# Studies on Facilities for Persistent Programming Languages and Their Implementations

**Masayoshi Aritsugi**

**Department of Computer Science and Communication Engineering**
**Kyushu University, 6-10-1 Hakozaki, Higashi-ku, Fukuoka 812-81 Japan**

**A Dissertation Submitted to the Division of Engineering**
**in Partial Fulfillment of the Requirements for the Degree of Doctor of Engineering**
**in Computer Science and Communication Engineering**

# Contents

# Abstract

Databases become widely applied to such areas as computer-aided design, computer-aided manufacturing, computer-integrated manufacturing, software engineering, and multimedia applications. Data structures and data processes handled in these areas are so complex that it is impossible to process such data efficiently with relational databases. Therefore, many researchers and commercial organizations have focused on object-oriented databases to benefit by their high abilities to model entities in the real world and good performance, and developed prototype and commercial systems.

In this dissertation, we exploit memory-mapped I/O environment instead of using buffer pool environment which almost other work employs to implement persistence of objects. In the buffer pool environment systems have to convert data formats between on primary memory and on secondary storage, and to decompose data bigger than the buffer size. On the contrary, we can avoid the problem with the memory-mapped I/O environment. This dissertation also discusses several implementations of persistent pointers. As a result, we know the fact that performance of non-swizzling approaches is not so poor compared with that of swizzling approaches. Moreover, with taking functionalities provided by non-swizzling approaches into account, a non-swizzling approach can be a good alternative in order to implement persistent pointers.

As we manipulate objects with a long life span, it is needed to change forms and/or behaviors of persistent objects so as to adapt existed objects to up-to-date requirements for applications. To this end, we introduce multiple type objects in this dissertation: any persistent object can get/lose their types dynamically in this concept.

Also, this dissertation proposes object-oriented views. The view mechanism in relational databases is quite convenient for users. The mechanism allows users to deal with relations as what they expect, and provides securities on the relations in some sense. Recently many researchers have tried to integrate the view mechanism in relational databases into object-oriented databases. We propose a view mechanism that is implemented by applying the multiple type concept to sets of objects. Update on a view can be automatically propagated to its base set in the implementation.

# Acknowledgments

First of all, I would like to express my sincere appreciation to Professor Akifumi Makinouchi at Kyushu University for his irreplaceable encouragement, constructive criticism, and guidance. He is my supervisor at Kyushu University, and led me to this research field. He made the course of my study I enjoyed even more enjoyable. The opportunities that he gave me for conducting research were outstanding.

I also would like to express my gratitude to Professor Tim Merrett at McGill University, Montréal, Canada, for his kindness of being my supervisor while I was attending McGill as a visiting doctoral student. I really enjoyed my stay in Montréal. He suggested me that type checking mechanism in multiple type concept could be one of key issues concerning our work.

I gratefully appreciate the careful reading of this dissertation by Professor Kazuo Ushijima and Professor Fumihiro Matsuo at Kyushu University. They were the committee members of this dissertation, and gave me many valuable comments.

My thanks also go to Associate Professor Norihiko Yoshida in the Department of Computer Science and Communication Engineering at Kyushu University for his valuable suggestions and comments on the studies. Associate Professor Hirofumi Amano in Computer Center at Kyushu University and Associate Professor Ge Yu in the Department of Computer at Northeastern University, China, who has stayed at Kyushu University as a visiting researcher, also deserve special thanks. They proofread this dissertation and gave me remarkable comments making this dissertation better.

Anna Lin and Martin Santavy helped me to start my life in Montréal. My life there was quite comfortable. Without their help, I could not have leaded such good time in Montréal. I really appreciate their help. Pung Chitra Hay and Hon-Hing Chen are good friends from when I was at McGill University. They were very kind enough to share much time with me at the university. I owed them and other friends at McGill University that I could touch diverse cultures and enjoy them. I was able to skill up my English by communicating with them.

I would like to offer special thanks to the members of Makinouchi and Yoshida Laboratories in the Department of Computer Science and Communication Engineering at Kyushu University. In particular, I thank Keiichi Teramoto (currently with Toshiba Co., Japan) for that he discussed various issues concerning most of this dissertation with me, and implemented prototype systems to concrete our ideas. I would also like to thank Dr. Masatoshi Arikawa (currently at Hiroshima City University) and Mr. Susumu Kuroki for all of the assistance that they provided during my graduate career.

Finally, I would like to express my sincere gratitude and thanks to my father and my mother for always believing in me and for helping me to believe that I could earn a degree of doctor. Getting a degree is extremely hard work, and I could not have made it without their unconditional supports.

# Chapter 1

# Introduction

## 1.1 Next Generation Databases

It is said that the first database management system in the world was IDS (Integrated Data Store) which was available from General Electric in the U. S. in 1963. In 1968, IBM brought out IMS (Information Management System), which was based on a hierarchical data model. In 1970, the relational data model was proposed by E. F. Codd [Codd70]. This data model has the following features:

- It is simple.
- It provides good data independence.
- It provides nonprocedural data management languages.
- It can be applied to distributed environments easily.
- It is based on mathematical principles.

In the 1980s, many relational database (RDB) systems were developed and put on the market. Since then, RDBs have been generally accepted around the world. These databases have been mainly used in the business area, for example, for the management of personnel matters and/or sales and inventory data in a company.

Also, in the 1980s, many researchers tried to apply database systems more and more widely to such areas as knowledge bases, artificial intelligence, software engineering, computer-aided design, computer-integrated manufacturing, and multimedia applications. Data processed in these systems have complex structures and it is difficult to handle such complex data efficiently with RDBs[1]. To model these new applications and to manage data in such applications efficiently, object-oriented databases (OODBs) [ABD+89] have been focused on. OODBs have their own shortcomings, though. For example, the model is not based on mathematical principles; studies of OODBs have tended to make actual systems and applications rather than to investigate mathematical aspects. Therefore, there is no theoretical way to design, compose, and decompose databases with the object-oriented concept. But, in our opinion, current research tends to explore with OODBs, and to use the OODBs to manage a large amount of complex data because of the high ability to model the new applications.

A goal of object-oriented database systems is to provide a computing environment merging programming languages and databases with which users can define and manipulate complex objects, and can make programs with which they can process what they really expect to do in order to support data-intensive applications. To this end, we have been designing and developing a persistent programming language; using this language users can handle persistent objects as easily as volatile objects. Persistent objects are the objects that can be stored on secondary storage and can be reused after the program which creates them terminates. The topic of the dissertation, persistent programming language, is an approach to object-oriented database systems.

## 1.2 Persistent Programming Languages

Persistent programming languages are the languages which can handle persistent objects and volatile objects in the same way, i.e., an object of any type in the languages can be stored on secondary storage and can be reused [AB87]. Persistent objects are the objects which exist beyond the life of the program that creates the objects, and users can make use of them again and again until the objects are deleted from secondary storage. Therefore, the languages include some storage mechanisms in addition to computational facilities that usual programming languages provide.

[1]There are some studies to extend the relational data model to manipulate complex data, e.g., [ERDB90].

Programming languages have provided convenience to define complex data and to process complex manipulations on them. However, it is difficult for the conventional programming languages to deal with persistent objects. Conventionally, there is no way in programming languages without using direct file I/O operations to store data on secondary storage and reuse them after the termination of the program which created the data. Thus, users have to make up some tricky devices to store and reuse complex data on secondary storage, or give up treating complex data. To reduce the load of users, persistent programming languages, or database programming languages, which handle persistent objects as easily as when handling non-persistent objects, which are called volatile objects, in terms of both functionality and performance, have been studied.

We have been designing and developing a persistent programming language called INADA. INADA is a part of our ongoing project named "Shusse-Uo" [AABJMT94]. Shusse-Uo is a Japanese word meaning fishes that are called by different names as they grow larger. Figure 1.1 shows the layers of systems in the project. INADA has such features as parallelism and distribution as well as persistence of objects and set-oriented processing as shown in Figure 1.1. We focus on the latter subjects in this dissertation, and do not touch on the parallelism and distribution.

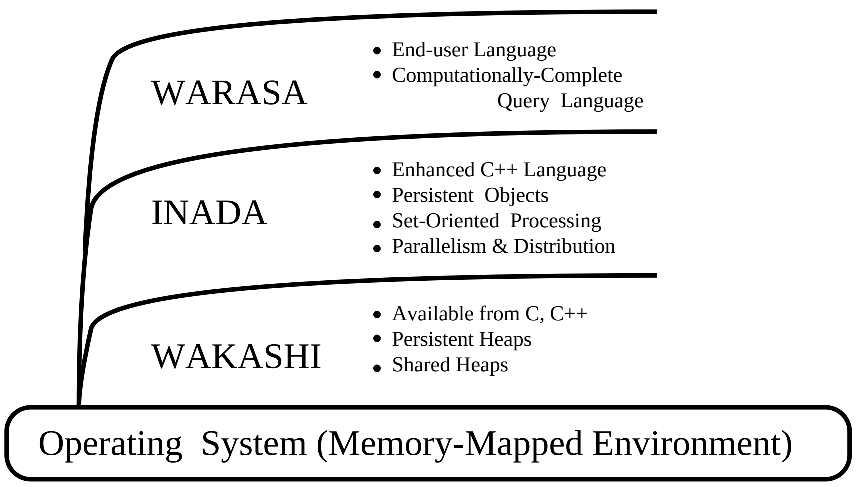
WARASA
• End-user Language
• Computationally-Complete Query Language
INADA
• Enhanced C++ Language
• Persistent Objects
• Set-Oriented Processing
• Parallelism & Distribution
WAKASHI
• Available from C, C++
• Persistent Heaps
• Shared Heaps
Operating System (Memory-Mapped Environment)

Figure 1.1: Layers of Shusse-Uo project

We have tackled plenty of issues which we must face when we consider handling a large number of persistent objects. This dissertation shows some of the issues, namely, how to implement persistent objects and persistent pointers, multiple type objects, and object-oriented views.

We adapted memory-mapped I/O environment to implement persistence of objects. The environment has not so far been studied so enough that we have to examine how to implement persistent pointers which are used to refer persistent objects and can be stored on secondary storage. To this end, we simulated several conversion mechanisms between in-memory and in-disk addresses in the environment.

As long as buffer pool approaches are used to make objects persistent, there are at least the following two problems.

- There are more than one data format in the buffer pool oriented approach [DMFV90]: in-disk data and in-memory data formats. Therefore, systems must build up some device for transferring persistent objects in in-disk forms to those in in-memory forms and vice versa.
- When you handle data larger than the size of a buffer, you have to decompose the data into fragments which fit in the buffer, and to re-form original data from the decomposed fragments when they are needed.

To avoid these problems, we decided to take a memory-mapped approach to implement persistence of objects. And, we examined several implementations of persistent pointers in the memory-mapped I/O environment. The several implementations include both pointer swizzling and nonswizzling techniques. Pointer swizzling techniques convert pointers from their disk format to a more efficient in-memory format (a direct memory address), and assign the in-memory address into the pointer variable to improve the performance of dereferencing persistent pointers.

We introduced the concept of multiple type objects. Persistent objects might be shared among many programs. However, each program uses the persistent objects in its own way. Furthermore, entities in the real world modeled by persistent objects will change as time goes by. Therefore, persistent programming languages should provide some mechanism with which users can handle persistent objects which may change their characteristics whenever needed. The multiple type object mechanism is introduced for this.

Any persistent object can get and/or lose any type, or class in C++ and INADA. Adding and deleting types can be done at any time they are needed. Thus, users can model real world entities which change themselves as time goes by with this concept. It has to be noticed that the multiple inheritance mechanism cannot cover all the facilities that the proposed multiple type object mechanism.

We also proposed views in the object-oriented framework. Many researchers have discussed integrating view mechanisms into object-oriented database systems [MM91, Rund92, SLT91, TYI88]. However, they could not implement the function of views completely. They tried to implement object-oriented views as virtual classes. In order to create a view, they needed to reconstruct the class hierarchy for integrating the virtual classes into the hierarchy. It is very difficult because there is one and only one class hierarchy in their models. We do not take this approach. Instead, we implement an object-oriented view as a virtual set. A set in INADA is an object having interfaces which are defined by the system so that the system can process set-oriented queries. More detailed explanation concerning this is given later on.

## 1.3 Organization of The Dissertation

The remainder of this dissertation discusses the issues mentioned above in detail.

Chapter 2 discusses how to make objects persistent and to implement persistent pointers within the framework of the memory-mapped file I/O environment. To examine several approaches to persistent pointer implementations, we simulated all the several approaches as using the class library of INADA. We show the result of the simulation, and discuss them in Chapter 2. The discussion concludes that a non-swizzling approach is one of good choices for implementing persistent pointers.

Chapter 3 introduces multiple type objects. The syntax and semantics in respect of the objects in INADA are explained in detail. An implementation of multiple type objects is also described. The implementation exploited the flexibility of the non-swizzling approach, which is a natural consequence of the discussion in Chapter 2.

Chapter 4 proposes object-oriented views. Some simple examples are employed to help readers understand the mechanism intuitively. Chapter 4 also proposes programming constructs to define views in INADA, and show how to implement them with functions provided by the persistent programming language. Moreover, a part of the results gotten from a simple simulation of the examples is shown in Chapter 4.

Chapter 5 concludes this dissertation, and presents some future work.

# Chapter 2

# Persistence of Objects

The programming languages which allow to treat with persistent objects as easily as volatile objects have been required. This chapter discusses techniques to implement persistence of objects and persistent pointers. In particular, memory mapped I/O environment is investigated as an underlying platform.

Several techniques for dereferencing persistent pointers have been proposed to improve performance of object-oriented persistent systems. This chapter describes performance experiments to compare several techniques of dereferencing persistent pointers including swizzling and nonswizzling approaches in a memory-mapped I/O environment, and discusses trade-offs among them. All techniques were implemented and evaluated in a persistent programming language called INADA, which exploits facilities provided by a memory-mapped I/O architecture for implementing persistence of objects. The experiments disclose the fact that the differences among the techniques compared in terms of performance are not significant enough to justify discarding nonswizzling techniques.

## 2.1 Introduction

With the capability of modeling entities in the real world, object-oriented database systems have been more and more widely accepted for such areas as computer-aided design, computer-aided manufacturing, geographic information systems, and multimedia applications. In these systems, structures of objects are likely to be complex, i.e., objects might have many pointers. Therefore, it must be the key whether a persistent system efficiently can store the pointers in secondary storage and translate their representations on primary memory into those on secondary storage and vice versa. Some existing systems adapt ‘pointer swizzling’ approach for that because it is believed that the total performance in a swizzling approach is better than in a nonswizzling one (e.g.,[Wil90, WD92, Moss92]).

We think, however, there are problems in the previous work. First, the techniques compared in the previous work are not implemented in a uniform system because the systems used for comparison may not be so flexible enough to do that. Secondly, systems using memory-mapped I/O utilities are not well investigated except for ObjectStore [LLOW91]. As a result, we wonder whether it is really always the case that a pointer swizzling approach outperforms a pointer nonswizzling one.

This chapter reports comparison among several pointer swizzling and nonswizzling techniques under a memory-mapped I/O environment showing how they are different experimentally in terms of performance, and discusses flexibility of the techniques. For the experiments, several techniques of persistent pointers were built into a persistent programming language called INADA [AA93, TAM94, AM95].

INADA is now under development at Kyushu University for supporting data-intensive applications. It is a C++-based language and built on a distributed paged-object storage server called WAKASHI [BM94], which has been developed using a memory-mapped I/O architecture. WAKASHI can avoid mainly the following two problems due to the facilities of the architecture (see Figure 2.1):

- Transformation between in-disk and in-memory formats.
  In buffer pool oriented architectures [DMFV90], there are two types of buffers, i.e., page buffer and object buffer. Since the format of an in-disk object stored in a page buffer is different from that of an in-memory object in an object buffer, transformation is needed whenever objects move between the buffers.

- Fragmentation of large data [Maki90].
  When handling a data larger than the size of a page, the data must be fragmented so as to be stored in a page. This causes inconvenience if it is expected that the whole data to be in a continuous address space for uniform processing. For example, a large image data would remind the readers of this problem.

