National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

An Object-Oriented Interface to a Relational Database for Multimedia Applications*

F. Béranger

Departement of Electrical Engineering
University of Ottawa
Ottawa, Ontario

*A thesis submitted to the School of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Applied Science in Electrical Engineering.

© François Béranger, Ottawa, Canada, 1992

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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

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

ISBN 0-315-80040-2

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

Emerging multimedia applications, such as CAD/CAM or real-time conferencing systems, no longer belong to the area of traditional applications, because they require new types of data and their importance places great demands on designers and programmers. The recent breakthrough of the object-oriented methodology is due to its ability to deal with both the problems of flexibility caused by multimedia applications, and the difficulty of programming ever larger applications. But object-oriented programmes lack persitence, and the traditional databases associated with typical office applications no longer suffice. The current research tries to find an answer to this problem, either with non-first normal form models, with object-oriented query languages, or with pure object-oriented databases.

In this thesis, we investigate the aspects of persistence, of the object-oriented approach and of multimedia data management. We show that there is currently no complete answer to the problems stated above. But partial solutions brought by relational database technology to the problem of efficient storage, combined with the ease of use of object-oriented techniques, allow us to implement multimedia applications without difficulty.

Our work consists of an analysis of the requirements of multimedia data management. Based on this, we build a data model which allows us to take advantage of both the relational technology and the object-oriented paradigm. We then build an architecture for that system, based on the client-server model. Finally, as a validation for our data model and achitecture, we present a prototype implementing our specifications, using a relational database management system and a BLOB (binary large object) storage manager.

Acknowledgements

First of all, I wish to express my gratitude to Dr. M. Goldberg, my supervisor, for the constant help and support he gave me during my stay at the University of Ottawa.

I also wish to thank Dr. A. Karmouch for his skilled advice, for the time he generously spent with me, as well as for his encouragement during this work.

I am grateful to the Telecommunications Research Institute of Ontario for its financial support.

My thanks also go to all the members of the Multimedia Communications Research Centre and to the assistants of the Department of Electrical Engineering, who gave me the opportunity to work with a friendly team.

Contents

1 Introduction 1

2 Traditional Database Systems 4

2.1 Database Vocabulary and Concepts 4

2.1.1 Data Models 5

2.1.2 Schema and Instances 5

2.1.3 Architecture 5

2.1.4 Data Independence 6

2.1.5 Languages 6

2.2 The Relational Model 7

2.2.1 Relational Terminology 8

2.2.2 Characterisation of the Relational Model 8

2.3 Conclusions 11

3 The Object-Oriented Paradigm 12

3.1 Concepts and Terminology 12

3.1.1 Basic Mechanisms 12

3.1.2 Inheritance 14

3.1.3 Concepts 15

3.1.4 Related Terms 16

3.2 Object-Oriented Database Systems 17
3.2.1 Persistence 18
3.2.2 Transactions 19
3.2.3 Concurrency Control 20
3.2.4 Recovery 21
3.2.5 Querying 22
3.2.6 Integrity 24
3.2.7 Performance 25
3.2.8 Other Capabilities 28
3.3 Conclusions 29

4 Multimedia and Databases **30**
4.1 Overview of Relational Database Systems 30
4.1.1 Strengths of Relational Database Systems 31
4.1.2 Problems Related to Relational Database Systems . . . 32
4.1.3 Summary 34
4.2 Object-Oriented Database Systems 34
4.2.1 Major Drawbacks 34
4.2.2 Benefits of Object-Orientation 37
4.2.3 Summary 40
4.3 Trends in Database Research 40
4.3.1 The AIM project 41
4.3.2 Relational-Based DBMSs 42
4.3.3 Full Object-Oriented DBMSs: The Example of O_2 . . . 44
4.4 Conclusions 45

5 Analysis of Requirements 46
5.1 Definition 46
5.2 Functional Requirements 47
5.2.1 Multimedia Requirements 48
5.2.2 Programming Requirements 51
5.3 Achievements 52
5.3.1 The Relational Features 52
5.3.2 Object-Oriented Features 53
5.3.3 Limitations of the System 53
5.4 Conclusions 54

6 Data Model and Architecture 55
6.1 Data Model 55
6.1.1 The External Model 55
6.1.2 The Implementation Model 56
6.1.3 The Physical Models 58
6.2 Architecture 58
6.2.1 The Communication Module 60
6.2.2 The Database System 60
6.3 Data Flow 62
6.3.1 Getting a New OID 62
6.3.2 Getting an Object from the Database 64
6.3.3 Updating an Object 64
6.4 Critical Analysis 65
6.4.1 Mapping Between Levels 65
6.4.2 Recovery 66
6.4.3 The Client-Server Model 66
6.5 Conclusion 66

7 Implementation of MOODS 68
7.1 Implementation Results 68
7.1.1 Programming Environment 69
7.1.2 Object Identity 69
7.1.3 Concurrency Control 70
7.1.4 Transactions 71
7.1.5 Database Design 71
7.1.6 Communications 72
7.1.7 Programmer's Entry Points 73
7.2 Evaluation 74
7.2.1 Transparency to the Programmer 75
7.2.2 Extensibility and Flexibility 75
7.2.3 Multimedia Requirements 76
7.2.4 Shortcomings 77
7.3 Conclusions 78

Conclusions 80

A Tables 83
A.1 Table `Chart` 83
A.2 Table `Shortt` 83
A.3 Table `Integert` 84
A.4 Table `Floatt` 84
A.5 Table `StringtS` 84
A.6 Table `StringtM` 84
A.7 Table `StringtL` 85
A.8 Table `Datet` 85

A.9 Table Blobt 85
A.10 Table Freet 86
A.11 Table UsedOid 86

B Classes **87**
B.1 Class Bullet 87
B.2 Class Client 88
B.3 Class Connect 88
B.4 Class ConnectionLoc 89
B.5 Class List 89
B.6 Class ListItem 90
B.7 Class Location 90
B.8 Class Message 91
B.9 Class OidMapItem 92
B.10 Class Operation 93
B.11 Class Permanent 93
B.12 Class Server 94
B.13 Class Server 94
B.14 Class Transaction 95

C Messages **96**

List of Figures

6.1 Example to illustrate the data model. 57
6.2 Architecture 59
6.3 Getting a new OID 63

7.1 Message Structure 74

Chapter 1

Introduction

Until the last decade, computers were mainly used in offices as enhanced accounting sheets, and by scientists as powerful calculators. But the field of application of computers has suddenly exploded with the introduction of low-cost components. Similarly, the applications have moved from either high-precision calculus or heavy administrative work such as automatic bill-paying computation or sale price evaluation to almost any possible domain: education, design, or document editing are some examples. Since a computer is now used in a hospital as well as in a lawyer's office or by moviemakers, it has to manipulate other data types than names and figures. This is what is usually called "multimedia".

Often presented as the collection of image, voice and text, the term *multimedia* encompasses—according to some authors—a very large range of information vehicles. Not everyone agrees on the definition of this word, but despite its somewhat loose meaning, and even if we restrict ourselves to the narrowest definition, there are more and more applications that fit in the frame of multimedia applications. Examples of these are CAD/CAM applications, multimedia extensions of hypertext for teaching purposes (usually called *hypermedia*), medical communications applications, desktop publishing, or video sequence editing.

