INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

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

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

USING OBJECTSTORE IN BUILDING C++ INTERFACE APPLICATION

Rita Vafadar Afshar

A MAJOR REPORT
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 2000

© RITA VAFADAR AFSHAR, 2000

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre reference

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-47854-8

Canada

Abstract

Using ObjectStore in building C++ Interface Application

Rita Vafadar Afshar

ObjectStore is an object oriented database management system. It provides a tightly integrated language interface to the traditional database management system features such as persistent distributed data access and associative queries. ObjectStore enables "ordinary" C and C++ programmers to add persistent database to their applications without having to learn a new language and without sacrificing any performance. This is an advantage over the relational database approach in which there is an impedance mismatch between database query language and the high level programming languages. ObjectStore and C++ programming language share the same data model, for instance base types, such as integers, characters, and pointers, as well as more complex types, such as structures and classes. The operators defined on the data types are also equivalent. Actions such as pointer dereferencing (- >) are valid for both persistent and transient data, with no difference to the programmer.

ObjectStore also ensures that the resulting unified interface yields access speed for persistent objects that are usually equal to that of in-memory dereferences of transient data.

Our objective in this report is to describe how to create database applications, using the fundamental features of ObjectStore such as, Database access, Transactions Exceptions, Collections, Queries, Indexes and Relationship facilities. The general information and instructions for generating schemas, compiling, linking and debugging ObjectStore are explained in details.

We develop an application, **Airline Reservation** to examine most of the database functions in ObjectStore system. This application has written in C++ language and uses ObjectStore API whenever needed to use database.

Contents

Acknowledgments

I would like to thank my supervisor, Gosta Grahne for introducing me to the concepts of ODBMS and encouraging me to finalize my project.

I am grateful to Mr. Swiercz Stanley (Applications and Information Systems Manager) for his enormous helps on compiling and debugging my ObjectStore application.

I would like to dedicate this project to my husband, and thanks him for his unfailing positive encouragement and support of me. Also my special thanks to my real friends and family, especially my Mom and my Dad.

Chapter 1

Introduction

1.1 What Is ObjectStore?

ObjectStore is an object-oriented database management system. It allows you to:

- Create and modify C++ objects (as well as C structures)instead of tables, columns, rows, and tuples.
- Access data in the format in which it exists in the application.
- Describe, store, and query complex data used in sophisticated computer applications, as well as data traditionally managed by relational database applications, such as MIS programs.
- Store data independently of the data type.

1.2 Persistent Storage

Persistent data is the data that survives after a process, which has created it terminates. ObjectStore stores persistent data in stable storage in databases, typically disks.

There are two kinds of databases. A **file database** which is a regular operating system file. A **rawfs database** (raw file systems) which resides in an ObjectStore file system, managed by an ObjectStore Server. **Rawfs databases** are not discussed

in this document.

Each database is made of **segments**, which are variable-sized regions of memory. They can be used as a unit of transfer from persistent storage to program, or transient memory. Each segment, in turn, is made of **pages**. A specified number of pages can be used instead of segments as the unit of transfer to program memory.

1.3 Language Integration

ObjectStore and C++ programming language share the same data model. The data types are the same for both, including base types, such as integers, characters, and pointers, as well as more complex types, such as structures and classes. The operators defined on the data types are also equivalent. Actions such as pointer dereferencing (- >) are valid for both persistent and transient data, with no difference to the programmer.

In the following code, we have defined two instances of class Passenger, one transient and one persistent (note the overloaded **new** operator). The manipulations of the objects are identical in each, both a part from the necessarily different persistent data definition, the piece of code looks exactly like ordinary C++:

```
os_database* db;
/*transient */
Passenger* passenger1 = new Passenger(1,"Julian Smith","911 A Street");
/* persistent */
Passenger* passenger2 = new(db,Passenger::get_os_typespec)
Passenger(2,"Bob Apple","30 5th Ave");
cout <<passenger1->address<<endl;
cout <<passenger1->address<<endl;
```

There is a tight integration between ODBMS and C++ programming language, which gives us the following features.

- **Re-usability** -We can use just one code for persistent and transient data. Also, there is no need to have additional statements in order to access persistent data, so binary compatibility with libraries and subroutines is possible.

- **No translation** -We are spared from having to write extra code to translate between the database and the application data types. This is referred to as single-level storage, and is in contrast to relational systems where code must be written to pick fields out of tuples and copy them into the data members of objects.

- **Ease of programming** - Because the constructs of the programming language are used as the data manipulation language of the ODBMS and because the type systems are identical, we are provided with an environment that is an easy transition from standard coding.

As illustrated in the code example above, each instance of class can be declared as persistent or transient through the explicit declarations and definitions, using the overloaded **new** operator.

1.4 Object Access

Providing efficient object access speed is a requirement deriving from the typical target application's accessing patterns. This type of application usually interleaves small database operations with small amounts of computation. It implies a large number of database accesses in comparison to the amount of data being fetched.

Object access in ObjectStore means dereferencing, like pointers in C or C++ . Performance for ObjectStore boils down to ensuring that the amount of time required to dereference pointers to persistent objects is as close as possible to that of transient objects. ObjectStore does this through the use of a unique **Virtual Memory Mapping Architecture (VMMA)** that involves sophisticated memory mapping, caching, and clustering techniques. These techniques provide the basis for database management services that match the highest performance levels of proprietary file management systems.

1.5 ObjectStore Processes

ObjectStore applications require two auxiliary processes for application execution, an ObjectStore Server and a Cache Manager.

A Server handles accesses to ObjectStore databases, including storage and retrieval of persistent data. When ObjectStore is installed, the system administrator typically arranges for each server to start when its host machine boots. A single application can use several databases, including databases on different file systems, handled by different servers. Most users never have to worry about starting and stopping servers.

A Cache Manager is started automatically when an ObjectStore application starts. The cache manager is a daemon that runs on the machine running the client application. Its function is to respond to server requests as a stand-in for the client application, in order to participate in the management of the application client cache. The client cache is the local holding area for data, mapped or waiting to be mapped into virtual memory.

If additional ObjectStore applications are started on the same machine, the same Cache Manager functions for those applications as well. Although the same machine can run several ObjectStore applications at once, only a single Cache Manager is ever running on a given machine. As with servers, most users never have to worry about starting or stopping Cache Managers.

1.6 ObjectStore Memory Mapping Architecture

With ObjectStore, data is transferred between database memory and program memory completely automatically in a manner transparent to the user. ObjectStore detects any reference in a running program to persistent data, and automatically transfers the page containing the referenced data (possibly together with adjacent pages) across the network to the application's cache. Then, the page containing the referenced data is mapped into virtual memory.

Sometimes the referenced data is already in the client cache (because data in the same pages was already used, and the required page was not swapped out of the cache), and all that is required is a virtual memory mapping. Sometimes the data

is already mapped into virtual memory (because data on the same page was already used in the current transaction)and then nothing additional is required to access it. Once data has been mapped into virtual memory, accesses to it will be as fast as accesses to regular, transient data.

ObjectStore achieves the combination of transparency and efficiency with a unique memory mapping architecture. All data is stored in an ObjectStore database, in its native C++ format. All pointers in a database take the form of regular virtual memory pointers.

In [7], more information is given on how the transfer of data between persistent and transient memory is handled.

1.7 Schemas

We have seen that segments and pages are used as the units of transfer between the server and its clients. We also know that storage entities contain the data structures, variables, and classes used by ObjectStore applications. The question is, how does ObjectStore gets this information?

ObjectStore needs to be able to identify which objects exist on a particular page as well as to determine their exact boundaries and the layout of their internal structures. Help is not available from C++ because classes are not run-time entities, and thus there is normally no location where this information can be placed. The natural solution is to store class layout information as C++ objects in the actual database, under the control of ObjectStore. ObjectStore refers to such class information as *schemas*.

1.8 ObjectStore ODBMS

As we mentioned, ObjectStore's architecture provides a unified programming interface to both persistently and transiently allocated data. The object access speeds are same for in-memory dereferences of transient data. The first design point was realized

by tightly integrating the Object-Store ODBMS with C++; the second, through an interesting virtual memory mapping architecture that also plays a vital role in merging the data models of both the language and the database. The result is an ODBMS that enables C and C++ programmers to add object-oriented database persistence to their applications without having to learn a new language and without sacrificing performance.

We can use the following database functions in ObjcetStore's applications:

Database access

Database access allows us to perform basic activities, such as opening and closing databases, and creating and accessing persistent objects.

Transactions

Transactions take the above basic activities and regroup them into logical units of work that are performed either all together or not at all. At the same time ObjectStore does not allow other transactions from viewing intermediate results.

Exceptions

If anything goes wrong during the database access activities, we can trap the ObjectStore-generated errors through an exception mechanism to provide our own error handling routines.

Collections

Grouping objects into various types of collections is simple with ObjectStore's library of collection classes. This class library enables us to define lists, arrays, sets, or bags of related objects.

Queries

Collections can be queried efficiently through ObjectStore's query facility, which selects objects on the basis of the values of their attributes (data members).

Indexes

To improve the performance of queries, indexes can be built on the collections.

Relationships

ObjectStore provides a relationship facility that automatically maintains the pointers used to specify relationships between objects.
We examine most of these functions as our sample application, *Thin Airline Reservation.*

Chapter 2

Introducing Sample Application

In this chapter, we describe a sample application **AirLine Reservation** which we developed to examine most of the ObjectStore concepts as we introduce them.