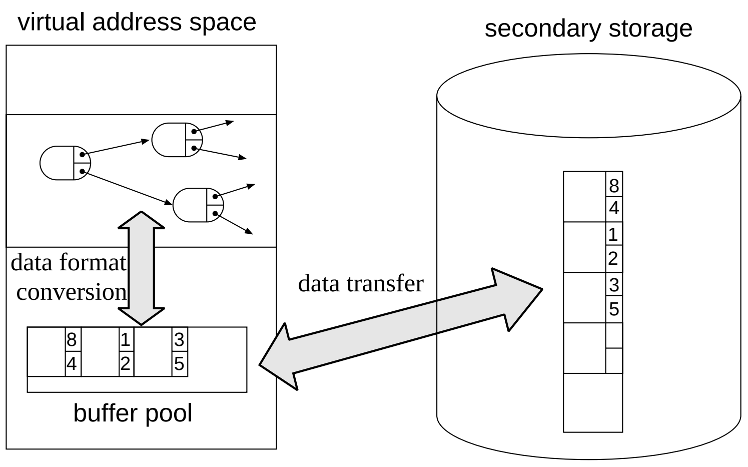

Figure 2.1: Transformation of data formats

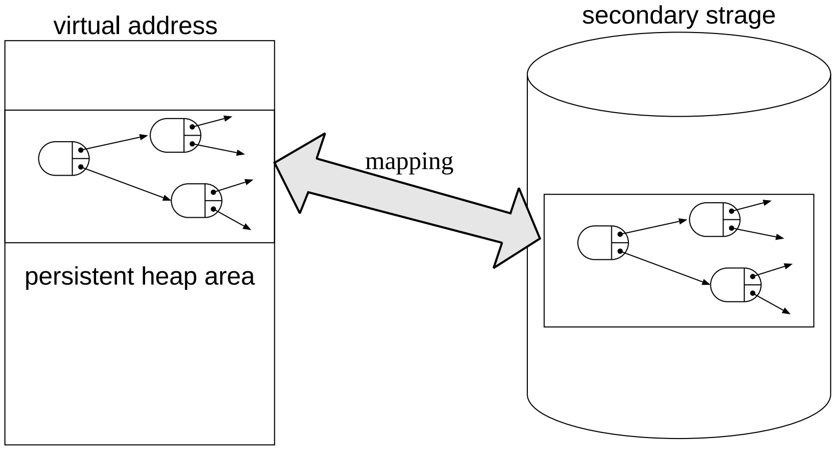

Figure 2.2: Memory-mapped I/O environment

Therefore, it is important to study the memory-mapped I/O architecture (see Figure 2.2).

The swizzling and nonswizzling techniques shown in this chapter are not new. The main contributions of this chapter are:

i) Comparison among several techniques in only one uniform platform and under a memory-mapped I/O environment.

ii) Experiments focusing on the cost of dereferencing persistent pointers.

The results show that the performance of nonswizzling techniques is not so significantly poorer than that of the compared swizzling techniques. In addition, nonswizzling techniques can offer greater flexibility/functionality in some applications.

The experiments used a simple but enough powerful benchmark, called *Dereferencing Benchmark*, instead of well-known benchmarks, e.g., OO1 [CS91] and OO7 [CDN94]. These famous benchmarks are much sophisticated that they can be used for measuring many aspects of OODBSs. However, the objective of using a benchmark in the experiment is to focus only on the cost of dereferencing persistent pointers, i.e., the most important factors are the number of persistent pointers in a page and the ratio of persistent pointers used in a transaction to the number. From the viewpoint, none of the famous benchmarks are suited for that. Thus, we designed and used Dereferencing Benchmark in which an object has persistent pointers, whose numbers can be parameterized, and dummy integers. Its details are described later on.

## 2.2 Persistence of Objects in a Memory-Mapped I/O Architecture

[DMFV90] discusses three alternative client/server architectures that are different in storage management for object-oriented database systems. In [DMFV90], there are two kinds of buffers for storage management, namely page buffer and object buffer (i.e., object cache). This approach has at least two drawbacks in addition to the problems mentioned in [KG+90]. One is the overhead of transformation between in-disk and in-memory formats. Since the format of an in-disk object stored in a page buffer is different from that of an in-memory object, some transformation mechanisms are required whenever objects move between the page buffer and the object buffer. The other problem occurs when treating long data [Maki90]. When a piece of data we want to handle is too long to be stored in a page, the data must be fragmented. This causes inconvenience when the whole data must be treated as a continuous data with uniform processing. The processing of such large image data reminds the readers this problem.

By using a virtual memory approach for storage management, we can avoid these problems effectively [Wil90]. This section describes how to implement persistence of objects in

INADA using the virtual memory approach in a memory mapped I/O architecture.

### 2.2.1 Persistent Heaps and Objects

INADA provides persistent heaps (PHs) [TAM94, AM95] which are portions of the virtual address space and mapped onto files on secondary storage under a memory mapped I/O environment. When a user makes an object be persistent, the user creates the object on a PH like as creating an object on a heap in C++. The difference between making a volatile object on a heap and making a persistent object on a PH is that a declaration of a persistent pointer has the keyword `persistent`, and the operator `new` has a parameter indicating the PH when creating persistent objects [AA93]. Since a PH is a part of the virtual address space, a user can code to manipulate persistent objects in the same way as ordinary volatile objects. In other words, a user can access and manage a file as it is just like as an ordinary heap (Figure 2.3).

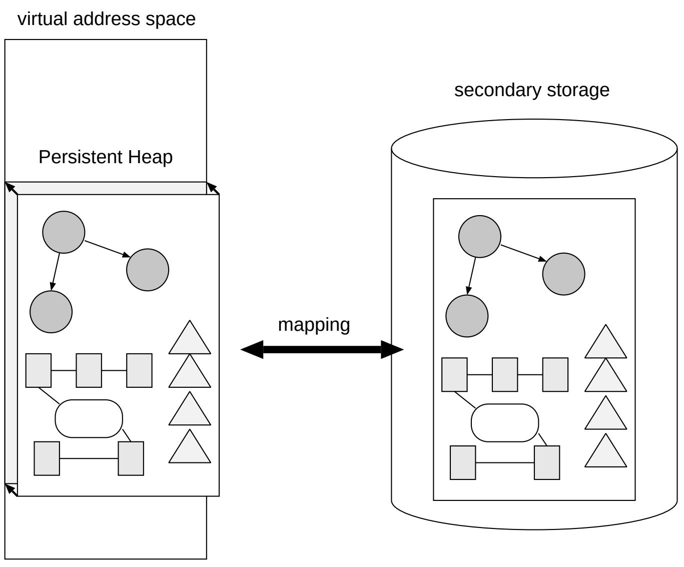

Figure 2.3: A persistent heap

PHs are implemented by using memory mapped I/O utilities. The utilities are currently provided by some operating systems such as Mach [Bar+] and SunOS. WAKASHI [BM94], a distributed paged-object storage server, enhances their memory mapped facilities so as to transfer data implicitly between secondary storage and physical memory, and INADA has

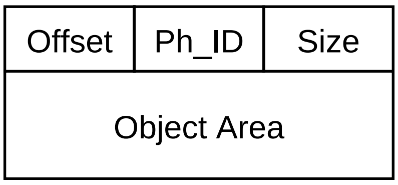

Figure 2.4: A memory block in a PH

been developed with exploitation of WAKASHI. Hence users can manage persistent objects without specifying any I/O operations explicitly.

### 2.2.2 Management of Persistent Heaps

The key issue in the management of PHs is how to allocate objects on the PHs. This subsection describes the way how to allocate objects on a PH used in all the techniques in the chapter.

To make an object persistent, a block including the object with some information for management of the space in a PH is created. Figure 2.4 depicts the block used in all the techniques in the experiment. `Offset` indicates the offset address referring the next block in a PH. In `Ph_ID` the identifier of the PH is held. This is not actually needed, since this can be calculated from the address of the next block, but for excluding the overhead the information was included in a block. `Size` stores `Object Area`'s size of the block. It is possible to do garbage collection of the PH with the information in the block. By making the list of blocks in a PH, we manage allocation and deletion of objects in the PH. Each size of `Offset`, `Ph_ID`, and `Size` is set to 4-byte, thus the size of the whole information is 12 bytes and the size of a block needed for a persistent object whose size is $S$ is $S + 12$ bytes.

## 2.3 Several Techniques for Dereferencing Persistent Pointers

As mentioned later on, creating and traversing persistent objects were evaluated. Thus this section discusses implementations and characteristics of several pointer swizzling and nonswizzling techniques especially on the two operations.

### 2.3.1 Assumptions

To compare several techniques, the following conditions were assumed in the experimentation.

- The environment of a memory-mapped I/O architecture is used to implement all the swizzling and nonswizzling techniques.
- The size of a persistent pointer variable is 4-byte. This is the same size of a C++ pointer (see the next subsection for the reason).
- All the techniques are implemented to allow an application program to use several PHs at a time.
- Only one benchmark program is used by means of building dereferencing mechanism into INADA class library.

The following five techniques were implemented and evaluated within INADA.

- Pointer swizzling techniques
  - $\texttt{PAGE}_S$: Page-at-a-time swizzling
  - $\texttt{ONE}_S$: One-at-a-time swizzling
- Pointer nonswizzling techniques
  - $\texttt{SLS}_N$: Single level store
  - $\texttt{OFF}_N$: OID holds an offset
  - $\texttt{ORT}_N$: OID holds an entry number of the object reference table (ORT)

A technique called "object-at-a-time" swizzling [Moss92] was not included in the experiment because its performance might be between those of the $\texttt{PAGE}_S$ and the $\texttt{ONE}_S$. These five techniques can simulate almost all known possible techniques.

Implementation of $\texttt{PAGE}_S$ and $\texttt{ONE}_S$ required modification of WAKASHI, because, as the readers can see later on, the storage server must record locations of persistent pointers on PHs with a 'log table'. The log table is manipulated by both a storage server process and a client process where an INADA program runs. Therefore, the table is located in the memory (denoted as 'shared memory' in the following) shared by the server and client processes. The current WAKASHI does not have such a mechanism. Note that a module of WAKASHI referred to as 'storage manager' in the following figures worked in the WAKASHI server. The evaluation were done in one-site environment; hence the functionality of WAKASHI as a distributed server was not used.

### 2.3.2 An Issue of Persistent Pointer Size

In INADA programs, a persistent pointer consumes 4 bytes, that is, the size of an object is same whenever the object holds either persistent pointers or ordinary C++ pointers. Thus users can code methods applicable to objects including any kind of pointers. Let us consider the following example.

```
class Part_A {
               char* name;
               int   p_id = 0;
               friend int check_id(void*);
};
class Part_B {
    persistent char* name;
               int   p_id = 1;
               friend int check_id(void*);
};
int get_p_id(void* ptr) {
// This can be applied both Part_A and Part_B
    int id = (int)(*((char*)ptr + sizeof(char*)));
    return id;
}
```

The method `get_p_id()` includes `sizeof(type_name)` function, which returns the size of `type_name`. Because the value of "`sizeof(char*)`" is equal to that of "`sizeof(persistent char*)`" due to the restriction, the method can be applied to objects of both `Part_A` and `Part_B`.

This assumption is important in order to allow users to avoid writing similar but redundant codes. Systems treating persistent objects should obey this rule as far as 64-bit address

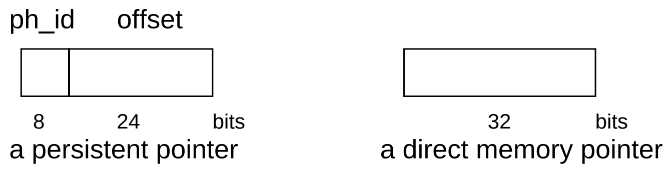

Figure 2.5: Pointer formats in $\texttt{PAGE}_S$

space cannot be widely used.

### 2.3.3 Page-at-a-time Swizzling : $\texttt{PAGE}_S$

In this scheme all persistent pointers in a page are translated into direct memory pointers at *page-fault time*. Therefore, programs can always see persistent pointers as virtual memory pointers after the target page is swapped in. [Wil90] describes several virtual memory techniques of pointer swizzling at page-fault time. An approach of this type is also used in ObjectStore [LLOW91] (its implementation details, however, are not currently available). Although the implementation in the experiment might be different from the others in detail, we believe that the basic idea is identical.

Figure 2.5 depicts the pointer formats used in the experiment. `ph_id` holds the identifier of the PH in which the object is located, and `offset` stores the offset address of the object from the top of the PH.

In $\texttt{PAGE}_S$, the storage manager not only keeps the correspondence between a PH and its file on secondary storage but must know the locations of all persistent pointers on the PH as well. To this end, a 'log file' (containing a 'log table') is used.

Figure 2.6 illustrates what happens when creating persistent objects. (1) If a reference to a persistent object (i.e., a persistent pointer) is created on a PH, the system records the location of that in the log file. (2) When the page is going to be swapped out, the storage manager looks for locations of all pointers on the page and (3) unswizzles them using the log file. Then, (4) the page is swapped out to secondary storage. Note that even if all the persistent pointers in the page have been swizzled, the page is not 'dirty' unless data on the page is changed at all. After the application program finishes, all references to the persistent

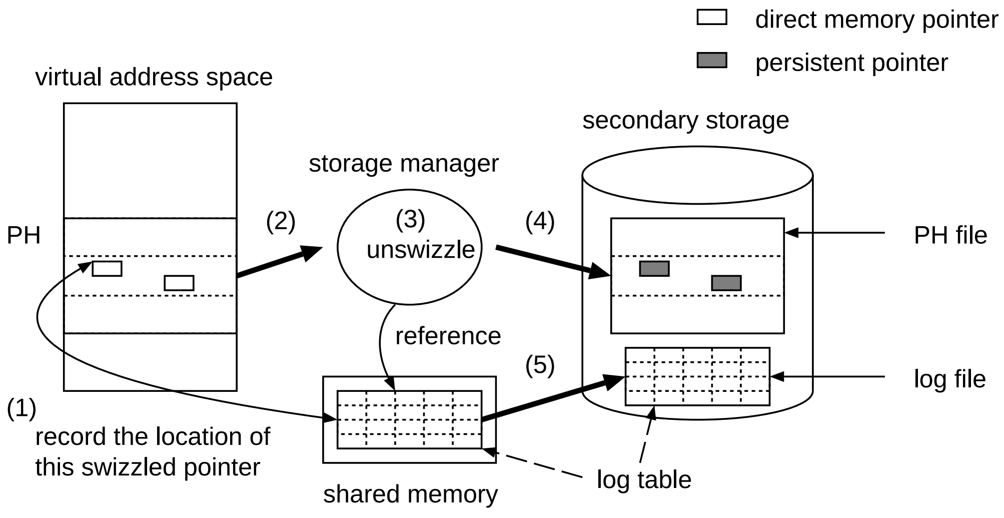

Figure 2.6: Creating objects in $\texttt{PAGE}_S$

objects in the PH are unswizzled and the PH is unmapped. Finally, (5) the log file is written into secondary storage. As shown in Figure 2.6, the log table is shared by the client and the storage manager; therefore, the table is implemented in a shared memory.

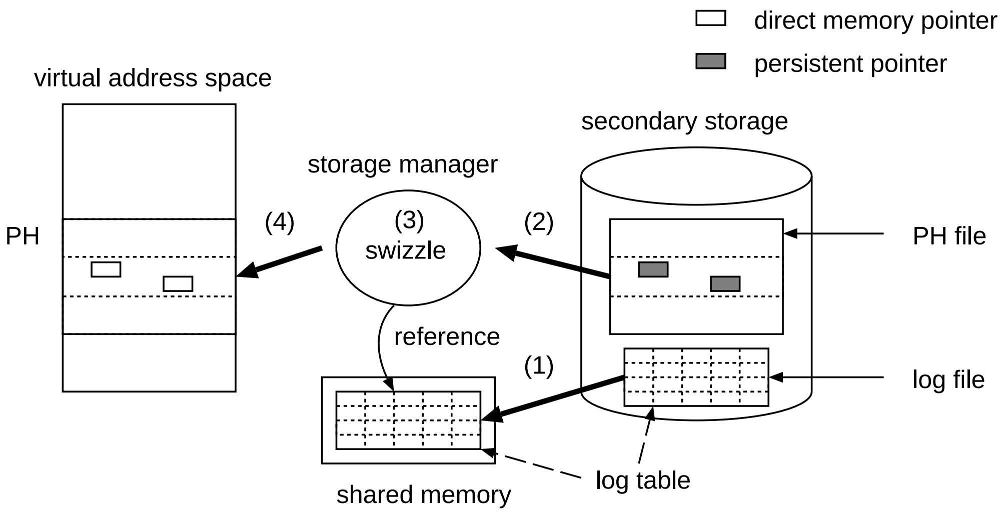

Figure 2.7: Traversing objects in $\texttt{PAGE}_S$

Figure 2.7 illustrates traversing persistent objects. When an application program accesses a page for the first time and a page fault occurs, all the persistent pointers contained in the page must be translated properly into direct memory pointers. For the translation, (1) the 'log file' is set into the shared memory before the program starts. When the accessed page is swapped into primary memory, (2) the storage manager looks for all the persistent pointers in the page with the 'log file' and (3) swizzles them. (4) After they are all swizzled, the page can be accessed by the application program.

One of the advantages of this technique is, obviously, that programs can access to persistent objects at memory speeds after swizzling.

A disadvantage, as described in [WD92], is that unnecessary swizzling and unswizzling overheads might arise. This is because swizzling and unswizzling are done at the granularity of page, and it is not likely to happen that a program accesses all the pointers located in a page.

### 2.3.4 One-at-a-time Swizzling : $\texttt{ONE}_S$

In this approach, persistent pointers are swizzled *one-at-a-time* when each of them is actually dereferenced. Therefore, it avoids the unnecessary swizzling and unswizzling overhead which may occur in $\texttt{PAGE}_S$. But, whenever a persistent pointer is referred, it must be checked whether the pointer has been already swizzled or not. Figure 2.8 shows the formats of pointer variables used for the experiment. If the most left bit value of a persistent pointer is 0 the pointer has been swizzled, and if the value is 1 the pointer has not been swizzled yet. One of the disadvantages is, therefore, that the number of PHs which an application program can use simultaneously is restricted to half of those in other approaches examined in the experiments.

