return mic_SICF;
}

//A function to display MIC implemented with a SICF coding technique
void show_mic_SICF(vector<mic_SICFc> pmic_SICF){
```

```
        vector<mic_SICFc> mic_SICF;
        mic_SICF.clear();
        mic_SICF = pmic_SICF;
        cout << "-- MIC2 with a SICF coding technique -- \n";
        cout << "(OID, forward_N, forward_D, backward_N, backward_D)\n";
        vector<mic_SICFc>::iterator s_m2 = mic_SICF.begin();
        while (s_m2 != mic_SICF.end()){
                cout << s_m2->mic_SICF_oid << " " << s_m2->a << " " << s_m2->b << " " << s_m2->c << " "
<< s_m2->d << "\n";
                s_m2++;
        }//end while
        cout << "\n";
}
```

```
/* ---mic_SICF_navigation.h---
 * Functions to navigate with MIC implemented with a SICF coding technique
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

#include "mic_SICF.h"
//A function for forward navigation with MIC implemented with a SICF coding technique
vector<int> forward_mic_SICF(int pbegin_oid, vector<mic_SICFc> pmic_SICF);

//A function for backward navigation with MIC implemented with a SICF coding technique
vector<int> backward_mic_SICF(int pbegin_oid, vector<mic_SICFc> pmic_SICF);
```

```
/* ---mic_SICF_navigation.cpp---
 * Functions to navigate with MIC implemented with a SICF coding technique
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

#include<iostream>
#include<vector>
using namespace std;

#include "mic_SICF_navigation.h"

//A function for forward navigation with MIC implemented with a SICF coding technique
vector<int> forward_mic_SICF(int pbegin_oid, vector<mic_SICFc> pmic_SICF){
        vector<mic_SICFc> mic_SICF;
        mic_SICF = pmic_SICF;
        int begin_oid;
        vector<int> end_oid;
        int N, oldD, D;

        begin_oid = pbegin_oid;
        vector<mic_SICFc>::iterator i = mic_SICF.begin();
        while (i != mic_SICF.end()){
                if (begin_oid == i->mic_SICF_oid && i->a > i->b){
                        N = i->a;
                        D = i->b;
                        end_oid.clear();
                        while (D!=0){
                                end_oid.push_back(N/D);
                                oldD = D;
                                D = N%D;
                                N = oldD;
                        }//end while
                }//end if
                else if (begin_oid == i->mic_SICF_oid && i->a < i->b){
                        N = i->b;
                        D = i->a;
                        end_oid.clear();
                        while (D!=0){
                                end_oid.push_back(N/D);
                                oldD = D;
                                D = N%D;
                                N = oldD;
                        }//end while
                }
                else if (begin_oid == i->mic_SICF_oid && i->a == 1){
                        end_oid.push_back(i->b);
                }
                i++;
        }//end while
        return end_oid;
}

//A function for backward navigation with MIC implemented with a SICF coding technique
vector<int> backward_mic_SICF(int pbegin_oid, vector<mic_SICFc> pmic_SICF){
        vector<mic_SICFc> mic_SICF;
        mic_SICF = pmic_SICF;

        int begin_oid;
        vector<int> end_oid;
        int N, oldD, D;

        begin_oid = pbegin_oid;
        vector<mic_SICFc>::iterator i = mic_SICF.begin();
        while (i != mic_SICF.end()){
                if (begin_oid == i->mic_SICF_oid && i->c > i->d){
                        N = i->c;
                        D = i->d;
                        end_oid.clear();
                        while (D!=0){
                                end_oid.push_back(N/D);
                                oldD = D;
                                D = N%D;
                                N = oldD;
```

```
                }//end while
            }//end if
            else if (begin_oid == i->mic_SICF_oid && i->c < i->d){
                N = i->d;
                D = i->c;
                end_oid.clear();
                while (D!=0){
                    end_oid.push_back(N/D);
                    oldD = D;
                    D = N%D;
                    N = oldD;
                }//end while
            }
            else if (begin_oid == i->mic_SICF_oid && i->c == 1){
                end_oid.push_back(i->d);
            }
            i++;
        }//end while
        return end_oid;
}
```

```
/* ---mic_SICF_support_Q.h---
 * A functions to a query Q by MIC implemented with a SICF coding technique
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

/* A function to support a query Q by MIC with a SICF coding technique
 * Q: Retreive details of journeys of an airplane HS-AAA including pilots of the journeys, details
 *    of engines and parts installed on the airplane, and details of mechanics who maintain
 *    the airplane.
 */
vector<access_resultc>  mic_SICF_support_Q(vector<mic_SICFc> pmic_SICF);
```

```
/* ---mic_SICF_support_Q.cpp---
 * A functions to a query Q by MIC implemented with a SICF coding technique
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

#include<iostream>
#include<list>
#include<vector>
using namespace std;

#include "result.h"
#include "mic_SICF_navigation.h"
#include "mic_SICF_support_Q.h"

/* A function to support a query Q by MIC with a SICF coding technique
 * Q: Retreive details of journeys of an airplane HS-AAA including pilots of the journeys, details
 *    of engines and parts installed on the airplane, and details of mechanics who maintain
 *    the airplane.
 */
vector<access_resultc>  mic_SICF_support_Q(vector<mic_SICFc> pmic_SICF){

        vector<access_resultc> access_result;                //Create an array of access results
        access_result.clear();

        vector<mic_SICFc> mic_SICF;
        mic_SICF = pmic_SICF;

        int VO;                                              //Define a Value Object(VO)
        char Value_Object[50];
        vector<int> temp, tempv, airplane_OID, engine_OID, part_OID;
        vector<int> partlc_OID, partonc_OID, mechanic_OID, journey_OID, person_OID, pilot_OID;
        list<int> templ, temp1, temp2;

        VO = 8;                                              //An OID of a VO representing a value of
                                                          //an attribute RegistrationName, HS-AAA
        strcpy(Value_Object, "Value Object (HS-AAA)");

       /* Retreive a storage object of a storage class AIRPLANEC whose value of an attribute
         * RegistrationName is HS-AAA by forward navigation from the corresponding VO.
         */
        airplane_OID = forward_mic_SICF(VO, mic_SICF);
        access_result.push_back(access_resultc(Value_Object, "Airplane Storage OID", VO, airplane_OID));


       /* Retreive storage objects of a storage class ENGINEC by forward navigation from
         * the corresponding storage object of a storage class AIRPLANEC
         */
        engine_OID = forward_mic_SICF(airplane_OID[0], mic_SICF);
        access_result.push_back(access_resultc("Airplane Storage OID", "Engine Storage OID", airplane_OID
[0], engine_OID ));

        templ.clear();
        for (unsigned int i=0; i<engine_OID.size(); i++){
                templ.push_back(engine_OID[i]);
        }
        templ.sort();
        templ.unique();
        engine_OID.clear();
        list<int>::iterator t = templ.begin();
        while(t != templ.end()){
                engine_OID.push_back(*t);
                t++;
        }
        temp.clear();
        templ.clear();
        temp2.clear();

       /* Retreive storage objects of storage classes LIFECYCLEPARTC and ONCONDITIONPARTC
         * by forward navigation from the corresponding storage objects of a storage
         * class ENGINEC
         */
        for (unsigned int i=0; i<engine_OID.size(); i++){
                partlc_OID.clear();
```

```
                partonc_OID.clear();
                temp.clear();
                temp = forward_mic_SICF(engine_OID[i], mic_SICF);
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 6){
                                partlc_OID.push_back(temp[j-1]);
                                temp1.push_back(temp[j-1]);
                        }
                        else if (temp[j] == 7){
                                partonc_OID.push_back(temp[j-1]);
                                temp2.push_back(temp[j-1]);
                        }
                }
                access_result.push_back(access_resultc("Engine Storage OID", "Life Cycle Part Storage
OID", engine_OID[i], partlc_OID ));
                access_result.push_back(access_resultc("Engine Storage OID", "On Condition Part Storage
OID", engine_OID[i], partonc_OID ));
        }
        temp1.sort();
        temp1.unique();
        temp2.sort();
        temp2.unique();
        temp.clear();
        list<int>::iterator t1 = temp1.begin();
        while(t1 != temp1.end()){
                temp.push_back(*t1);
                t1++;
        }

       /* Retreive storage objects of a storage class PARTC by forward navigation from
         * the corresponding storage objects of a storage class LIFECYCLEPARTC
         */
        for (unsigned int i=0; i<temp.size(); i++){
                tempv.clear();
                tempv = forward_mic_SICF(temp[i], mic_SICF);
                part_OID.clear();
                for (unsigned int j=0; j<tempv.size(); j++){
                        if (tempv[j] == 2){
                                part_OID.push_back(tempv[j-1]);
                                access_result.push_back(access_resultc("Life Cycle Part Storage OID",
"Part Storage OID", temp[i], part_OID));
                        }
                }
        }
        temp.clear();
        list<int>::iterator t4 = temp2.begin();
        while(t4 != temp2.end()){
                temp.push_back(*t4);
                t4++;
        }

       /* Retreive storage objects of a storage class PARTC by forward navigation from
         * the corresponding storage objects of a storage class ONCONDITIONPARTC
         */
        for (unsigned int i=0; i<temp.size(); i++){
                tempv.clear();
                tempv = forward_mic_SICF(temp[i], mic_SICF);
                part_OID.clear();
                for (unsigned int j=0; j<tempv.size(); j++){
                        if (tempv[j] == 2){
                                part_OID.push_back(tempv[j-1]);
                                access_result.push_back(access_resultc("On Condition Part Storage
OID", "Part Storage OID", temp[i], part_OID));
                        }
                }
        }
        temp.clear();

        //Backward navigation from the corresponding storage object of a storage class AIRPLANEC
        temp = backward_mic_SICF(airplane_OID[0], mic_SICF);

        for (unsigned int i=0; i<temp.size(); i++){
                if (temp[i] == 3){
                      //Retreive storage objects of a storage class MECHANICC
                        mechanic_OID.push_back(temp[i-1]);
```