2.1 Outline of requirement for the Airline Reservation System

Airline reservation application

The database keeps track of employees, aircraft, flights and passengers. Each aircraft is described by its serial number, date of manufacturing, manufacturer, accumulated flight hours, and the spare parts it needs regularly. A spare part has a serial number, a specification and a place. Each aircraft is a plane type. A pale type has a name(e.g. MD-11), a manufacturer, a maximum speed, a length, and a number of seats. A flight has a number(e.g. AC 47), a source and destination, a duration, and a distance. A particular flight on a particular date is called a departure. A departure is carried by an aircraft. Departures have passengers booked onto them. The booking includes the price of the ticket, the travel class, and the seat number. A passenger is described by his or her name, address, and amount of accumulated frequent flyer miles. A departure is furthermore flown by one or more pilots. A pilot has an employee number, name, address, salary, and license number. A pilot is qualified to fly a number of plane types. For each plane type he or she is qualified, his or her accumulated flight hours on that type is recorded. A technician is described by his or

her employee number, name, address, salary, team he or she is working on, and the type of planes he or she is qualified to service.

2.2 Application Operations

Airline reservation application supports the following type of queries, update and on-going report generation operations.

1. Add a new aircraft. Delete an aircraft.
2. Add/delete/modify a booking.
3. Add/delete/modify a pilot.
4. Add/delete/modify a technician.
5. List all passengers booked on a departure.
6. List the names and addresses of all pilots who are qualified to operate a departure. Assign a pilot-crew for a departure.
7. List the name and address of all technicians who are qualified to service an aircraft.
8. List all the plane types a passenger is booked on to fly in.
9. Assign a technician-crew for a departure.

2.3 Application Design

Figure 1 shows the relationship between the classes, which have been used in Airline reservation application.

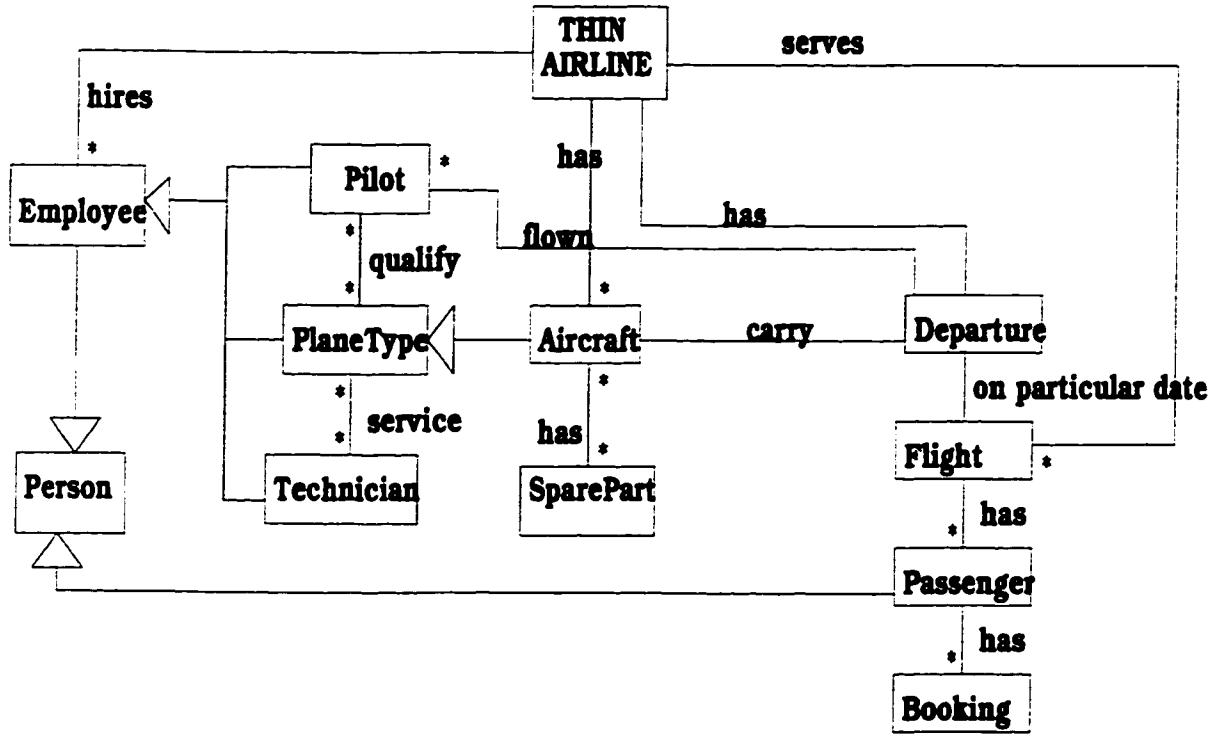

Figure 1: **Airline Reservation**

Chapter 3

General Instructions for Building An Application

An ObjectStore application is a C++ program that uses ObjectStore. We perform the following steps to develop an application with ObjectStore. These instructions assume that we are using a makefile to build our application.

1. Modify the source.

Modify your application source code to make ObjectStore API calls. See the ObjectStore C++ API User Guide for information about using ObjectStore APIs. Note that you must modify our makefile to find ObjectStore header files. (See section 3.2 for more information.)

2. Create the schema source file.(See section 3.3 for details)

The schema source file is a C++ file with a specified format used as input to the schema generator (ossg). The schema source file includes the files that define:

- Classes that have instances stored by the application in persistent memory.
- Classes that have instances read by the application from persistent memory. We can include the type itself, or the base types of the class.
- Classes that appear in library interface query strings or index paths.

3. Generate schema with **ossg**.

Modify your makefile to run the ObjectStore schema generator (ossg). The input for this step includes

- Schema source file
- ObjectStore library schemas The output from this step is the:
  - Application schema database
  - Application schema source file. If you are using Visual C++, the output is an object file referred to as the application schema object file. This file records the location of the application schema database along with the names of the application's virtual function dispatch tables, the names of discriminant functions, and the definitions for any get_os_typespec() member functions.

4. Compile the application schema source file.

Make sure your makefile enables you to compile the application schema source file. This creates the application schema object file.

5. Link.

Your makefile should contain likes to the following (to create the executable):

- Application object files
- Application schema object file
- Application libraries
- ObjectStore libraries
- System libraries

3.1 Overview of Source Files

We build an ObjectStore application from the following source files:

- Source files that contain code that we write.
- Header files, provided with ObjectStore, that we have included in our source files.
- Header files that we write to define our persistent C++ classes.
- Schema source file that specifies our persistent classes for the schema generator. We create this file according to ObjectStore rules.

Building an ObjectStore application requires the generation of schema information. This is information about the classes of objects the application stores in or reads from persistent memory. ObjectStore generates schema information according to the schema source file that we create. (See section 3.4 for more information.)

3.2 ObjectStore Header Files

ObjectStore provides header files that we must include in our source code. The ObjectStore features we use, determine which header files to include. Be sure to include the files in the given order. We must always include ostore/ostore.hh.

If We Use This Feature	Include These Header Files
Any ObjectStore feature	**ostore/ostore.hh**
Collection	**ostore/ostore.hh, ostore/coll.hh**
Compactor	**ostore/ostore.hh, ostore/compact.hh**
Database utilities	**ostore/ostore.hh, ostore/dbutil.hh**
Metaobject protocol	**ostore/ostore.hh, ostore/mop.hh**
Relationships	**ostore/ostore.hh, ostore/coll.hh, ostore/relat.hh**
Schema evolution	**ostore/ostore.hh, ostore/manschem.hh, ostore/schmevol.hh**

3.3 Determining Types in a Schema

The schema source file determines the types that are in a schema. In the schema source file, we use a macro to mark the types to be included in the schema.

The types that we mark are the types on which we can perform persistent new(). See also section 3.5.1 for more information.

Which Types to Mark

As a minimum, we should mark the following types:

- Classes on which the application might perform persistent new() to create a direct instance of the class.
- Classes that have instances read by the application from persistent memory. We can directly mark the type itself or mark the base types of the class.
- Classes appearing in a query string or in an index path.

Omissions in the schema source file can cause run-time errors. For example, we might try to persistently store a type that we did not mark or that is not in the schema. It is not necessary to mark ObjectStore classes except for collection classes [2].

3.4 Creating Schema Source Files

The schema source file specifies the C++ classes that our code reads from or writes to persistent memory. We create the schema source file according to a specified format. The schema source file can contain only valid C++ code. It is a good practice to compile our schema source file to verify that it is compilable, but compilation is not required.

When we run the schema generator, we specify the name of the schema source file. Our executable program does not include the schema source file. The schema source file is only for input to the schema generator (ossg).

Schema Source File Format

Before creating the schema source file, we determine the types in the application that we are going to mark in the schema source file. Use the tables in 3.3, then follow these steps to create a schema source file.

1. Create a text file.

2. In the text file, specify **#include** to include ObjectStore header files required by the features that will be used . The required order is in 3.2.

3. Specify #include to include the **manschem.hh** file provided with ObjectStore.

4. Specify #include to include the files that define the following types:

- The types that we are going to mark.
- Any types embedded in types that we are going to mark.

We are not required to include the definitions for all reachable types. However, not including the class definition for a type that is in the application schema causes that:

- ObjectStore cannot check the class for compatibility with a database class definition (if one exists).
- ObjectStore cannot make virtual function table pointers (**vftbls**) and discriminant functions available for the class.

For efficiency, create header files which contain only class definitions and include the header files in the schema source file. This speeds schema generation because there is nothing extra for the schema generator to examine.

5. Mark certain included types with a call to the macro **OS_MARK_SCHEMA_TYPE**. Each call is on its own line and has the format:

OS_MARK_SCHEMA_TYPE(type-name);

For additional information about OS_MARK_SCHEMA_TYPE() and OS_MARK_SCHEMA_TYPESPEC() chapter 4, System-Supplied Macros of [5].