The emergence of multimedia applications has two major consequences. First of all, it emphasizes the limits of traditional programming methods. The complexity of programmes has reached a point where a programmer

cannot understand a whole programme, and where the number of concepts present at the same time extends far beyond human possibilities. As a result, software production costs increased (in the late seventies), as well as maintenance costs. Meanwhile, the quality and reliability of programmes slowly decreased because of their complexity, thus following the traditional law of nature that the most complex entities are less robust to environmental changes. The late eighties saw the generalisation of a methodology called the "object-oriented approach": by tightly associating the data with the functions which manipulate them, one creates objects that are independent entities of a programme. This permits programmers to cope with the complexity of programmes, provides a natural representation for problems, and simplifies the maintenance task.

Secondly, the data manipulated by multimedia applications are now far too complex to be destroyed when the application terminates and recreated when it is restarted. Sometimes, it is even impossible: a design application for parts of an aircraft cannot forget about the tens of thousands of pieces that represent months of work. Therefore, the need for adapted storage is more and more pressing. But storage management systems, or *databases*, evolved at a slower pace than applications. In an effort to catch up with them, research explored all possible directions without inter-coordination. However, after a few years, the techniques for multimedia data storage management began to follow schematically three paths, as we will explain in Chapter 4.

In this thesis, we propose a system that allows programmers to benefit from the three concepts now necessary in new applications: the object-oriented approach, multimedia and persistence. The goal of this system is to bring these concepts to the programmer in the most simple way so that he/she can concentrate on the design of his/her application.

Our approach is, first, to understand and analyse the issues involved in databases. It clearly appeares that a database is not only a storage management system, but must also provide the user with functionalities such as concurrency control. Then we chose, among the set of already proposed solutions, features that will allow our system to answer the problems addressed by multimedia data management in an object-oriented fashion. We finally checked the validity of our model with the implementation of the interface between a relational database and object-oriented applications.

The thesis is organized as follows: Chapter 2 is devoted to the analysis of traditional database concepts and to an overview of the relational technology. This includes also a brief presentation of the related models: the hierarchical model, the network model and the entity-relationship model. Chapter 3 explains the concept of object and the bases of the object-oriented paradigm: essentially classes, methods and encapsulation. Then, in Chapter 4, we detail how far the previous models can be applied to multimedia data management, and what their particular limitations are. Additionally, we study the existing approaches to multimedia that will be used in Chapter 6 as a basis for our data model. Before we can specify a data model, we need to clearly define the term *multimedia* and also to analyse the requirements that our interface should fulfill. This is done in Chapter 5. We then provide, in Chapter 6, a description of our data model at each of the three levels: the physical level, the implementation level and the external level. This chapter also details the architecture of the interface. It supports remote applications, and is thus based on the client-server architecture. Chapter 7 is dedicated to the study of the solutions that were used for the implementation: we have tried to take as much advantage as possible from the relational model and the object-oriented database systems. Finally, we summarise the present work and propose a few ideas for further research in the last chapter.

Chapter 2

Traditional Database Systems

In our work, we built an interface atop a relational database system. This interface will therefore benefit from the positive aspects of the relational DBMS as well as suffer from its deficiencies. It is thus important to study the characteristics of relational technology. This chapter is devoted to that task. We first introduce the general terminology and concepts of conventional databases. Then we present the peculiarities of relational systems. Naturally, alternative models must be studied too. We found it more convenient to present them in Chapter 4, after the introduction of the concept of the object-oriented approach and multimedia. The topics tackled in this chapter are extensively analysed in the literature; for more details, refer to [29, 35, 41].

2.1 Database Vocabulary and Concepts

A complete *database system* is comprised of a *database*, the repository of data, and a *database management system*, the programme in charge of storing, maintaining and retrieving the data in the database. There is a general consensus on the meaning of the terms that we introduce here, and they are presented in any detailed book about database system. We have found that of Elmasri and Navathe particularly clear, and we used it for this part [35].

2.1.1 Data Models

A database generally provides the users with some level of abstraction. Because of the difference between the physical organization of data and their structure when presented to the end-user, a database system is associated with a *data model*, which is a set of concepts describing the operations and the structure of the database, *i.e.* the data types, the relationships and constraints that should apply to the data. Although a number of data models have been proposed, a categorisation in three layers is now generally accepted. These are the conceptual data model, the implementation data model, and the physical data model.

1. The first one is often called an object-based data model, because it uses high-levels concepts such as *entities*, whose properties are described with *attributes*. Enties are interrelated with *relationships*.

2. The second type of data models are called record-based models, as opposed to the object-based models, because they represent data in record structures. The hierarchical, network and relational models belong to this class.

3. Finally, the physical data models describe how data are permanently stored: the format of records, their ordering and access paths.

2.1.2 Schema and Instances

At any level, the database is described with a *schema*. A schema is specified during database design, and, unlike the content of the database, is not subject to frequent changes for the duration of the database. The base building block of a schema is the *construct*. Data in the database define *instances* of this schema construct. Another name for the schema is the *intension*, and the occurences of an intension form the *extension*.

2.1.3 Architecture

The *architecture* of databases varies with the systems. The ANSI/SPARC committee proposed a three-schema architecture that is now widely used [46]:

1. The low-level schema is called the *internal* schema and thus corresponds to the third layer of the model. It specifies the full details of data storage.

2. The *conceptual* schema describes the structure of the whole database. It extensively describes all the entities, attributes and relationships of the database. Therefore, the corresponding data model can be either an implementation data model or a high-level data model.

3. The *external* or *view* level includes a number of external schemata or user views. They describe the database as viewed from the standpoint of a particular group of users. Both models used in the conceptual schema can be used here. Consequently, DBMSs generally use the same data model for both schemata.

These three schemata should be regarded as a description of data. Data actually exist only at the physical level, and data used at another level must be mapped to comply with the description given by this level.

2.1.4 Data Independence

One of the goals of the use of schemata is to achieve independence in two ways. First, the external level must be independent of the conceptual level. This means that the conceptual schema can be changed without changes to the external schema. This is called *logical data independence*. Second, the *physical data independence* is the independence that exists between the conceptual level and the internal level. So, the internal schema can be modified without affecting the conceptual schema.

2.1.5 Languages

Specifying a schema is done by the database administrator with a particular language called the *storage definition language* at the internal level, the *data definition language* at the conceptual level, or the *views definition language* at the external level [35]. Once the schemata are designed, the end users exchange data with the database with a *data manipulation language*. This

is generally a high-level declarative language. Most often, all these different languages have the same syntax and are embedded in a general database language. SQL provides a good example of such a language [5].

2.2 The Relational Model

We describe here an implementation model. It is customary to present four different successive models of database systems to place the relational model in its context. These are the hierarchical model, the network model, the relational model, and the entity-relationship model. The first two are now obsolete and of little interest to our study. The fourth model does not deserve much consideration, because there are few products using it as a basis for the database. We will therefore describe fully the basics of the relational model only, after giving a brief explanation of the other models.

The hierarchical model. This is the oldest model (about 1960), based on the observation that many entities in the real world are hierarchically organized. Data are stored into records (fixed length "boxes") that can have any number of children but only one parent (hence the hierarchy). Therefore, 1:N relationships (where one record is in relation with many others; for example, a department with many employees working exclusively in the department) are well represented. But M:N relationships are extremely cumbersome to represent, at the cost of either extraneous data duplication or pointer manipulations across hierarchies. (In this type of relationship, several records of one type are in relation with several records of another type. This is the case of the employee-project relationship, where an employee works on several projects, but a project hires several employees.) This limitation caused the progressive disappearance of this model.