```
                }
                else if(temp[i] == 4){
                        //Retreive storage objects of a storage class JOURNEYLOGC
                           journey_OID.push_back(temp[i-1]);
                }
        }
        access_result.push_back(access_resultc("Airplane Storage OID", "Mechanic Storage OID",
airplane_OID[0], mechanic_OID ));
        if (journey_OID.size()!=0){
                access_result.push_back(access_resultc("Airplane Storage OID", "Journey Storage OID",
airplane_OID[0], journey_OID ));
        }

       /* Retreive storage objects of a storage class PERSONC by forward navigation from
         * the corresponding storage objects of a storage class MECHANICC
         */
        for (unsigned int i=0; i<mechanic_OID.size(); i++){
                temp.clear();
                temp = forward_mic_SICF(mechanic_OID[i], mic_SICF);
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 2){
                                person_OID.push_back(temp[j-1]);
                                access_result.push_back(access_resultc("Mechanic Storage OID", "Person
Storage OID", mechanic_OID[i], person_OID));
                        }
                }
                person_OID.clear();
        }

       /* Retreive storage objects of a storage class PILOTC by forward navigation from
         * the corresponding storage objects of a storage class JOURNEYLOGC
         */
        for (unsigned int i=0; i<journey_OID.size(); i++){
                temp.clear();
                temp = forward_mic_SICF(journey_OID[i], mic_SICF);
                pilot_OID.clear();
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 5){
                                pilot_OID.push_back(temp[j-1]);
                                templ.push_back(temp[j-1]);
                        }
                }
                access_result.push_back(access_resultc("Journey Storage OID", "Pilot Storage OID",
journey_OID[i], pilot_OID));
        }
        templ.sort();
        templ.unique();
        pilot_OID.clear();
        list<int>::iterator t3 = templ.begin();
        while(t3 != templ.end()){
                pilot_OID.push_back(*t3);
                t3++;
        }

       /* Retreive storage objects of a storage class PERSONC by forward navigation from
         * the corresponding storage objects of a storage class PILOTC
         */
        for (unsigned int i=0; i<pilot_OID.size(); i++){
                temp.clear();
                temp = forward_mic_SICF(pilot_OID[i], mic_SICF);
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 2){
                                person_OID.push_back(temp[j-1]);
                                access_result.push_back(access_resultc("Pilot Storage OID", "Person
Storage OID", pilot_OID[i], person_OID));
                        }
                }
                person_OID.clear();
        }
        return access_result;
}
```

```
/* ---mic_start_stop.h---
 * - A class mic_start_stopc for MIC implemented with a Start/Stop data compression technique (Pigeon,
2001)
 * - A function to create MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
 * - A function to display MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
 *
 * Programmer: Weena Nusdin
 * Last Modified: 6 February 2004
 */

#include "pigeon_code.h"

 //A class mic_start_stopc for MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
class mic_start_stopc{
public:
        int mic_start_stop_oid;
        char ffanout_type;
        char bfanout_type;
        vector<pigeon_codec> for_oid;
        vector<pigeon_codec> bac_oid;
        int fcode_size, bcode_size;

        mic_start_stopc(int pmic_start_stop_oid, char pffanout_type, vector<pigeon_codec> pfor_oid, char
pbfanout_type, vector<pigeon_codec> pbac_oid, int pfcode_size, int pbcode_size);
};

//A function to create MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
vector<mic_start_stopc> create_mic_start_stop();

//A function to display MIC implemented with a Start/Stop data compression technique (Pigeon, 2001)
void show_mic_start_stop(vector<mic_start_stopc> pmic_start_stop);
```

```
/* ---mic_start_stop.cpp---
 * This program creates MIC implemented with a Start/Stop data compression technique (Pigeon,2001)
 * The program also provides a function to display MIC implemented with a Start/Stop data
 * compression technique (Pigeon,2001)
 *
 * Programmer: Weena Nusdin
 * Last Modified: 6 February 2004
 */

#include<iostream>
#include<vector>
using namespace std;

#include "mic_start_stop.h"
#include "input.h"
#include "count_ref.h"

/*A constructor function of a class mic_start_stopc for MIC implemented  with a Start/Stop data
 * compression technique (Pigeon,2001)
 */
mic_start_stopc::mic_start_stopc(int pmic_start_stop_oid, char pffanout_type, vector<pigeon_codec>
pfor_oid, char pbfanout_type, vector<pigeon_codec> pbac_oid, int pfcode_size, int pbcode_size){
        mic_start_stop_oid = pmic_start_stop_oid;
        ffanout_type = pffanout_type;
        for_oid = pfor_oid;
        bfanout_type = pbfanout_type;
        bac_oid = pbac_oid;
        fcode_size = pfcode_size;
        bcode_size = pbcode_size;
}

//A function to create MIC implemented with a Start/Stop data compression technique (Pigeon,2001)
vector<mic_start_stopc> create_mic_start_stop(){
        vector<referencec> reference;
        reference.clear();
        reference = load_reference();
        vector<objectc> object;
        object.clear();
        object = load_object();
        vector<mic_start_stopc> mic_start_stop;
        mic_start_stop.clear();

        int pmic_start_stop_oid, pfcode_size, pbcode_size;
        char pffanout_type, pbfanout_type;
        vector<pigeon_codec> pfor_oid, pbac_oid;
        vector<int> same_f, same_b;
        int r_4, r_10, o_run_f, o_run_b;

        vector<objectc>::iterator r_1 = object.begin();
        while (r_1 != object.end()){
                pfor_oid.clear();
                pbac_oid.clear();
                pmic_start_stop_oid = r_1->oid;
                if (count_fref(pmic_start_stop_oid) == 0){   //no forward reference
                        pffanout_type = '-';
                }//end if
                else if (count_fref(pmic_start_stop_oid) > 1){        //multiple forward reference
                        pffanout_type = 'm';
                        vector<referencec>::iterator r_2 = reference.begin();
                        while (r_2 != reference.end()){
                                if (r_2->begin_oid == pmic_start_stop_oid && r_2->reference_flag != 0)
{
                                        pfor_oid.push_back(encode_pigeon(r_2->end_oid));
                                        pfor_oid.push_back(encode_pigeon(r_2->reference_flag));
                                        vector<objectc>::iterator r_3 = object.begin();
                                        while (r_3 != object.end()){
                                                if (r_3->oid == r_2->end_oid && r_3->condition_flag
!= 0){
                                                        pfor_oid.push_back(encode_pigeon(r_3->
condition_flag));
                                                        break;
                                                }//end if
                                                r_3++;
                                        }//end while
                                }//end if
```

```
                                else if (r_2->begin_oid == pmic_start_stop_oid && r_2->reference_flag
== 0){
                                        pfor_oid.push_back(encode_pigeon(r_2->end_oid));
                                        vector<objectc>::iterator r_3 = object.begin();
                                        while (r_3 != object.end()){
                                                if (r_3->oid == r_2->end_oid && r_3->condition_flag
!= 0){
                                                        pfor_oid.push_back(encode_pigeon(r_3->
condition_flag));
                                                        break;
                                                }//end if
                                                r_3++;
                                        }//end while
                                }//end else if
                                r_2++;
                        }//end while
                }//end else if
                else if (count_fref(pmic_start_stop_oid) == 1){       //single forward reference
                        pffanout_type = 's';
                        same_f.clear();
                        o_run_f = pmic_start_stop_oid;
                        r_4 = 0;
                        while (r_4 == 0){
                                same_f.push_back(o_run_f);
                                vector<referencec>::iterator r_5 = reference.begin();
                                while (r_5 != reference.end()){
                                        if (r_5->begin_oid == o_run_f && r_5->reference_flag != 0){
                                                o_run_f = r_5->end_oid;
                                                pfor_oid.push_back(encode_pigeon(o_run_f));
                                                pfor_oid.push_back(encode_pigeon(r_5->
reference_flag));
                                                vector<objectc>::iterator r_6 = object.begin();
                                                while (r_6 != object.end()){
                                                        if (r_6->oid == o_run_f && r_6->
condition_flag != 0){
                                                                pfor_oid.push_back(encode_pigeon
(r_6->condition_flag));
                                                                break;
                                                        }//end if
                                                        r_6++;
                                                }//end while
                                                break;
                                        }//end if
                                        else if (r_5->begin_oid == o_run_f && r_5->reference_flag ==
0){
                                                o_run_f = r_5->end_oid;
                                                pfor_oid.push_back(encode_pigeon(o_run_f));
                                                vector<objectc>::iterator r_6 = object.begin();
                                                while (r_6 != object.end()){
                                                        if (r_6->oid == o_run_f && r_6->
condition_flag != 0){
                                                                pfor_oid.push_back(encode_pigeon
(r_6->condition_flag));
                                                                break;
                                                        }//end if
                                                        r_6++;
                                                }//end while
                                                break;
                                        }//end else if
                                        r_5++;
                                }//end while
                                if(count_fref(o_run_f) != 1){
                                        r_4 = 1;
                                        break;
                                }//end if
                                else {
                                        vector<int>::iterator r_7 = same_f.begin();
                                        while(r_7 != same_f.end()){
                                                if (*r_7 == o_run_f){
                                                        r_4 = 1;
                                                        break;
                                                }//end if
                                                r_7++;
                                        }//end while
                                }//end else
```