6. Mark parameterised types with multiple arguments with a call to the macro OS_MARK_SCHEMA_TYPESPEC. This macro is similar to OS_MARK_SCHEMA_TYPE in syntax and function. except that we must enclose the type and its arguments in parentheses. Each call is on its own line and has the format:

OS_MARK_SCHEMA_TYPESPEC((type-name< x,y >));

7. Save the schema source file.
Here is the schema source file for the **Airline Reservation** application which was described in section 2. As we can see, types can be marked without putting them in any dummy function.

schema.cc:

```
#include <ostore/ostore.hh >
#include <ostore/manschem.hh >
#include <ostore/coll.hh >
#include "smallclass.hh"
OS_MARK_SCHEMA_TYPE(PlaneType);
OS_MARK_SCHEMA_TYPE(Aircraft);
...
OS_MARK_SCHEMA_TYPE(os_Set<PlaneType* >);
OS_MARK_SCHEMA_TYPE(os_Set<Aircraft* >);
...
```

3.5 Overview of Schema Generation

A schema contains information about a set of classes. ObjectStore defines the following kinds of schemas:

- Application schemas

- Component schemas
- Library schemas
- Compilation schemas
- Database schemas

We use the ObjectStore schema generator to generate application, component, library, and compilation schemas. ObjectStore creates database schemas.

ObjectStore stores each application, component, library, and compilation schema in its own ObjectStore database. Database schemas are stored in the associated database or in a separate database that we specify. Each schema database must be accessible to an ObjectStore server.

Application Schemas

An application schema contains descriptions of

- Classes the application stores in or reads from persistent memory
- Classes that a library in the application links with, stores in or reads from persistent memory

ObjectStore uses the application schema during run time to:

- determine the layout of objects being transferred between the database and the application
- validate the database schema to ensure that the application schema matches the database schema

For simple applications, we can use a single invocation of **ossg** to generate an application schema. In more complex applications, we might need to use library schemas to store schema information before constructing the application schema.

Component Schemas

A component schema is a type of application schema that can be loaded and unloaded dynamically at run time. Typically, a component schema is also a self-contained schema associated with a DLL.

Library Schemas

ObjectStore must have all the information about persistently used type in a application. If the application uses a library that stores or retrieves persistent data, and the library does not supply its own component schema, use the schema generator to create a library schema for that library. This library schema should be used when creating application schema in order to have all the persistently used types in the application.

In addition to the library schemas we create, ObjectStore provides library schemas for its libraries that use persistent data. If we link our application with an ObjectStore library that has a library schema, we must specify library schema when generating the application schema.

Compilation Schemas

Compilation schema works like library schema. It contains information about the application's persistent types, but does not contain information about any persistent types used by any libraries that the application links with.

Database Schemas

ObjectStore creates a database schema from the application and component schemas of all applications that allocate objects in the database. The database schema consists of the definitions of all types of objects that have ever been stored, or are expected to be stored in the database.

Keeping Database Schemas and Application Schemas Compatible

Compatibility means that if a class exists in both schemas

- its data members must have identical definitions and ordering in each schema,
- both class definitions must either define at least one virtual function, or virtual functions must be absent from both definitions.

3.5.1 Generating an Application Schema

Input

To generate an application schema, specify your schema source file as input to **ossg** . If you are linking with libraries that have library schemas, you must also specify those library schemas as input to **ossg** .

Output

The output from **ossg** is

- **Application schema database** The schema generator creates the application schema and stores it in an ObjectStore database.
- **Intermediate schema source file** This file records the location of the application schema database along with the names of the application's virtual function dispatch tables, the names of discriminant functions, and the definitions for any **get_os_typespec()** member functions.

We must compile this file and then link the resulting object file with our application. See section 3.6 for details.

Invoking ossg to generate an application schema

The **ossg** command syntax for generating application, library, and compilation schemas appears below. More options for **ossg** is given in [5]

ossg [neutralizer_options] [additional_app_options] [-assf app_schema_source_file -asof app_schema_object_file.obj] -asdb app_schema_db schema_source_file [lib_schema_db1 ... lib_schema_dbn]

-assf app_schema_source_file or -asof app_schema_object_file.o	Specifies the name of the application schema source file or application schema object file to be produced by ossg. For all compilers except Visual C++, the schema generator produces a source file that we must compile. When we use Visual C++, the schema generator directly produces the object file. Required. No default.
-asdb app_schema_db.adb	Specifies the name of the application schema database to be produced by ossg. If the schema database exists and is compatible with the type information in the input files, the database is not modified. This pathname must be local to a host running an ObjectStore Server. The pathname should have the extension . If we want to specify an existing application schema database with ossg, the application schema must have .adb as its extension. Required. No default.
schema_source_file	Specifies the C++ source file that designates all the types we want to include in the schema. It should include all classes that the application uses in a persistent context. No default.
lib_schema.ldb ...	Specifies the pathname of a library schema database. The name must end in .ldb. This can be an ObjectStore-provided library schema or a library schema that we created with ossg. The schema generator reads schema information from the library schema database specified and modifies the application schema database to include the library schema information. We can specify zero or more library schema databases. Optional. The default is that library schemas are not included.

Here is an example of **ossg** :

ossg -assf $(APP_SCHEMA_SRC) -asdb $(APP_SCHEMA_DB) $(OSSCHEMA_FLAGS) $(CPPFLAGS) $(SCHEMA_SRC) $(LIB_SCHEMA_DBS)

where,
APP_SCHEMA_SRC = osschema.cc
APP_SCHEMA_DB= $(OS_SCHEMA_DB_DIR) airline.adb
OSSCHEMA_FLAGS=
CPPFLAGS=-I$(OS_ROOTDIR)/include
SCHEMA_SRC= schema.cc
LIB_SCHEMA_DBS= $(OS_ROOTDIR)/lib/liboscol.ldb
$(OS_ROOTDIR)/lib/libosqry.ldb

Specifying ObjectStore Library Schemas

ObjectStore provides library schemas for the ObjectStore libraries that store or retrieve persistent data. If we are linking our application with an ObjectStore library that has a schema, we must specify that library schema when we generate the application schema. The following table shows how to specify an ObjectStore, given a library schema. When we specify a library schema, we must always specify the full path. For example, $(OS_ROOTDIR)/lib/liboscol.ldb on UNIX.

For This Feature	Specify This Library Schema	Link with This Library
Collections	liboscol.ldb	-loscol
Compactor	liboscmp.ldb	-loscmp
Queries	libosqry.ldb	-losqry
Schema evolution	libosse.ldb libosqry.ldb liboscol.ldb	-losse -losqry -loscol

3.6 Compiling and Linking Programs

UNIX

This section provides information for compiling and linking ObjectStore applications on UNIX platforms. The material refers to all UNIX platforms that support ObjectStore C++ interface Release 5.1.

3.6.1 Linking with ObjectStore Libraries

ObjectStore includes libraries that must be linked when building the applications. Libraries allow multiple programs to share code without redundantly compiling the source. Applications use libraries by specifying them at link time.

Requirement

We always link with the **libos** library. If we are using full ObjectStore, *$OS_ROOTDIR/lib/libos* is used; if we are using ObjectStore/Single, *$OS_ROOTDIR/libsngl/libos* is used. For definitions of *single* and *full* objectstore, see chapter one in [2].

We must also link with either the **libosthr** or **libosths** libraries. Additional ObjectStore libraries are linked according to the features we use in our code, as shown in the following table. If there are more than one library, they have to be specified in the order given.

If Our Application Uses This Feature	Link with These Libraries in This Order
Any ObjectStore feature	libos {libosthr or libosths}
Collections	liboscol libos {libosthr or libosths}
Queries and indexes	libosqry liboscol libos {libosthr or libosths}
Relationships	liboscol libos {libosthr or libosths}
C run-time library	libos {libosthr or libosths}

C++ run-time library

We must link **libos** before **libC**, the C++ run-time library. On some platforms it is possible a wrong order is obtained when shared libraries are used. Linking first with **libos** prevents this problem.

Specifying libraries

Use the **-l** (lowercase **l** as in link) option to pass library names to the linker. When we specify an ObjectStore library, do not include the **lib** portion of the library name, it means instead of using **libos** use **-los** . For example, in Airline Reservation application, we use collections, relationships and queries as the following:

$(CCC) $(TFLAGS) -o generate generate.o implement.o $(APP_SCHEMA_OBJ) $(LDFLAGS) $(LDLIBS)
which is:

```
CCC= CC
OSSCHEMA_FLAGS=
LIBOSC_LD =
LIBOSTHR_LD=-losth
TFLAGS= -DPTHREADS -g -KPIC -pta -vdelx -mt $(DEBUG_OPT)
LDFLAGS= -L$(OS_ROOTDIR)/lib $(OS_LINK_FLAGS)
LDLIBS= -losqry -losmop -loscol -los -losth $(LIBOSVND) \
        $(LIBOSTHR_LD) $(LIBOSC_LD)
APP_SCHEMA_OBJ= osschema.o
```

Note that the **-L$(OS_ROOTDIR)/lib** option begins with an uppercase L as in Library.

3.6.2 Sun C++ Compiler Options

Sun C++ 4.0 has a compile-time option,**-pto** , that creates all template instantiations in the current object file. Do not use this option when developing ObjectStore applications, because it makes everything (including **vtbls**) static. Since the 3.7 are static, ObjectStore cannot get at them and gets the wrong **vtbl** , which leads to an error.

-vdelx compiler option

When we are using SPARC Pro-Compiler C++, we must always specify the **-vdelx** compiler option.

The **-vdelx** option to CC generates the correct calling sequence for persistent vector deletes. For example:

```
delete [] persistent_array;
```

Without **-vdelx** , the compiler generates a direct call to the Sun vector delete routine. This routine returns an error message indicating that it did not allocate the array:

```
error: delete [] does not correspond to any `new'
```

If we use CC to link (by means of **ld**), then the linker receives the correct libraries.

On Solaris 2.x systems, we must compile (and link) ObjectStore applications with real thread support. Specify **-losthr** on the link line to supply the proper library for real thread support. The **libosths** library, a stub threads library, can also be used with Solaris 2.x systems. Using this library can result in higher performance in some circumstances because it turns off thread locking.

3.6.3 Solaris 2 Linking

ObjectStore supports linking with and without threads. This means there are two ObjectStore thread support libraries on Solaris 2 - libosths and libosthr.

- Use the libosthr library in linking an application that links with **-mt** (or **-lthread**).
- Use the libosths library in linking an application that does not link with **-mt** (or **-lthread**).