The network model. Successfully representing M:N relationships, the network model replaced the hierarchical model in the seventies. It was proposed in 1971 in the CODASYL Data Base Task Group (DBTG [30]). It still uses records, but with several parents and children. The problem was that querying required the knowledge of the data organization at the physical level, and in particular the access path.

The entity-relationship [E.-R. model.] This model appeared after the relational model [18]. Unlike the preceding models, it uses two concepts: entities and relationships. It was proposed to unify the view of data irrespective of the actual model. As a result, it is used for its attractive and popular diagrammatic design technique. There exist simple algorithms to transform an E.-R. diagram into either a hierarchical, a DBTG or a relational implementation. It was later enhanced with the inclusion of categories (ECR model). Categories allow refinements in the storage of data. For example cars and trucks have the same basic information (licence number, color, make), but trucks have additional information such as weight, while cars have a given number of passengers. The notion of category allows common information to be grouped without structure duplication.

2.2.1 Relational Terminology

A special terminology is used for relational databases. Data are organized in rows, forming a table. The table is called a *relation*, and a row is called a *tuple*. A column defines an *attribute* which takes its values in a *domain*. An attribute cannot take its values outside of the defined domain.

2.2.2 Characterisation of the Relational Model

We can summarize the main characteristics of relational databases as follows:

Atomicity constraint. The basic element in a relation is the *atom*. An atom is an instance of an attribute. The constraint imposed by the relational model is that an atom can only take *one* value in the domain. When we will introduce the normal forms, this constraint will be termed "first normal form".

Keys. In the relational model, everything is a relation, including the relationships between entities. Relationships are achieved by establishing cross-references through relations. Therefore, a referenced relation must have a *key* (primary key), and the referencing relation uses a *foreign key*. The database ensures that primary keys are not null (entity

integrity constraint). Some systems also check that foreign keys always refer to existing tuples (referential integrity constraint). This last type integrity is not always automatically maintained and must be included in the application.

Relational algebra. The good performance of relational systems is chiefly due to their sound mathematical foundations. Based on the relational algebra formalism, design principles have been investigated to reduce data redundancy in the database, to allow better performance for updates, and to ensure data consistency. The outcome is a set of design rules called the "normal forms". Without going into great detail, we can give here the definitions of the normal forms as they are used now in all the relational systems. Codd introduced the first three normal forms [22]. His definition of the third normal form, however, was not satisfying and he later refined it with Boyce [23] (hence its name). One generally considers that every relation should be in BCNF. The concept of multivalued dependencies was the base of the fourth normal form [36], defined by Fagin [37]. The multivalued dependency (MVD) is a consquence of the first normal form, which states that an atom cannot be in relation with a list. It logically follows that if two atoms are in relation with the same list, that list is duplicated. This is a potential source of update problems. The fifth normal form expresses the fact that some tables may be losslessly decomposed in more than two other tables. This is called a *join dependency* and is a generalisation of MVDs.

1NF A relation is in first normal form if and only if all underlying domains contain atomic values only. This simply says that all data must be atomic, and aggregates or structured data types are not allowed.

2NF A relation is in second normal form if and only if it is in first normal form and every nonkey attribute is fully dependent on the primary key.

3NF A relation is in third normal form if and only if it is in second normal form and every nonkey attribute is nontransitively dependent on the primary key.

BCNF A relation is in Boyce-Codd normal form if and only if every determinant is a candidate key.

4NF A relation is in fourth normal form if and only if every multivalued dependency in the relation is a consequence of the candidate keys of this relation.

5NF A relation is in fifth normal form if and only if every join dependency in the relation is a consequence of the candidate keys of this relation.

It has been stated that the fifth normal form is the "final" one.[1] It solves the problem of join dependencies [3]. The last two normal forms are not mandatory for a system to work, but they should be achieved when possible. We must, however, bear in mind the fact that there are some cases where normalising all the way up to the fifth normal form is not advisable (cf. [29, p. 391] and [19, p. 83]). It is up to the database administrator to decide, for reasons of efficiency, if the normalisation process should be completed or not.

Query language. We stated earlier that SQL is a good example of a homogenous database language that integrates four languages, each dedicated to a particular level of the database. The main advantage of this language is its ability to separate the entities acted upon from their location. In other words, a query in SQL specifies what operation is wanted instead of where to put the data (or get them) and how to do it.

To conclude this characterisation, we can say that the relational model has had considerable success. Major products such as Ingres or Oracle are the present state of the art, although their origin is now twenty years old [21].

[1]To slightly paraphrase Date [29], if we restrict ourselves to the join and projection operations in relational algebra, there is no greater dependency than the join dependency. In other words, there is no dependency such that the join dependency is a special case of this "higher form" of dependency. Therefore, we cannot go any further than the fifth normal form.

2.3 Conclusions

We have presented the basics of databases. After introducing the terminology and the concepts of traditional databases, we presented the most widespread technology, the relational model. We defined the particular vocabulary and sketched the important points of relational database systems.

Although the relational model had a firm base very early and is now mature, research is still very productive in this area (storage structures, data models, integrity constraints, are a few examples). Some research is also carried out to add new features to the relational databases. We will review only the most relevant fields for our study. We are interested in using variable size and/or weakly structured data such as image, voice or video. There are three main areas of research regarding this issue: new data models, multimedia databases and object-oriented database systems. The former essentially consists of non-first normal form models (abbreviated NF^2). Multimedia databases very often use many object-oriented features, so that it is extremely difficult to present them without introducting first the object-oriented paradigm.

Chapter 3

The Object-Oriented Paradigm

In this chapter, we review the object-oriented methodology and present the features of the object-oriented databases. The literature is abundant, but a few works of major importance provide sufficient details [7, 33, 1]. In the first section of this chapter, we introduce the basic terms and the concepts. Then, we have a closer look at the features of object-oriented databases.

3.1 Concepts and Terminology

3.1.1 Basic Mechanisms

Objects

Traditional programming methods separate the data and the procedure. A typical consequence of this is the organization of binary COFF files (Common Object File Format) produced by traditional compilers: information is separated in segments, two of which are called "text" and "data". One contains the programme code and the other the necessary data. However, these files also represent the first step towards object-oriented programming, and it is not a coincidence if these files are called "object files", even though the object-oriented paradigm did not exist at the time this name was given. Indeed, they roughly comply with the characteristics of objects, that we give here, and support some object-oriented features such as encapsulation.

Unlike classical programmes, object-oriented programmes mix the conventional data with the programme which uses them. *Objects* are thus particular entities that have their own data *(attributes)* and behaviour *(methods)*. There are generally two kinds of objects. *Passive* objects act only upon request, while *active* objects can initiate actions of their own (alert messages for example).

Messages and Methods

Action is triggered in objects at the reception of *messages*. There are three types of messages, but they are all treated equally by the objects. A first type of message is the *information* message type. As the name says, information messages inform an object that something has changed in its environment (*i.e.* the other objects that the current object is related to). For example, in a driving simulation programme, the object "wheel" can receive a message telling it that it has new air pressure because the external temperature has increased or the driver simply re-inflated it. A second type of message is the *questioning* message: information is extracted from the object. In our example, it would be something like "what is the current adherence?" which depends on the temperature of the external rubber, the depth of the tread and if the soil is wet, iced or dry. Finally, there are *behavioural* messages which are an explicit request to a specific behaviour. A typical example is to apply a given torque to the wheel, or to turn it a number of degrees left or right. The set of messages that an object understands is called the *protocol* of the object.