```
                }//end while
        }//end else if
        if (count_bref(pmic_start_stop_oid) == 0){    //no backward reference
                pbfanout_type = '-';
        }//end if
        else if (count_bref(pmic_start_stop_oid) > 1){        //multiple backward reference
                pbfanout_type = 'm';
                vector<referencec>::iterator r_8 = reference.begin();
                while (r_8 != reference.end()){
                        if (r_8->end_oid == pmic_start_stop_oid && r_8->reference_flag !=0){
                                pbac_oid.push_back(encode_pigeon(r_8->begin_oid));
                                pbac_oid.push_back(encode_pigeon(r_8->reference_flag));
                                vector<objectc>::iterator r_9 = object.begin();
                                while (r_9 != object.end()){
                                        if (r_9->oid == r_8->begin_oid && r_9->
condition_flag != 0){
                                                pbac_oid.push_back(encode_pigeon(r_9->
condition_flag));
                                                break;
                                        }//end if
                                        r_9++;
                                }//end while
                        }//end if
                        else if (r_8->end_oid == pmic_start_stop_oid && r_8->reference_flag ==
0){
                                pbac_oid.push_back(encode_pigeon(r_8->begin_oid));
                                vector<objectc>::iterator r_9 = object.begin();
                                while (r_9 != object.end()){
                                        if (r_9->oid == r_8->begin_oid && r_9->
condition_flag != 0){
                                                pbac_oid.push_back(encode_pigeon(r_9->
condition_flag));
                                                break;
                                        }//end if
                                        r_9++;
                                }//end while
                        }//end else if
                        r_8++;
                }//end while
        }//end else if
        else if (count_bref(pmic_start_stop_oid) == 1){        //single backward reference
                pbfanout_type = 's';
                same_b.clear();
                o_run_b = pmic_start_stop_oid;
                r_10 = 0;
                while (r_10 == 0){
                        same_b.push_back(o_run_b);
                        vector<referencec>::iterator r_11 = reference.begin();
                        while (r_11 != reference.end()){
                                if (r_11->end_oid == o_run_b && r_11->reference_flag != 0){
                                        o_run_b = r_11->begin_oid;
                                        pbac_oid.push_back(encode_pigeon(o_run_b));
                                        pbac_oid.push_back(encode_pigeon(r_11->
reference_flag));
                                        vector<objectc>::iterator r_12 = object.begin();
                                        while (r_12 != object.end()){
                                                if (r_12->oid == o_run_b && r_12->
condition_flag != 0){
                                                        pbac_oid.push_back(encode_pigeon
(r_12->condition_flag));
                                                        break;
                                                }//end if
                                                r_12++;
                                        }//end while
                                        break;
                                }//end if
                                else if (r_11->end_oid == o_run_b && r_11->reference_flag ==
0){
                                        o_run_b = r_11->begin_oid;
                                        pbac_oid.push_back(encode_pigeon(o_run_b));
                                        vector<objectc>::iterator r_12 = object.begin();
                                        while (r_12 != object.end()){
                                                if (r_12->oid == o_run_b && r_12->
condition_flag != 0){
```

```
                                                                pbac_oid.push_back(encode_pigeon
(r_12->condition_flag));
                                                                break;
                                                        }//end if
                                                        r_12++;
                                                }//end while
                                                break;
                                        }//end else if
                                        r_11++;
                                }//end while
                                if(count_bref(o_run_b) != 1){
                                        r_10 = 1;
                                        break;
                                }//end if
                                else {
                                        vector<int>::iterator r_13 = same_b.begin();
                                        while(r_13 != same_b.end()){
                                                if (*r_13 == o_run_b){
                                                        r_10 = 1;
                                                        break;
                                                }//end if
                                                r_13++;
                                        }//end while
                                }//end else
                        }//end while
                }//end else if

                pfcode_size = 0;
                vector<pigeon_codec>::iterator fbit = pfor_oid.begin();
                while(fbit != pfor_oid.end()){
                        vector<int> fbit_size;
                        fbit_size = fbit->pigeon_code;
                        pfcode_size = pfcode_size + fbit_size.size();
                        fbit++;
                }

                pbcode_size = 0;
                vector<pigeon_codec>::iterator bbit = pbac_oid.begin();
                while(bbit != pbac_oid.end()){
                        vector<int> bbit_size;
                        bbit_size = bbit->pigeon_code;
                        pbcode_size = pbcode_size + bbit_size.size();
                        bbit++;
                }

                mic_start_stop.push_back(mic_start_stopc(pmic_start_stop_oid, pffanout_type, pfor_oid,
pbfanout_type, pbac_oid, pfcode_size, pbcode_size));
                r_1++;
        }//end while
        return mic_start_stop;
}

//A function to display MIC implemented with a Start/Stop data compression technique (Pigeon,2001)
void show_mic_start_stop(vector<mic_start_stopc> pmic_start_stop){
        vector<mic_start_stopc> mic_start_stop;
        mic_start_stop.clear();
        mic_start_stop = pmic_start_stop;

        cout << "-- MIC with a Start/Stop data compression technique -- \n";
        cout << "(OID, forward fanout type, encoded referenced OIDs, backward fanout type, encoded
referencing OIDs, bits for forward code, bits for backward codes)\n";
        vector<pigeon_codec> show_f_oid, show_b_oid;
        vector<mic_start_stopc>::iterator s_m3 = mic_start_stop.begin();
        while (s_m3 != mic_start_stop.end()){
                cout << s_m3->mic_start_stop_oid << " " << s_m3->ffanout_type << " [";
                show_f_oid = s_m3 -> for_oid;
                vector<pigeon_codec>::iterator s_fbit = show_f_oid.begin();
                while(s_fbit != show_f_oid.end()){
                        vector<int> show_f_bit;
                        show_f_bit = s_fbit->pigeon_code;
                        for (unsigned int i=0; i<show_f_bit.size(); i++) cout << show_f_bit[i];
                        cout << " ";
                        s_fbit++;
                }
                cout << "] ";
```

```
                cout << s_m3->bfanout_type << " [";
                show_b_oid = s_m3 -> bac_oid;
                vector<pigeon_codec>::iterator s_bbit = show_b_oid.begin();
                while(s_bbit != show_b_oid.end()){
                        vector<int> show_b_bit;
                        show_b_bit = s_bbit->pigeon_code;
                        for (unsigned int i=0; i<show_b_bit.size(); i++) cout << show_b_bit[i];
                        cout << " ";
                        s_bbit++;
                }
                cout << "] ";
                cout << s_m3->fcode_size << " ";
                cout << s_m3->bcode_size << " \n";
                s_m3++;
        }
}
```

```
/* ---mic_start_stop_navigation.h---
 * Functions to navigate with MIC implemented with a Start/Stop data compression technique
 * (Pigeon, 2001)
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

#include "mic_start_stop.h"
/* A function for forward navigation by MIC implemented with a Start/Stop data compression technique
(Pigeon, 2001)
 */

vector<int> forward_mic_start_stop(int pbegin_oid, vector<mic_start_stopc> pmic_start_stop);

/* A function for backward navigation by MIC implemented with a Start/Stop data compression technique
(Pigeon, 2001)
 */
vector<int> backward_mic_start_stop(int pbegin_oid, vector<mic_start_stopc> pmic_start_stop);
```

```
/* ---mic_start_stop_navigation.cpp---
 * Functions to navigate with MIC implemented with a Start/Stop data compression technique
 * (Pigeon, 2001)
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

#include<iostream>
#include<vector>
using namespace std;

#include "mic_start_stop_navigation.h"

//A function for forward navigation by MIC with a Start/Stop data compression technique (Pigeon,2001)
vector<int> forward_mic_start_stop(int pbegin_oid, vector<mic_start_stopc> pmic_start_stop){
        vector<mic_start_stopc> mic_start_stop;
        mic_start_stop = pmic_start_stop;
        int begin_oid;
        vector<int> end_oid, temp3;
        vector<pigeon_codec> end_bits;
        int temp_end_oid;

        begin_oid = pbegin_oid;
        vector<mic_start_stopc>::iterator i = mic_start_stop.begin();
        while (i != mic_start_stop.end()){
                if (begin_oid == i->mic_start_stop_oid){
                        end_bits = i->for_oid;
                }
                i++;
        }
        vector<pigeon_codec>::iterator j = end_bits.begin();
        while(j != end_bits.end()){
                temp3 = j->pigeon_code;
                temp_end_oid = decode_pigeon(temp3);
                end_oid.push_back(temp_end_oid);
                j++;
        }
        return end_oid;
}

//A function for backward navigation by MIC with a Start/Stop data compression technique (Pigeon,2001)
vector<int> backward_mic_start_stop(int pbegin_oid, vector<mic_start_stopc> pmic_start_stop){
        vector<mic_start_stopc> mic_start_stop;
        mic_start_stop = pmic_start_stop;

        int begin_oid;
        vector<int> end_oid, temp3;
        vector<pigeon_codec> end_bits;
        int temp_end_oid;

        begin_oid = pbegin_oid;
        vector<mic_start_stopc>::iterator i = mic_start_stop.begin();
        while (i != mic_start_stop.end()){
                if (begin_oid == i->mic_start_stop_oid){
                        end_bits = i->bac_oid;
                }
                i++;
        }
        vector<pigeon_codec>::iterator j = end_bits.begin();
        while(j != end_bits.end()){
                temp3 = j->pigeon_code;
                temp_end_oid = decode_pigeon(temp3);
                end_oid.push_back(temp_end_oid);
                j++;
        }
        return end_oid;
}
```

```
/* ---mic_start_stop_support_Q.h---
 * A functions to a query Q by MIC implemented with a Start/Stop data compression technique
 * (Pigeon, 2001)
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

/* A function to support a query Q by MIC with a Start/Sop data compression technique (Pigeon, 2001)
 * Q: Retreive details of journeys of an airplane HS-AAA including pilots of the journeys, details
 *    of engines and parts installed on the airplane, and details of mechanics who maintain
 *    the airplane.
 */
vector<access_resultc> mic_start_stop_support_Q(vector<mic_start_stopc> pmic_start_stop);
```

```
/* ---mic_start_stop_support_Q.cpp---
 * A functions to a query Q by MIC implemented with a Start/Stop data compression technique
 * (Pigeon, 2001)
 *
 * Programmer: Weena Nusdin
 * Modified: 6 February 2004
 */

#include<iostream>
#include<list>
#include<vector>
using namespace std;

#include "result.h"
#include "mic_start_stop_navigation.h"
#include "mic_start_stop_support_Q.h"