On Solaris 2.x, we must always specify the **-mt** compiler option on the CC command line. This is required for a successful compilation.

Here is an example of a program (foo) compiled and linked in both configurations.

Linking with threads

with threads:

CC -mt -I$(OS_ROOTDIR)/include $(CCFLAGS) -o **foo** -L$(OS_ROOTDIR)/lib -los -losthr

Note that to link an application to use threads.

- Use **-mt** (or **-lthread**) on the link line.
- Link with **libosthr** .

Linking without threads

without threads:

CC -I$(OS_ROOTDIR)/include $(CCFLAGS) -o **foo** -L$(OS_ROOTDIR)/lib -los -losths

Note that to link an application that does not use threads,

- Do not use **-mt** on the link line.
- Do not explicitly place **-lthread** in the link line.
- Link with **libosths** .

3.6.4 Sample Makefile Template

In an ObjectStore makefile, in the **LDLIBS** line, we must specify each library with which we are linking. Then, in the line of the makefile where **ossg** generates the application schema, we must specify the library schemas needed by our application schema. For each library schema that is specified in the **ossg** command line, we must specify the corresponding library in the **LDLIBS** line. Note that the reverse is not

true. For each library that we specify in the **LDLIBS** line, we do not necessarily specify a library schema in the **ossg** command line. This is because every library does not necessarily have a library schema. Only those libraries that store or retrieve persistent data have associated library schemas.

Application schema database

In makefiles, we should not specify an existing ObjectStore database as the application schema database.
Doing so can corrupt our build process if the server log has not been propagated to the database.

Makefile template

```
APPLICATION_SCHEMA_PATH= app-schema-db

/* we only assign the name to application schema */
/* database app-schema-db, the body will be generated by ObjectStore.*/

LDFLAGS= -L$(OS_ROOTDIR)/lib $(OS_LINK_FLAGS)
LDLIBS= -losqry -losmop -loscol -los -losth  [other libraries]
SOURCES = .cc  files
OBJECTS = .o  files
EXECUTABLES =  executables
CCC=CC
all: ${EXECUTABLES}
executable: $(OBJECTS) os_schema.o
      $(CCC) -o executable $(OBJECTS) os_schema.o   \
      $(LDLIBS)  $(OS_POSTLINK) executable
.o  files: .cc files
      ${CCC} $(CPPFLAGS) -c .cc  files
os_schema.o: os_schema.cc
      $(CCC) $(CPPFLAGS) -c os_schema.cc
```

```
os_schema.cc: schema.cc

        ossg -assf os_schema.cc -asdb $(APPLICATION_SCHEMA_PATH) \
              $(CPPFLAGS) schema.cc $(OS_ROOTDIR)/lib/libosqry.ldb \
              $(OS_ROOTDIR)/lib/liboscol.ldb
clean:
        osrm -f ${APPLICATION_SCHEMA_PATH}
        rm -f ${EXECUTABLES} ${OBJECTS} os_schema.*
```

For more information see chapter 4 in [5].

3.7 Working with Virtual Function Table (VTBL) Pointers

There is a special case in which ObjectStore needs to know, at run time, the locations of information in our application program's executable: **Virtual function tables (vtbls)**

When we declare a class to have virtual functions or, in some cases, to have virtual base classes, it acquires an invisible data member, a virtual function table pointer. (Virtual function table is usually abbreviated as **vtbl** , pronounced veetable. On some platforms vtbls are called **vfts** for virtual function tables. Vtbl and vft indicate the same thing.) The vtbl points to a table of function pointers that the application uses to dispatch calls to virtual functions. The C++ compiler arranges for the correct function pointers to be placed in the virtual function table.

Persistent storage

When we persistently store an object belonging to a class with virtual functions, ObjectStore cannot store the vtbl pointer literally, since it is a pointer to the text or data segment of the current executable - a tra ient pointer. When the same program runs another time, or a different program opens the database, the vtbl might have a different location.

3.7.1 Missing VTBLs

Depending on our platform, missing vtbls can cause errors at compile time or at run time. It is better to find such errors at compile time. By default **ossg** reports these errors at compile time.

Symbols missing when linking ObjectStore applications

Sometimes when linking, particularly with optimisations enabled, we are told that various symbols required by the application schema object file are missing. These symbols on non-Windows platforms (including OS/2) begin with _vtbl or _vft, or end with _vtbl.

This happens because ObjectStore needs to access the virtual function tables (vfts) for some classes, and the C++ compiler does not recognize these tables as being needed. The easiest way to get these symbols is to add a non-static dummy function such as the following:

```
void foo_force_vfts(void*){
   force_vfts(new A);
   force_vfts(new B);
}
```

Creating instances of a class causes the class's vfts, as well as those of bases having out-of-line default constructors, to be created in this file.

Abstract classes

If one of our classes is abstract, a variant of the above approach is needed, since we cannot allocate an abstract class. We can provide an out-of-line constructor for the class, or we can allocate a non abstract derived class in such a way that in-line constructors are used for the abstract class. For example, if the original class definitions were:

```
class A {
     virtual void foo() = 0;
      };
```

```
class B : public A {
     virtual void foo();
      };
```

,then class A might be missing its vft. However, an unoptimized new B would call A's in-line default constructor, which would reference the vft for A. But if class B had an out-of-line constructor, this would not work. Then it would be easiest to make an out-of-line constructor for A:

```
class A {
      virtual void foo() = 0;
      A(){}
      friend void force_vfts(void*);
      A(void*);
};
class B : public A {
    virtual void foo();
  };
```

and define A::A(void*){} in some file.

Instantiating Collection

If we are using a parameterized collection class, we must instantiate the other collection classes because they have casting to each other. A work around is to declare the following and link them. For example:

```
void foo_force_vfts(void*) {
force_vfts(new os_Array<missing-type*>);
force_vfts(new os_Bag<missing-type*>);
force_vfts(new os_Collection<missing-type*>);
force_vfts(new os_List<missing-type*>);
force_vfts(new os_Set<missing-type*>);
force_vfts(new missing-type*);
....
}
```

In our Airline Reservation example (as described in chapter 2 we have:

```
/* A function to force vftable inclusion for collections templates */

void foo_force_vfts(void*) {
force_vfts(new os_Array<Flight*>);
force_vfts(new os_Bag<Flight*>);
force_vfts(new os_Collection<Flight*>);
force_vfts(new os_List<Flight*>);
force_vfts(new os_Set<Flight*>);
force_vfts(new Flight*);
....
}
```

3.7.2 Run-Time Errors from Missing VTBLs

On some platforms (without weak symbol support), we find out about the missing **vtbls** at link time. The vtbls are marked in the schema output file, but are not marked in the application. This is frequently the case for parameterised collections classes (os_Set, os_List, and so on).

Sometimes an executable does not have vtbls for all classes with virtual functions in the schema. When a **vtbl** pointer for a class is not available, ObjectStore fills in the **vtbl** pointer for the class's instance with a special **vtbl** that signals an error when any of the virtual functions is called.

No constructor

Missing vtbls can occur when our application calls a virtual function on an instance of a class for which no constructor call appears in the source. Since a call to the class's constructor does not appear in the source, the linker does not recognise the class as being used and does not link in its implementation. But an ObjectStore application can use a class whose constructor never calls it by reading an instance of the class from a database. To avoid this situation, put a call to the class constructor

inside a dummy function that is never called.

In our **Airline Reservation** application, for example, class **Employee** is an abstract class. By separating class declarations file from their methods implementation, we will not have missing **vtbl** error anymore.

Class not in schema

Missing vtbls can also occur when the class is not included in the application's schema, either because its definition was not included in the source or because the class was only reachable from explicitly marked classes by means of void* pointers. In this case, the solution is to include a definition of the class, or to explicitly mark it with **OS_MARK_SCHEMA_TYPE()** .

Marking Employee class in schema source file is looks like the following:

```
OS_MARK_SCHEMA_TYPE(Employee);
```

Chapter 4

Environment for ObjectStore

We want to show you the different features of ObjectStore application including:

- How we can have database operations, like create a database, destroy, open, and close
- Create objects in the database
- Retrieve objects through database roots.

Note that we always assumes that we have an existing C++ program and we want to store objects in a database and retrieve them later.

We use our Airline reservation application as an example to describe more clearly each part.

4.1 Include Files

To convert a C++ program to an ObjectStore application first add the ObjectStore header files. You can find about ObjectStore header files in 3.2. Our sample application includes :

```
#include <ostore/ostore.hh >
#include <ostore/coll.hh >
```

#include "smallclass.hh"
and "smallclass.hh" includes all class definitions.

4.2 ObjectStore Initialisation

As the first statement to be executed in any ObjectStore program must be the static member function **objectstore::initialize()**. If the application uses the collection class library supplied by ObjectStore. it must call **os_collection::initialize()** as well.

Our Airline reservation application contains these two calls in the beginning of main() function:

```
main(int argc, char** argv)
 {
  objectstore::initialize();
  os_collection::initialize();
  ....
  }
```

4.3 Application Schema Source File

By creating application schema source file. ObjectStore will have all nessecary schema information in its schema database. see section 3.4 for more information.

4.4 ObjectStore Database operators

Before we access persistent memory, we must set the stage by performing a few other operations:

- A database must be created or opened.
- A transaction must be started.
- A database root must be retrieved or created.

All these operations are described in this section.

4.4.1 Creating a Database

We create a database by calling the static member function **os_database::create()** giving the name of the database as an argument:

os_database *db= os_database::create("/oodb/airline/thinair.db");

This function also opens a newly created database. We can call this function from either inside or outside a transaction, although, in general, it is best to create databases outside a transaction.

The create function returns a pointer to an object of type **os_database** . We use this pointer as the **this** argument to other non static member functions of the class **os_database**, such as **os_database::close()** (this function is used to close a database).