The response of an object to a message, whatever its type, is to trigger the appropriate procedure, which is called a *method*. This is where the internal state of the object can be changed (see the term "encapsulation" further on). Methods can be arbitrarily complex and it is not unusual for them to send messages to other objects. The sender of the message has no notion of what the method consists of, and therefore considers the object as a black box. This is all the more true because, as we will see later, the internal data of an object cannot be accessed by other objects. In other words, a message tells an object what is wanted, and the methods define how to do the task; the sender of a message is not aware of the *how*.

Classes, Instances and Instance Variables

It is common for several objects to have the same structure and the same methods. In our wheel example, we have at least two similar objects: the front wheels. In this case, instead of duplicating the structure and the methods, we would like to have a "meta-object" that describes what the object will be (its data, methods and protocol), and that can be instanciated into a specific object. Such entities are named "classes". Classes may also receive some types of messages. These are basically: "create new object" and "delete object".

An object is an *instance* of a class. Therefore, all the data (necessarily derived from the class) which are specific to an object and not to another object instanciated from the same class are called *instance variables*. Some data are the same for all the objects in a class (here the word "class" takes another meaning and stands for the collection of the objects instanciated from the same parent class), and are not replicated throughout the objects. They belong to the class itself and are therefore called *class variables*. These two types of data define the *state* of the object.

Classes and types are two different things. The first one describes an object while the second one can only be applied to the data within that object. Types can have a similar hierarchy to classes and thus have common properties (the next paragraph describes the concept of inheritance, which only applies to classes).

3.1.2 Inheritance

A characteristic of a class is that it can be an extension of another class. For example, the class *FrontWheel* describes the same object as the class *RearWheel*, with the addition that front wheels can be turned right or left, and that—on modern cars—they receive a positive torque from the engine block. Instead of constructing two indentical classes and adding features to one of them, it is better to let the class *FrontWheel* "inherit" from the class *RearWheel* and write only the difference between the two classes in the class *FrontWheel*. The inheritance mechanism applies to both data and methods. If they are not found in the parent class of the object receiving a message,

they are searched for in its parent class and then in the parents of the parent class, etc. The parent class of a class is called the *superclass*.

A class can have multiple superclasses. This is called *multiple inheritance*. Multiple inheritance is a problem for the resolver in charge of finding a method or data in the superclasses: if they are present in at least two of the superclasses, which one should be chosen? The answer is language dependent. Generally the first matching element is chosen, and the language specifies if it is using a breadth-first or depth-first search in the class tree.

3.1.3 Concepts

The basic mechanisms outlined above are the foundation of the object-oriented paradigm. The three concepts summarised here are the direct consequence of these mechanisms. They are polymorphism, abstraction and encapsulation.

Polymorphism

The messages are not dependent on the particular object they are aimed at, so that any message can be sent to any object. Message recognition is simply not guaranteed. But the same message can be recognised by different objects. It will therefore produce different effects. For example, a text object receiving the method "print" will produce a stream of ASCII characters for a printer, but the same method applied to an image will produce a bitmap with control codes the printer can understand. Polymorphism contributes to the "black box effect" because there is a separation between the request itself and how it is processed.

Abstraction

Because of inheritance, the tree of classes can be very high. Each level provides an additional level of abstraction for the programmer (and for the user also), so that the same object can be considered in more or less detail. For example, the set of an engine, four wheels, a steering system and an

almond-green body can be regarded more abstractly as a particular car, as a vehicle, or as an object someone owns, or simply as an inanimated thing.

Encapsulation

Encapsulation describes the fact that data and methods are hidden from other objects and can only be accessed by sending messages. This allows objects to maintain their integrity. Additionally, the contents of a method or of an object can be modified without incidence on the rest of the programme.

3.1.4 Related Terms

Many new terms are associated with the object-oriented paradigm. We shall consider here two of them, which are of particular importance in multimedia applications.

BLOBs

"BLOB" stands for Binary Large OBject. This indicates a data type (not a class in itself), used to store large chunks of data. They are often used in the context of databases to indicate a stream of binary information considered by the database system as meaningless. BLOBs can become objects, provided that they are associated with a class and methods. These can be part of the BLOB itself, but they have then to be bound with the rest of the programme. This is the topic of the next paragraph.

Dynamic Binding

Dynamic binding is a feature offered by most object-oriented languages, and also in some traditional languages. It is based on the assumption that classes can be dynamically created, and that some methods can not be known when the programme is compiled. Therefore, it is necessary to acquire the methods at the time they are called. This is called late, or dynamic, binding. Some authors consider that object-oriented languages must support dynamic binding, but the overhead this causes during the execution for the look-up of

functions, the retrieval and binding might be unacceptable in certain applications. On the other hand, dynamic binding allows smaller executables and easy upgrade of programmes, because the executable does not have to be recompiled.

3.2 Object-Oriented Database Systems

In this section, we present the issues in object-oriented databases. Before we introduce them, we would like to point out a common confusion between the terms "object-oriented database" and "database of objects". They refer to two different things that have nothing in common. The confusion actually originates in the term "database" which is used either as "database system" or as "contents of the database". An object-oriented database is a *database system* which can catch the meaning of objects, but does not necessarily store objects, although the natural entity manipulated by such a database is the object. We would say that the implementation or the conceptual model is object-oriented, whereas the physical model is not necessarily so. On the other hand, a database of objects is a database whose *contents* is only made up of objects. At the physical level, data are objects, but not necessarily at the implementation or conceptual level. A database of objects can be managed by a programme which may not be object-oriented, but which stores objects in its database. It is possible to combine both to have an object-oriented database of object; or we can just build a non-object-oriented database of objects or an object-oriented database of traditional data, etc. We consider here object-oriented databases, because we are interested in using the object-oriented paradigm for applications programmes.

Many attempts have been made in the literature to define the capabilities of object-oriented systems [1, 7]. We are in a situation where there is no emerging standard for object-oriented systems, unlike relational DBMSs which are all based on the definitions given by Codd before their appearance [21, 22, 23]. We shall present in this section what seem to be the most commonly agreed upon capabilities of object-oriented systems and we will illustrate some of them with examples of existing implementations. This description is important in that it gave us the main directions of development for the implementation stage of our system. As with relational databases, it

appears more relevant to us to analyse the state of the art in object-oriented technology after we have presented the background notions in multimedia.

3.2.1 Persistence

Persistence is known as the fact of keeping track of data throughout the life of an executing entity. There are several levels of persistence, just as there are several levels of executing entities in a programme:

Procedure: A procedure is the smallest executing entity which has its own data. Their scope is the procedure itself. Data (or objects) are created at the beginning of the procedure and destroyed at the end.

Transaction: A transaction is a set of database operations which is considered by the database management system as a single operation. This allows better control of integrity since, for the DBMS, a transaction is either totally executed or not executed at all. A transaction thus possesses its own workspace in which objects are created or deleted when the transaction commits or aborts.

Session: A session is the normal frame of execution of transactions, and the largest executing entity. A user initiates a session and terminates it when he/she logs off. Many objects are created, such as windows, and other data are used to specifiy options for the duration of the session. These objects and data must be recreated for each session.

Permanent Objects: Finally, objects persisting through different sessions and persisting even through system and disk failure are permanent objects. Their expectancy is theoretically infinite, unless their owner decides to discard them from the database.

The implementation of persistence varies with the models, but we distinguish basically two large categories called by Khoshafian and Abnous "persistent extensions" and "persistence through reachability", which are not orthogonal [1].