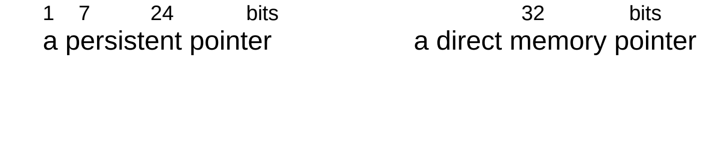

Figure 2.8: Pointer formats in $\texttt{ONE}_S$

Figure 2.9 shows what is done when persistent objects are created in a PH and no access

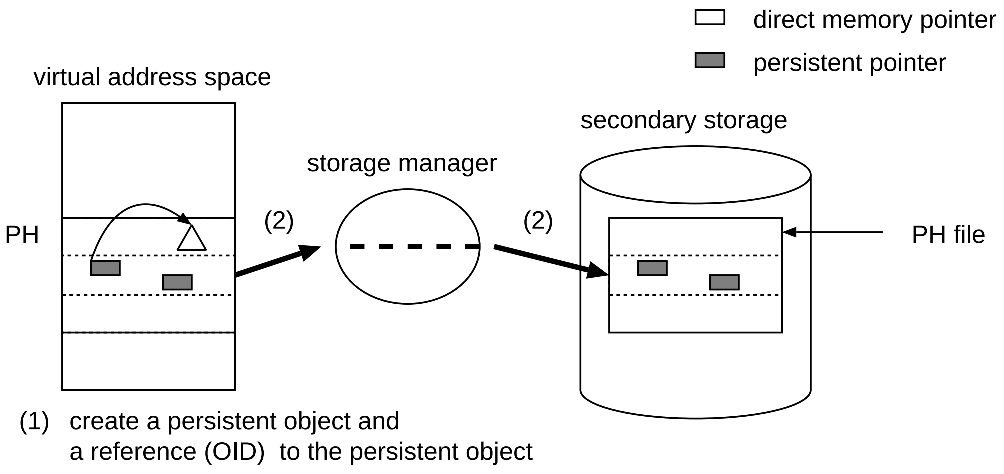

Figure 2.9: Creating objects in ONE$_S$

to the objects occurs. (1) When a persistent object is created, the virtual address of the object is converted to a persistent pointer value with setting the value of its tag to 1, and the value is assigned to a persistent pointer variable. When the dirty page in which the persistent pointer is located is going to be swapped out of the virtual memory, (2) the storage manager writes the page on secondary storage as it is. After creating objects, (2) such dirty pages in the PH are all written back to secondary storage as they are.

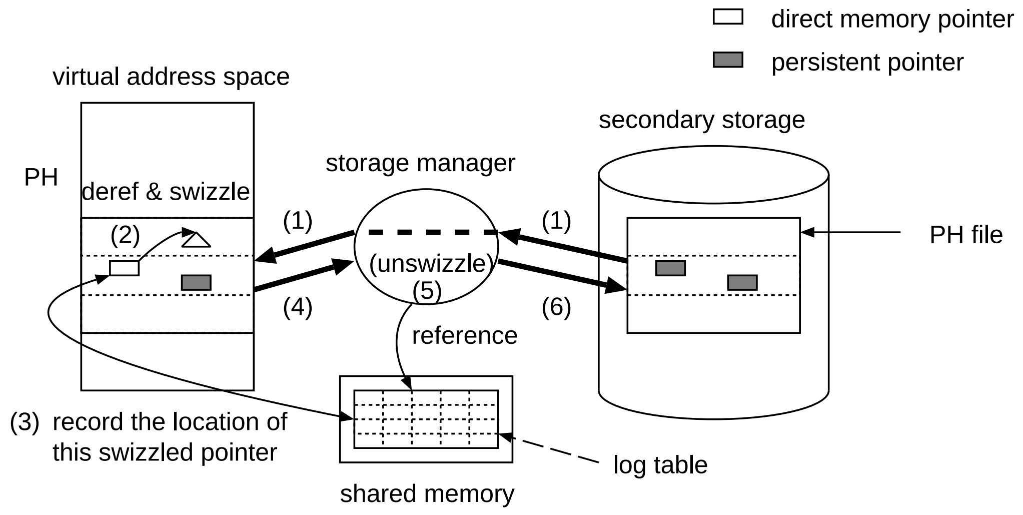

Figure 2.10: Traversing objects in ONE$_S$

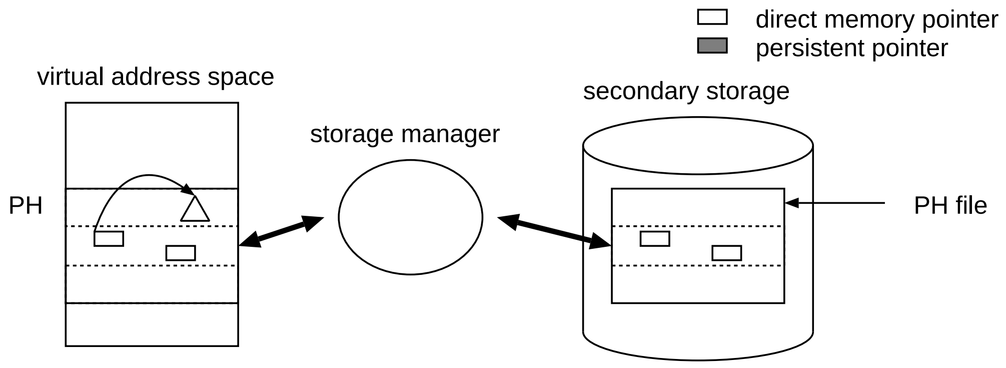

Figure 2.11: Non Swizzling $\mathtt{SLS}_N$

Figure 2.10 shows what happens when a program traverses persistent objects. (1) When the program accesses a page in a PH for the first time and the page will be swapped into primary memory, no persistent pointer in the page is swizzled and the page is swapped as it is. When a persistent pointer in the page is referred (2) it is checked whether the pointer has been swizzled or not. Naturally the pointer is not swizzled yet, (3) it is swizzled and the location of the reference is recorded in the log table. Note that the check is always performed whenever the pointer is referred. After execution of the program, (4) the storage manager looks for all swizzled pointers in the PH using the log table and (5) unswizzles them. Then, (6) all pages of the PH are written back into the disk. It should be noted that any page including at least one swizzled pointer has to be marked as 'dirty' and be written back on the disk with its unswizzled pointer, even if the application program has not modified any data in the page. While the log table used for $\mathtt{PAGE}_S$ should be persistent, the log table for $\mathtt{ONE}_S$ does not need to be.

### 2.3.5 Non Swizzling Technique (1) : $\mathtt{SLS}_N$

This technique maps a database file always on the same portion of the virtual address space, as shown in Figure 2.11. [SZ90] describes Cricket database storage system which provides the abstraction of a shared, transactional *single-level store* and argues for the effectiveness of the store.

With the single-level store, there is no need of pointer conversion. There is no distinction between persistent pointers and non-persistent pointers. As a result, this technique can eliminate the overhead which would be incurred in any other approach, and the cost of manipulating persistent objects through persistent pointers would be identical to that of through ordinary C++ pointers.

However, it has some problems which may arise when using multiple PHs. One problem is that every database files must always be mapped onto the same places in the virtual address space of each application. Next is that an application has to map all the PHs, which have been created, onto its virtual space when the application wants to create a new PH, because all PHs must be access-protected before creating a new PH to avoid overlapping of PHs' spaces. Another problem is that expanding a PH between two consecutive PHs is practically impossible.

Although `SLS`$_N$ is not so practical for 32-bit address space, it should be reconsidered in future when 64-bit address space becomes available and the time could not be so remote. We include `SLS`$_N$ in evaluation for only comparison.

### 2.3.6 Non Swizzling Technique (2) : `OFF`$_N$

The persistent pointer format used for this technique is shown in Figure 2.12.

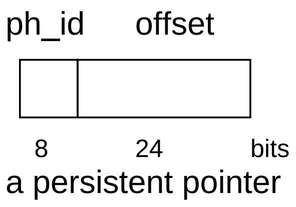

Figure 2.12: Persistent pointer format in `OFF`$_N$

This format consists of two fields. One is `ph_id` which indicates the identifier of the PH that contains the persistent object pointed to by the pointer. The other field is `offset` which holds the offset address of the object from the beginning of the PH. Whenever an application program accesses a persistent pointer, the virtual address is calculated by adding the `offset` in the pointer to the virtual address of the PH whose identifier is `ph_id`.

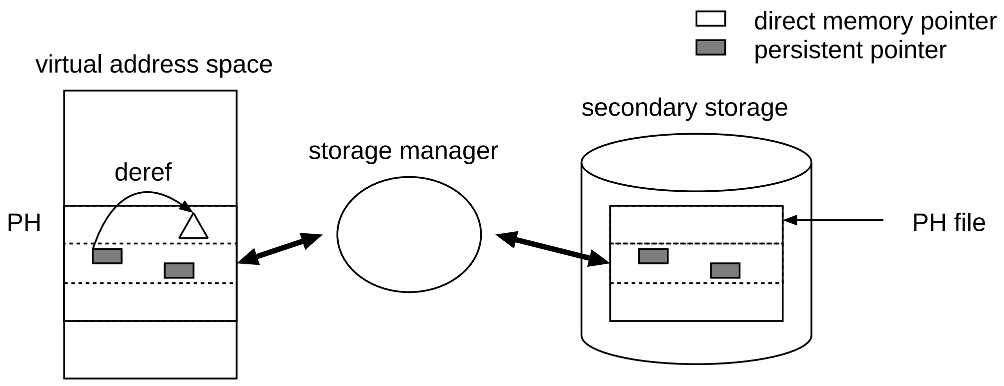

Figure 2.13: Non swizzling $\texttt{OFF}_N$

An advantage of this technique is that the cost for dereferencing seems relatively small, since only addition operation is involved.

The main disadvantage is that relocation of objects in a PH is difficult and costly, if not impossible ($\texttt{SLS}_N$ has the same difficulty).

### 2.3.7 Non Swizzling Technique (3) : $\texttt{ORT}_N$

The persistent pointer format used is shown in Figure 2.14. A persistent pointer used consists of two fields: `ph_id` and `ort_id`. `ph_id` represents the identifier of the PH in which the persistent object referred to by the pointer is allocated. `ort_id` indexes the object reference table (ORT) of the PH. The ORT is a large array and an entry of it contains the offset address of the object.

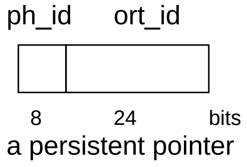

Figure 2.14: Persistent pointer format in $\texttt{ORT}_N$

In the technique, whenever a persistent pointer is used the entry in the ORT is looked for from the `ort_id`, then the virtual address can be calculated by adding the offset value stored in the entry to the top address of the PH. Thus, this type of pointer dereferencing, referred to by $\texttt{ORT}_N$ in this dissertation, does not swizzle any persistent pointer. The technique is adapted in the current system of INADA.

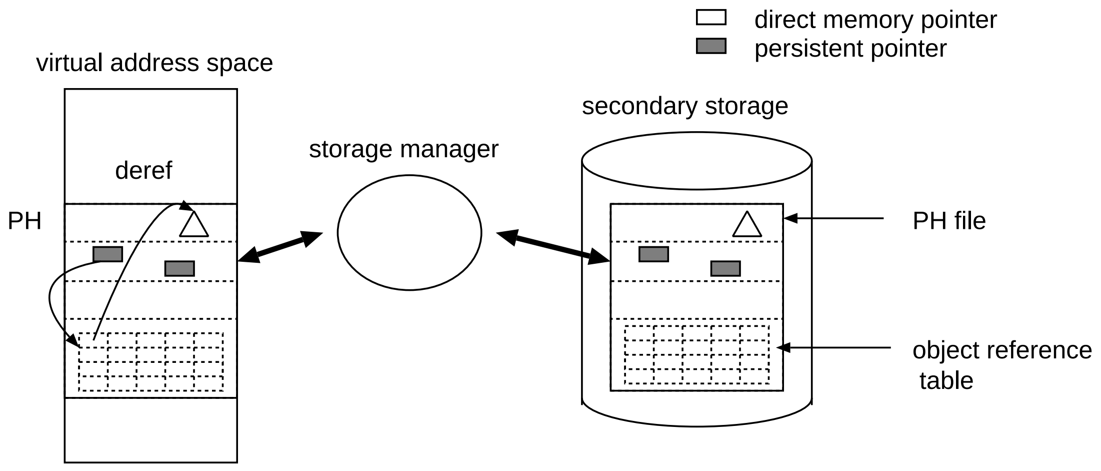

Figure 2.15: Non swizzling $\texttt{ORT}_N$

Figure 2.15 shows what is going on when dereferencing persistent pointers. Dereferencing persistent pointers costs more than $\texttt{OFF}_N$ because (1) more PH space is needed to store the ORT, and (2) table-look-up is required for getting actual addresses of objects. However, this technique has several advantages in terms of management of PHs. One is that $\texttt{ORT}_N$ allows easy relocation of objects in a PH space, so that it can support garbage collection very easily. This benefits such applications as repeatedly allocate and deallocate persistent objects. Also, $\texttt{ORT}_N$ could perform dereferencing persistent pointers more safely. For example, entries of the table could hold information to check whether the persistent pointer value is set correctly. Moreover, the INADA system can avoid the overhead caused by the calculation by doing that the references are processed through a volatile pointer whose value is assigned to by processing simple operation with the persistent pointer [TAM94]. Note that this optimization was never used to be fair in the experiments.

## 2.4 Experiments and Results

### 2.4.1 Environment Used

All experiments presented in this section were performed on an OMRON LUNA-88K workstation running under Uni-OS Mach Ver 1.34, which is actually Mach 2.5. The system had 4 MC88100 CPUs with 25.0 MHz clock and 32 megabytes main memory. The disk used to store both benchmark program and its data was a HITACHI DK515C (330 megabytes). The virtual memory swap area on this machine was located in a HITACHI DK312C disk, and was up to 100 megabytes in size. The page size was 8 Kbytes. GNU C++ compiler version 2.6.3 and libg++, which was the GNU C++ class library, version 2.6.2 were used.

### 2.4.2 Dereferencing Benchmark

To evaluate and compare several implementations of persistent pointers in the memory-mapped I/O environment, a simple benchmark program, called *Dereferencing Benchmark*, was made. Not a few benchmark programs such as OO1 [CS91], OO7 [CDN94], Hyper Model Benchmark [AB+90], and a complex object benchmark [DMFV90] have been proposed so far. They are much sophisticated that they can be used for measuring many aspects of object-oriented database systems. However, the aim of this experiment is to compare several strategies of dereferencing persistent pointers, and an ideal benchmark for that should be so simple that we can investigate the cost of dereferencing alone. In fact, to evaluate the cost the most important thing is that we could know exactly how many pointers are in a page and how many pointers of them are really used in a transaction. Hence we designed the Dereferencing Benchmark.

In the benchmark, a persistent list is used. Each node in the list consists of *NumOfPointer* persistent pointers and *NumOfInteger* integers as dummy data items. One of the pointers is linked to a neighboring node, and the rest of them are linked to the node holding the pointers (see Figure 2.16). We set *NumOfPointer* + *NumOfInteger* = 31 in the experiment for that the object size needed could be constant even if the numbers were changed.

The form of list was adopted so that traversing a linear list could cause page faults very easily, and $NumOfPointer - 1$ persistent pointers in a node could represent unused

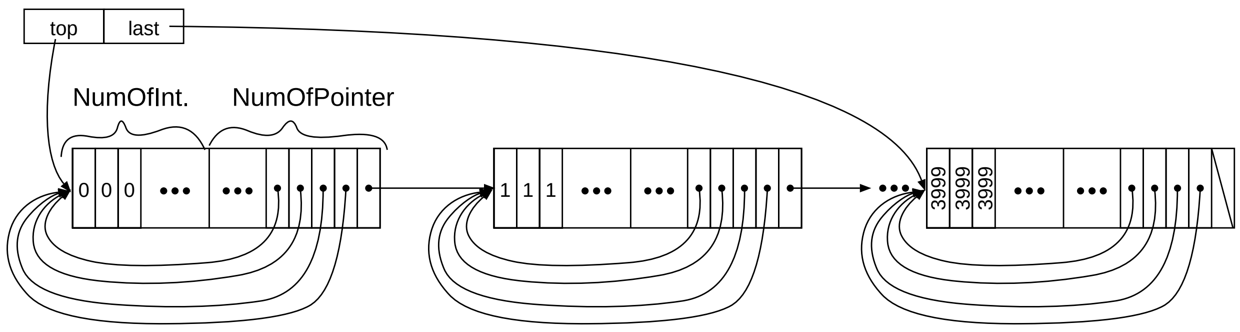

Figure 2.16: Dereferencing Benchmark

persistent pointers for a transaction.

The benchmark program consists of two work sessions. One is the create session where the persistent list is created, and the other is the traversal session where the list is traversed from node to node. No method invocation without returning the neighboring node is involved when visiting nodes. Each session includes two or three sub-sessions as follows:

i) create session

  (1) preparation

  (2) creation of objects

ii) traversal session

  (1) preparation

  (2) cold traversal

  (3) hot traversal (20 times traversals)

The preparation sub-sessions in both the create and traversal session include the cost of initializing PHs and a log table, if necessary.