/* A function to support a query Q by MIC with a Start/Stop data compression technique (Pigeon, 2001)
 * Q: Retreive details of journeys of an airplane HS-AAA including pilots of the journeys, details
 *    of engines and parts installed on the airplane, and details of mechanics who maintain
 *    the airplane.
 */
vector<access_resultc> mic_start_stop_support_Q(vector<mic_start_stopc> pmic_start_stop){
        vector<access_resultc> access_result;        //Create an array of access results
        access_result.clear();

        vector<mic_start_stopc> mic_start_stop;
        mic_start_stop = pmic_start_stop;

        int VO;                                      //Define a Value Object(VO)
        char Value_Object[50];
        vector<int> temp, tempv, airplane_OID, engine_OID, part_OID;
        vector<int> partlc_OID, partonc_OID, mechanic_OID, journey_OID, person_OID, pilot_OID;
        list<int> templ, templ, temp2;

        VO = 8;                                      //An OID of a VO representing a value of
                                                  //an attribute RegistrationName, HS-AAA
        strcpy(Value_Object, "Value Object (HS-AAA)");

       /* Retreive a storage object of a storage class AIRPLANEC whose value of an attribute
         * RegistrationName is HS-AAA by forward navigation from the corresponding VO.
         */
        airplane_OID = forward_mic_start_stop(VO, mic_start_stop);
        access_result.push_back(access_resultc(Value_Object, "Airplane Storage OID", VO, airplane_OID));

       /* Retreive storage objects of a storage class ENGINEC by forward navigation from
         * the corresponding storage object of a storage class AIRPLANEC
         */
        engine_OID = forward_mic_start_stop(airplane_OID[0], mic_start_stop);

        access_result.push_back(access_resultc("Airplane Storage OID", "Engine Storage OID", airplane_OID
[0], engine_OID ));

        templ.clear();
        for (unsigned int i=0; i<engine_OID.size(); i++){
                templ.push_back(engine_OID[i]);
        }
        templ.sort();
        templ.unique();
        engine_OID.clear();
        list<int>::iterator t = templ.begin();
        while(t != templ.end()){
                engine_OID.push_back(*t);
                t++;
        }
        temp.clear();
        templ.clear();
        temp2.clear();

       /* Retreive storage objects of storage classes LIFECYCLEPARTC and ONCONDITIONPARTC
         * by forward navigation from the corresponding storage objects of a storage
         * class ENGINEC
         */
        for (unsigned int i=0; i<engine_OID.size(); i++){
                partlc_OID.clear();
```

```
                partonc_OID.clear();
                temp.clear();
                temp = forward_mic_start_stop(engine_OID[i], mic_start_stop);
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 6){
                                partlc_OID.push_back(temp[j-1]);
                                temp1.push_back(temp[j-1]);
                        }
                        else if (temp[j] == 7){
                                partonc_OID.push_back(temp[j-1]);
                                temp2.push_back(temp[j-1]);
                        }
                }
                access_result.push_back(access_resultc("Engine Storage OID", "Life Cycle Part Storage
OID", engine_OID[i], partlc_OID ));
                access_result.push_back(access_resultc("Engine Storage OID", "On Condition Part Storage
OID", engine_OID[i], partonc_OID ));
        }
        temp1.sort();
        temp1.unique();
        temp2.sort();
        temp2.unique();
        temp.clear();
        list<int>::iterator t1 = temp1.begin();
        while(t1 != temp1.end()){
                temp.push_back(*t1);
                t1++;
        }

       /* Retreive storage objects of a storage class PARTC by forward navigation from
         * the corresponding storage objects of a storage class LIFECYCLEPARTC
         */
        for (unsigned int i=0; i<temp.size(); i++){
                tempv.clear();
                tempv = forward_mic_start_stop(temp[i], mic_start_stop);
                part_OID.clear();
                for (unsigned int j=0; j<tempv.size(); j++){
                        if (tempv[j] == 2){
                                part_OID.push_back(tempv[j-1]);
                                access_result.push_back(access_resultc("Life Cycle Part Storage OID",
"Part Storage OID", temp[i], part_OID));
                        }
                }
        }

        temp.clear();
        list<int>::iterator t4 = temp2.begin();
        while(t4 != temp2.end()){
                temp.push_back(*t4);
                t4++;
        }

       /* Retreive storage objects of a storage class PARTC by forward navigation from
         * the corresponding storage objects of a storage class ONCONDITIONPARTC
         */
        for (unsigned int i=0; i<temp.size(); i++){
                tempv.clear();
                tempv = forward_mic_start_stop(temp[i], mic_start_stop);
                part_OID.clear();
                for (unsigned int j=0; j<tempv.size(); j++){
                        if (tempv[j] == 2){
                                part_OID.push_back(tempv[j-1]);
                                access_result.push_back(access_resultc("On Condition Part Storage
OID", "Part Storage OID", temp[i], part_OID));
                        }
                }
        }
        temp.clear();

        //Backward navigation from the corresponding storage object of a storage class AIRPLANEC
        temp = backward_mic_start_stop(airplane_OID[0], mic_start_stop);

        for (unsigned int i=0; i<temp.size(); i++){
                if (temp[i] == 3){
                      //Retreive storage objects of a storage class MECHANICC
```

```
                        mechanic_OID.push_back(temp[i-1]);
                }
                else if(temp[i] == 4){
                      //Retreive storage objects of a storage class JOURNEYLOGC
                        journey_OID.push_back(temp[i-1]);
                }
        }
        access_result.push_back(access_resultc("Airplane Storage OID", "Mechanic Storage OID",
airplane_OID[0], mechanic_OID ));
        if (journey_OID.size()!=0){
                access_result.push_back(access_resultc("Airplane Storage OID", "Journey Storage OID",
airplane_OID[0], journey_OID ));
        }

       /* Retreive storage objects of a storage class PERSONC by forward navigation from
         * the corresponding storage objects of a storage class MECHANICC
         */
        for (unsigned int i=0; i<mechanic_OID.size(); i++){
                temp.clear();
                temp = forward_mic_start_stop(mechanic_OID[i], mic_start_stop);
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 2){
                                person_OID.push_back(temp[j-1]);
                                access_result.push_back(access_resultc("Mechanic Storage OID", "Person
Storage OID", mechanic_OID[i], person_OID));
                        }
                }
                person_OID.clear();
        }

       /* Retreive storage objects of a storage class PILOTC by forward navigation from
         * the corresponding storage objects of a storage class JOURNEYLOGC
         */
        for (unsigned int i=0; i<journey_OID.size(); i++){
                temp.clear();
                temp = forward_mic_start_stop(journey_OID[i], mic_start_stop);
                pilot_OID.clear();
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 5){
                                pilot_OID.push_back(temp[j-1]);
                                templ.push_back(temp[j-1]);
                        }
                }
                access_result.push_back(access_resultc("Journey Storage OID", "Pilot Storage OID",
journey_OID[i], pilot_OID));
        }

        templ.sort();
        templ.unique();
        pilot_OID.clear();
        list<int>::iterator t3 = templ.begin();
        while(t3 != templ.end()){
                pilot_OID.push_back(*t3);
                t3++;
        }