An *extension* is defined as a group of data of the same type. Therefore, as soon as one creates an object of type T, it belongs to the extension of T

and becomes a permanent object. In object-oriented languages, an extension is usually a class associated with a method to traverse all the objects created in that class. Persistent extensions require particular language primitives for iteration and selection. The problem of extensions is that there is no clear difference between the type of an object, its class and its extension. The example of O_2 is probably the most interesting of all examples because it distinguishes a class from its extension [31, 48].

More often than not, modern or widespread object-oriented languages do not consider classes as extensions for their objects, *i.e.* classes do not keep track of their objects. Objects, however, do keep track of the objects they are composed of. Thus, the principle of persistence through reachability is that if an object belongs to a persistent object, it becomes persistent too. Therefore, the character of persistence follows the hierarchy of objects, independently of the classes and extensions. There must be, of course, a root object at the top of the tree which is artificially declared persistent.

3.2.2 Transactions

A transaction is an execution entity which has the property of being considered by the programme as only one atomic instruction. If it is interrupted in the course of its execution, the state of the programme is restored to what was backed up before the first instruction of the transaction (rollback). Otherwise, when the transaction finishes, changes are said to be committed. Different implementations generally agree on the definition of transactions and in the type of operations they allow. But new propotypes, as well as some commercial products, accept several levels of transactions: a transaction may be composed of several different sub-transactions. It commits, if its sub-transactions either commit or abort. But more importantly, the scope of updates is visible only in the parent transaction, until the parent transaction itself commits.

The implementation of transactions is not difficult unless we take into consideration that very large data can be modified in one transaction. The solution that is generally proposed, for the problem of nested transactions, is to grant each transaction its private workspace, which is then copied into the private workspace of the upper level of transaction and eventually in the

database itself. This is a good solution for traditional transactions which typically involve reasonably sized sets of small data. We will see in Section 7.1.4 how we implemented transaction control for large data types.

3.2.3 Concurrency Control

Typically, transactions occur concurrently, and it is likely that two or more of them will try to update the same data at the same time. There exist many variations in the implementation of concurrency control which we can roughly classify in three categories. More details can be found for algorithms specific to object-oriented databases in the bibliography [56, 1].

Time-Stamping: Each transaction receives a time-stamp, and transactions are executed in the temporal order based on the time-stamp values. If an early but long transaction happens to modify some chunks of data already updated by an older but shorter transaction that already committed, the younger one cannot commit. The problem with this solution is to associate a time-stamp to each data. This processing is often cumbersome and the extra information greatly hinders the performance.

Optimistic Methods: Some applications do not require a great number of simultaneous transactions. In addition, these transactions very rarely conflict. In such cases, it is better to let the transaction work in a private workspace and then copy it into the database. If the data that were modified have been updated since the beginning of the transaction by some other transaction, the current transaction aborts. These methods are called "optimistic" because they work based on the assumption that no conflict will occur. However, if a transaction must abort, all its work is lost. Such algorithms are, therefore, charaterised by a greater cost, in terms of computations and checking, in cases of collisions between transactions, but low cost in general cases.

Pessimistic Methods: Just as the name of the methods suggests it, these are best used when the chance of two transactions modifying the same data is great. The protection of data is done by locks: each piece of data updated by a transactions must be locked by the transaction

from beginning to the time it commits. Locks cannot be broken, but the transaction holding them releases them after it terminates. This algorithm is therefore extremely safe, but carries with it the danger of potential deadlocks. The implementation of pessimistic methods must thus include a special part dedicated to deadlock detection in order to abort some transactions.

Determining the right method is a difficult task where parameters such as the available memory, the size of the storage space, the size of data, the load of the system and application-dependent characteristics must be accounted for. Therefore, some systems provide user-definable parameters to the concurrency controls algorithms. In ORION for example, locks apply to a set of data, but they have variable size in order to achieve the best balance between the overhead necessary (to apply the lock) and the level of concurrency which depends on the granularity of the lock [9, 8, 11]. GemStone offers the possibility to choose whether data have an optimistic concurrency control or a pessimistic one [51, 50]. This is not a database-wide option and can be refined for each attribute of an object.

3.2.4 Recovery

The role of a database is to provide applications with persistency. This means not only that data have to survive the applications, but also that the state of a database is not changed by the various types of crashes a computer experiences. We distinguish two orders of object's remanence, as described by Thatte [59]: the *persistence* is the ability of an object to survive the application that created it. The *resilience* goes further in that objects can survive process, system and media failure. We usually represent both these meanings with the terms of "persistence" or "persistent object", unless specified. Unlike other capabilities provided by databases, the problems of recovery from failure do not offer many solutions. Those described in GemStone are a good example [51, 50].

The lowest degree of failure, the transaction failure, is in the scope of the primitive meaning of *persistence* as Thatte defined it. If, indeed, a transaction fails, the newly created objects cannot survive the application. Generally, the recovery procedure is part of the transaction manager. For simple

transactions, the best solution is often to use a private workspace in memory, after the data that will be updated have been written on the secondary storage. This is certainly not enough as transactions become more and more complicated. Since memory is a scarce resource (rapidly insufficient), the main solution to the problem of transaction failure is the log file. In a log file, the transaction records the image of data before and/or after the modifications (some algorithms need both images, while others need only one).

Other faults relate to the resilience of data. First of all, a process failure happens when a process is suddenly interrupted while in the middle of its operation. Second, the operating system itself may be subject to breakdown. Again, log files are a convenient way of gracefully recovering from these two kinds of faults. Last, the media failure is a defect of the storage unit itself. There is no other solution to this problem than to copy the database to another physical storage unit (or to another partition in case of hard disks) and to force transactions to modify the two database images before they commit.

3.2.5 Querying

The query language is the normal way for database users to select specific information. It must offer high-level syntax, and let the user concentrate on what he/she wants from the database instead of how to get it. As we said earlier, SQL is now a standard query language for relational databases. Because of the difference of views of the data between the user and the DBMS in relational systems, the query is converted from the upper level (conceptual level) to the physical level, and then data are transferred across these levels back to the user [41, p. 232–236 and 285–288]. Bertino *et al.* [13], Khoshafian *et al.* [1] as well as Zdonik *et al.* [52] provide detailed information about querying in object-oriented databases (see also [6, 34, 38, 50, 62]).

Since the explosion of object-oriented or extended relational systems, the rigid frame of SQL no longer suffices. Some research groups try to unify the physical layer with the conceptual layer, and also to reduce the gap between the host programming language of external applications and the database query language. Others try to stick to the three-layer representation of relational systems and the SQL standard. This generally leads to three developments, two of which are strongly bound to the implementation method: the

first is used in systems that are true object-oriented DBMSs, while the other is characteristic of an implementation consisting of an object-oriented shell atop a relational system.

- Some implementations, like O_2, VOOD or GemStone create a new query language from scratch, with its own syntax and its own concepts [48, 31, 62]. In particular, because of the use of object identity, the notion of equality is changed. Objects can be equal because their attributes have the same value, or they can be equal because they *share* the same value. Moreover, objects can be just equal or equal "in depth", in which case the sub-objects are also compared. We find the same difference in LISP.

- ORION [11, 9], Iris [12, 40], AIM [27], Postgres [42] or VBASE [6] give several examples of extensions to SQL which are not necessarily object-oriented extensions. These extensions are either an addition of set manipulators (by "set" we mean set, list, tuples, aggregation or other types having more than one atomic entity), or aggregate types constructors, or function definitions or predicate manipulations, so that the language is eventually closer to a programming language than to the original SQL, which was specifically designed to allow people without any programming background to use a database.