There are two types of traversal; *cold* and *hot*. The cold traversal means the first traversal of the list, which is not cached yet in primary memory. The hot traversal represents the total of 20 times traversals of the same list after the cold traversal. In this case, the whole, or a part of, database is cached.

Three different pairs of ($NumOfPointer$, $NumOfInteger$) were evaluated; (10, 21), (20, 11), and (30, 1). Therefore the memory block for a node consumes $31 \times 4 + 12$ bytes (see Section 2.2). The evaluated database had 4000 nodes, thus about 80 pages were needed to store the node objects. The whole database could be memory resident in the environment used.

For the sake of brevity, only one PH was used in each experiment, although all the implemented techniques allowed to manipulate several PHs as known from the assumption mentioned before. In addition, none of such utilities as transaction, concurrency control, and recovery provided by the storage server was included in the evaluation. Therefore, the results showed the overhead for dereferencing persistent pointers alone.

### 2.4.3 Benchmark Results and Discussion

The time reported in this subsection is the average elapsed time of ten runs of each session.

#### Creation

Table 2.1 presents the create session times of databases in which the pairs of ($NumOfPointer$, $NumOfInteger$) in a node are (10, 21), (20, 11), and (30, 1). As the number of pointers increases, the costs of swizzling techniques increase more than those of nonswizzling ones.

In creating objects, several operations are involved.

i) Allocating persistent objects (i.e., nodes) in the PH.

ii) (1) Assigning pointer values to persistent pointer variables in the persistent objects in the cases of $\mathtt{SLS}_N$ and $\mathtt{PAGE}_S$, or

(2) Assigning proper OID values calculated from the pointer values to persistent pointer variables in the cases of $\mathtt{ONE}_S$, $\mathtt{OFF}_N$ and $\mathtt{ORT}_N$.

iii) Dereferencing persistent pointers in order to link nodes, and so on.

$\mathtt{SLS}_N$ has the best performance obviously. The costs of $\mathtt{OFF}_N$ and $\mathtt{ONE}_S$ are next, since they use the similar ways for pointer conversion. Both $\mathtt{PAGE}_S$ and $\mathtt{ORT}_N$ incur larger overhead. In $\mathtt{PAGE}_S$ the cost of getting entries in the log table in order to keep tracks of persistent pointers

Table 2.1: Creation (in milli-seconds).

$(NumOfPointer,NumOfInt.)$=(10,21)

| *Technique* | *Prep.* | *Create* | *Total* |
|---|---|---|---|
| $\texttt{PAGE}_S$ | 395 | 1020 | 1415 |
| $\texttt{ONE}_S$ | 267 | 838 | 1105 |
| $\texttt{SLS}_N$ | 58 | 802 | 860 |
| $\texttt{OFF}_N$ | 47 | 835 | 882 |
| $\texttt{ORT}_N$ | 45 | 1555 | 1600 |

$(NumOfPointer,NumOfInt.)$=(20,11)

| *Technique* | *Prep.* | *Create* | *Total* |
|---|---|---|---|
| $\texttt{PAGE}_S$ | 397 | 1248 | 1645 |
| $\texttt{ONE}_S$ | 263 | 852 | 1115 |
| $\texttt{SLS}_N$ | 54 | 823 | 877 |
| $\texttt{OFF}_N$ | 47 | 844 | 891 |
| $\texttt{ORT}_N$ | 40 | 1559 | 1599 |

$(NumOfPointer,NumOfInt.)$=(30,1)

| *Technique* | *Prep.* | *Create* | *Total* |
|---|---|---|---|
| $\texttt{PAGE}_S$ | 395 | 1444 | 1839 |
| $\texttt{ONE}_S$ | 266 | 872 | 1138 |
| $\texttt{SLS}_N$ | 53 | 840 | 893 |
| $\texttt{OFF}_N$ | 49 | 862 | 911 |
| $\texttt{ORT}_N$ | 42 | 1557 | 1599 |

for (1) of ii) is the reason, and in $\texttt{ORT}_N$ the costs of both getting entries in the object reference table to construct persistent pointer formats for (2) of ii) and dereferencing pointers used to link nodes for iii) are the reasons. While they are similar in getting entries, there are many different aspects. In the creation, $\texttt{PAGE}_S$ creates $4000 \times NumOfPointers + 2$ entries in the log table for (1) of ii). On the other hand, $\texttt{ORT}_N$ creates $4000 + 1$ entries in the ORT in i), performs calculation of the persistent pointer value from the virtual address 4000 times in (2) of ii), and dereferences pointers 4000 times in iii). It is interesting to note that the costs of all techniques except for $\texttt{ORT}_N$ increase as the number of pointers created increases.

**Traversal**

Table 2.2 reports the times of the traversal sessions of databases in which the pairs of ($NumOfPointer$, $NumOfInteger$) in a node are (10, 21), (20, 11), and (30, 1). Each preparation times are roughly same as in the creation session of all pairs. Cold(1) stands for the time of the cold traversal, and Hot(20) does the total time of the 20 traversals after the cold traversal.

Since all pointers are swizzled before the hot traversals in $\texttt{PAGE}_S$, it incurs essentially no overhead, that could be revealed by comparing with $\texttt{SLS}_N$.

Both $\texttt{OFF}_N$ and $\texttt{ORT}_N$ techniques involve pointer value conversion whenever persistent objects are accessed. Especially in $\texttt{ORT}_N$, dereferencing a persistent pointer entails more interaction with the *object reference table* manager to obtain address of the referenced object.

Obviously, the more the number of objects increases, the worse the performance of $\texttt{PAGE}_S$ must become. In the experiments the number of nodes was 4000 so that all data were memory resident. If all data could not be on primary memory, the performance of the swizzling techniques would become worse.

## 2.5 Related Work and Summary

This chapter studied the way of achieving persistence of objects, and examined several pointer swizzling and nonswizzling techniques within a memory-mapped I/O architecture. Implementations of such different techniques in the INADA framework were easy owing to INADA's flexibility. However, some slight modification of WAKASHI was required for the implementations.

[Wil90] describes a pointer swizzling scheme based on a virtual memory technique. In the scheme, persistent pointers are swizzled to normal pointers at page fault time. A similar approach is used in ObjectStore [LLOW91].

Table 2.2: Traversal (in milli-seconds).

$(NumOfPointer, NumOfInt.) = (10,21)$

| *Technique* | *Prep.* | *Cold(1)* | *Hot(20)* |
|---|---|---|---|
| PAGE$_S$ | 398 | 713 | 81 |
| ONE$_S$ | 348 | 708 | 187 |
| SLS$_N$ | 41 | 590 | 80 |
| OFF$_N$ | 36 | 637 | 167 |
| ORT$_N$ | 37 | 1288 | 230 |

$(NumOfPointer, NumOfInt.) = (20,11)$

| *Technique* | *Prep.* | *Cold(1)* | *Hot(20)* |
|---|---|---|---|
| PAGE$_S$ | 401 | 777 | 81 |
| ONE$_S$ | 345 | 707 | 188 |
| SLS$_N$ | 38 | 590 | 80 |
| OFF$_N$ | 36 | 640 | 162 |
| ORT$_N$ | 41 | 1270 | 232 |

$(NumOfPointer, NumOfInt.) = (30,1)$

| *Technique* | *Prep.* | *Cold(1)* | *Hot(20)* |
|---|---|---|---|
| PAGE$_S$ | 400 | 836 | 79 |
| ONE$_S$ | 348 | 719 | 183 |
| SLS$_N$ | 41 | 583 | 80 |
| OFF$_N$ | 38 | 644 | 166 |
| ORT$_N$ | 39 | 1267 | 226 |

[Moss92] presents a detailed analysis on performance of some swizzling and nonswizzling schemes. The author takes an object-at-a-time approach to swizzling under which all pointers in an unswizzled object are swizzled immediately when the object is first accessed.

[WD92] describes the relative performance of several versions of EPVM (E Persistent Virtual Machine) 2.0 and compares it with several alternative software architectures including

ObjectStore V1.2. EPVM 2.0 supports a pointer-at-a-time swizzling scheme (referred as the one-at-a-time scheme in this dissertation) and uses software checks to distinguish swizzled and unswizzled pointers.

[KK93] classifies and evaluates different pointer swizzling approaches. The paper, however, uses an object manager called GOM, which is based on the EXODUS storage manager, or on a buffer pool oriented architecture.

The pointer swizzling techniques described in this chapter come from these previous work. Comparison among several techniques has been already done in [WD92] and [Moss92]. The key difference of our work from theirs is that the comparison was done in a same framework although the benchmark program was fairly simple. In our work, all the swizzling and nonswizzling techniques compared were built into INADA in the memory-mapped I/O architecture. In [WD92], different systems with different architectures were compared. In [Moss92], the software used was not based on memory-mapped I/O architecture, but on also a traditional buffer pool oriented architecture.

Our experimental results show that a nonswizzling approach outperforms swizzling ones in cold case. Also, the nonswizzling approaches are not much behind the swizzling ones in hot case. The pointer swizzling approach is believed to give much better performance when database is loaded in primary memory. Our experiments show that it is not the case when the architecture is based on the memory-mapped I/O. In the environment a swizzling approach like $\texttt{PAGE}_S$ has the big overhead for unnecessary swizzling and unswizzling, since the size of data moving from secondary storage to primary memory and vice versa is not the size of buffer pool but the size of page in the environment.

Taking the flexibility provided by $\texttt{ORT}_N$ into account, the $\texttt{ORT}_N$ technique can be one of good choices for handling persistent pointers. Table 2.3 indicates an overall evaluation of the five techniques. The numerical values stand for the ratio to the performance of $\texttt{SLS}_N$ in the case that $NumOfPointer = 20$.

Table 2.3: Overall evaluation

| | performance | | | | functionality | |
|---|---|---|---|---|---|---|
| | *prep.* | *create* | *cold* | *hot* | (i) | (ii) |
| $\texttt{PAGE}_S$ | $\times\times$ | 1.5 | 1.3 | 1.0 | $\times$ | $\surd$ |
| $\texttt{ONE}_S$ | $\times\times$ | 1.0 | 1.2 | 2.4 | $\times$ | $\surd$ |
| $\texttt{SLS}_N$ | $\surd$ | 1 | 1 | 1 | $\times$ | $\times$ |
| $\texttt{OFF}_N$ | $\surd$ | 1.0 | 1.1 | 2.0 | $\times$ | $\surd$ |
| $\texttt{ORT}_N$ | $\surd$ | 1.9 | 2.2 | 2.9 | $\surd$ | $\surd\surd$ |

(i) Garbage collection

(ii) Flexibility for managing of PHs

| Symbols: | | |
|---|---|---|
| | $\surd\surd$ | Very Well |
| | $\surd$ | Well |
| | $\times$ | Poorly |
| | $\times\times$ | Very Bad |

# Chapter 3

# Multiple Type Objects

In general, it takes a lot of time to decide the forms of classes, or types, in database design. This is because the forms of objects stored in a database can hardly be changed. If the objects can get and lose types dynamically, this can be solved. This chapter describes design of multiple type objects in INADA. Any persistent objects in INADA may get any types at any time the types are needed, and may lose any unnecessary types dynamically. INADA is an enhanced C++ language; it borrows the object model of C++ and extends it to provide facilities needed for processing on a large amount of persistent objects. Also, implementation of multiple type objects is shown in this chapter. This implementation exploits the type system of C++ just as it is.

## 3.1 Introduction

For the ability to model real world entities, many database researchers have investigated object-oriented databases as designing and developing C++-based database or persistent programming languages, e.g., E [RC89], O++ [AG89], $O_2$ [Deu+90], ONTOS [Ont94], and ObjectStore [LLOW91]. This is mainly because C++ [ES91] is a general purpose programming language based on C: in addition to the facilities provided in C, C++ provides classes, inline functions, operator overloading, function name overloading, constant types, and templates. Besides, C libraries can be used from a C++ program, and most tools that support programming in C can be used with C++.

However, none of them has any abilities to model real world entities' facets which change dynamically as time passes. Things in this world have several roles and facets generally, and use them properly suitable for circumstances in which the things are. For example, a person who works at a laboratory can enter into a university as a graduate student without quitting his job. While in the university he is a student and has his student number and grades, in the laboratory he has his employee number as an employee. Note that no one can make the accurate forecast and, therefore, design all of such roles of the person when he comes into the world. Since a person has generally a long life span, an object modeling the person in programming languages should be persistent. Therefore, a persistent programming language should have some capabilities to model and expressing such facets.

An object-oriented persistent programming language called INADA [AA93, TAM94, ATBM95] is designed for writing data-intensive applications, and it is now under development. INADA is an extended C++; it borrows the object model of C++ and extends it to provide facilities for manipulating a large amount of persistent objects. In INADA, any persistent object can get and lose any types, or classes in C++ terms. Such objects are called *multiple type objects*. Multiple type objects can be implemented with no modification of the type system of C++ except for the runtime check whether a persistent object has the type. In other words, the facility provided by the type system of C++ can be available in INADA. This chapter describes the design of multiple type objects and shows that the concept can be implemented in a very simple way.

## 3.2 Multiple Type Concept

This section clarifies the necessity of multiple type objects and describes the design policies kept when introducing the concept into INADA.

### 3.2.1 Why Needed?

A persistent programming language allows users to manipulate persistent objects just like volatile objects. Once a persistent object is created and stored on secondary storage, the object may be used by the program which created it or by other programs, i.e., the persistent object may be shared among several programs. In general, declared and manipulated objects in a program have different forms, or information, from those in another program. Let us consider a case of modeling persons. When a person enters a university, the person *becomes* a student. In the university one has one's name, age, student id number, office room number, and office phone number. At the same time one can work outside the university as a part-time employee. In the environment one may have one's name, age, department name one belongs to, office phone number, and an amount of one's pay (see Figure 3.1). Note that although the name and age in both environments must be the same, the office phone numbers are different. In addition, the person is naturally identical.

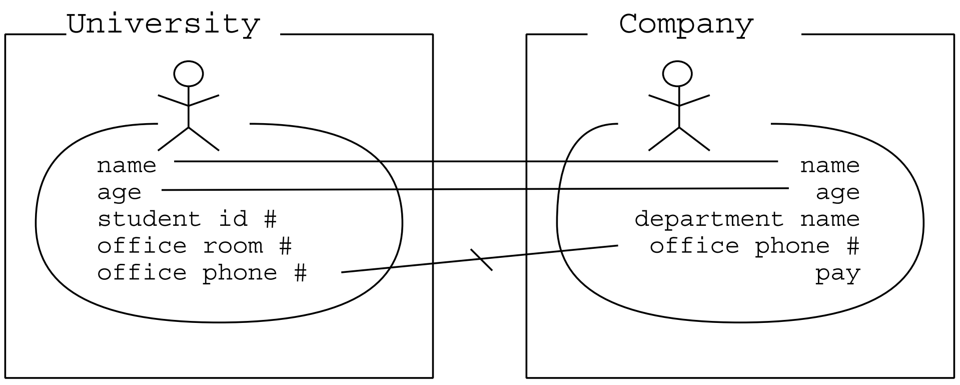

Figure 3.1: An example of a person

One solution is that a database designer designs structures of objects, which correspond *types* in terms of programming languages, useful for all programs that will manipulate the objects before creating a persistent object of the types. In other words, a persistent object includes all information and a user projects it to certain form or type suitable for her/his own application when accessing the object. For the values of some attributes which cannot be decided or should not be set, users may use so-called null values or flags (see Figure 3.2). For instance, consider the case that all persons are supposed to be modeled as persistent objects of '`University_Company`' and you must model a person who is a student but not an employee at the time. The values of department name, office phone number and pay attributes as an employee must be set to null.

The solution seems to work apart from the fact that the database designer must be well aware of all programs. However, another problem arises in investigating the characteristics

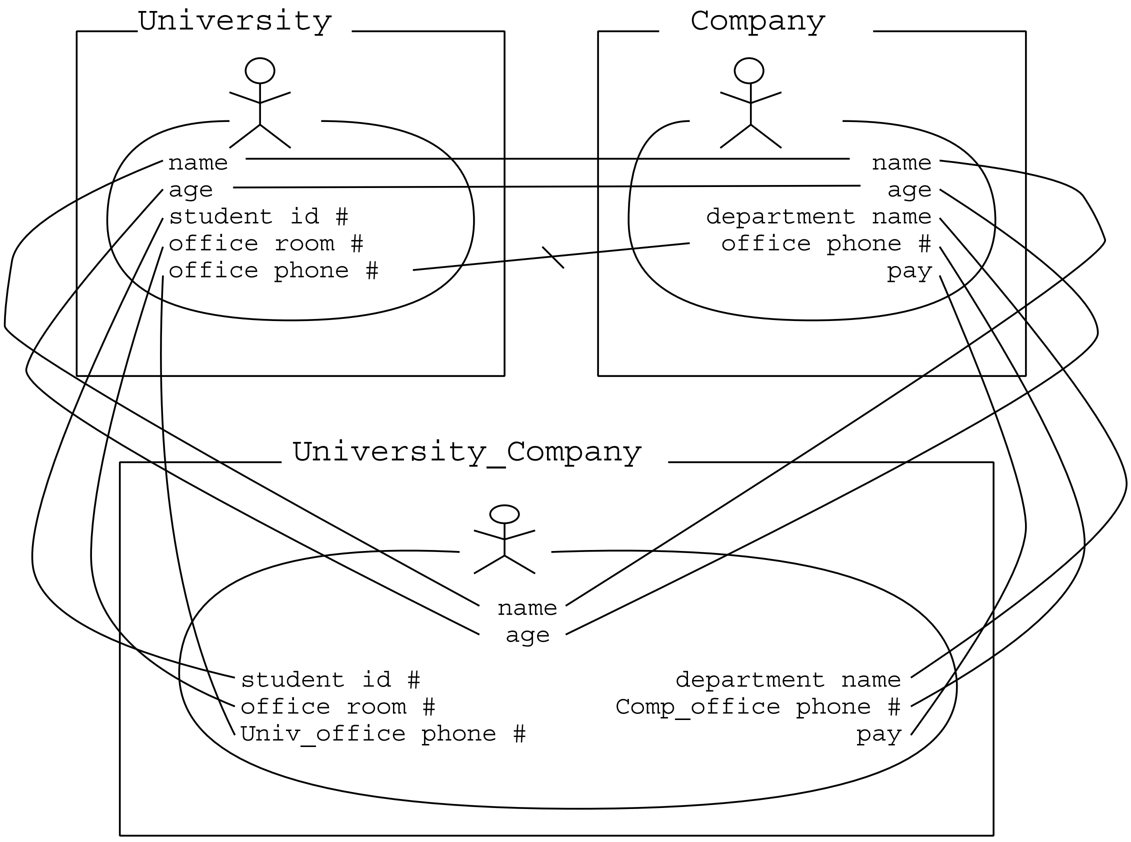

Figure 3.2: One solution for expressing many roles

of real world entities to be modeled.