       /* Retreive storage objects of a storage class PERSONC by forward navigation from
        * the corresponding storage objects of a storage class PILOTC
        */
        for (unsigned int i=0; i<pilot_OID.size(); i++){
                temp.clear();
                temp = forward_mic_start_stop(pilot_OID[i], mic_start_stop);
                for (unsigned int j=0; j<temp.size(); j++){
                        if (temp[j] == 2){
                                person_OID.push_back(temp[j-1]);
                                access_result.push_back(access_resultc("Pilot Storage OID", "Person
Storage OID", pilot_OID[i], person_OID));
                        }
                }
                person_OID.clear();
        }
        return access_result;
}
```

APPENDIX D

INPUT DATA

Details of Value Objects (VOs)

Integer numbers 8-19 are chosen to be OIDs of value objects that represent values of an attribute RegistrationName of a storage class AIRPLANEC because a query Q that is supported in the implementation specifies a condition on the attribute RegistrationName.

A query Q is as follows:

Q: *Retrieve details of journeys of an airplane HS-AAA including pilots of the journeys, details of engines and parts installed on the airplane, and details of mechanics who maintain the airplane.*

An Integer number 8 is chosen to be an OID of a value object that represents a value of HS-AAA of an attribute RegistrationName of a storage class AIRPLANEC.

Details of Reference Flags (RFs) are as follows.

Flags Values	Represented references
2	A reference type inheritance
3	A reference name Maintain
4	A reference name flight airplane
5	A reference name flight pilot
6	A reference name LifeCyclePart
7	A reference name OnConditionPart

Storage Class Objects (SCOs), and Value and Storage Class Flags (VFs and SCFs) are not used in the implementation, as described in chapter seven.

Table D.1 Storage Objects Identifiers (OIDs) and (Value/Storage Class) flags

OIDs	Flags	OIDs	Flags	OIDs	Flags	OIDs	Flags	OIDs	Flags
8	0	46	0	84	0	122	0	160	0
9	0	47	0	85	0	123	0	161	0
10	0	48	0	86	0	124	0	162	0
11	0	49	0	87	0	125	0	163	0
12	0	50	0	88	0	126	0	164	0
13	0	51	0	89	0	127	0	165	0
14	0	52	0	90	0	128	0	166	0
15	0	53	0	91	0	129	0	167	0
16	0	54	0	92	0	130	0	168	0
17	0	55	0	93	0	131	0	169	0
18	0	56	0	94	0	132	0	170	0
19	0	57	0	95	0	133	0	171	0
20	0	58	0	96	0	134	0	172	0
21	0	59	0	97	0	135	0	173	0
22	0	60	0	98	0	136	0	174	0
23	0	61	0	99	0	137	0	175	0
24	0	62	0	100	0	138	0	176	0
25	0	63	0	101	0	139	0	177	0
26	0	64	0	102	0	140	0	178	0
27	0	65	0	103	0	141	0	179	0
28	0	66	0	104	0	142	0	180	0
29	0	67	0	105	0	143	0	181	0
30	0	68	0	106	0	144	0	182	0
31	0	69	0	107	0	145	0	183	0
32	0	70	0	108	0	146	0	184	0
33	0	71	0	109	0	147	0	185	0
34	0	72	0	110	0	148	0	186	0
35	0	73	0	111	0	149	0	187	0
36	0	74	0	112	0	150	0	188	0
37	0	75	0	113	0	151	0	189	0
38	0	76	0	114	0	152	0	190	0
39	0	77	0	115	0	153	0	191	0
40	0	78	0	116	0	154	0	192	0
41	0	79	0	117	0	155	0	193	0
42	0	80	0	118	0	156	0	194	0
43	0	81	0	119	0	157	0	195	0
44	0	82	0	120	0	158	0	196	0
45	0	83	0	121	0	159	0		

Table D.2 References and (reference) flags

Begin_OID	End_OID	Flag	Begin_OID	End_OID	Flag
8	57	0	57	47	0
9	58	0	58	48	0
10	59	0	58	49	0
11	60	0	59	50	0
12	116	0	59	51	0
13	117	0	59	52	0
14	118	0	60	53	0
15	119	0	60	54	0
16	175	0	61	27	5
17	176	0	61	28	5
18	177	0	61	57	4
19	178	0	62	27	5
20	25	0	62	28	5
25	20	0	62	57	4
27	20	2	63	27	5
28	21	2	63	28	5
29	22	2	63	58	4
29	59	3	64	27	5
30	23	2	64	28	5
30	58	3	64	58	4
31	24	2	79	84	0
31	57	3	84	79	0
31	60	3	86	79	2
32	65	2	87	80	2
33	66	2	88	81	2
34	67	2	88	118	3
35	68	2	89	82	2
36	69	2	89	117	3
37	70	2	90	83	2
38	71	2	90	116	3
39	72	2	90	119	3
40	73	2	91	124	2
41	74	2	92	125	2
42	75	2	93	126	2
43	76	2	94	127	2
44	77	2	95	128	2
45	78	2	96	129	2
46	32	6	97	130	2
46	33	7	98	131	2
47	34	6	99	132	2
47	35	7	100	133	2
48	36	6	101	134	2
49	37	6	102	135	2
50	38	7	103	136	2
51	39	6	104	137	2
52	40	7	105	91	6
53	41	6	105	92	7
54	42	7	106	93	6
55	43	6	106	94	7
56	44	7	107	95	6
57	46	0	108	96	6

Begin_OID	End_OID	Flag
109	97	7
110	98	6
111	99	7
112	100	6
113	101	7
114	102	6
115	103	7
116	105	0
116	106	0
117	107	0
117	108	0
118	109	0
118	110	0
118	111	0
119	112	0
119	113	0
120	86	5
120	87	5
120	116	4
121	86	5
121	87	5
121	116	4
122	86	5
122	87	5
122	117	4
123	86	5
123	87	5
123	117	4
138	143	0
143	138	0
145	138	2
146	139	2
147	140	2
147	177	3
148	141	2
148	176	3
149	142	2
149	175	3
149	178	3
150	183	2
151	184	2
152	185	2
153	186	2
154	187	2
155	188	2
156	189	2
157	190	2
158	191	2
159	192	2
160	193	2
161	194	2

Begin_OID	End_OID	Flag
162	195	2
163	196	2
164	150	6
164	151	7
165	152	6
165	153	7
166	154	6
167	155	6
168	156	7
169	157	6
170	158	7
171	159	6
172	160	7
173	161	6
174	162	7
175	164	0
175	165	0
176	166	0
176	167	0
177	168	0
177	169	0
177	170	0
178	171	0
178	172	0
179	145	5
179	146	5
179	175	4
180	145	5
180	146	5
180	175	4
181	145	5
181	146	5
181	176	4
182	145	5
182	146	5
182	176	4

APPENDIX E

IMPLEMENTED MIC

Table E.1 MIC implemented with no coding technique

OID	Forward Fan-out Type	A List of Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Referencing OIDs (Backward Navigation)	Bits
8	s	[57]	-	[]	48
9	s	[58]	-	[]	48
10	s	[59]	-	[]	48
11	s	[60]	-	[]	48
12	s	[116]	-	[]	48
13	s	[117]	-	[]	48
14	s	[118]	-	[]	48
15	s	[119]	-	[]	48
16	s	[175]	-	[]	48
17	s	[176]	-	[]	48
18	s	[177]	-	[]	48
19	s	[178]	-	[]	48
20	s	[25 20]	m	[25 27 2]	112
21	-	[]	s	[28 2]	64
22	-	[]	s	[29 2]	64
23	-	[]	s	[30 2]	64
24	-	[]	s	[31 2]	64
25	s	[20 25]	s	[20]	80
26	-	[]	-	[]	32
27	s	[20 2 25 20]	m	[61 5 62 5 63 5 64 5]	224
28	s	[21 2]	m	[61 5 62 5 63 5 64 5]	192
29	m	[22 2 59 3]	-	[]	96
30	m	[23 2 58 3]	-	[]	96
31	m	[24 2 57 3 60 3]	-	[]	128
32	s	[65 2]	s	[46 6 57]	112
33	s	[66 2]	s	[46 7 57]	112
34	s	[67 2]	s	[47 6 57]	112
35	s	[68 2]	s	[47 7 57]	112
36	s	[69 2]	s	[48 6 58]	112
37	s	[70 2]	s	[49 6 58]	112
38	s	[71 2]	s	[50 7 59]	112
39	s	[72 2]	s	[51 6 59]	112
40	s	[73 2]	s	[52 7 59]	112
41	s	[74 2]	s	[53 6 60]	112

OID	Forward Fan-out Type	A List of Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Referencing OIDs (Backward Navigation)	Bits
42	s	[75 2]	s	[54 7 60]	112
43	s	[76 2]	s	[55 6]	96
44	s	[77 2]	s	[56 7]	96
45	s	[78 2]	-	[]	64
46	m	[32 6 33 7]	s	[57]	112
47	m	[34 6 35 7]	s	[57]	112
48	s	[36 6 69 2]	s	[58]	112
49	s	[37 6 70 2]	s	[58]	112
50	s	[38 7 71 2]	s	[59]	112
51	s	[39 6 72 2]	s	[59]	112
52	s	[40 7 73 2]	s	[59]	112
53	s	[41 6 74 2]	s	[60]	112
54	s	[42 7 75 2]	s	[60]	112
55	s	[43 6 76 2]	-	[]	96
56	s	[44 7 77 2]	-	[]	96
57	m	[46 47]	m	[8 31 3 61 4 62 4]	176
58	m	[48 49]	m	[9 30 3 63 4 64 4]	176
59	m	[50 51 52]	m	[10 29 3]	128
60	m	[53 54]	m	[11 31 3]	112
61	m	[27 5 28 5 57 4]	-	[]	128
62	m	[27 5 28 5 57 4]	-	[]	128
63	m	[27 5 28 5 58 4]	-	[]	128
64	m	[27 5 28 5 58 4]	-	[]	128
65	-	[]	s	[32 2 46 6 57]	112
66	-	[]	s	[33 2 46 7 57]	112
67	-	[]	s	[34 2 47 6 57]	112
68	-	[]	s	[35 2 47 7 57]	112
69	-	[]	s	[36 2 48 6 58]	112
70	-	[]	s	[37 2 49 6 58]	112
71	-	[]	s	[38 2 50 7 59]	112
72	-	[]	s	[39 2 51 6 59]	112
73	-	[]	s	[40 2 52 7 59]	112
74	-	[]	s	[41 2 53 6 60]	112
75	-	[]	s	[42 2 54 7 60]	112
76	-	[]	s	[43 2 55 6]	96
77	-	[]	s	[44 2 56 7]	96
78	-	[]	s	[45 2]	64
79	s	[84 79]	m	[84 86 2]	112
80	-	[]	s	[87 2]	64
81	-	[]	s	[88 2]	64
82	-	[]	s	[89 2]	64
83	-	[]	s	[90 2]	64
84	s	[79 84]	s	[79]	80
85	-	[]	-	[]	32
86	s	[79 2 84 79]	m	[120 5 121 5 122 5 123 5]	224