- Interactive querying is another kind of "query language". It is used in addition to one of the two implementations above, or simply to replace them, like in Intermedia [32]. However, interactive query is not suitable for all kinds of application and general purpose DBMSs should never implement it alone, without support of a good query language or, better, of a good programming interface.

Adding an interactive query system might be an interesting idea, but such an interface may be difficult to integrate into larger softwares whose purposes are not primarily centred on the database. Nevertheless, experienced database designers forecast the use of SQL-based data for at least another five to ten years [10, 57], so the current trend is to follow the path of SQL. We wonder, however, if this will not lead to another sort of standard in query languages, designed more to comply to industrial demand than to the real requirements of the models.

3.2.6 Integrity

Because of the variety of data types, the issues of integrity in object-oriented databases encompasses a much wider set of problems than in relational systems. There are five levels of integrity [1]:

1. **Uniqueness** refers to the fact that two objects in the database must not be equal. We saw earlier in this chapter that there are several types of equality. In general, and particularly in true object-oriented databases (as opposed to relational-based systems), equality is by pointer, or object identifier. Uniqueness was already very well implemented in relational DBMSs and nothing new has been done since.

2. **Object pointers** (or identifiers) must be checked so that they do not refer to an object that no longer exists or is out of date (*e.g.* in systems supporting versioning). This is the weakness of relational systems and the solution often has to be implemented in their applications. In object-oriented systems, this problem does not exist with some identifiers called *surrogates*, as Khoshafian *et al.* call them [1, 25, 26]. The same authors also remark that the use of surrogates totally eliminates the need for referential integrity checking. The use of garbage collection is another way of dealing with the problem caused by the deletion of an object: an object is only removed from the database when no other object points to it.

3. Another feature is frequently well implemented in relational databases, namely **non-null constraint enforcement.** This was created by the need for tuples to have a key in order to accelerate data location. In object-oriented databases, the need for indexes does not disappear, though it is less important. Different algorithms are implemented, but since this operation is never a problem, we will not detail them.

4. More interesting are the **domain constraints.** These express the fact that some attributes have a limited set of value (the age of someone cannot be negative, or more than about 150). There is much to say on this topic, but we will just make two remarks. First, some relational systems already implement this operation. But the fact that

they manipulate only "classic" data types makes it simpler, if not simple. Domain constraints on objects are much more difficult. Second, they generally express constraints of the real world. We thus know (for objects are a good attempt to model the real world) that it would not be wise to implement this constraint in the database, because it does not always catch the meaning of objects as well as they do it themselves. Therefore, domain constraints are better enforced from within the objects than from the outside.

5. Finally, all the other integrity constraints that do not fit in the previous categories are called "**general integrity constraints**". Like the domain constraints, they reflect the real world's restrictions. They had to be implemented in the applications using relational systems, but object-oriented applications do not have to take them into account. The objects themselves can take care of it, and in a much more efficient way thanks to encapsulation.

Integrity checking is, therefore, a very simple operation in object-oriented databases, because most of the job is done by the objects themselves. The other almost trivial enforcements are already very well implemented in relational DBMSs, and there are currently no development or emerging research for new methods.

3.2.7 Performance

Performance is an important issue in object-oriented databases, and in his presentation of such systems, Bancilhon concludes that although they are a good step forward following the relational step, the performance problem is still one of the four most important issue to be solved soon [10]. We can say, to the credit of object-oriented DBMSs, that relational systems raised the same problems when they first appeared in the seventies. Performance can be improved in three domains, as explained hereafter [1]. We will make only a brief overview of the existing issues, because this expertise is not relevant to our purpose. More details can be found in the literature (see [26, 28] for index problems, [17, 20] about storage management and [11] regarding query optimization).

Indexes and Accelerators

As we just said, the use of object identifiers does not make indexes obsolete. Objects can still be searched for some specific characteristics, as tuples are. Indexing is a complex problem in relational databases, especially at the physical level, but it grows in complexity with object-oriented systems because of the variety of data types and storage devices. Indexes must also cope with the problem of equality and identity.

The difficulty of index management is overcome by specifying the index at the time the extension is specified (whether it is a type, a class or a hierarchy). But some systems allow for very complex index specifications [51, 50]. The indexes are generally implemented as B-trees [27], or similar structures (there are a number of variants) for classic data types. AIM's index manager has a special part dedicated to text indexes, and some hypertext or hypermedia systems include also special indexing elements [27, 1]. However, a good general-purpose indexing technique that could be used for any kind of data does not exist.

Storage Management

Storage management, which is still improving for relational databases, is more and more difficult because of the existence of BLOBs. Adding to this complexity are the facts that: the structure of objects is very irregular, and objects are nested in one another and reference each other in a non-predictable way. Abnous *et al.* give a summary of these issues [1], and Valduriez *et al.* give some detailed results of various algorithms [26].

1. The main problem addressed by storage managers is to convert structure of objects and data into disk pages (or into an atomic element of storage for other devices than hard disks), because a page is the ultimate entity (in this respect, the DBMS can be seen as a wire with a converter in the middle and two different ends: one that can receive or send objects, and the other that can handle fixed blocks of data).

2. In solving the problem just stated above, the storage manager may find it useful to use some auxiliary data or structures, such as indexes or accelerators, as explained in the previous section. In this respect,

this extra information is application-dependent, and no important steps were taken since pure relational DBMS appeared. It is indeed still up to the database designer to acquire the knowledge about the way the database works at the physical level, and to deduct, according to the goals of the application, what extra information should be given to the storage manager to optimize its operation.

3. This "solution" also applies to the problem of clustering, which defines how data are grouped on disk (or any other storage device) for better efficiency. It is very similar to the problem of indexing: indexing data and grouping them on the same page, if they are often accessed together, are two operations based on statistical information. The storage manager, on some systems, may analyze the statistics of data access and decide to create an index or to cluster some objects. This is called "dynamic strategies". It seems very attractive but the parameters, such as the threshold values for decision making, cannot be fixed without a lot of experience or extremely careful computation. In static strategies, the DDL should be precise enough to allow the database designer to choose between various strategies (GemStone is a good example [51, 50]).

Query Optimization

There are two goals in query optimization: to minimize accesses to storage units, and to minimize CPU usage for executing the query [1]. For this, the query optimizer uses three indicators: statistical data such as the number of elements in a given extension, their correlation and so on; clustering information and location of objects related to each other; and information on the indexes. Based on these indicators, the query optimizer chooses between different join algorithms (there is an important number of proposed algorithms because the join is a very expensive operation). It is sometimes necessary to fragment the objects and operate a traditional query optimization of the decomposed object, and to reconstruct it after the query is performed [24].

Optimization in general, and query optimization in particular, is an extremely complex problem. We have presented the three main issues, but research is very active in others fields such as caching algorithms. The problems are far more complex than in relational databases. These already have

efficient optimization methods and we will show in the next chapters how we intend to benefit from them.