Let us consider the person example again. A person generally has a bright future. That is to say no one can make the accurate forecast of the person's future. In the case where additional attributes are restricted to be used by a few objects in a large number of persistent objects, the solution using null values or flags wastes a large amount of disk space. In conclusion, the design of the structure for persistent objects should not be static but *augmentable*. Therefore, some mechanisms are required to add characteristics to objects for implementing incremental database designs and software modeling.

This is basically different from multiple inheritance. One can use the multiple inheritance to model objects which have many types. The multiple inheritance, however, has problems, e.g., a derived class must have all members of super classes. Furthermore, an object once created cannot change its type throughout its life span. As a result, the cases where a person *becomes* a student and *resigns* his job cannot be modeled with the multiple inheritance.

### 3.2.2 Design Policies

On the basis of things described above, the following principles were kept when introducing *multiple type objects* into INADA.

- Any persistent object can get and lose any types.
  Any persistent object gets and loses any types dynamically. There might exist persistent objects without any type in INADA.

- A persistent object has an OID (Object IDentifier).
  One persistent object has only one OID, and not have several OIDs for maintain multiple types of the object. The OID is used as an identifier and a reference.

- The type system of C++ should not be modified.
  We do not intend to modify the type system of C++ for introducing the concept; the type system can be employed just as it is and only the check whether a persistent object has certain type is added.

- C++ codes are available also in INADA.
  INADA allows the use of C++ codes already existed without any modification.

### 3.2.3 Syntax and Semantics

Manipulation of volatile objects is definitely same as in C++. A persistent object can be manipulated through a persistent pointer (whose value is actually an OID for a persistent object in INADA). A persistent pointer variable is declared just as a pointer variable in C++ with the prefix keyword '`persistent`'. For example, if you want to declare a persistent pointer variable '`person`' which will refer a persistent object whose class (or type) is '`Person`' (the way of class definitions is also same as in C++, and here assume the class has been defined properly elsewhere), write

```
persistent Person * person;
```

When creating a persistent object, you use the '`new`' operator with an argument to indicate where you want to allocate the new object. You can put a pointer variable which refers to a set object or a persistent heap object as the argument. INADA introduces set objects, which are instance objects of classes designed for collections of objects. A persistent heap (PH) is a portion of the virtual address space and mapped onto files on secondary storage under a memory mapped I/O environment [ATBM95], and used as a heap in C and C++ for manipulating persistent objects dynamically. Since a PH is a collection of objects, the PH is a set object. '`new(set)`', where '`set`' is a pointer to a set object, creates an object in the set object as an element of the set object and returns its OID (see [AA93] for more details). The following statement creates a '`Person`' object in a persistent heap stood for by '`pho`', and assigns its OID to the variable '`person`'.

```
person = new(pho)Person("Ari",27,....);
```

As far as creating persistent objects, the difference between INADA and C++ is that INADA has the keyword '`persistent`' for declarations of persistent pointer variables and '`new`' operator takes an argument whenever creating persistent objects. The type system used is that of C++ itself. For example,

```
person = new(pho)Part(...);
```

is detected as an error at compiling time because it attempts to create a '`Part`' object and assign its OID to '`person`', which has been already declared as a variable pointing a '`persistent Person`' object. Of course,

```
person -> Name();
```

is understood as a correct statement if '`Person`' has the method '`Name()`', and

```
person -> PartID();
```

is detected as an error, too, if '`Person`' does not have '`PartID()`', at compiling time by the type system of C++.

For the multiple type concept, '`as`' and '`transforms`' are introduced as programming constructs into INADA. The '`as`' construct is used to let the OID of a persistent object be a specific type. Its syntax is:

`OID as` TYPE

where '`OID`' means a persistent pointer variable, or '`this`' in class definition. The '`transforms`' is used as a statement of the following form:

```
new(set)Type(arguments) transforms OID;
```

which is exploited in order to add '`Type`' to the object referred to by the '`OID`' and to store data into the '`set`'. Let us explain it using the example shown in Figure 3.3. Figure 3.4 illustrates correspondence between attributes of '`CLASSI`' and '`CLASSJ`' in similar manner to Figure 3.1.

```
class CLASSI{
   int a;
   char b[10];
   int c;
public:
   CLASSI(int, char*, int);//constructor
   int A(){return a;}
   char* B(){return b;}
   int C(){return c;}
};
class CLASSJ{
   int a;
   char d[15];
public:
   CLASSJ(int, char*); //constructor
   int A(){return a;}
   char* D(){return d;}
   int JA(){return this as CLASSI -> A();}
   //A() in CLASSI is accessed with different name
   char* B(){return this as CLASSI -> B();}
   //B() in CLASSI is accessed with same name
   //C() in CLASSI isn't accessed through CLASSJ
};
```

Figure 3.3: Definitions of CLASSI and CLASSJ

In the code, the points should be noted are as follows.

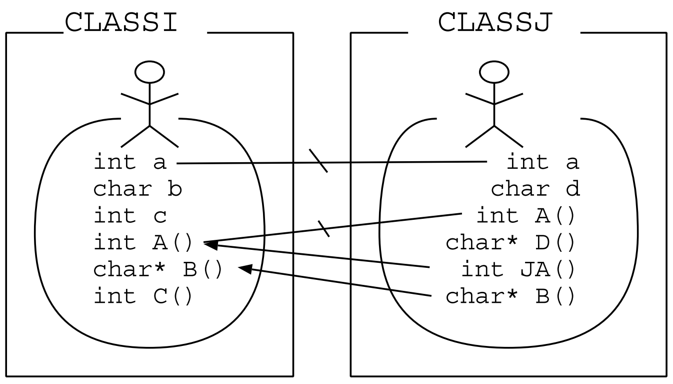

Figure 3.4: Dependency between CLASSI and CLASSJ

- The definition of '`CLASSI`' is identical to that in C++, i.e., the definition is available in C++ just as it is.
- '`CLASSJ`' can have unique attributes including data and methods.
- Attributes in '`CLASSI`' can be accessed through attributes in '`CLASSJ`'. The attributes in '`CLASSJ`' need not to have the same names of those in '`CLASSI`'.
- '`CLASSJ`' need not to have all attributes in '`CLASSI`'. This is one of the different points from the inheritance mechanism.

A persistent object of '`CLASSI`' in Figure 3.3 can be created as follows:

```
persistent CLASSI * pi = new(pho)CLASSI(1,"fuk",4);
```

To add the type '`CLASSJ`' to the object, write

```
persistent CLASSJ * pj;
pj=new(pho)CLASSJ(1000,"kyu") transforms pi;
```

or

```
pi as CLASSJ=new(pho)CLASSJ(1000,"kyu");
```

The difference between the two styles is that the variable '`pj`' is made in the former style, while no variable for the addition is made in the latter. Note that the values '`pj`' made in the former and 'pi' are the same in INADA, since both are pointing to an identical persistent object.

```
// correct statements
pi -> A();  // 1                  ----(1)
pj -> D();  // "kyu"              ----(2)
pj -> A();  // 1000               ----(3)
pi as CLASSJ -> D(); // "kyu"     ----(4)

// wrong statements
// (detected at compiling time)
pi -> D();                        ----(5)
pj as CLASSI -> D();              ----(6)
```

Statement (1) sends the method '`A()`' to the object which is pointed to by '`pi`' that is declared as the type '`persistent CLASSI *`'. Statement (5) can be detected as an error at compiling time as same as in C++ because (5) is going to send '`D()`' to the object whose type can be decided as '`CLASSI`' by the declaration, '`persistent CLASSI * pi`'. As shown in Figure 3.3, the two '`A()`'s in '`CLASSI`' and in '`CLASSJ`' are different, and (1) and (3) may return different results. In the example, (1) returns 1 and (3) returns 1000. Statement (4) is a correct code in INADA. It sends '`D()`' to the object pointed to by '`pi`' as a '`CLASSJ`' object. However, statement (6) is a wrong statement, since '`CLASSI`' does not have the method '`D()`'. Note that the statements (5) and (6) can be decided as wrong statements when a user compiles programs including them, and the detection is done by the type system of C++.

When a persistent object is accessed through types which the object does not have, for example,

```
persistent CLASSI * pi = new(pho)CLASSI(1,"fuk",4);  ----(1)
pi as CLASSJ -> D();                                 ----(2)
```

the evaluation is failed at runtime of the program just because the object pointed to by '`pi`' does not have such a type and the absence of the type cannot be detected until the program runs.

In the example described above, a persistent object of '`CLASSJ`' is made after the creation of a '`CLASSI`' object. However, in general, the order of object creation does not have any relation of dependency between classes. For example, with using the two classes, users can create a '`CLASSJ`' object and then add '`CLASSI`' to the object:

```
persistent CLASSJ * pj  = new(pho)CLASSJ(1000,"kyu");               ----(1)
persistent CLASSI * pi  = new(pho)CLASSI(1,"fuk",4) transforms pj;  ----(2)
```

In this case, however, runtime failures may be occurred if there are some statement(s) using the object created in statement (1) before the evaluation of statement (2). One is that, for example, '`pj -> B();`' is a correct statement in INADA, but the method evaluation is failed because the object has no information defined in '`CLASSI`' at the moment. This is just like as what happened when heap objects are accessed before they are initialized in C++. The other is the same one mentioned before.

In the example using '`CLASSI`' and '`CLASSJ`', '`CLASSJ`' depends on '`CLASSI`'. Multiple type objects may have types which do not have any dependency among each other. For example, for adding type '`CLASSK`' shown below to a persistent object which has '`CLASSI`' and pointed to by '`pi`',

```
class CLASSK{
  int a;
public:
  CLASSK(int);
  int A();
};
```

a programmer can write

```
persistent CLASSK * pk = new(pho)CLASSK(4) transforms pi;
```

Besides, there may be mutual dependencies among classes in INADA. For instance, '`CLASSI`' in Figure 3.3 can have a method to access attributes in '`CLASSJ`' as follows:

```
class CLASSI{
   // data members are the same in Figure 3.3
 public:
   CLASSI(int, char*, int); // constructor
   int A(){ return a; }
   char* B(){ return b; }
   int C(){ return c; }
   char* D(){return this as CLASSJ -> D();}
};
```

Consequently, programmers can design classes independently if the classes have no dependency among them and also can integrate all persistent objects already stored in secondary storage independently into one schema using the multiple type concept.

Inheritance mechanism provided by C++ is also available for the concept. For example,

```
class CLASSL : CLASSJ{
  int aa;
public:
  CLASSL(int ja, char* jd, int la): CLASSJ(ja,jd){aa=la;}
  int AA(){return aa;}
};
```

is a completely valid class definition in INADA.

To delete persistent objects, all users have to do is to use '`delete`' operator as they do when deleting heap objects in C++. For instance,

```
delete pi;
```

deletes the persistent object pointed to by '`pi`'. If the object has many types, all information for the types are deleted. Similarly, the '`delete`' is used when deleting certain type from persistent objects:

```
delete CLASSI of pi;
```

removes the type '`CLASSI`' from the object without deleting the object itself. If the object pointed to by '`pi`' does not have such type, a runtime failure will occur because of the same reason described before.

In addition, persistent objects with no type can exist in INADA. Thus users can express the existence of objects whose characteristics are not so clear at the point with the no type objects.

## 3.3 Implementation Details

This section gives implementation details of multiple type objects in persistent programming languages, in particular, matters related to the multiple type concept and shows the key idea for implementation of multiple type objects of a prototype system of INADA. INADA itself consists of a translator, a class library and runtime routines. The translator translates INADA programs into C++ codes, which in turn are compiled by a C++ compiler.

### 3.3.1 Persistent Heaps and Pointers

In INADA, persistent objects are implemented with "*persistent heaps*". The term "heap" is employed because the space looks from users just like the heap in C and C++ programming languages where data are dynamically allocated and deleted by the user's programs. A persistent heap (PH) is a part of virtual address space of processes and is mapped to a file on secondary storage of local or remote sites. This is implemented with WAKASHI [BM94], a distributed paged-object server, which is based on the memory mapped file I/O mechanism supported in operating systems such as SunOS and Mach. A persistent heap basically consists of three parts depicted in Figure 3.5.

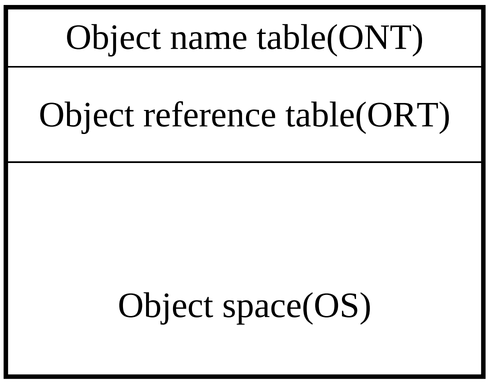

Figure 3.5: A physical segment layout of a PH

Object name table (ONT) binds names and persistent objects allocated in it. The names must be unique in a PH. All objects in it need not to be named. Object reference table (ORT) is a hash table[1] which is described in detail later on. The remaining space of a PH (OS: Object Space) is used for allocating and storing persistent objects. ORT and OS can be extended in the current prototype system of INADA.

As described before, persistent objects are manipulated through their persistent pointers. There have been many discussions concerning pointer swizzling and several methods for that were proposed [Moss92, Wil90, WD92] and many object-oriented database programming languages support some kinds of methods for swizzling [LLOW91, AB87]. However, a non-

[1]The hash mechanism in INADA is not mentioned in this chapter because this is not very significant to the implementation of multiple type objects

swizzling approach for dereferencing persistent pointers is adopted instead of a swizzling technique in INADA. This design decision mostly depends on the observations:

i) The performance of runtime pointer conversion depends on the scheme for storage management, the structure of persistent pointer, and the conversion mechanism.

ii) The runtime conversion based on the memory mapped file I/O approach shows comparatively low overhead, when using a simplified structure of persistent pointers [ATBM95].

iii) There is flexibility benefiting us to be able to implement many functionalities such as multiple type objects and garbage collection in a PH easily in the non-swizzling approach.[2]

A persistent pointer consists of two fields (see Figure 3.6 ). The first 1 byte represents `ph_id` which indicates the identifier of the PH in which the persistent object referred to by the pointer is allocated. The rest 3-byte stores `ort_id` from which the entry of the ORT for the object pointed to by the persistent pointer can be calculated with a hashing mechanism.

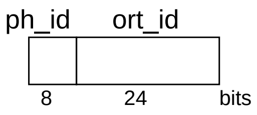

Figure 3.6: A persistent pointer format

A persistent pointer consumes 4-byte long, which has the same size as that of a pointer variable in C++ in the environment of 32-bit virtual address space. This assures size compatibility of volatile objects and persistent objects that are structurally identical except for pointer types. This limitation seems to be a bit severe, but it will be loosened in the near future when 64-bit address computers are widely used.

[2]In fact, instead of supporting automatic swizzling, INADA provides programmers with operations to convert persistent pointers to C++ pointers (i.e., virtual addresses) and vice versa for improving performance of their programs. This is described in [TAM94].

### 3.3.2 Strategy

Before introducing the functionality of multiple type objects, the most significant data held in ORT was only the offset address of the object from the top of the PH. For implementing multiple type objects, an entry in the ORT is extended for holding the offset address, the TypeID, the OID, and the NextID (Figure 3.7 (a)), and each size of them is 4-byte in the current system. The TypeID table (Figure 3.7 (b)) is also needed when translating from INADA codes into C++. The table is unique in a database and stored in secondary storage.