We can call **os_database::create()** function by more arguments to overwrite the default mode:
os_database *db= os_database::create("/oodb/airline/thinair.db",0666,1,1);

The second argument specifies a protection mode. The defaults mode is **0664**

By specifying a nonzero value (true) as the value of third argument, we direct **os_database::create()** to overwrite any existing database with the same name, instead of signaling an exception. This argument defaults to **0** (false).

Nonzero value of the fourth argument specifies that, the schema information of the newly created database to store in the specified schema database. However if this argument is **0** , schema information is stored in the new database which we are creating.

4.4.2 Destroying Databases

We can destroy a database with the **os_ database::destroy()** function.

void destroy();

This function takes no arguments (other than the **this** argument), and has no return value.

```
os_database* db1;
....
db1->destroy();
```

If the database is open at the time of the call, destroy() closes the database before deleting it. Note that to help ensure program portability, we should call the destroy function from outside any transaction.

4.4.3 Opening Databases

Before we can read or write data, we must open the database in which the data resides. A database is automatically opened when we create it with os_database::create(). If we want to open a previously created database, we use the function os_database::open().

Like **os_database::create()**, we should specify pathname of the database. By setting an optional argument, **create_mode** to nonzero, in case that there is no database by specified pathname, instead of signaling an exception, a new database is created. We can open the database in a read only mode by giving a nonzero integer as the second argument. The default mode is **0** for read/write.

os_database *db= os_database::open("/oodb/airline/thinair.db", 0, 0664);

4.4.4 Closing Databases

We close a database by calling **os_database::close()**.

void close() ;

This makes the database data inaccessible (assuming we have not performed nested opens- see [3] for information about Nested Database). If the call to **os_database::close()** occurs inside a transaction, the database is not actually closed

until the end of the outermost transaction. That is, the data remains accessible until the outermost transaction terminates.

```
os_database *db1;
...
OS_BEGIN_TXN(tx1, 0, os_transaction::update)
...
db1->close();
/* data in db1 remains accessible */
...
OS_END_TXN(tx1);
/* end of transaction, so db1 data is now inaccessible */
```

4.5 Persistent Objects in Database

ObjectStore uses overloaded new operators to create persistent objects:

void* operator new(size_t, os_database*, os_typespec*)

The first argument of type **size_t** is always provided by ObjectStore. The second argument tells ObjectStore where to put the object. The Third argument tells ObjectStore the type of object to create and its storage layout.

To create an array of persistent objects we have to specify an additional argument for **new** operator that determines the number of elements. For example, to create a persistent character string containing 10 symbols we write:

char* pstring = new(db,os_typespec::get_char(),10) char[10];

4.5.1 Using Typespecs

Typespecs, instances of the class **os_typespec** help maintain type safety when you are manipulating database roots. A typespec represents a particular type, such as

char.

ObjectStore provides some special functions for retrieving typespecs for types such as:

```
static os_typespec* get_char();
static os_typespec* get_short();
static os_typespec* get_int();
static os_typespec* get_long();
```

Here is an example in which we create a persistent integer object in a database by calling **os_typespec::get_int()**:

```
int* number = new( db,os_typespec::get_int()) int(0);
```

4.5.2 Typespecs For Classes

For user-defined classes such as **aircraft** class we create an os_typespec as follows:

```
os_typespec* Aircraft_type = new os_typespec("Aircraft");
```

Typespecs should only be allocated transiently: you should not create a typespec with persistent new.

```
aircraft=new(db,Aircraft_type)Aircraft(na,Pma,ms,len,sn,sno,mdate,Ama,f);
```

Instead of creating typespec objects in each application we can declare a static member function with the name **get_os_typespec** for each user defined class. The ObjectStore schema generator automatically supplies a body for this function. The **get_os_typespec** function is particularly useful when we want to define a parameterised typespec for class templates. The get_os_typespec function also highly simplifies the management of os_typespecs.

For example, the class declaration of **aircraft** contains the following static function declaration:

```
class Aircraft:public PlaneType
{
private:
    int          serialNo;
    int          manufacturdate;
    char *       Amanufacturer;
    int          flighthours;
    ....

public:
/* Define layout for ObjectStore */
        static  os_typespec*  get_os_typespec();
        ....
};
```

Using this class declaration, we create a new aircraft using:

```
Aircraft* an_aricraft = new(db,Aircraft_type) Aircraft("jet","Homa",6000,
4000,50,1,20,"HH",35);
```

4.5.3 Clustering

Sometimes we want to specify more precisely where we want to allocate the storage for our **new** object. When an ObjectStore database is created, it contains two segments, one the schema segment and another the default segment. The schema segment contains schema information used internally by ObjectStore and all the database roots. The default segment contains the persistent objects we create by giving a database pointer as the first argument of the overloaded **new** operator.

The class **os_segment** provides a useful function, **os_segment::of()**, that allows us to determine the segment in which a given object resides.

Here is an example which uses a version of the **new** operator that allows us to determine the place of our new object:

void* operator new(size_t, os_segment*, os_typespec*)

This operator is used in the following example. which shows the body of our constructor for passenger:

```
Passenger::Passenger(char* na, char* ad,...)
{
long len = strlen(na) + 1;
name = new(os_segment::of(this),os_typespec::get_char(),len)char[len];
strcpy(name, na);
len = strlen(ad) + 1;
address = new(os_segment::of(this),os_typespec::get_char(),len)char[len];
strcpy(address, ad);
.....
}
```

4.5.4 Transient Database

By passing pointer to a transient database in ObjectStore overloaded **new** operator, we can create transient object using exactly same code.

We get a pointer to the transient database by calling the os_database::get_transient_database() function.

4.6 Database Entry Points and Data Retrieval

The only ways you can retrieve your stored object from persistent database is either by *Navigation* or *Query*. But neither of them can be done unless you have an **entry point** to the object. Once you have retrieved an entry point object, all objects reachable from it, will be automatically retrieved when needed.

An entry point is an object which is given a persistent name using a root. A root is an instance of the ObjectStore class **os_database_root** and its aim is to associate an object witn a name.

4.6.1 Creating Database Roots

The first step is to create a root, then associate an object, which we want to be an entry point with a name and finally retrieve our object using the entry point.

Here is how we create a root for our sample application:

```
os_database* db =....open(...);
os_Set<Passenger *> * Passenger_extent;
/* Create the passenger extent collection as a persistent object */
Passenger_extent = &os_Set<Passenger *>::create(db);
/*Define the root "passenger" */
os_database_root* aRoot = db->create_root("passenger");
```

As we see, the root is created in current persistent database that the entry point is stored. Transient database causes the error, **err_database_not_open**. If we try to create a root with a name that already exists in the database, the **err_root_exists** is signaled.

The second step was to associate an object, which we want to be an entry point with a name. The name which we give to create_root() function will be assigned to the entry point by calling set_value() function:

```
/* Associate the root with the passenger extent entry point */
aRoot->set_value(Passenger_extent);
```

In our Airline reservation application we use **extents** as entry points. For example, Passenger_extent is a collection (set) that contains pointers to all objects of a class. Through this entry point we can reach all objects whose pointers have been inserted into this collection.

4.6.2 Retrieving Entry Points

We retrieve the root by giving its name to the **find_root()** function. To access the value of this root (entry point object) use **get_value()** function as illustrated in the following code extract:

```
os_database* db =....open(...);
os_Set<Passenger *> * Passenger_extent;
/* Find the "passenger" root*/
 os_database_root * a_root = db->find_root("passenger");
if(a_root)
   Passenger_extent= (os_List<Passenger *> *)a_root->get_value();
...
```

The **os_database::find_root()** function takes a root name as an argument and returns a pointer to the root if the root can be found in the database; otherwise it returns a 0.

The function **os_database_root::get_value()** returns a void*, a pointer to the entry point object associated with the specified root. Since the returned value is typed as void*, a cast is usually required when retrieving it.

Now we can navigate through *Passenger_extent* entry point object and get all passengers information that will be described in 7.5:

```
/* cursor on passenger extent */
os_Cursor<Passenger*> p(*Passenger_extent);
/* iterate cursor */
for (Passenger* aPass=p.first(); aPass; aPass=p.next())
    /* display each passenger */
     aPass->display();
```

In Airline Reservation application, we declare **extent** as static member and all accesses to extent is being done in a static member function, access_extent():

```
void Passenger::access_extent(os_database *db)
{
  os_database_root * a_root = db->find_root("passenger");
  if(a_root)
      extent= (os_List<Passenger *> *)a_root->get_value();
  else
    {
```

```
    extent=&os_List<Passenger *>::create(db);
    db->create_root("passenger")->set_value(extent);
    }
}
```

4.6.3 Type-Safety for Database Roots

If we want to have some type-safety for roots, we can add **os_typespec*** as a second argument to **set_value()** and **get_value()** functions. The type of the entry point should match with the **os_typespec** given in **get_value**. Otherwise an **err_type_mismatch** exception is signaled.
Here is an example:

```
os_database* db;
Passenger* aPassenger;
....
aPassenger = new(db, Passenger::get_os_typespec())
                Passenger("Robin","Forest",...);
os_database_root* aRoot = db->create_root("my_passenger");
aRoot->set_value(aPassenger, Passenger::get_os_typspec());
....
aRoot = db->find_root("my_passenger");
if (aRoot)
 aPassenger =(Passenger*)(aRoot->get_value(Passenger::get_os_typespec()));
```

Chapter 5

Using Transactions

We access to persistent storage in a boundary of a transaction. There are two ways we can specify this boundary:

- **Lexical** transactions: Using transaction macros.
- **Dynamic** transactions: Using members of the class **os_transaction**.

5.1 Using Lexical Transactions

We begin and commit lexical transactions with the following macros:

OS_BEGIN_TXN (identifier,exception**,transaction-type)
OS_END_TXN (identifier)

For example, in our Airline Reservation application:

```
OS_BEGIN_TXN(t1, 0, os_transaction::update)
{
// List of airplane
....
}
OS_END_TXN(t1)
```

The **identifier** argument is used to distinct the transaction from possible other transactions in the same function.