87	s	[80 2]	m	[120 5 121 5 122 5 123 5]	192
88	m	[81 2 118 3]	-	[]	96
89	m	[82 2 117 3]	-	[]	96
90	m	[83 2 116 3 119 3]	-	[]	128
91	s	[124 2]	s	[105 6 116]	112

OID	Forward Fan-out Type	A List of Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Referencing OIDs (Backward Navigation)	Bits
92	s	[125 2]	s	[105 7 116]	112
93	s	[126 2]	s	[106 6 116]	112
94	s	[127 2]	s	[106 7 116]	112
95	s	[128 2]	s	[107 6 117]	112
96	s	[129 2]	s	[108 6 117]	112
97	s	[130 2]	s	[109 7 118]	112
98	s	[131 2]	s	[110 6 118]	112
99	s	[132 2]	s	[111 7 118]	112
100	s	[133 2]	s	[112 6 119]	112
101	s	[134 2]	s	[113 7 119]	112
102	s	[135 2]	s	[114 6]	96
103	s	[136 2]	s	[115 7]	96
104	s	[137 2]	-	[]	64
105	m	[91 6 92 7]	s	[116]	112
106	m	[93 6 94 7]	s	[116]	112
107	s	[95 6 128 2]	s	[117]	112
108	s	[96 6 129 2]	s	[117]	112
109	s	[97 7 130 2]	s	[118]	112
110	s	[98 6 131 2]	s	[118]	112
111	s	[99 7 132 2]	s	[118]	112
112	s	[100 6 133 2]	s	[119]	112
113	s	[101 7 134 2]	s	[119]	112
114	s	[102 6 135 2]	-	[]	96
115	s	[103 7 136 2]	-	[]	96
116	m	[105 106]	m	[12 90 3 120 4 121 4]	176
117	m	[107 108]	m	[13 89 3 122 4 123 4]	176
118	m	[109 110 111]	m	[14 88 3]	128
119	m	[112 113]	m	[15 90 3]	112
120	m	[86 5 87 5 116 4]	-	[]	128
121	m	[86 5 87 5 116 4]	-	[]	128
122	m	[86 5 87 5 117 4]	-	[]	128
123	m	[86 5 87 5 117 4]	-	[]	128
124	-	[]	s	[91 2 105 6 116]	112
125	-	[]	s	[92 2 105 7 116]	112
126	-	[]	s	[93 2 106 6 116]	112
127	-	[]	s	[94 2 106 7 116]	112
128	-	[]	s	[95 2 107 6 117]	112
129	-	[]	s	[96 2 108 6 117]	112
130	-	[]	s	[97 2 109 7 118]	112
131	-	[]	s	[98 2 110 6 118]	112
132	-	[]	s	[99 2 111 7 118]	112
133	-	[]	s	[100 2 112 6 119]	112
134	-	[]	s	[101 2 113 7 119]	112
135	-	[]	s	[102 2 114 6]	96
136	-	[]	s	[103 2 115 7]	96
137	-	[]	s	[104 2]	64
138	s	[143 138]	m	[143 145 2]	112
139	-	[]	s	[146 2]	64
140	-	[]	s	[147 2]	64
141	-	[]	s	[148 2]	64

OID	Forward Fan-out Type	A List of Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Referencing OIDs (Backward Navigation)	Bits
142	-	[]	s	[149 2]	64
143	s	[138 143]	s	[138]	80
144	-	[]	-	[]	32
145	s	[138 2 143 138]	m	[179 5 180 5 181 5 182 5]	224
146	s	[139 2]	m	[179 5 180 5 181 5 182 5]	192
147	m	[140 2 177 3]	-	[]	96
148	m	[141 2 176 3]	-	[]	96
149	m	[142 2 175 3 178 3]	-	[]	128
150	s	[183 2]	s	[164 6 175]	112
151	s	[184 2]	s	[164 7 175]	112
152	s	[185 2]	s	[165 6 175]	112
153	s	[186 2]	s	[165 7 175]	112
154	s	[187 2]	s	[166 6 176]	112
155	s	[188 2]	s	[167 6 176]	112
156	s	[189 2]	s	[168 7 177]	112
157	s	[190 2]	s	[169 6 177]	112
158	s	[191 2]	s	[170 7 177]	112
159	s	[192 2]	s	[171 6 178]	112
160	s	[193 2]	s	[172 7 178]	112
161	s	[194 2]	s	[173 6]	96
162	s	[195 2]	s	[174 7]	96
163	s	[196 2]	-	[]	64
164	m	[150 6 151 7]	s	[175]	112
165	m	[152 6 153 7]	s	[175]	112
166	s	[154 6 187 2]	s	[176]	112
167	s	[155 6 188 2]	s	[176]	112
168	s	[156 7 189 2]	s	[177]	112
169	s	[157 6 190 2]	s	[177]	112
170	s	[158 7 191 2]	s	[177]	112
171	s	[159 6 192 2]	s	[178]	112
172	s	[160 7 193 2]	s	[178]	112
173	s	[161 6 194 2]	-	[]	96
174	s	[162 7 195 2]	-	[]	96
175	m	[164 165]	m	[16 149 3 179 4 180 4]	176
176	m	[166 167]	m	[17 148 3 181 4 182 4]	176
177	m	[168 169 170]	m	[18 147 3]	128
178	m	[171 172]	m	[19 149 3]	112
179	m	[145 5 146 5 175 4]	-	[]	128
180	m	[145 5 146 5 175 4]	-	[]	128
181	m	[145 5 146 5 176 4]	-	[]	128
182	m	[145 5 146 5 176 4]	-	[]	128
183	-	[]	s	[150 2 164 6 175]	112
184	-	[]	s	[151 2 164 7 175]	112
185	-	[]	s	[152 2 165 6 175]	112
186	-	[]	s	[153 2 165 7 175]	112
187	-	[]	s	[154 2 166 6 176]	112
188	-	[]	s	[155 2 167 6 176]	112
189	-	[]	s	[156 2 168 7 177]	112
190	-	[]	s	[157 2 169 6 177]	112
191	-	[]	s	[158 2 170 7 177]	112

OID	Forward Fan-out Type	A List of Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Referencing OIDs (Backward Navigation)	Bits
192	-	[]	s	[159 2 171 6 178]	112
193	-	[]	s	[160 2 172 7 178]	112
194	-	[]	s	[161 2 173 6]	96
195	-	[]	s	[162 2 174 7]	96
196	-	[]	s	[163 2]	64
					20,064

m - multiple references

s - single reference

- - no reference

Table E.2 MIC implemented with a SICF coding technique

OID	Forward SICF code		Backward SICF code		Bits
	N	D	N	D	
8	1	57	0	0	80
9	1	58	0	0	80
10	1	59	0	0	80
11	1	60	0	0	80
12	1	116	0	0	80
13	1	117	0	0	80
14	1	118	0	0	80
15	1	119	0	0	80
16	1	175	0	0	80
17	1	176	0	0	80
18	1	177	0	0	80
19	1	178	0	0	80
20	501	20	55	1,377	80
21	0	0	57	2	80
22	0	0	59	2	80
23	0	0	61	2	80
24	0	0	63	2	80
25	501	25	1	20	80
26	0	0	0	0	80
27	20,941	1,022	159,232,270	9,744,912,851	144
28	43	2	159,232,270	9,744,912,851	144
29	359	8,076	0	0	80
30	353	8,294	0	0	80
31	63,152	1,546,951	0	0	96
32	131	2	15,835	343	80
33	133	2	18,457	400	80
34	135	2	16,178	343	80
35	137	2	18,857	400	80
36	139	2	16,810	349	80
37	141	2	17,159	349	80
38	143	2	20,759	414	80
39	145	2	18,164	355	80
40	147	2	21,587	414	80
41	149	2	19,193	361	80
42	151	2	22,794	421	80
43	153	2	331	6	80
44	155	2	393	7	80
45	157	2	0	0	80
46	1,399	45,000	1	57	80
47	1,483	50,668	1	57	80
48	30,235	836	1	58	80
49	31,517	848	1	58	80
50	38,257	1,003	1	59	80
51	34,153	872	1	59	80

OID	Forward SICF code		Backward SICF code		Bits
	N	D	N	D	
52	41,387	1,031	1	59	80
53	36,885	896	1	60	80
54	44,629	1,059	1	60	80
55	39,713	920	0	0	80
56	47,983	1,087	0	0	80
57	47	2,163	5,788,406	46,491,995	112
58	49	2,353	5,970,803	53,934,078	112
59	2,653	132,702	88	883	96
60	54	2,863	94	1,037	80
61	163,154	4,437,559	0	0	112
62	163,154	4,437,559	0	0	112
63	165,994	4,514,803	0	0	112
64	165,994	4,514,803	0	0	112
65	0	0	1,040,251	32,013	96
66	0	0	1,249,819	37,314	96
67	0	0	1,127,944	32,699	96
68	0	0	1,352,847	38,114	96
69	0	0	1,239,694	33,969	96
70	0	0	1,299,838	34,667	96
71	0	0	1,614,175	41,932	96
72	0	0	1,448,801	36,683	96
73	0	0	1,765,107	43,588	96
74	0	0	1,607,820	38,747	96
75	0	0	1,955,172	46,009	112
76	0	0	29,055	668	80
77	0	0	35,285	793	80
78	0	0	91	2	80
79	6,637	79	173	14,534	80
80	0	0	175	2	80
81	0	0	177	2	80
82	0	0	179	2	80
83	0	0	181	2	80
84	6,637	84	1	79	80
85	0	0	0	0	80
86	1,061,524	13,353	1,146,022,020	137,751,469,201	160
87	161	2	1,146,022,020	137,751,469,201	144
88	713	58,108	0	0	80
89	707	58,326	0	0	80
90	251,657	21,012,821	0	0	112
91	249	2	73,301	697	96
92	251	2	85,481	813	96
93	253	2	73,998	697	96
94	255	2	86,294	813	96
95	257	2	75,338	703	96
96	259	2	76,041	703	96
97	261	2	90,261	827	96
98	263	2	78,108	709	96

OID	Forward SICF code		Backward SICF code		Bits
	N	D	N	D	
99	265	2	91,915	827	96
100	267	2	80,199	715	96
101	269	2	94,361	834	96
102	271	2	685	6	80
103	273	2	806	7	80
104	275	2	0	0	80
105	3,877	353,452	1	116	96
106	3,961	369,032	1	116	96
107	146,937	1,544	1	117	96
108	149,635	1,556	1	117	96
109	177,674	1,829	1	118	96
110	155,103	1,580	1	118	96
111	184,108	1,857	1	118	96
112	160,667	1,604	1	119	96
113	190,654	1,885	1	119	96
114	166,327	1,628	0	0	96
115	197,312	1,913	0	0	96
116	106	11,131	63,525,275	763,006,539	112
117	108	11,557	64,915,284	844,625,363	112
118	12,211	1,331,110	265	3,713	96
119	113	12,657	271	4,068	80
120	1,017,769	87,731,222	0	0	112
121	1,017,769	87,731,222	0	0	112
122	1,026,509	88,484,606	0	0	112
123	1,026,509	88,484,606	0	0	112
124	0	0	13,477,510	147,299	112
125	0	0	15,888,781	171,775	112
126	0	0	13,902,447	148,693	112
127	0	0	16,385,988	173,401	112
128	0	0	14,456,343	151,379	112
129	0	0	14,743,401	152,785	112
130	0	0	17,681,114	181,349	112
131	0	0	15,456,758	156,925	112
132	0	0	18,372,958	184,657	112
133	0	0	16,191,499	161,113	112
134	0	0	19,239,517	189,556	112