3.2.8 Other Capabilities

Two other important topics were not yet addressed, namely: the version control and security. This last feature is not very relevant in our case, because we do not intend to build a real DBMS, but this is not the main reason. Current systems do not have a lot of flexibility regarding security. Users are granted privileges that correspond to the operations they can perform. There are basically four classes of privileges: retrieve, update, insert data into table, and the DBA privilege (may do everything, including grant privileges to other users). Let us recall the view of Tsichritzis and Nierstrasz about objects protection [54, p. 532–533]:

> It is improper to talk about *protection* of objects. Dynamic, moving objects are not passive units that need protection. They can actively organize their own defense. An object can refuse to allow certain operations. It can hide information. It can provide disinformation. It can move outside a context where it cannot be reached. It can completely erase itself. An object can even defend itself by directly or indirectly attacking the intruding object. All these possibilities give protection mechanisms a new meaning.

According to Tsichritzis and Nierstrasz, the protection of objects, like the maintenance of integrity constraints should be done from within the objects. This would allow more powerful and adaptable protection to the context, while freeing the database from a role it does not have to play. This is the chief reason why we do not implement protection.

Finally, we present the problem of version control. The literature proposes several solutions for versioning [14], but no real system outlines the future trends of versioning systems. However, some implementations propose that a dedicated and independent class based on tree manipulation primitives should allow versioning. This solution seems the most adapted to the varying situations a DBMS may experience. Therefore, it seems more reasonable to implement versioning from the outside of the database system, instead of trying to make it a fixed feature of the database manager.

3.3 Conclusions

In this section, we intended to clarify the concepts of the object-oriented approach, and also to point out the capabilities of databases of objects. We showed various implementations derived from relational technology in order for an object-oriented database to provide the following capabilities: persistence of objects, transaction and concurrency control, recovery from failures, querying, enforcing integrity, optimize storage management, query and indexation, and ensure proper protection and version control.

This is a very basic overview, the purpose of which is to set the direction of our research. Much more could be said because of the complexity of the problems, which is mainly due to the variety of data, data types and to the nature of the object-oriented paradigm itself. We observed that relational systems have generally found a solution to most of the problems, and that other difficulties are solved by transferring the implementation into the objects (in particular, the requirements of multimedia applications are better fulfilled by the object-oriented paradigm than by any other). In such a case, the solution found is often simpler and more flexible than that proposed by pure relational DBMSs. Our work will be to solve the remaining difficulties, using what appears to us to be the most appropriate solutions.

Chapter 4

Multimedia and Databases

As we stated in the introduction, the term of "multimedia" is very loosely defined in the literature. Its signification extends from the simple association of text, graphics and sometimes voice, to anything containing a piece of information regardless of its form [15, 53]. The boundary between object-oriented database systems and multimedia database systems is not clear at all, because implementation is often done with both ideas in mind. The goal of this chapter is to present the state of the art in multimedia or object-oriented databases. It is organized as follows: we will first demonstrate how far relational and object-oriented database systems fulfill the requirements of multimedia data management. This will show that they are complementary and that one system alone does not provide a satisfying answer. Then we will show how current research tries to overcome this difficulty with the example of three different implementations summarizing the current trends. These have been a source of inspiration for our data model and implementation described in the following chapters.

4.1 Overview of Relational Database Systems

Although they are known to be less adapted to multimedia data management than to traditional "business" data, it is still reasonable to attempt to use

the relational experience for multimedia applications. The question arises: "what, in relational databases, can be used for multimedia data management, and what features should be added or changed?" In this section, we will answer the first part of this question.

4.1.1 Strengths of Relational Database Systems

Standardization

First, standardization plays a major role because it is the balanced achievement of several extreme solutions and often the best choice when crucial dilemmas arise. Standardization concerns, in particular, the languages or the implementation model. Products related to database management systems are highly standardized. This is a sign of their maturity and reliability. Standardization enhances production rates, while reducing risks of misbehaviour. In a multimedia environment, because the number of devices and data exchange is much more important than in a traditional "business" application, it is important to achieve a good standardization at every level. This is already provided by relational database systems.

Normal Forms

The role of the normal forms is to minimize the replication of data and external references (across tables). Consequently, there are fewer possible errors in the operations of data creation, modification or deletion. Normal forms are thus a support to efficient representations. They also allow easy clustering and caching and therefore are the base for compact data storage as well as fast data manipulations (insertion, update or removal).

Interface

The data definition and data manipulation languages are both encompassed in SQL, which became a *de facto* standard after IBM introduced it with DB2. It is also now the basis of the X3H2 database standardization committee. Among the qualities of SQL, one can stress the ease of use of this language for the database administrator, the application developers, and the final users.

Special Features of Relational Databases

Relational databases provide efficient use of indices. This feature is artificially implemented in SQL to facilitate the use of primary and secondary keys. Efficient indexing is very difficult to achieve. However, after several years of intensive research, most of the problems are now well mastered. This operation has been made possible by the use of the first normal form, wich requires that each complex entity is decomposed into its atomic components. The process of atomisation of data allows applications to use rigidly structured representations. Therefore, only simple data types are encountered in a relational database. The process actually goes further because the size of an element is also fixed for the whole table. Since all data are stored in a fixed format, the access time is optimized. This makes seeking addresses easily calculable. For this reason, both indexing and seeking are very fast operations.

4.1.2 Problems Related to Relational Database Systems

First Normal Form

From a general point of view, relational systems suffer from those features that make them most efficient: the main problems come from the requirements of the normal forms, and in particular the first normal form.

Data are flattened into tables and lose their structure. Modern programming languages use different kinds of structures and aggregate types, as opposed to simple data types such as integers and strings. There is, therefore, an important gap between the way the relational model works and the way the applications handle the data. More importantly, the absence of structured data prevents the use of pointers. In other words, the use of structures and aggregate types is often combined with references by pointer (instead of giving the value of a structure, its address is used in order to minimize the transfer of data), but such reference disappears in a relational database.

Referential Integrity

The decomposition into atomic entities and the separation of objects into different tables also prevent us from expressing the semantics of complex entities. This is a very important disadvantage of relational DBMSs [35]. As a direct consequence of the lack of any semantic information associated with the data, complex integrity checking must be implemented in the application rather than at the database level. This is a major problem in large applications, because the normalisation process forces us to use external references across a large number of tables.

Performance

Finally, normal forms indirectly lead to problems in terms of performance. They require that the data be broken up into different tables, but these tables will eventually be merged again by an operation called *join*. In a large system, join operations are very long, because of the increasing number of tables.

"Frozen Configuration"

In addition to the problems caused by the normal forms, relational systems are unable to handle new data types created by the user. For this reason, they are not open to changes in data types, or in media types. Nor are they able to handle different storage devices. A relational database is configured to use a magnetic hard disk, with relatively low access time (compared to other devices), and the possiblity to rewrite data as many times as necessary. Managing WORMs or magnetic tapes such as backup devices or VCRs is problematic. Moreover, controlling different devices at the same time is not possible, unless under the assumption that there is only one device per computer (in which case we have a distributed database).

Language Mismatch

We also have to take into account other disadvantages, such as "language mismatch" [1, 19, 33]: there is a strong difference between what the query

language can do and what the host language can do. While the first is very specialized and oriented towards efficient data retrieval, the latter is a multi-purpose tool. Unifying these two concepts is often difficult and artificial. Another important drawback lies in the links between entities: they are made with matching values of a field. This forces the use of unnatural joins, and cannot be used for creating new kinds of relationships such as the one described above.

4.1.3 Summary

In this section, we showed that relational systems take advantage of their long life and experience to provide efficiency and reliability, while object-oriented systems are not yet able to compete in these domains, as we will see next. In addition, the normal form theory is a strong support to an easy design stage. But it is important to note that although their strengths are well adapted to the manipulation of traditional business data, they become weaknesses when such systems are used for multimedia data management. From this point of view, object-oriented DBMSs have many advantages, as we shall now see.