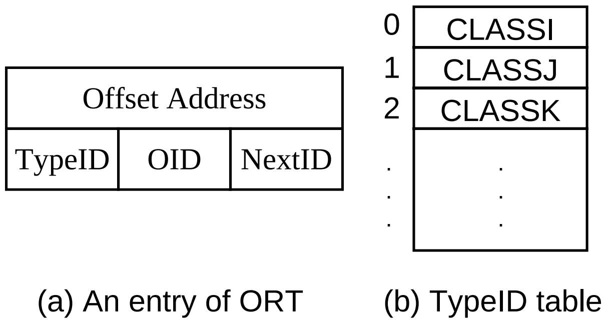

Figure 3.7: ORT entry and TypeID table

Algorithms for creating, adding a type to, manipulating, removing a type from, and deleting a persistent object are as follows.

**Algorithm 1** *Create*

i) Find a statement `persistent class-name* var-name`, check the TypeID table whether '`class-name`' has been registered or not. If not registered, register the class. Get the TypeID for the '`class-name`'.

ii) Find an entry in the ORT on the PH specified as the argument of '`new()`' for the new object.

iii) Allocate space in its object space, and assign the offset value from the top of the PH, the TypeID, and the OID to the entry correctly. The value of the OID refers to the entry. The NextID of the entry is set to null.

**Algorithm 2** *Add a type*

i) Check the TypeID table whether '`class-name`' has been registered or not. If not registered, register the class. Get the TypeID for the '`class-name`'.

ii) Find the entry in which the value of NextID is null with following NextID chain from the OID which points to the object that is going to be added the type.

iii) Find an entry in ORT on the PH specified as the argument of '`new()`' for the type.

iv) Allocate space in its OS, and assign the offset value, the TypeID, and the OID to the entry correctly. The NextID of the entry is set to null. And set the NextID found in ii) as points to the entry in ORT found in iii).

**Algorithm 3** *Manipulate an object through a persistent pointer*

i) Get the TypeID of the type through which the object is going to be manipulated from the TypeID table. The type is the specified one when users use the '`as`' construct, or the default one specified at the declaration of the persistent pointer variable.

ii) Find the entry in ORT which has the TypeID in the NextID chain. If it is not found, terminate because the object does not have the type, and this means a runtime error. If found, return the virtual address.

**Algorithm 4** *Remove a type*

i) Find the entry whose TypeID stands for the type in the NextID chain. If it is not found, terminate because the object does not have such type.

ii) Release the space pointed to by the offset value.

iii) Remove the entry from the NextID chain properly.

iv) Be invalid the entry for recycling.

**Algorithm 5** *Delete*

i) As following the Next ID chain, release all spaces pointed to by offset values.

ii) As following the Next ID chain, be invalid all entries for recycling.

Algorithms 1 through 5 implement the multiple type concept except for the '`this as` TYPE' construct, which may be appeared in class definition (for instance, see Figure 3.3). Since '`this`' is the same as that of C++, i.e., '`this`' is not a persistent pointer but a virtual address, Algorithm 3 cannot be applied. Hence, we need the following algorithm.

**Algorithm 6** *A statement* '`this as` TYPE'

i) After a statement '`this as` TYPE' is found, find the ORT entry which pointing the space for the object, whose type is the class where '`this`' is appeared, by calculating from its virtual address ('`this`').

ii) Get OID from the OID field in the ORT entry.

iii) Do Algorithm 3.

Figure 3.8 illustrates what is going on when creating a persistent object whose type is '`CLASSI`' and adding '`CLASSJ`' to the object (in the figure, ONT is ignored for simplicity).

In the TypeID table, the TypeIDs of '`CLASSI`' and '`CLASSJ`' are 0 and 1, respectively (Figure 3.8 (a)). Figure 3.8 (b) shows the status of the PH where a persistent object whose type is '`CLASSI`' is created and (c) shows where '`CLASSJ`' is added to the object.

## 3.4 Related Work

The system adapting the most similar implementation of persistent objects to INADA is probably ObjectStore [LLOW91]. It is based on the memory mapped file I/O architecture, too. While ObjectStore supports a swizzling technique for dereferencing persistent pointers, INADA employs a non-swizzling technique and prepare operations for improving performance. The flexibility of the non-swizzling technique brings about a very simple implementation of multiple type objects. ObjectStore does not support such the functionality at all.

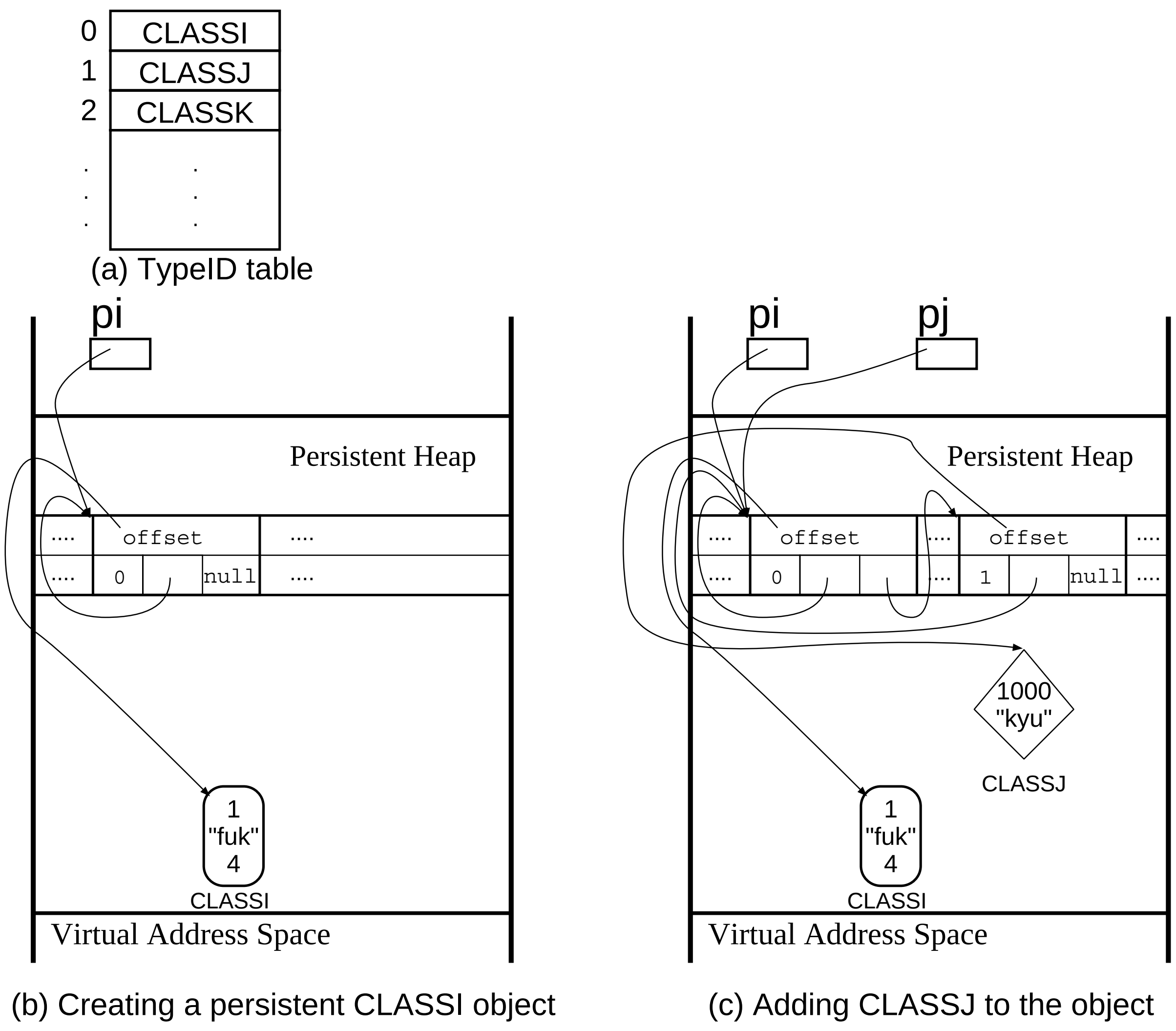

Figure 3.8: Creating an object and adding a type to the object

Iris [Fish+87] is the first system that allows an object to obtain and lose types dynamically. However, name conflict occurs when an object belongs to many types and the types have the method of identical name [RS91]. In contrast, there is no such problem in INADA, since any persistent object is manipulated only through a persistent pointer, which is declared with certain type, and INADA provides '`OID as` TYPE' construct when referring the object through 'TYPE'.

Aspect concept [RS91] is designed to extend objects to support multiple roles. In their data model, type of an object is separated from its implementation. The implementation of an object is practically similar to C++ class definition. The type provides only interfaces for accessing the object. An aspect is defined to be an implementation of a type, say `A`, with the base type, say `B`. Using this definition, an object, `a`, can be created for type `A` as an aspect of the already created object, `b`, whose type is `B`.

Although `a` and `b` represent different aspects of the identical object (e.g., `Employee` and `Person`), `a` and `b` have their own references. However, the value called OID is given to an object so as to identity two different aspects.

The work described in this chapter was motivated by [RS91], and we tried to integrate this mechanism into INADA without changing C++ object-oriented framework and its type system. However, at least two problems were found when simulating the functions described in [RS91].

i) The strategy of separating OID from reference is not compatible with volatile objects in C++. The identifier of a C++ object is actually a pointer that is used as the reference, too.

ii) In the aspect mechanism, classes defined independently cannot serve as an aspect to each other.

[AB+93] describes Fibonacci, a strongly typed object-oriented database programming language with ability to model objects with roles. As noted in [RS91], [RS91] does not support inheritance mechanism, but Fibonacci and INADA do. The key difference between Fibonacci and INADA is that INADA is based on C++, which has been already diffused widely in a large number of users, while Fibonacci has their own data model. Therefore programmers can use the ability to model multifaceted objects in INADA easily if they have known about C++, whereas they have to learn a completely new data model from the beginning to use Fibonacci.

## 3.5 Summary

This chapter described the design of multiple type objects in INADA, a new persistent programming language under development at Kyushu University, Japan. Any persistent objects in INADA may get and lose types dynamically. INADA is an enhanced C++ language. We have also shown the implementation of multiple type objects in a simple manner, and the type checking which can be processed statically by the type system of C++.

There are a lot of merits by introducing multiple type objects that are not shown in this chapter. One is that they are useful for views in object-oriented databases [AA93, AAM95]. Views can be defined as virtual set objects, and manipulation on the views are translated into manipulation on sets of actual existing objects. This is discussed in detail in the next chapter.

Some open problems are left. INADA will need more facilities concerning the multiple type concept. One is that INADA should be able to handle semantic constraints among types. For example, adding type '`Part`' to a '`Person`' object does not make sense in general applications. Users might want to describe something like that objects whose type is '`A`' cannot be added type '`B`'. The other is that an object cannot have a couple of same types in the current system. For example, there can be a person who works two places in the world, but INADA cannot implement the two jobs in one type because there cannot exist an object having two identical types, say '`Employee`'.

# Chapter 4

# Object-Oriented Views

Persistent objects may be shared among many application programs, but they may be handled in different ways in each of the application programs; only needed attributes can be selected from persistent objects and processed. View is the mechanism that allows users to deal with data as they like. This chapter describes how to implement object-oriented views in INADA. INADA has rich functions and high extensibility to provide facilities of processing queries on sets of objects. In INADA, views are implemented as virtual sets of objects, and manipulation on views are translated into manipulation on sets of actual existing objects. The concept of virtual set attributes is also proposed in this chapter.

## 4.1 Introduction

View is an important mechanism in databases. Relational databases successfully provide views to their users. In the last decade, view mechanisms for object-oriented database management systems have been proposed [e.g.,AbBo91,HeZd88,MaMe91,Rund92]. In these studies, a view is defined as a virtual class. These approaches are based on the object model in which a container of objects is a class, and hence a view must be defined on the whole objects of a class.

In relational databases a view may be regarded as a virtual table, i.e., a table that does not exist in its own right but looks to users as if it does. A view is defined on base table(s) or other view(s). Note that a base table is not a template of tuples but a set of tuples, that is, a container of tuples. Thus, we can create multiple sets of tuples on a single schema and use them independently to define different views in the relational model. This leads us to the belief that object-oriented views should also be defined on sets of objects stored in physical storage, and never on classes which specify properties and behaviors of those objects as templates of objects belonging to the classes.

This chapter proposes a view mechanism in an object-oriented persistent programming language INADA [AAM92, ATAM93, AAM95]. INADA borrows and extends the object model of C++ so that it provides facilities for handling a large amount of sharable persistent objects and processing queries on collections of objects. In INADA, a class is a type definition and not a collection of objects of the type: this is the same as C++.

To manipulate collections of objects, INADA provides *set objects*, which have a special interface defined by the system. A set object is a container of objects in the object model of INADA. Throughout this chapter the term 'set' is used for representing a collection of objects, that is, representing either a set in mathematical sense or a multi-set [Knu69].

Any persistent object in INADA may have multiple types as described in the previous chapter. Any type can be attached to and deleted from a persistent object. In the object model of INADA a user can create and delete views by applying this mechanism to set objects.

## 4.2 Set Objects

In the case where we have to manipulate a large amount of persistent objects, it is very important to retrieve only the objects satisfying certain conditions from the set of persistent objects. To this end, we need set functionality of retrieving certain objects. Generally speaking, we cannot define and implement a set which make it possible to provide efficient functionality for all kinds of applications. Therefore, INADA does not provide system-defined sets but define interfaces as standard which a set object must possess for the functionality. Such classes that have the interfaces are called as set classes in the language.

This section explains only a part of the interfaces, which are the most basic ones, so that the readers can easily follow the rest of this chapter.

- `Iterator* OpenScan(Iterator* i)`

The method opens a new iterator for a set object and returns the reference to it. `i` indicates where a user intends to open the iterator, whose default value is null. In the case where `i` is null, the iterator referring to the top of a set object is returned. If the set has no element, null is returned.

- `Type* GetElement(Iterator* i)`

  The method returns a pointer to the element referred to by the iterator, which is referred to by `i`. The name of members' type should be written on the place of '`Type`'.

- `Iterator* Next(Iterator* i)`

  The method makes the iterator referred to by `i` point to the next member and returns the reference to the iterator.

- `void CloseScan(Iterator* i)`

  The iterator referred to by '`i`' is closed by this method.

In the above description, `Iterator` and `Type` stand for the type of iterator of the set class and the type of element of the set class, respectively.

INADA provides '`for all`' syntax as an extended '`for`' statement in C++ for iteration over a set object. This syntax allows users to write a program where they can apply manipulation only on objects which satisfy certain condition(s). Suppose `domain` is a pointer referring a set object in which the type of elements is `Type`,

```
for all Type x in domain
such that  condition
do  manipulation
```

means that the element `x` in the set referred to by `domain` satisfies *condition* and *manipulation* is applied on only such `x`. The above expression would be translated into C++ statements which use the interfaces mentioned above.

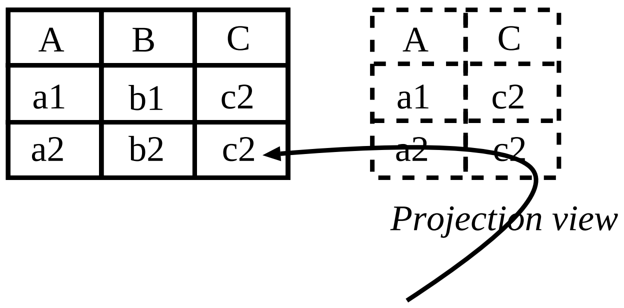

Figure 4.1: A view as a virtual table

## 4.3 Views and Their Implementation

Relational databases provide set-oriented manipulation and query languages on a large amount of data. Several researchers have tried to integrate such kind of facilities into object-oriented systems [AIM+90, Deu+90, Fish+87, KG+90]. Also, there are several discussions regarding views in object-oriented databases [AB87, MM91, Rund92, SLT91, TYI88]. However, none of them can implement the function of views completely.

There are mainly two types of view implementations in relational databases, i.e., views exist as real relations and as virtual ones. In the former, update on base relations which include real data is not reflected automatically on the views; therefore, we must have some mechanism for propagating the update to the views. On the other hand, in the second case, updates on base relations are automatically reflected to the views when we access the views (Figure 4.1). We can implement this by means of translating queries on the views into queries on the base relations.

Object-oriented database researchers attempted to integrate view facilities as virtual classes [SLT91, Rund92]. But, if a class is a container of the class's objects, this attempt is unlikely to succeed. Because there is only one class hierarchy and must not be more than one in their object models, the virtual classes, which construct views actually, must be built in the class hierarchy (see Figure 4.2). The inheritance of attributes in an object-oriented database is determined as the class hierarchy. Thus, it is a big problem how to put the virtual classes, which are defined by queries, to proper position in the class hierarchy already existed.