135	0	0	141,037	1,376	96
136	0	0	167,563	1,619	96
137	0	0	209	2	80
138	19,735	138	291	41,615	80
139	0	0	293	2	80
140	0	0	295	2	80
141	0	0	297	2	80
142	0	0	299	2	80
143	19,735	143	1	138	80
144	0	0	0	0	80
145	5,485,639	39,608	3,730,590,770	668,521,038,801	160

OID	Forward SICF code		Backward SICF code		Bits
	N	D	N	D	
146	279	2	3,730,590,770	668,521,038,801	144
147	1,067	149,912	0	0	96
148	1,061	150,130	0	0	96
149	565,478	80,579,811	0	0	112
150	367	2	172,539	1,051	96
151	369	2	201,239	1,226	96
152	371	2	173,590	1,051	96
153	373	2	202,465	1,226	96
154	375	2	175,638	1,057	96
155	377	2	176,695	1,057	96
156	379	2	208,497	1,240	96
157	381	2	179,824	1,063	96
158	383	2	210,977	1,240	96
159	385	2	182,977	1,069	96
160	387	2	214,662	1,247	96
161	389	2	1,039	6	80
162	391	2	1,219	7	80
163	393	2	0	0	80
164	6,355	954,308	1	175	96
165	6,439	979,800	1	175	96
166	347,183	2,252	1	176	96
167	351,297	2,264	1	176	96
168	414,559	2,655	1	177	96
169	359,597	2,288	1	177	96
170	424,297	2,683	1	177	96
171	367,993	2,312	1	178	96
172	434,147	2,711	1	178	96
173	376,485	2,336	0	0	96
174	444,109	2,739	0	0	96
175	165	27,061	232,347,816	3,719,120,963	112
176	167	27,723	235,947,965	4,012,706,072	112
177	28,731	4,826,978	442	7,959	96
178	172	29,413	448	8,515	80
179	2,568,584	372,957,695	0	0	112
180	2,568,584	372,957,695	0	0	112
181	2,583,224	375,083,419	0	0	112
182	2,583,224	375,083,419	0	0	112
183	0	0	52,091,889	346,129	112
184	0	0	61,160,543	403,704	112
185	0	0	53,104,702	348,231	112
186	0	0	62,344,333	406,156	112
187	0	0	54,434,920	352,333	112
188	0	0	55,115,980	354,447	112
189	0	0	65,453,001	418,234	112
190	0	0	56,811,451	360,711	112
191	0	0	67,075,629	423,194	112
192	0	0	58,539,634	367,023	112

OID	Forward SICF code		Backward SICF code		Bits
	N	D	N	D	
193	0	0	69,106,022	430,571	112
194	0	0	336,563	2,084	96
195	0	0	397,309	2,445	96
196	0	0	327	2	80
					17,936

N - Numerator

D - Denominator

Table E.3 MIC implemented with a Start/Stop coding technique

OID	Forward Fan-out Type	A List of Coded Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Coded Referencing OIDs (Backward Navigation)	Bits
8	s	[110100101]	-	[]	41
9	s	[110100110]	-	[]	41
10	s	[110100111]	-	[]	41
11	s	[110101000]	-	[]	41
12	s	[111000100000]	-	[]	44
13	s	[111000100001]	-	[]	44
14	s	[111000100010]	-	[]	44
15	s	[111000100011]	-	[]	44
16	s	[111001011011]	-	[]	44
17	s	[111001011100]	-	[]	44
18	s	[111001011101]	-	[]	44
19	s	[111001011110]	-	[]	44
20	s	[110000101 110000000]	m	[110000101 110000111 010]	71
21	-	[]	s	[110001000 010]	44
22	-	[]	s	[110001001 010]	44
23	-	[]	s	[110001010 010]	44
24	-	[]	s	[110001011 010]	44
25	s	[110000000 110000101]	s	[110000000]	59
26	-	[]	-	[]	32
27	s	[110000000 010 110000101 110000000]	m	[110101001 100001 110101010 100001 110101011 100001 110101100 100001]	122
28	s	[110000001 010]	m	[110101001 100001 110101010 100001 110101011 100001 110101100 100001]	104
29	m	[110000010 010 110100111 011]	-	[]	56
30	m	[110000011 010 110100110 011]	-	[]	56
31	m	[110000100 010 110100101 011 110101000 011]	-	[]	68
32	s	[110101101 010]	s	[110011010 100010 110100101]	68
33	s	[110101110 010]	s	[110011010 100011 110100101]	68
34	s	[110101111 010]	s	[110011011 100010 110100101]	68
35	s	[110110000 010]	s	[110011011 100011 110100101]	68
36	s	[110110001 010]	s	[110011100 100010 110100110]	68
37	s	[110110010 010]	s	[110011101 100010 110100110]	68
38	s	[110110011 010]	s	[110011110 100011 110100111]	68
39	s	[110110100 010]	s	[110011111 100010 110100111]	68
40	s	[110110101 010]	s	[110100000 100011 110100111]	68
41	s	[110110110 010]	s	[110100001 100010 110101000]	68
42	s	[110110111 010]	s	[110100010 100011 110101000]	68
43	s	[110111000 010]	s	[110100011 100010]	59
44	s	[110111001 010]	s	[110100100 100011]	59
45	s	[110111010 010]	-	[]	44

OID	Forward Fan-out Type	A List of Coded Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Coded Referencing OIDs (Backward Navigation)	Bits
46	m	[110001100 100010 110001101 100011]	s	[110100101]	71
47	m	[110001110 100010 110001111 100011]	s	[110100101]	71
48	s	[110010000 100010 110110001 010]	s	[110100110]	68
49	s	[110010001 100010 110110010 010]	s	[110100110]	68
50	s	[110010010 100011 110110011 010]	s	[110100111]	68
51	s	[110010011 100010 110110100 010]	s	[110100111]	68
52	s	[110010100 100011 110110101 010]	s	[110100111]	68
53	s	[110010101 100010 110110110 010]	s	[110101000]	68
54	s	[110010110 100011 110110111 010]	s	[110101000]	68
55	s	[110010111 100010 110111000 010]	-	[]	59
56	s	[110011000 100011 110111001 010]	-	[]	59
57	m	[110011010 110011011]	m	[100100 110001011 011 110101001 100000 110101010 100000]	98
58	m	[110011100 110011101]	m	[100101 110001010 011 110101011 100000 110101100 100000]	98
59	m	[110011110 110011111 110100000]	m	[100110 110001001 011]	77
60	m	[110100001 110100010]	m	[100111 110001011 011]	68
61	m	[110000111 100001 110001000 100001 110100101 100000]	-	[]	77
62	m	[110000111 100001 110001000 100001 110100101 100000]	-	[]	77
63	m	[110000111 100001 110001000 100001 110100110 100000]	-	[]	77
64	m	[110000111 100001 110001000 100001 110100110 100000]	-	[]	77
65	-	[]	s	[110001100 010 110011010 100010 110100101]	68
66	-	[]	s	[110001101 010 110011010 100011 110100101]	68
67	-	[]	s	[110001110 010 110011011 100010 110100101]	68
68	-	[]	s	[110001111 010 110011011 100011 110100101]	68

OID	Forward Fan-out Type	A List of Coded Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Coded Referencing OIDs (Backward Navigation)	Bits
69	-	[]	s	[110010000 010 110011100 100010 110100110]	68
70	-	[]	s	[110010001 010 110011101 100010 110100110]	68
71	-	[]	s	[110010010 010 110011110 100011 110100111]	68
72	-	[]	s	[110010011 010 110011111 100010 110100111]	68
73	-	[]	s	[110010100 010 110100000 100011 110100111]	68
74	-	[]	s	[110010101 010 110100001 100010 110101000]	68
75	-	[]	s	[110010110 010 110100010 100011 110101000]	68
76	-	[]	s	[110010111 010 110100011 100010]	59
77	-	[]	s	[110011000 010 110100100 100011]	59
78	-	[]	s	[110011001 010]	44
79	s	[111000000000 110111011]	m	[111000000000 111000000010 010]	80
80	-	[]	s	[111000000011 010]	47
81	-	[]	s	[111000000100 010]	47
82	-	[]	s	[111000000101 010]	47
83	-	[]	s	[111000000110 010]	47
84	s	[110111011 111000000000]	s	[110111011]	62
85	-	[]	-	[]	32
86	s	[110111011 010 111000000000 110111011]	m	[111000100100 100001 111000100101 100001 111000100110 100001 111000100111 100001]	137
87	s	[110111100 010]	m	[111000100100 100001 111000100101 100001 111000100110 100001 111000100111 100001]	116
88	m	[110111101 010 111000100010 011]	-	[]	59
89	m	[110111110 010 111000100001 011]	-	[]	59
90	m	[110111111 010 111000100000 011 111000100011 011]	-	[]	74
91	s	[111000101000 010]	s	[111000010101 100010 111000100000]	77
92	s	[111000101001 010]	s	[111000010101 100011 111000100000]	77
93	s	[111000101010 010]	s	[111000010110 100010 111000100000]	77

OID	Forward Fan-out Type	A List of Coded Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Coded Referencing OIDs (Backward Navigation)	Bits
94	s	[111000101011 010]	s	[111000010110 100011 111000100000]	77
95	s	[111000101100 010]	s	[111000010111 100010 111000100001]	77
96	s	[111000101101 010]	s	[111000011000 100010 111000100001]	77
97	s	[111000101110 010]	s	[111000011001 100011 111000100010]	77
98	s	[111000101111 010]	s	[111000011010 100010 111000100010]	77
99	s	[111000110000 010]	s	[111000011011 100011 111000100010]	77