The second argument **exception**** is used in case you want to have exception handling when a transaction aborted. To ignore special handling of retries, you can give a 0 as the second argument (as shown in our example above).

transaction-type is one of the following enumerators, defined in the scope of os_transaction:

os_transaction::update specifies a transaction in which updates to persistent memories are allowed.

os_transaction::read_only specifies a transaction in which any attempt to update persistent memory signals the exception err_write_permission_denied.

os_transaction::abort_only specifies a transaction in which writes to persistent memory is allowed, but the transaction cannot be committed.

5.2 Using Dynamic Transactions

This kind of transaction allows programs to begin a transaction dynamically, and end the transaction in a different function. We begin and commit the transaction using call the member functions **begin()** and **commit()** of the **os_transaction** class.

5.3 Pointers

There are several kinds of pointers in an ObjectStore application.

Pointers to persistent objects are one kind of these pointers. Typically they are invalidated at the end of a transaction. We can make this to not happen by calling the database method **objectstore::retain_persistent_address()**. This function introduce additional overhead and storage to our application.

We can have pointers from persistent memory to transient memory. As we know the transient memory is a temporary storage and will be discarded at the

end of our process. We also know that at the end of a transaction, persistent objects are written to a disk. If some of these persistent objects have a pointer value to the transient database, these pointers are no longer valid. By calling **db->set_null_illegal_pointers** we can ask ObjectStore to set all pointers from objects in database **db** to transient memory to null at the end of a transaction. We have to do this every time we start a new process that accesses this database.

In ObjectStore applications we can access several databases simultaneously. Therefore we may have some pointers between these databases. We named these pointers as **cross database pointers**.

If we want to keep the cross database pointers at the end of our transaction, we should call **db1- >allow_external_pointer(1)** to enable ObjectStore to store a pointer to another database (db2) in an object of database db1.

For more about Transactions see [7] and [8].

Chapter 6

Exception Facility

ObjectStore supports the use of two kinds of exceptions:

- C++ exceptions
- TIX exceptions

C++ exceptions are parts of the C++ language supported by some C++ compilers. On any platform whose compiler supports exceptions, we can signal and catch C++ exceptions in ObjectStore applications.

TIX exceptions are parts of the ObjectStore application-programming interface. ObjectStore API functions sometimes signal predefined TIX exceptions when error conditions arise, and our programs can handle these exceptions with macros and member functions provided by ObjectStore. These predefined exceptions are listed in Appendix B, Predefined TIX Exceptions of [5].

In addition, we can

- Define new TIX exceptions
- Signal predefined or user-defined TIX exceptions
- Handle user-defined TIX exceptions

Exceptions allow us to define alternative return paths to handle errors and unusual cases. Using exceptions, we distinguish two different code paths:

- Normal execution path, which is executed if no error or unusual situation occurs.
- Exception execution path, executed in the case of a recognized and predefined error situation.

With these two paths, we can handle exceptional cases separately and keep the code for the normal execution path simple. Without exceptions, a single return mechanism has to deal with both normal return values and error values.

6.1 Macros

ObjectStore provides a set of macros to catch and handle exceptions:
TIX_HANDLE(identifier)

Declares a region of "protected code" where a specific type of exception will be detected. This is the normal execution path. This region of code is enclosed in brackets to specify the handler's scope: it defines the "exception-catching" block. The **identifier**, given as a parameter is the name of the exception to be handled. This exception as well as any sub exceptions in the hierarchy will be caught.

EXCEPTION

Specifies the block of code that handles the exception. This "exception-handling" block is also enclosed in brackets.

TIX_END_HANDLE

Marks the end of the exception-catching and exception-handling blocks.

Note that we must avoid blanks in writing these macros.

We have used the exception-handling feature in different parts of the Airline Reservation application. For example, we have defined exception-catching and exception-handling blocks when opening the database at the beginning of the application:

```
main(int argc, char** argv)
{
objectstore::initialize();
os_collection::initialize();
if (argc != 2)
{
cout << "Usage: " << argv[0] << " <dbname >" << endl;
exit(1);
}
TIX_HANDLE (err_database_not_found)
{
/* normal execution path */
db = os_database::open(argv[1]);
}
TIX_EXCEPTION
{
/* exception execution path */
cerr << "Database " << argv[1] << " not found. Create it ..." << endl;
db = os_database::create(argv[1]);
generate();
cout << endl << " generate OK" << endl;
}
TIX_END_HANDLE
/* end of exception handling */
....
db- >close();
}
```

We catch the predefined exception **err_database_not_found** and handle the exception by creating the database and calling **generate()** function to initialize the new database with some objects.

In Addition to the ObjectStore's exception we also can define our own exceptions.

We haven't used user-defined exception in our application, if you are interested see Appendix A, of [5].

Chapter 7

Collections

Collections contain set of pointers to objects, rather than the objects themselves. We can have collections both in transient and persistent memory, depending on the needs of our application.

ObjectStore has a set of collection classes like *array, set, bag,* and *list* that they inherit from the class **os_collection**. Some of the classes like arrays and lists can have duplicate elements and some like sets can not. Some like lists and arrays have ordered element and some like sets do not. Lists and arrays are same in ObjectStore, except that arrays can have null values but list can not.

7.1 Collections Requirements

In order to use ObjectStore collection classes include the **ostore/coll.hh** file after including the basic ObjectStore classes:

```
#include <ostore/ostore.hh >
#include <ostore/coll.hh >
```

We also must call the initialize method before we can use collections:

```
os_collection::initialize();
```

As we mentioned in 3.6 we also have to update our make file.

7.2 Creating Collections

After we choose the kind of collections we need, we use **create()** method to create each collection. **Create()** method is a static method of os_Set<T>, where T stands for the type of the elements of the collection. It creates an empty set in the database.

As we said, collections are classes in ObjectStore. We can have objects of collections or classes that have collections as their data members. In our Airline Reservation application, we have created a persistent collection, a set that contains pointers to all passenger objects in our database. We have called this set **Passenger_extent**:

```
os_database* db:
os_Set<Passenger* >& Passenger_extent = os_Set<Passenger* >::create(db);
```

The only required argument for the **create()** method is the name of the database. It returns a reference to the generated instance.

Collections can be defined as static multivalue data members of objects. We declare the collection as static member because we want it to be shared by all objects of the class and not part of each objects of the class.

In our application, class *Aircraft* contains a collection, called **Aircraft_extent**.

```
class Aircraft:public PlaneType
 {
public:
        static        os_Set<Aircraft *> *Aircraft_extent;
        os_backptr    bkptr;
private:
       int            serialNo;
       char           * manufacturdate;
```

```
        char            * Amanufacturer;
        int             flighthours;
        ....
};
```

Note that the static member must be initialized before we can access it. In order to be able to access it we should define a root. We described how to create a root in 4.6.1.

7.3 Inserting Collection Elements

The insertion is used by calling **insert()** method:

void insert(const E) ;

In our sample application, we have inserted any newly created object into the *extent* collection. So, first we make sure we have an entry point to this collection by calling *access_extent()*.

```
SparePart::SparePart(int seno, char *sp, int pr)
{
long lent= strlen(sp) +1;
....
if (!extent)
access_extent(os_database::of(this));
TIX_HANDLE(err_not_assigned)
{
 extent->insert(this);
 ...
}
}
```

Because *extent* has been defined as a **set** , there is no order in it. If we want to have order in which objects are created, we should declare *extent* to be a **list**.

Array Collections

Ordered collections such as lists or arrays have additional insertion methods that allow us to control the position of the new element, such as *insert_first()* and *insert_last()*. The insert methods *insert_before()* and *insert_after()* allow us to place an object relative to the position of a cursor.

The create() method for array creations takes more arguments like initial cardinality that determines the number of slots available (3 in following example). All other parameters are default values.

```
os_Array<Kinds* >& KindArray =
os_Array<kinds* >::create(db,0,0,0,os_collection::dont_associate_policy,3,0);
```

7.4 Removing Collection Elements

Remove() method is used to delete an element from a collection. In our sample application class Passenger has a method called *deleteBooking()* which removes a booking from the set of the passenger's bookings.

```
void Passenger::deleteBooking(Booking *bo)
{
/* find that this booking is already in the passenger's bookingset or not, if it is not
say error */
bookingSet.remove(bo);
/* if there is no other passenger's that have booked for this booking */
/* may be we can delete it from extent of booking in the Booking class */
...
delete bo;
}
```

There are various kinds of remove methods based on the type of the collections. We can use cursor positions or a numerical indexes to specify an element to be removed.

Numerical indexes can be used only with ordered collections.

7.5 Navigating Collections using Cursors

The ObjectStore class library provides cursor classes. Cursors are associated with collections. They can be moved across the collections to access each elements of collections. Whenever we attach a cursor to a collection, the type of the cursor and the type of the collection elements should match.

The following code of our sample application displays all the elements of a collection *parts* using a cursor *sparecur*.

```
void Aircraft::display()
{
os_Set<SparePart * > parts ;
...
if(sparepartpntr)
parts=sparepartpntr;
os_Cursor<SparePart * > sparecur(parts);
// Attach the cursor sparecur to the os_Set<SparePart * > sparepartpntr
for(SparePart * sp= sparecur.first();sparecur.more(); sp=sparecur.next())
if(sp)
sp- >display();
cout<<endl;
}
```

os_Cursor::first()

The program has a loop. The traversal is performed with a loop. The initialization part of the loop header is an assignment involving a call to the member function **os_Cursor::first()**:

```
p = c.first()
```

This function positions the cursor at the collection's first element, and returns that element. If there is no first element, since the collection is empty, **first()** makes the cursor null and returns 0.

os_Cursor::next()

The incrementation part of the loop header is an assignment involving a call to the member function **os_Cursor::next()**:

```
p = c.next()
```

It positions the cursor at the collection's next element, and returns that element. If there is no next element, next() makes the cursor null and returns 0.

os_Cursor::more()

The loop's condition is a call to the member function **os_Cursor::more()**:

```
c.more()
```

This function returns a nonzero 32-bit integer (true) when the cursor is still positioned at some element of the collection, and 0 (false) when it is null.