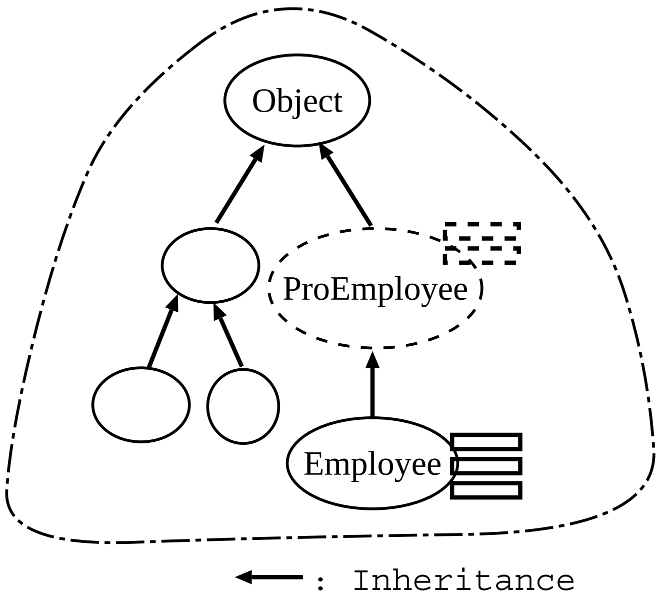

Figure 4.2: Reconstruction of a class hierarchy

Incidentally, INADA is an enhanced C++; there can be more than one class hierarchy, i.e., we do not have to care the number of class hierarchies (see Figure 4.3). Also, classes are not containers of objects. Instead, INADA has set objects which actually become containers of objects. Since set objects do not have any relations to class hierarchy, the problem does not arise.

We have the following two issues: how to define views and how to implement the views by means of INADA's facilities. Set objects are created from classes which are defined by users, and '`for all`' statement which is used to manipulate elements of set objects is a quite primitive function. Thus, a concrete solution for the issues is proposed in this chapter.

This section employs several concrete examples regarding views, how to define them in INADA, and how to implement them in INADA.

As mentioned earlier, views can be implemented as virtual sets which are calculated and realized when treating the views. Although we must calculate whenever a view is used, update on data which are really stored and existed is reflected to the view automatically in the approach. Therefore, this approach was adopted for integrating the view mechanism into INADA.

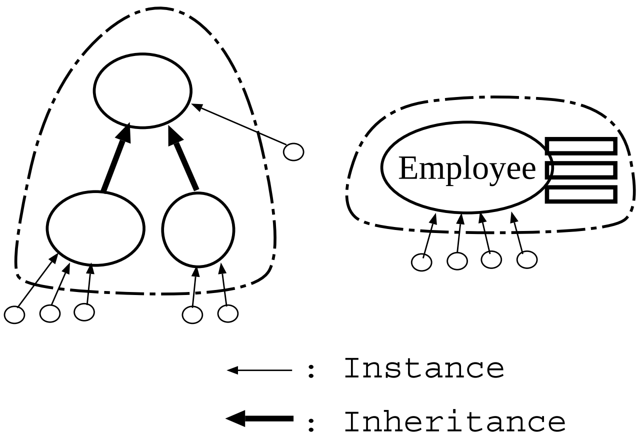

Figure 4.3: Class hierarchies in the data model of C++

INADA provides the following expression in order to define views.

```
view ViewSet on BaseSet{
                a_{i_1} for a_{j_1};
                a_{i_2} for a_{j_2};
                          :
                a_{i_k} for a_{j_k};
                  where
                    condition
                      };
```

where $a_{i_l}(l = 1, 2, \cdots, k)$ stands for an attribute of views and $a_{j_l}$ $(l = 1, 2, \cdots, k)$ means an attribute of base objects. The keyword `view` reveals that this is a view definition like as the keyword `class` in C++ and that the following description defines a class. $a_{i_l}$ `for` $a_{j_l}(l = 1, 2, \cdots, k)$ means "the view has the attribute $a_{i_l}$, which corresponds to the attribute $a_{j_l}$ in a base object." Both $a_{i_l}$ and $a_{j_l}(l = 1, 2, \cdots, k)$ can be member variables and methods in a base object, but $a_{j_l}(l = 1, 2, \cdots, k)$ must be public.

This section discusses views like those which are defined with selection and projection in relational databases (hereafter, the two types of views are referred to as selection views and projection views, respectively). For implementing the selection and projection views, you have to take the following steps.

i) For projection views, you define first the C++ class `View`, which has attributes $\mathtt{a}_{i_1}$, $\mathtt{a}_{i_2}$, ⋯, $\mathtt{a}_{i_k}$. Instance objects of this class are elements of instance objects of the class `ViewSet` defined in ii), and exist virtually. Instance objects of `ViewSet` construct a view of instance objects of `BaseSet` class, which is supposed to be defined already elsewhere.

ii) You define class `ViewSet` in INADA. The class is like that the type of its instance objects is `Base` in the case of selection views, or is `View` in the case of projection views. `Base` class is supposed to be defined already elsewhere. In a method of `ViewSet` class the corresponding methods to those of `BaseSet` class are evaluated by means of the syntax '`OID` as `Type`.' An interface in `ViewSet` as a selection view includes selection predicate(s) which appear in the `condition` part.

iii) You create an object of `ViewSet` class as an object of another class (or type) of a base object of `BaseSet` class. In other words, you create a multiple type object by adding type `ViewSet` to an object of `BaseSet` type. This multiple type object is a view of the base set object.

iv) A '`for all`' statement which accesses the view is translated into a '`for all`' statement accessing the base set object. This can be done automatically, since both `BaseSet` and `ViewSet` have the same interfaces which set classes must have in INADA.

What users have to do with this approach to views is to create a set object as a view of a base set and to delete it when it is no longer needed after manipulating it. Although the class of elements of a view set is defined in i), any object for this class is not actually created; therefore, there is no overhead regarding it. Let us consider the following example.

**Example 1** *Suppose that you must model the employees in a company. You model each employee using* `Employee` *class which has attributes of name, age, department number, and pay. And suppose that the set object, which is referred to by* `eset` *pointer variable, holds many objects of the class* `Employee`*.*

Figure 4.4 shows a sample code for this example (for the sake of simplicity, detailed codes of methods and so forth are omitted). `Set` is implemented as a template class. In the constructor of Employee default values are set in all member variables, i.e., the default values are used if users do not put the value(s) when creating an object. Change_Dept() method is an update method to change an employee's department. In Figure 4.4 five persistent objects of Employee are created as elements of a set object of Set<Employee>. The persistent heap referred to by pho is mapped on EmpFile file. Figure 4.5 shows this conceptually.

The rest of this section presents some concrete views on the set in Example 1.

**View 1** *A view which allows users to get names, ages, and departments, and cannot retrieve pays from a set of employees. (A projection view)*

For implementing this view, users can define Proview, for example, as follows:

```
view Proview on Set<Employee>{
    char* Name() for Name();
    int AGE() for Age();
    int DeptNo() for DeptNo();
};
```

where char* Name() for Name() means "the view has the method named Name(), which returns a value whose type is char*, and the method corresponds to the Name() in its base set." It is interesting to note that names of methods in a view need not to have the same names as the methods in its base class (int AGE() for Age() in the example is an example).

Figure 4.6 shows the code translated from the view definition. As the figure shows, the data of pays cannot be accessed from the class Proview_Employee. The class Proview whose elements' type is the Proview_Employee is used to implement the view.

**View 2** *A view which has only employees whose pays are equal to or more than two thousand. (A selection view)*

This view Selview can, for example, be expressed as follows:

```
    view Selview on Set<Employee>{
        char* Name() for Name();
        int Age() for Age();
        int DeptNo() for DeptNo();
        int Pay() for Pay();
        where Pay() >= 2000;
    };
```

```
class Employee{
public:
    char name[40];
    int age;
    int deptno;
private:
    int pay;

public:
    Employee(char* n="No-Name",int a=0,int d=-1,int p=-1)
        { strcpy(name, n);
          age=a;
          deptno=d;
          pay = p;} // Constructor
    ~Employee(); // Destructor
    char* Name(){ return name; }
    int Age(){ return age; }
    int DeptNo(){ return deptno; }
    int Pay(){ return pay; }
    Change_Dept(int d){ deptno=d; } // Update
};

template <class T>
class Set{
public:
    Set(); // Constructor
    T* GetElement(iterator<T>* i);
    iterator<T>* Next(iterator<T>* i);
    iterator<T>* OpenScan(iterator<T>* i);
    void CloseScan(iterator<T>* i);
        :
};

main(){
    PHFT* phft = new PHFT();
    PHO* pho = new PHO(phft, "EmpFile");
    persistent Employee *emp;
    persistent Set<Employee>* eset=new(pho)Set<Employee>();
    emp=new(eset)Employee("Ari",26,4,1000);
    emp=new(eset)Employee("Kyu",50,2,3000);
    emp=new(eset)Employee("Csc",34,4,1800);
    emp=new(eset)Employee("Dat",28,1,1500);
    emp=new(eset)Employee("Kum",55,3,3800);
        :
}
```

Figure 4.4: Class definitions of example 1 and creation of instance objects

In this view definition, you can see that all members of the base class are specified. This means that an element of the view has the same members as those of the base class. In this case, the interpreted class in C++ is the same as the base class, i.e., the interpreted class is unnecessary. To avoid this, the following definition can be used:

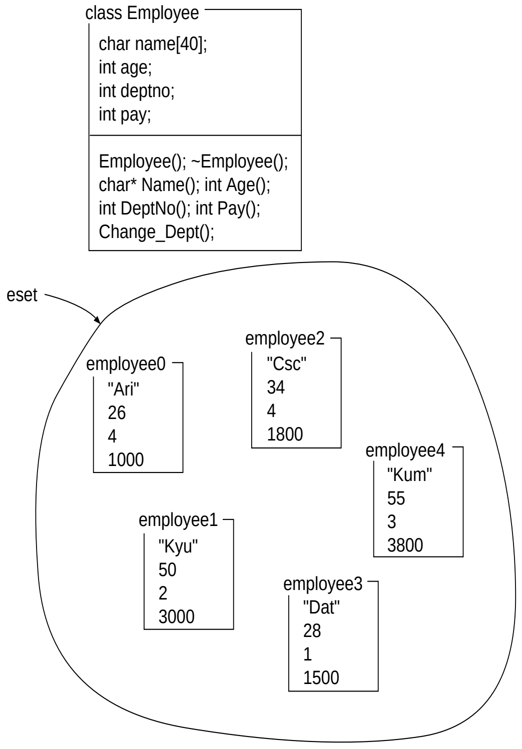

Figure 4.5: A conceptual figure of example 1

```
view Selview on Set<Employee>{
    where Pay() >= 2000;
};
```

As you see, INADA provides a view definition which has `where` clause and no member definition. This view can be translated as Figure 4.7.

If a view definition has no members, INADA translates that with using the base element class as it is. The set which is created from the translation returns only a set of elements which satisfy the condition(s) specified. In Figure 4.7, the condition defined in the view definition can be seen in the interface of the class `Selview`.

**View 3** *A view which returns names of employees each of whose age is equal to fifty or more than that, and users cannot access the rest of members of class* `Employee`. *(A view by combination of selection and projection)*

```
class Proview_Employee{
private:
    char name[40];
    int age;
    int deptno;
    int pay;

public:
    char* Name(){return ((Employee*)this)->Name();}
    int AGE(){return ((Employee*)this)->Age();}
    int DeptNo(){return ((Employee*)this)->Deptno();}
};

class Proview{
public:
    Proview(){}
    Proview_Employee* GetElement(iterator<Proview_Employee>* i){
        return (Proview_Employee*)
            (this as Set<Employee>->GetElement((iterator<Employee>*)i));}
    iterator<Proview_Employee>* Next(iterator<Proview_Employee>* i){
        return (iterator<Proview_Employee>*)
            (this as Set<Employee>->Next((iterator<Employee>*)i));}
    iterator<Proview_Employee>* OpenScan(iterator<Proview_Employee>* i){
        return (iterator<Proview_Employee>*)
            (this as Set<Employee>->OpenScan((iterator<Employee>*)i);}
    void CloseScan(iterator<Proview_Employee>* i){
        this as Set<Employee>->CloseScan((iterator<Employee>*)i);}
};
```

Figure 4.6: A projection view

This view `Combiview` can be expressed in INADA as follows:

```
view Combiview on Set<Employee>{
    char* Name() for Name();
    where
        Age() >= 50;
};
```

This definition has both its own members and `where` clause in which a condition is specified. It should be noticed that this is a view combining both projection and selection predicates. This is translated as shown in Figure 4.8. The class `Combiview_Employee` has only one public method named `Name()` due to the view definition. The condition `Age()>=50` is built in the standard interface of the class `Combiview`.

```
class Selview{
public:
    Selview(){}
    Employee* GetElement(iterator<Employee>* i){
        return this as Set<emplyee>->GetElement(i);}
    iterator<Employee>* Next(iterator<Employee>* i){
        iterator<Employee>* tmp
            =this as Set<Employee>->Next(i);
        if (!tmp){
            CloseScan(tmp);
            return 0;}
        if((GetElement(tmp)->Pay())>=2000)
            return tmp;
        else tmp = Next(tmp);
        return tmp;}
    iterator<Employee>* OpenScan(iterator<Employee>* i){
        iterator<Employee>* tmp
            =this as Set<Employee>->OpenScan(i);
        if(!tmp){
            CloseScan(tmp);
            return 0;}
        if((GetElement(tmp)->Pay())>=2000)
            return tmp;
        tmp = Next(tmp);
        return tmp;}
    void CloseScan(iterator<Employee>* i){
        this as Set<Employee>->CloseScan(i);}
};
```

Figure 4.7: A selection view

## 4.4 Experimental Results

The table 4.1 shows one of experimental results obtained by running the examples described in the previous section on a Sun Sparc 2. In the experiment, 10000 objects were created and managed with a simple linear list. The table shows the time required for displaying names of employees who satisfy a condition. And the condition was changed to vary the amount of the employees. The manipulation on projection views took more time than that on the base set, since the projection views had a projection processing. Speaking of selection views, it took more time to display than to evaluate a selection predicate. Therefore, the amount of time required for processing selection views was less than that for processing the base set.

```
class Combiview_Employee{
private:
    char name[40];
    char age;
    char deptno;
    int pay;

public:
    char* Name(){return ((Employee*)this)->Name();}

protected:
    int Age(){return ((Employee*)this)->Age();}
    friend class Combiview;
};

class Combiview{
public:
    Combiview_Employee* GetElement(iterator<Combiview_Employee>* i){
        return (Combiview_Employee*)
            (this as Set<emplyee>->GetElement((iterator<Employee>*)i));}
    iterator<Combiview_Employee>*
        Next(iterator<Combiview_Employee>* i){
            iterator<Employee>* tmp
                =this as Set<Employee>->Next((iterator<Employee>*)i);
            if (!tmp){
                CloseScan( (iterator<Combiview_Employee>*)tmp );
                return 0;}
            if((GetElement((iterator<Combiview_Employee>*)tmp)->Age())>=50)
                return (iterator<Combiview_Employee>*)tmp
            else tmp
                =(iterator<Employee>*)Next((iterator<Combiview_Employee>*)tmp);
            return (iterator<Combiview_Employee>*)tmp;}
    iterator<Combiview_Employee>*
        OpenScan(iterator<Combiview_Employee>* i){
            iterator<Employee>* tmp
                =this as Set<Employee>->OpenScan((iterator<Employee>*)i);
            if(!tmp){
                CloseScan( (iterator<Combiview_employee>*)i);
                return 0;}
            if((GetElement((iterator<Combiview_Employee>*)tmp)->Age())>=50)
                return (iterator<Combiview_Employee>*)tmp;
            tmp=(iterator<Employee>*)Next((iterator<Combiview_Employee>*)tmp);
            return (iterator<Combiview_Employee>*)tmp;}
    void CloseScan(iterator<Combiview_Employee>* i){
        this as Set<Employee>->CloseScan((iterator<Employee>*)i);}
};
```

Figure 4.8: A combination view of projection and selection

## 4.5 Insertion and Deletion through Views

This section discusses insertion and deletion through views. Insertion and deletion in this section stand for insertion of elements with `new` operator and deletion of them with `delete`,

Table 4.1: Evaluation of projection and selection and their combination views (in seconds)

| Selectivity | Base set | Projection view | Selection view | Combination view |
|---|---|---|---|---|
| 0% | 1.94 | 2.15 | 0.54 | 0.56 |
| 20% | 1.95 | 2.24 | 0.66 | 0.67 |
| 40% | 1.95 | 2.24 | 1.04 | 1.06 |
| 60% | 1.97 | 2.24 | 1.45 | 1.46 |
| 80% | 1.95 | 2.23 | 1.85 | 1.87 |
| 100% | 1.95 | 2.24 | 2.24 | 2.26 |

respectively.

Insertion of an element is carried out by creating an object of the class as the element of a base set and inserting the object to the view. The following things have to be considered.

- In the selection views

  Insertion of objects which do not satisfy condition(s) is not allowed. This can be implemented by the system checking this when evaluating a sentence including constructor. By this mechanism, insertion through views are stable.

- In the projection views

  The system automatically complement the values which cannot be accessed through the views by the default values defined in the definition of the constructor.

Deletion is the delete process of an element object from a base set. The followings, like insertion, have to be considered.

- In the selection views

  The objects which satisfy selection predicate(s) are deleted. This is obvious because the OIDs processed by the operator `delete` are obtained through the views including the selection predicate(s).

- In the projection views

  The values of all members of an element which is going to be deleted are deleted regardless that every member can be accessed through the views.

```
class Proview_Employee{
private:
    char name[40];
    int age;
    int deptno;
    int pay;

public:
    char* Name(){return ((Employee*)this)->Name();}
    int AGE(){return ((Employee*)this)->Age();}
    int DeptNo(){return ((Employee*)this)->Deptno();}
    Change_Dept(int D){((Employee*)this)->Change_Dept(D);}
    Proview_Employee(char* N, int A, int D)
        {((Employee*)this)->Employee(N,A,D);}
};
```

Figure 4.9: A projection view with a method to update

Let us consider the following example, which obtained by modifying the View 1.