100	s	[111000110001 010]	s	[111000011100 100010 111000100011]	77
101	s	[111000110010 010]	s	[111000011101 100011 111000100011]	77
102	s	[111000110011 010]	s	[111000011110 100010]	65
103	s	[111000110100 010]	s	[111000011111 100011]	65
104	s	[111000110101 010]	-	[]	47
105	m	[111000000111 100010 111000001000 100011]	s	[111000100000]	80
106	m	[111000001001 100010 111000001010 100011]	s	[111000100000]	80
107	s	[111000001011 100010 111000101100 010]	s	[111000100001]	77
108	s	[111000001100 100010 111000101101 010]	s	[111000100001]	77
109	s	[111000001101 100011 111000101110 010]	s	[111000100010]	77
110	s	[111000001110 100010 111000101111 010]	s	[111000100010]	77
111	s	[111000001111 100011 111000110000 010]	s	[111000100010]	77
112	s	[111000010000 100010 111000110001 010]	s	[111000100011]	77
113	s	[111000010001 100011 111000110010 010]	s	[111000100011]	77
114	s	[111000010010 100010 111000110011 010]	-	[]	65
115	s	[111000010011 100011 111000110100 010]	-	[]	65
116	m	[111000010101 111000010110]	m	[101000 111000000110 011 111000100100 100000 111000100101 100000]	113
117	m	[111000010111 111000011000]	m	[101001 111000000101 011 111000100110 100000 111000100111 100000]	113

OID	Forward Fan-out Type	A List of Coded Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Coded Referencing OIDs (Backward Navigation)	Bits
118	m	[111000011001 111000011010 111000011011]	m	[101010 111000000100 011]	89
119	m	[111000011100 111000011101]	m	[101011 111000000110 011]	77
120	m	[111000000010 100001 111000000011 100001 111000100000 100000]	-	[]	86
121	m	[111000000010 100001 111000000011 100001 111000100000 100000]	-	[]	86
122	m	[111000000010 100001 111000000011 100001 111000100001 100000]	-	[]	86
123	m	[111000000010 100001 111000000011 100001 111000100001 100000]	-	[]	86
124	-	[]	s	[111000000111 010 111000010101 100010 111000100000]	77
125	-	[]	s	[111000001000 010 111000010101 100011 111000100000]	77
126	-	[]	s	[111000001001 010 111000010110 100010 111000100000]	77
127	-	[]	s	[111000001010 010 111000010110 100011 111000100000]	77
128	-	[]	s	[111000001011 010 111000010111 100010 111000100001]	77
129	-	[]	s	[111000001100 010 111000011000 100010 111000100001]	77
130	-	[]	s	[111000001101 010 111000011001 100011 111000100010]	77
131	-	[]	s	[111000001110 010 111000011010 100010 111000100010]	77
132	-	[]	s	[111000001111 010 111000011011 100011 111000100010]	77
133	-	[]	s	[111000010000 010 111000011100 100010 111000100011]	77

OID	Forward Fan-out Type	A List of Coded Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Coded Referencing OIDs (Backward Navigation)	Bits
134	-	[]	s	[111000010001 010 111000011101 100011 111000100011]	77
135	-	[]	s	[111000010010 010 111000011110 100010]	65
136	-	[]	s	[111000010011 010 111000011111 100011]	65
137	-	[]	s	[111000010100 010]	47
138	s	[111000111011 111000110110]	m	[111000111011 111000111101 010]	83
139	-	[]	s	[111000111110 010]	47
140	-	[]	s	[111000111111 010]	47
141	-	[]	s	[111001000000 010]	47
142	-	[]	s	[111001000001 010]	47
143	s	[111000110110 111000111011]	s	[111000110110]	68
144	-	[]	-	[]	32
145	s	[111000110110 010 111000111011 111000110110]	m	[111001011111 100001 111001100000 100001 111001100001 100001 111001100010 100001]	143
146	s	[111000110111 010]	m	[111001011111 100001 111001100000 100001 111001100001 100001 111001100010 100001]	119
147	m	[111000111000 010 111001011101 011]	-	[]	62
148	m	[111000111001 010 111001011100 011]	-	[]	62
149	m	[111000111010 010 111001011011 011 111001011110 011]	-	[]	77
150	s	[111001100011 010]	s	[111001010000 100010 111001011011]	77
151	s	[111001100100 010]	s	[111001010000 100011 111001011011]	77
152	s	[111001100101 010]	s	[111001010001 100010 111001011011]	77
153	s	[111001100110 010]	s	[111001010001 100011 111001011011]	77
154	s	[111001100111 010]	s	[111001010010 100010 111001011100]	77
155	s	[111001101000 010]	s	[111001010011 100010 111001011100]	77
156	s	[111001101001 010]	s	[111001010100 100011 111001011101]	77
157	s	[111001101010 010]	s	[111001010101 100010 111001011101]	77

OID	Forward Fan-out Type	A List of Coded Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Coded Referencing OIDs (Backward Navigation)	Bits
158	s	[111001101011 010]	s	[111001010110 100011 111001011101]	77
159	s	[111001101100 010]	s	[111001010111 100010 111001011110]	77
160	s	[111001101101 010]	s	[111001011000 100011 111001011110]	77
161	s	[111001101110 010]	s	[111001011001 100010]	65
162	s	[111001101111 010]	s	[111001011010 100011]	65
163	s	[111001110000 010]	-	[]	47
164	m	[111001000010 100010 111001000011 100011]	s	[111001011011]	80
165	m	[111001000100 100010 111001000101 100011]	s	[111001011011]	80
166	s	[111001000110 100010 111001100111 010]	s	[111001011100]	77
167	s	[111001000111 100010 111001101000 010]	s	[111001011100]	77
168	s	[111001001000 100011 111001101001 010]	s	[111001011101]	77
169	s	[111001001001 100010 111001101010 010]	s	[111001011101]	77
170	s	[111001001010 100011 111001101011 010]	s	[111001011101]	77
171	s	[111001001011 100010 111001101100 010]	s	[111001011110]	77
172	s	[111001001100 100011 111001101101 010]	s	[111001011110]	77
173	s	[111001001101 100010 111001101110 010]	-	[]	65
174	s	[111001001110 100011 111001101111 010]	-	[]	65
175	m	[111001010000 111001010001]	m	[101100 111001000001 011 111001011111 100000 111001100000 100000]	113
176	m	[111001010010 111001010011]	m	[101101 111001000000 011 111001100001 100000 111001100010 100000]	113
177	m	[111001010100 111001010101 111001010110]	m	[101110 111000111111 011]	89
178	m	[111001010111 111001011000]	m	[101111 111001000001 011]	77
179	m	[111000111101 100001 111000111110 100001 111001011011 100000]	-	[]	86
180	m	[111000111101 100001 111000111110 100001 111001011011 100000]	-	[]	86

OID	Forward Fan-out Type	A List of Coded Referenced OIDs (Forward Navigation)	Backward Fan-out Type	A List of Coded Referencing OIDs (Backward Navigation)	Bits
181	m	[111000111101 100001 111000111110 100001 111001011100 100000]	-	[]	86
182	m	[111000111101 100001 111000111110 100001 111001011100 100000]	-	[]	86
183	-	[]	s	[111001000010 010 111001010000 100010 111001011011]	77
184	-	[]	s	[111001000011 010 111001010000 100011 111001011011]	77
185	-	[]	s	[111001000100 010 111001010001 100010 111001011011]	77
186	-	[]	s	[111001000101 010 111001010001 100011 111001011011]	77
187	-	[]	s	[111001000110 010 111001010010 100010 111001011100]	77
188	-	[]	s	[111001000111 010 111001010011 100010 111001011100]	77
189	-	[]	s	[111001001000 010 111001010100 100011 111001011101]	77
190	-	[]	s	[111001001001 010 111001010101 100010 111001011101]	77
191	-	[]	s	[111001001010 010 111001010110 100011 111001011101]	77
192	-	[]	s	[111001001011 010 111001010111 100010 111001011110]	77
193	-	[]	s	[111001001100 010 111001011000 100011 111001011110]	77
194	-	[]	s	[111001001101 010 111001011001 100010]	65
195	-	[]	s	[111001001110 010 111001011010 100011]	65
196	-	[]	s	[111001001111 010]	47
					13,332

m - multiple references

s - single reference

· - no reference

APPENDIX F

RESULTS OF USING MIC TO SUPPORT A QUERY Q

Results of using MIC implemented with no coding technique, a SICF coding technique and a Start/Stop data compression technique (Pigeon, 2001) to support a query Q are the same and presented as the following.

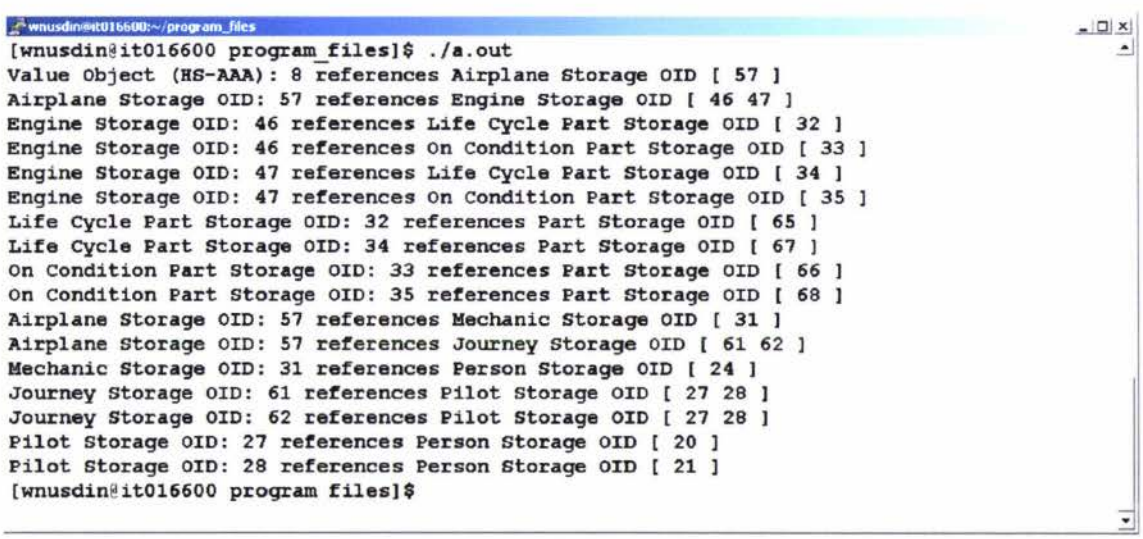


Figure F.1 Results of Using MIC Implemented with the Extended POS to Support a Query Q