After next() is applied to the collection's last element, the cursor becomes null, and more() then returns false, terminating the loop.

Chapter 8

Queries and Indexes

Among the database services provided by ObjectStore is support for query processing. A query facility with adequate performance must go beyond support for linear searches. So ObjectStore provides a query optimizer, which formulates efficient retrieval strategies, minimizing the number of objects examined in response to a query. The query facilities are used from within C++ programs, and they treat persistent and non-persistent data in an entirely uniform manner.

8.1 Simple Queries

In Airline reservation application, if we want to find a passenger's information, instead of iterating through Passenger::extent looking for an entry with name "Bita", we let ObjectStore do the work by using **os_Collection::query** facility to find the right passenger object in our database:

```
os_Set<Passenger* >& result = Passenger::extent- >query (
"Passenger*","strcmp(name,"Bita")==0");
```

Calls to the function can take the form:

collection-expression.query(element-type-name,query-string, schema-database)

collection-expression

It defines the collection over which the query will be run. (e.g. *Passenger::extent in the above example

element-type-name

It is a string indicating the element type of the collection being queried.(e.g. Passenger*)

query-string

The query-string is a C++ control expression indicating the query selection criterion. In this example it is **strcmp(name,"Bita")==0** .

Any string consisting of an int-valued C++ expression is allowed, as long as

- There are no variables that are not data members.
- There are no function calls, except calls to **strcmp()** or **strcoll()**, calls involving a comparison operator for which the user has defined a corresponding rank function and/or hash function. See section 8.4 for queries with function calls.

schema-database

The schema-database is a database whose schema contains all the types mentioned in the selection criterion. This database provides the environment in which the query is analyzed and optimized. The database in which the collection resides is often appropriate.

Return value

The return value of **query()** refers to a collection that is allocated on the heap. So when we no longer need the resulting collection, we should reclaim its memory with **::operator delete()** to avoid memory leaks. The resulting collection has the same behavior as the collection being queried. The order of the elements in the result cannot be guaranteed to be the order of the elements in the collection being queried.

8.2 Single-Element Queries

If we know that the answer for the query has just a single element, it might be more convenient to use **as_Collection::query_pick()**. Same format for calls to **query_pick()** as for calls to **query()**:

E query_pick(char*, char*, os_database*) const;

where E is the element type parameter.

Example to query_pick()

```
PlaneType * a_planetype;
a_planetype= PlaneType::extent- >query_pick ("PlaneType*",strcmp("jet",name),db);
```

A pointer to PlaneType object, with the attribute name *jet* is returned by the above query. If there are more than one answer for query_pick(), one of them is picked and returned. If there is no answer for query, **0** is returned.

8.3 Existential Queries

In some cases we just want to check whether a particular element exists in a collection or not. For such cases, we should use **os_Collection::exists()**.

Again it has the same syntax as **query()** and **query_pick()**, but instead of returning a collection or element, they return a nonzero **os_int32**(**int** or **long**, whichever is 32 bits on our platform) for true, and 0 for false.

We can also have **Nested Queries**[8].

8.4 Pre-analyzed Queries

The restrictions in simple queries cause that we cannot ask for the name of passenger, read it into a program variable, and look it up using a query. Also the cost of

analyzing a query is likely to be a relatively expensive operation. If the same query is performed several times, perhaps with different values for the free variables each time, and perhaps on different collections each time, we should use a preanalyzed query.

To work with preanalyzed query, first we create a query object, an instance of class **os_coll_query**. This query will be analyzed upon creation. It is created with one of the static member functions **os_coll_query::create()**, **os_coll_query::create_pick()** or **os_coll_query::create_exists()**.

We give the type of the target collection, a query string and the name of the schema database where this query will be executed. Variables (including data members) can appear in a query string, as long as the type of each variable (except data members) is specified explicitly with a cast.

For example:

const os_coll_query & query1 = os_coll_query::**create_pick**("PlaneType*","strcmp((char *)arg1.name)==0",db);

Once we have a preanalyzed query, we have to bind a value to the free reference by creating a bound query. We can create a bound query at any time, using the constructor for the class os_bound_query. The bound query object can then be used as a parameter for the query method, applied to a collection with the appropriate element type:

```
/* We have planetype's name in variable 'na'*/
....
os_bound_query bound_query1(query1, os_keyword_arg("arg1",na));
a_planetype= PlaneType::extent- >query_pick(bound_query1);
```

The bound query constructor takes two arguments: a preanalyzed query and a *keyword_arg* list, an instance of **os_keyword_arg_list**.

Using os_keyword_arg(), it substitutes the free reference, arg1, with an actual program variable,*na*, for the query execution.

We can also use a function call, as long as the specified function returns the right type. This allows method calls to be used in bound queries. Whenever the value of the program variable changes, we have to provide a new binding.

We can have **more than one** free reference in a query string. In such a case, the constructor takes a list of os_keyword_arg as parameters:

```
/* na and ad are name and address of passenger */
....
const os_coll_query & query1 =
os_coll_query::create_pick("Passenger*","strcmp((char *)s1,name)==0
&& strcmp((char *)s2,address)==0",os_database::of(this));
os_bound_query bound_query1(query1, (os_keyword_arg("s1",na),os_keyword_arg("s2",ad)));
a_passenger= Passenger::extent- >query_pick(bound_query1);
```

When our query objects are no longer needed, we should delete them from the heap using the method call **os_coll_query::destroy**:

```
os_coll_query::destroy(query1);
```

8.5 Query Optimization

If we want to optimize queries over collections, we can use ObjectStore's index facility. To tell ObjectStore which data member to use to build an index, we have to create an **index path** object. Using the simplest kind of path, we can base an index key or iteration order on the value of some data member or simple member function. For example, we can iterate through a set of **Aircraft** in order of the Aircraft's serial numbers (Aircraft with lower numbers precedes Aircraft with higher numbers).

The collection library contains a class **os_index_path** that provides a static member function **create()**. The definition of the index path for the part example looks like this:

```
os_index_path& idx_path1 = os_index_path::create("Aircraft*","serialNo",db);
```

Here, **Aircraft*** is the path's type string, which names the element type of collections, whose elements can serve as path starting points. *SerialNo* is a path string indicating the data member itself. The argument **db** is a database whose schema contains the definition of the class *Aircraft*.

8.5.1 Explicit Index

In order to perform an ordered traversal over a collection, we can build an index. To request an index into the set **extent**, we specify the key of the index as the value of the data member, serialNo. We use the member function **os_collection::add_index()**:

```
os_Set<Aircraft* > *extent;
. . .
os_index_path &path1 = os_index_path::create("Aircraft*", "serialNo",db);
extent- >add_index(path1);
```

Once we invoke this function, any query over **extent** involving lookup by *serialNo* is optimized.

Having explicit indexes in a collection slow down updates to the collection. This is because index maintenance is performed whenever such an update occurs. Therefore, it is advisable to remove an index whenever we have update to the collection and insert before querying. The method call for dropping an index looks like this:
Aircraft::extent- >**drop_index(path1)**;

8.5.2 Implicit Index

To print our list of **Aircraft**, ordered by serialNo, we now define a cursor using an index path object:

```
/* List Aircraft, ordered by serialNo */
Aircraft::access_extent(db);
os_index_path& idx_path1 = os_index_path::create("Aircraft*","serialNo".db);
os_Cursor<Aircraft* >airc(*Aircraft::extent,idx_path1);
for (Aircraft* a1 = airc.first(); airc.more(); a1 = airc.next())

        a1- >display();
```

We perform an ordered traversal over a collection using a cursor. The os_Cursor classes provide a constructor that takes a path as an argument. To support this iteration, ObjectStore builds an index to access the **aircraft** in the right order. When the iteration is finished, the index is automatically deleted.

8.6 Index Options

The default index in ObjectStore is implemented as hash tables (unordered indexes). But if your application performs range queries involving the indexed data member, you can request an ordered index, implemented using a B-tree. With a B-tree, queries involving <, <=, >, or >= comparisons on indexed data member can be computed more efficiently than with an index implemented only with hash tables.

The os_index_path::ordered Enumerator

You request an ordered index by specifying **os_index_path::ordered** as the second argument to **add_index()** when requesting the index:

```
os_Set<Aircraft* > *extent;
. . .
os_index_path &a_path =
```

os_index_path::create("Aircraft*","serialNo", db);
extent- >add_index(a_path, **os_index_path::ordered**);

The default value for this parameter is *os_collection::unordered.*

8.7 Index Maintenance

Whenever you use a path, for each data member mentioned in the path string, except **const** and collection-valued members, you must perform or enable index maintenance. Note that failing to perform or enable index maintenance can result in corrupted indexes, incorrect query results, and program failures.

8.7.1 Automatic Index Maintenance

Before we can create *index_path* objects we have to extend the class declarations. Each data member that appears in an index path has to be declared as indexable. To make a data member indexable, you add to the class whose data member you want to be indexable a public or private data member of type **os_backptr**. The declaration of the data member of type **os_backptr** must precede the declaration of the data member (or member functions) you want to make indexable:

```
class Aircraft
{
public:
        os_backptr baackp;
private:
        static   os_Set<Aircraft*>* extent;
        os_indexable_member (Aircraft,serialNo,int) serialNo;
        char* manufacturdate;
        char* Amanufacturer;
        int             flighthours;
        ....
};
```

The last thing you must do to make a data member indexable is call the macro **os_indexable_body()** to instantiate the bodies of the functions that provide access to the indexable member. These functions ensure that any indexes keyed by that data member are properly updated when the member is updated. The macro call should appear (at top level) in a file associated with the class defining the indexable member:

os_indexable_body(Aircraft,serialNo,int,os_index(Aircraft,backp));

Note: Since some macro arguments are used (among other things) to concatenate unique names, avoid white space in macro arguments.