**View 4** *A view which has name, age, and department, and users cannot retrieve pay of an employee (a projection view). Also, users can update department and insert an employee through the view.*

This can be defined as follows:

```
view Proview on Set<Employee>{
    char* Name() for Name();
    int AGE() for Age();
    int DeptNo() for DeptNo();
    Change_Dept(int D) for Change_Dept(D);
    Proview(char* N, int A, int D) for Employee(N, A, D);
};
```

In this definition, you see the constructor for the insertion and the update method for the update process through the view. The result of the translation from this definition by the system is different from that part of the class `Proview_Employee`. The translation of the part is shown in Figure 4.9.

An update through the view is processed by an update to the base object. For example, an insertion

```
new(peset)Proview("Mas",26,4);
```

gives the same result as given by evaluating the following statement:

```
        new(peset)Employee("Mas",26,4,-1);
```

## 4.6 Virtual Set Attributes

The previous sections have shown that selection and projection views can be implemented as virtual sets by using the multiple type object mechanism on set objects. Each of them is a view to one set object. This section discusses a virtual set based on more than one set object.

Let us take a look at the following example.

**Example 2** *Suppose that you have to model the set of departments of a company. Each department holds department name, department number, and department manager as its members, and is modeled by the class* `Dept`. *And, suppose that the set object, which is referred to by* `dset` *pointer variable that is a persistent object of the class* `Set<Dept>`, *holds a number of objects of the class* `Dept`.

A simple code for this example is shown in Figure 4.10. Now let us consider combining the set object shown in Figure 4.10 and a set of `Employee` objects in order to obtain *employees belonging to certain department.* Since users cannot express two members belonging two types as one member of a type with the multiple type concept, this view cannot be implemented by the approach taken to implement views created from selection and/or projection predicates. This is because object-oriented concept does not allow to express the relationships of two objects in an object, though the concept is useful to encapsulate data and behaviors applicable to the data in an object. Thus, we must build relationship of two or more objects into one object to manipulate the relationship.

To implement this, virtual set attributes are introduced. The virtual set attributes have a virtual set object as their return values. When the attributes are evaluated, the values become real. To implement the combination in the above example, a user code the virtual set attribute `Member()` of the class `ExtDept` as follows:

```
class ExtDept{
public:
    char* Name() for Dept::Name();
    int DeptNo() for Dept::DeptNo();
    char* Mng() for Dept::Mng();
```

```
class Dept{
    char name[40];
    int deptno;
    char mng[40];
public:
    Dept(); // Constructor
    char* Name();
    int DeptNo();
    char* Mng();
};

main(){
                :
    persistent Dept *dpt;
    persistent Set<Dept>* dset=new(pho)Set<Dept>();
    dpt=new(dset)Dept("Design",4,"Ari");
    dpt=new(dset)Dept("Sale",1,"Dat");
                :
}
```

Figure 4.10: Class definitions of example 2 and creation of instance objects

```
         Set<Employee>* Member()
             where Employee::DeptNo()
                 == Dept::DeptNo();
     };
```

By adding this class to an object of the class `Dept` and accessing the object through `ExtDept`, you can retrieve all employees belonging to certain department.

## 4.7 Summary

This chapter discussed how to consider views in object-oriented framework, and how to implement object-oriented views in the persistent programming language INADA. The multiple type concept is one of the functions which should be included in persistent programming languages. Object-oriented views can be implemented easily by applying the concept to set objects.

INADA is an enhanced C++ programming language. Most of all functions in INADA are affected by many other work, e.g., [AG89, Deu+90, AIM+90, LLOW91, RC89]. The obvious different points between INADA and the other work are

i) INADA provides multiple type objects.

ii) Views are implemented as virtual sets; a view is implemented as a type of a base set.

All other work [HZ88, Rund92, SLT91, TYI88] think that views in object-oriented database systems are implemented as virtual classes made by some queries, because in their data models only one class hierarchy can exist and a class is a container of objects. Thus, they have to tackle on the problem that they build much complex class hierarchy over and over again whenever creating a new view; therefore, in this sense, their approaches to implement object-oriented views are not successful.

The new approach presented in this chapter is quite different from them. In INADA, a class is not a container of objects, and views are implemented as virtual sets. Therefore, a class hierarchy which is a database schema does not need to be rebuilt. In the approach proposed in this chapter what you have to do to create a view is to add a type which is for the view to a set object. When the view becomes no longer needed, you just delete the type from the set object.

[SS89] tries to implement views by allowing an object to have several interfaces while keeping its OID. However, [SS89] does not support the view mechanism that the relational database model supports.

INADA thinks of treating with sets of objects, and implements views to the sets. Manipulation on views, or virtual sets, are translated into that on their base sets, as shown in this chapter.

The examples of the translation shown in this chapter, however, use the most primitive way. In INADA you can define and handle set objects implemented in various ways so as to fix your application. Integration this flexibility into the translation must be consider as future work.

# Chapter 5

# Conclusions

In this dissertation design and implementation of the facilities that persistent programming languages should possess have been studied. The issues described in this dissertation are a part of results we have obtained during designing the persistent programming language INADA and developing it practically on real systems.

We have studied mainly the following three topics concerning persistent programming languages in this dissertation.

i) Persistence of objects

We investigated memory-mapped I/O environment as a platform on which we implemented the persistent programming language INADA, and examined several implementations of persistent pointers in the environment. The memory-mapped I/O enables us to implement persistent heaps which are portions of virtual address space and correspond to files on secondary storage in the local site or remote sites. Because a persistent heap is a contiguous address space, we can handle huge persistent objects efficiently. The results we obtained from the examination disclose that non-swizzling pointer techniques are not so inefficient to discard the techniques when comparing with swizzling pointer techniques. In addition, non-swizzling pointer techniques have an advantage over swizzling pointer ones in that they provide flexibilities allowing us to build useful functions. As a result, we adopted a non-swizzling pointer technique to implement persistent pointers in INADA.

ii) Ability to change characteristics of objects

We proposed the framework in which a persistent object can get and/or lose its type(s) dynamically, and called such objects multiple type objects. Since persistent objects are shared by many applications, we need the mechanism to change forms of persistent objects suitable for each applications. In addition, we can model changes of objects as time goes on with multiple type objects. This is significant because persistent objects have so long life spans that requirement for applications manipulating the objects might change. We also proposed an implementation of this mechanism, and built it into INADA.

iii) Views in object-oriented systems

We discussed object-oriented views, and proposed a new implementation of views in object-oriented systems. Little work, to our knowledge, have succeeded so far to implement this mechanism. In INADA, a view is implemented by adding a type to a set object. Also, we designed definition constructs for views in the language.

There are still issues to investigate regarding the above three topics. Future work includes the followings.

- Optimization of treating persistent pointers

  We adapted the ORT approach to implement persistent pointers in INADA. We have to improve its performance. An idea is, as you can see in [INADA], to assign the virtual address calculated from a persistent pointer into a pointer variable when referring the persistent object referred to by the persistent pointer. If we can change statements like this when translating a program in INADA into C++, the performance would be improved.

- Integrating inheritance into multiple type object mechanism

  When accessing a multiple type object by certain type we adapted type name matching algorithm. In this implementation, however, inheritance relations between classes are ignored. In future, we need to integrate these two mechanisms.

- Optimization of looking up certain type from a multiple type object

  We proposed an implementation of multiple type objects in this dissertation. A linear link is used in the implementation; therefore, it might take long time to look up certain type from a multiple type object in some cases. We would have to improve holding multiple types and make up some better ways to retrieve certain types.

- Developing more user-friendly view definition languages

  The view definition proposed in this dissertation might not be easy-to-use enough because the definition is based on the programming language INADA. As a matter of fact the definition way is the most primitive programming construct. To make INADA a more user-friendly language, easy-to-use descriptions as syntax sugars must be needed.

- Improving of view processing

  There are several ways to implement set objects, and users can adapt their own implementations of set objects so as to fix their applications in INADA. Thus there must be several processes of views by using proper sets so as to get better performance. Also, we would be able to improve performance with rewriting view definitions, particularly in the case of combinations of selection and projection views.

# Bibliography

[AA93] M. Aritsugi and H. Amano : "Views in an Object-Oriented Persistent Programming Language," *Proc. of the International Symposium on Next Generation Database Systems and Their Applications*, Fukuoka, Japan, pp.18-25, Sep. 1993.

[AABJMT94] H. Amano, M. Aritsugi, et al. : "Shusse Uo: a Persistent Project of Developing an Flexible Platform for Advanced Database Systems and Applications," *Tech. Rep. of IEICE*, Mar. 1994, DE93-62.

[AAM92] M. Aritsugi, H. Amano, and A. Makinouchi : "Multitype Objects in Persistent Programming Language INADA," *Proc. of Advanced Database System Symposium*, Tokyo, Japan, pp.93-100, Dec. 1992 [in Japanese].

[AAM95] M. Aritsugi, H. Amano, and A. Makinouchi : "Implementation of Views in the Persistent Programming Language INADA," *Trans. of the Information Processing Society of Japan*, 36(4), pp.971-980, Apr. 1995 [in Japanese].

[AB87] M.P. Atkinson and O.P. Buneman : "Types and Persistence in Database Programming Languages," *ACM Computing Surveys*, 19(2), pp.105-190, Jun. 1987.

[AB+90] T.L. Anderson, A.J. Berre, et al.: "The HyperModel Benchmark," *EDBT*, pp.317-331, 1990.

[AB91] A. Abiteboul, and A. Bonner : "Objects and Views," *Proc. the 1991 ACM SIGMOD Int'l Conf. on Management of Data*, pp.238-247, May 1991.

[AB+93] A. Albano, R. Bergamini, et al. : "An Object Data Model with Roles," *Proc. of the 19th International Conference on VLDB*, pp.39-51, Aug. 1993.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, et al. : "The Object-Oriented Database System Manifesto," *The First Int'l Conf. on DOOD*, kyoto, Japan, pp.40-57, 1989.

[AG89] R. Agrawal and N.H. Gehani : "ODE (Object Database and Environment): The Language and the Data Model," *Proc. of ACM SIGMOD Conference on Management of Data*, pp.36-45, May 1989.

[AIM+90] M. Aoshima, Y. Izumida, A. Makinouchi, F. Suzuki, and Y. Yamane : "The C-based Database Programming Language Jasmine/C," *Proc. 16th VLDB Conf.*, pp.539-551, Aug. 1990.

[AM95] Aritsugi, M. and Makinouchi, A. : "Design and Implementation of Multiple Type Objects in a Persistent Programming Language," *IEEE 19th Annual International Computer Software and Applications Conference (COMPSAC '95)*, pp.70-76, Aug. 1995.

[ATAM93] M. Aritsugi, K. Teramoto, H. Amano, and A. Makinouchi : "Multiple Type Objects and Set Objects in an Object-Oriented Persistent Programming Language," *Technical Report CSCE-93-C02, Dept. of Comp. Sci. and Comm. Eng., Kyushu University*, Mar. 1993.

[ATBM95] M. Aritsugi, K. Teramoto, G. Bai, and A. Makinouchi : "Several Implementations of Persistent Pointers in a Memory-Mapped I/O Environment," *Proc. 6th International Conference on Database and Expert Systems Applications (DEXA '95)*, Lecture Notes in Computer Science 978, pp.490-501, Sep. 1995.

[Bar+] R.V. Baron, et al. : *MACH Kernel Interface Manual*, 25 Oct. 1988.

[BM94] G. Bai and A. Makinouchi : " WAKASHI/D: A Distributed Storage Server for New Generation Database Systems," *Proc. of the International Symposium on Advanced Database Technologies and Their Integration*, pp.137-144, Oct. 1994.

[CDN94] M.J. Carey, D.J. DeWitt, and J.F. Naughton : "The OO7 Benchmark," *CS Tech. Rep.* Univ. of Wisconsin-Madison, 1994.

[Codd70] E.F. Codd : "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, 13(6), pp.377-387, 1970.

[CS91] R.G.G Cattell and J. Skeen : "Object Operations Benchmark," *ACM Trans. on DS 17(1)*, pp.1-31, 1992.

[Deu+90] O. Deux, et al. : "The Story of O2," *IEEE Trans. on Knowledge and Data Engineering*, 2(1), pp.91-108, Mar. 1990.

[DMFV90] D.J. DeWitt, D. Maier, P. Futtersack, and F. Velez : "A Study of Three Alternative Workstation Server-Architectures for Object Oriented Database Systems," *VLDB*, pp.107-121, 1990.

[ERDB90] The Committee for Advanced Database Function : "Third-Generation Database System Manifesto," *SIGMOD RECORD*, 19(3), pp.31-33, Sep. 1990.

[ES91] M.A. Ellis, B. Stroustrup : *The Annotated C++ Reference Manual,* Addison-Wesley, 1991.

[Fish+87] D.H. Fishman, et al. : "Iris: An Object-Oriented Database Management System," *ACM Trans. on Office Information Systems*, 5(1), pp.48-69, Jan. 1987.

[HZ88] S. Heiler, and S. Zdonik : "Views, Data Abstraction, and Inheritance in the FUGUE Data Model," *Proc. the 2nd Int'l Workshop on Object-Oriented Database Systems*, Lecture Notes in Computer Science 334, pp.225-241, Sep. 1988.

[INADA] *INADA User Guide, Release 1.0*, Dept. of computer Science and Communication Eng., Kyushu University, 1994 [in Japanese].

[KG+90] W. Kim, J.F. Garza, et al. : "Architecture of the ORION Next-Generation Database System," *IEEE Tran. on Knowledge and Data Eng.*, 2(1), pp.109-124, Mar. 1990.

[KK93] A. Kemper, and D. Kossmann : "Adaptable Pointer Swizzling Strategies in Object Bases," *ICDE*, pp.155-162, 1993.

[Knu69] Knuth, D. : *The Art of Computer Programming. Seminumerical Algorithms*, Addison-Wesley, Reading, Mass., 1969.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb : "The ObjectStore Database System," *COMM. of the ACM*, 34(10), pp.51-63, Oct. 1991.

[Maki90] A. Makinouchi : "Architectures of the Object-Oriented Database Management Systems," *Information Processing 32(5)* pp.514-522, 1991 [in Japanese].

[MM91] J.-C. Mamou and C.B. Medeiros : "Interactive Manipulation of Object-oriented Views," *Proc. the 7th Int'l Conf. on Data Engineering*, Kobe, Japan, pp.60-69, Apr. 1991.

[Moss92] J. Eliot B. Moss : "Working with Persistent Objects: To Swizzle or Not to Swizzle," *IEEE Trans. on Software Engineering*, 18(8), pp.657-673, Aug. 1992.

[Obj91] Object Design, Inc. : "ObjectStore User Guide," Release 1.1, Mar. 1991.

[Ont94] Ontologic Inc. : *Ontos Object Database Version 3.0 Developer's Guide*, Ontologic Inc., Burling Mass, 1994.

[RC89] J.E. Richardson and M.J. Carey : "Persistence in the E Language: Issues and Implementation," *Software-Practice and Experience*, 19(12), pp.1115-1150, Dec. 1989.

[RS91] J. Richardson and P. Schwarz : "Aspects: Extending Objects to Support Multiple, Independent Roles," *Proc. of ACM SIGMOD Conference on Management of Data*, pp.298-307, May 1991.

[Rund92] Rundensteiner, E. A. : "*MultiView*: A Methodology for Supporting Multiple Views in Object-Oriented Databases," *Proc. the 18th VLDB Conf.*, Vancouver, British Columbia, Canada, pp.187-198, 1992.

[SB86] M. Stefik and D. G. Bobrow : "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, 6(4), 1986.

[SLT91] M.H. Scoll, C. Laasch, and M. Tresch : "Updatable Views in Object-Oriented Databases," *Proc. on the 2nd Int'l Conf. on DOOD*, Munich, Germany, pp.189-207, Dec. 1991.

[SS89] J. J. Shilling and P. F. Sweeney : "Three Steps to views: Extending the Object-Oriented Paradigm," *Proc. OOPSLA '89, ACM SIGPLAN Notices*, 24(10), pp.353-361, Oct. 1989.

[SZ90] E. Shekita, and M. Zwilling : "Cricket: A Mapped, Persistent Object Store," *Proc. of the 4th Int'l Workshop on Persistent Object Systems*, pp.89-102, 1990.

[TAM94] K. Teramoto, M. Aritsugi, and A. Makinouchi : "Design, Implementation, and Evaluation of the Persistent Programming Language INADA," *Tech. Rep. CSCE-94C-02*, Dept. of Comp. Sci. and Comm. Eng., Kyushu University, Mar. 1994.

[TYI88] K. Tanaka, M. Yoshikawa, and K. Ishihara : "Schema Virtualization in Object-Oriented Databases," *Proc. the 4th Int'l Conf. on Data Engineering*, pp.22-29, Feb. 1988.

[WD92] S. White and D. DeWitt : "A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies," *Proc. 18th VLDB Conference*, Canada, pp.419-431, 1992.

[Wil90] P.R. Wilson : "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Space on Standard Hardware," *Tech. Rep. UIC-EECS-90-6*, Dec. 1990.