8.7.2 User-Controlled Index Maintenance

Using the functions **os_backptr::break_link()** and **os_backptr::make_link()** whenever you update the member, allows ObjectStore to perform index maintenance under user control.

You still have to add an os_backptr data member to the declaration of the class for which you want to build an index:

```
class Aircraft:public PlaneType
{
public:
     static        os_Set<Aircraft *>*extent;
     os_backptr    bkptr ;
private:
     int           serialNo; /*indexable data*/
     char*         manufacturdate;
     char*         Amanufacturer;
     int           flighthours;
     ...
}
```

Whenever a new object of this class is created, a new index entry has to be generated and inserted into the index. This is done using the **os_backptr** member function

make_link, which should be called in the body of the constructor of the class after you assign a value to the data member for which the index is maintained:

```
   /* Constructor */

Aircraft::Aircraft(char *na, char * Pma,...)
{
long len;
serialNo=sno;
manufacturdate=mdate;
lent=strlen(Ama) + 1;
...
/* explicit maintenance for index, keyed on serialNo */
bkptr.make_link(&serialNo, &serialNo, os_index(Aircraft,bkptr)
        -os_index(Aircraft,serialNo));
/* add new Aircraft into Aircraft::extent */
if (!extent)
     access_extent(os_database::of(this));
TIX_HANDLE(err_not_assigned)
    {
    extent->insert(this);
    }
 ...
   }
```

Deleting an object requires the removal of the corresponding entry from the index by calling **os_backptr::break_link()**.

```
   /* Destructor*/

Aircraft::~{}Aircraft()
   {
    /* explicit maintenance for index, keyed on serialNo*/
    bkptr.break_link(&serialNo, &serialNo, os_index(Aircraft,bkptr)-
                os_index(Aircraft,serialNo));
```

```
delete [] Amanufacturer;
remove Aircraft from Aircraft::extent
TIX_HANDLE(err_not_assigned)
    {
    extent->remove(this);
     ....
    }
}
```

You should call **break_link()** just before making a change to an indexable data member (this removes an entry from each relevant index). Call **make_link()** just after making the change (this inserts a new entry into each relevant index, indexing the object by the new value of the relevant path). You can ensure that this happens by encapsulating these calls in a member function for setting the value of the data member.

The functions **make_link()** and **break_link()** take the same arguments: the first two arguments are identical and they are the address of the indexed data member, which in this case is name; the third argument is of type **int** and is always of the form:

os_index(Classname,backptrname) - os_index(Classname,membername)

where

- Classname stands for the class name (e.g. Aircraft)
- backptrname is the name of the os_backptr member of the class (bkptr)
- membername is the name of the data member for which the index exists(serialNo).

This is all we have to do to maintain indexes with key name, because it is not possible to change the name after an object has been created.

ObjectStore allows you to specify different behaviors for indexes [8].

Chapter 9

Data Integrity

There are many ways in which database integrity might be jeopardized. We show an example that may cause a problem in **referential integrity**. In our sample application, take the many-to-one relationship between Passengers and their one or many booked seats.

```
class Passenger{
public:
        os_Set<Booking*> books;
        ....
};

class Booking {
private:
        Passenger * client;
        ....
};
```

Any updating to passenger's booking, like deleting one and assigning a new booking into the books collection of passenger object should be done carefully. You must remember to change the inverse client pointer in the Booking object to point to the new-booked object.

There are some kinds of database pointers that can cause **dangling reference**, the pointer to a deleted object, as well as the incorrectly typed pointer.

If all of the data members that point to the deleted object are not updated, they will point to unallocated space, or perhaps even worse, to the valid space of another process.

ObjectStore provides facilities to help deal with these integrity maintenance problems.

9.1 Inverse Data Members

ObjectStore has relationship facility, which allows you to declare two data members as inverses of one another, so they stay in sync with each other according to the semantics of binary relationships. This works for pairs of data members that represent one-to-one, one-to-many, and many-to-many relationships.

In order to use relationship we should include the following libraries in this order: ostore/ostore.hh, ostore/coll.hh, ostore/relat.hh

9.2 Defining Relationship

There are two types of macros, which are used to define relationships in classes. The first type of macro, which are called as relationship member macros define the data members as well as the different access functions used to get and set the data member values.

They are as follow:
os_relationship_1_1() - for one-to-one relationships
os_relationship_1_m() - for one-to-many relationships
os_relationship_m_1() - for many-to-one relationships
os_relationship_m_m() - for many-to-many relationships

The second macro types are body macros. They provide the instantiations for the relationship's accessory functions.

The corresponding function body macros are:
os_rel_1_1_body() - for one-to-one relationships
os_rel_1_m_body() - for one-to-many relationships
os_rel_m_1_body() - for many-to-one relationships
os_rel_m_m_body() - for many-to-many relationships
Here is an example of our Airline application:

```
class Passenger{
private:
....
os_relationship_m_1(Passenger,books,Booking,client,os_Set<Booking*>)books;
....
};
```

Relationship member macros require five arguments:

- Name of the class defining the member (Passenger)
- Name of the member (books)
- Name of the class defining the inverse member (Booking)
- Name of the inverse member (client)
- Type of the data member itself (os_Set<Booking* >)

We also need to use the matching function body macro in the class implementation file:

```
os_rel_m_1_body(Passenger,books,Booking,client);
```

The arguments for body macro are the first four arguments of its corresponding member macro.

The other side of relationship also has to be defined. In above example, in **Booking** class we should define one-to-many member macro relationship with its matching body macro:

```
class Booking{
private:
....
os_relationship_1_m(Booking,client,Passenger,books, Passenger*)client;
....
 };
```

function body to instantiate the access functions is defined:

os_rel_1_m_body(Booking,client,Passenger,books);

So, as we see the relationship macros should always come in fours. This means that for instance, a one-to-many relationship member must also have a one-to-many relationship body, as well as a many-to-one inverse member, which itself must have a many-to-one relationship body.

9.3 Relationship Interface Styles

After you define a binary relationship in a class, you can present this relationship for end user in different interfaces.

We take the previous **Booking** class, which has a single valued relationship **client** that points to the **Passenger** class.

```
Passenger* a_passenger;
Booking* a_book = new(db,Booking::get_os_typespec())Booking(...);
a_book- >client = a_passenger;
```

ObjectStore's relationship facility would automatically ensure that the equivalent of the following operation was performed:

```
/*insert new book into set of bookings done by a passenger*/
(a_passenger- >books).insert(a_book);
```

9.3.1 Simple Data Member

In this style, relationship is like a simple data member of the class. It can be accessed using the "." or "- >" operator. The end user need not be aware that special *inverse-update* processing is occurring as showing below:
Relationship definition

```
os_relationship_m_1(Passenger,books,Booking,client,os_Set<Booking*>)books;
Passenger*  a_passenger;
Booking*        a_book;
....
a_passenger=a_book->client;
```

Note that there are two restrictions of using this style. The first restriction is that you can't initialize this data member in the initialization list of a constructor. They must be explicitly set in the body of the constructor.

The second situation is that, this style is only available in the C++ library interface, not the C library interface, because it relies on the C++ capability to define coercion operators and to overload operator=().

9.3.2 Relationship Interface Style

In this style, we treat relationship like objects in its own right. There are ObjectStore particular functions **getvalue() and setvalue()**, which are used to set or get the value of the relationships:

Passenger* a_passenger;
Booking* a_book;
a_passenger = a_book- >client.getvalue();
Notice that, **a_book- >client** will be the relationship's instance. This interface solves the restrictions in the first style. It also could be considered a more consistent approach, because it allows relationships to be treated like real objects rather than simple memory pointers.

9.3.3 Functional Interface Style

In functional interface style, we let to the end user to access the relationship just using functions defined on the class. You can do this, by defining relationship in private part of the class.

This style offers a more object-oriented interface, because it takes away the impression of direct member access that is associated with the simple data member and relationship interface styles. Here is an example using a function *set_book()* to set the value of the relationship:

```
Passenger* a_passenger;
Booking* a_book;
a_book=a_passenger- >set_book();
```

Important:

The function body macros for many-to-many and.many-to-one relationships must not be placed in files that could be included more than once in the build of a single application because some of the functions defined by these macros are virtual functions. Multiple definitions of these functions can cause the compiler to produce redundant versions of the tables that are used to resolve calls to such functions.

You can also have ability to have different behavior, like **indexable inverse member**, **delete propagation**, and so on. Descriptions of all these can be found in chapter 6 of [7].

Chapter 10

Conclusion

We described the concept of ObjectStore, an Object Oriented Database Management System. ObjectStore supports essential functions to create database applications. Database access feature allows us to perform basic activities, such as opening and closing databases, and creating and accessing persistent objects. Transaction features take the same basic activities and regroup them into logical units of work that are performed either all together or not at all. At the same time ObjectStore does not allow other transactions from viewing intermediate results.

Grouping objects into various types of collections is simple with ObjectStore's library of collection classes. This class library enables us to define lists, arrays, sets, or bags of related objects. Collections can be queried efficiently through ObjectStore's query facility, which selects objects on the basis of the values of their attributes (data members). To improve the performance of queries, indexes can be built on the collections.

ObjectStore provides a relationship facility that automatically maintains the pointers used to specify relationships between objects.

By developing Airline reservation application, we examined these fundamental features of ObjectStore.

Bibliography

[1] ObjectStore C++ Release Notes.

[2] ObjectStore Building C++ Interface Applications.

[3] ObjectStore Management.

[4] ObjectStore Component Server Framework User Guide.

[5] ObjectStore C++ API Reference.

[6] ObjectStore Collections C++ API Reference.

[7] ObjectStore C++ API User Guide.

[8] ObjectStore Advanced C++ API User Guide.

The publications listed below are considered particularly suitable for more detailed discussions of the topics covered in this document.
Note: All the references are on_line and can be found in

`http://www.unik.no/~mdbase/OS_doc_cc`