4.2 Object-Oriented Database Systems

After describing some of the major points of relational systems, we now look at the capabilities of object-oriented systems in the same fields. Most of the benefits are related to the object-oriented approach in general, rather than to object-oriented DBMSs themselves, and complement the weaknesses of relational systems.

4.2.1 Major Drawbacks

Lack of Maturity and Standardization

Object-oriented systems are barely emerging from laboratories and are neither mature nor very reliable [43]. These products are too young for standardization to take place. In particular, there is no standard query language [57].

The DDL and DML as defined to date remain incomplete or very difficult to use. Some companies replace this lack of a good query language with a database browser. This is in fact an application, and not part of the database itself; thus it is not really useful for other purposes and cannot be easily included in a foreign application. The same is true for object-oriented languages in general: although C++ appears to be a *de facto* standard, there is currently no ANSI standard emerging [33].

Difficulty of Design

Unlike the relational databases, object-oriented systems have no theoretical bases [43]. It is more relevant in this case to consider the design of such a database almost as an art: feelings and experience take the place of methodology, and several iterations are necessary before the right model is found [47]. There is currently no way of strictly organizing data, because of the inheritance mechanism. Class design becomes therefore a very difficult task. In addition, the number of possible solutions is very large in comparison with that of relational design, whereas the number of "good" solutions (in the sense that they provide at the same time good performance and little duplication of data) is not significantly larger.

One may also object that programmers have to learn an extensive class library before being able to programme. In traditional programming, re-using the basic entity (the procedure) is not straightforward and it must often be adapted, resulting in a loss of time. In object-oriented programming, however, re-using the base building block (the class) is easy and immediate. So the overhead of learning a class library soon becomes negligible. Once again, this disadvantage is minor and, in the long term, is not an obstacle to the development of object-oriented programmes.

Performance

Because of their young age, object-oriented systems are not designed for speed and efficiency [43]: the main concern of the designers is first and foremost to make something that works. Although performance of object-oriented DBMSs capable of clustering and caching are expected to be about a

hundred times better than that of relational systems (especially when data complexity increases [33]) such efficiency has not yet been achieved [1].

Because of inheritance, data are not replicated throughout objects having the same classes or superclasses. The non-replication of data saves storage space, but may greatly slow down the retrieval or updating process, because it prevents the database system from using efficient clustering or caching.

One can generally notice that object-oriented databases or programming languages are slower than their classical counterparts. There are two main reasons for this: (1) late binding; each time a new object is accessed, the code of its method must be searched on the mass storage and bound to the running programme, and (2) polymorphism added to multiple inheritance slows the process of reaching the right method. The resolution mechanism must search the class tree, which may sometimes be very large. One must not exaggerate the loss of performance and there is good hope that this will not be a problem in the future: the first compilers were very inefficient compared to direct assembly programming, but nobody uses assembly language any more. A good sign of the evolution of object-oriented languages is the use of C++ by AT&T and Sun Microsystems for their future versions of Unix [33].

"Semantic Relativism" and Indexation

Two difficulties are inherent in the object-oriented approach: having different views of the same data, and indexation. First, object-oriented systems lack "semantic relativism" [45], because they tend to force the users to follow built-in links within data [60]. It is very difficult to index objects, because of the variety of their content. Similarly, caching is also a problem for database designers: should the objects belonging to the same class be grouped together, or should only the related objects be clustered, independently of their classes? Moreover, given the time needed to retrieve a 1 Mb large record, caching or clustering is probably not as relevant as with smaller pieces of data.

Lack of Relational Interface

Finally, let us quote F. Bancilhon, who expresses a point of view shared by many [10, 57].

> The absence of relational interface on object-oriented databases is a drawback. It might seem strange to mention this as a problem. However from an industrial point of view it is certainly an issue. In the years to come, relational systems will keep increasing their share of the market. SQL is becoming a *de facto* standard. It will certainly become a standard for exchange of data between heterogeneous systems. So, even if objects take the world, we will keep exchanging data between systems in relational form.

Even if the development of pure object-oriented DBMSs is important from a research standpoint, such systems have probably no industrial future.

4.2.2 Benefits of Object-Orientation

When something new in the field of computers comes up, many people presume that it is much better than anything that existed before, and that all the old problems have been solved. In short, they say that the new system is a panacea. However, we saw above that the weaknesses of object-oriented systems are such that one cannot say it is the ultimate and perfect tool for everything. But what was until now no more than a programming method [16] now applies to any field of computer engineering. It is important to notice the strengths of the object-oriented approach (they are very well described by Winblad *et al.* [33]), all the more so, since they give a very good answer to the problems of multimedia data management which we defined before. On this topic, more can be found in some books and articles included in the bibliography [50, 10, 1, 33].

Software Production

From the viewpoint of software production, the object-oriented approach brings many solutions and is an extremely important step towards fast, efficient and clean development of any kind of application: there are as many differences between object-oriented and procedural programming as between procedural and sequential programming. In addition to all the benefits of

procedural programming, the object-oriented approach increases the flexibility, the reusability, the extensibility and the maintainability of an application, and therefore reduces development costs. We know for example that about 80% of the cost of a large application consists of pure maintenance (this is the result of a survey by the U.S. Department of Defense of its major software production in the late seventies [4]), and that reusing an already written code has always been a problem until ADA first appeared. The object-oriented design methodology thus saves time at several stages of the life-cycle of a software: the creation, because of improved reusability; the testing phase, because we are using already tested classes; the maintenance phase, because changes happen in just one place of the programme (in procedural languages, one generally considers that a change in one function implies a number of other changes, which grows exponentially with the total number of procedures in the software).

Model

The decomposition of a problem into objects rather than into tasks is a much more natural decomposition. Thus, the representation at the conceptual level maps better the real-world organization than other models. Moreover, there is a full correspondence between the conceptual level and the logical level: both use objects and methods to represent a solution to a problem. This one-to-one relationship allows faster and more reliable design.

Types

We saw at the beginning of this chapter that multimedia applications should be able to accept very easily new types of data. Object-oriented systems are the perfect answer to this requirement. They allow any new class to be attached to the hierarchy of existing classes. By doing this, one can virtually add any kind of media, with its own displaying method (by overriding the superclasses' displaying method) and with its own characteristics. In addition, object relationships are maintained by direct reference. For this reason, object-oriented systems can manipulate complex data sets as simply as traditional data types. This explains the superiority of the object-oriented approach regarding multimedia data management.

Database Capabilities of Object-Oriented DBMSs

Object-oriented databases have numerous capabilities which can be organized into four categories [33, 1, 10]:

Database Integrity. Direct references between objects as well as encapsulation are two strong elements which permit both extended constraint checking and database integrity maintenance (external references from one object to another). This last point, which is relatively difficult to achieve in large relational databases without extensive programming in the application, can be automatically implemented in the object-oriented database manager.

Sharing. Object sharing is one of the major aims of an object database. Direct references from one object to another prevent complex updating problems. In addition, concurrency control is very easy because, unlike relational systems where locks apply in several places in the database, the lock only applies to one object, to one location where all the relevant data are stored.

Performance. Memory consumption is reduced to the minimum: with dynamic binding, objects are loaded only when needed and can be discarded afterwards. They do not stay in the memory more than the time necessary for them to be processed. This exchange of information is a penalty for the execution rate of a programme, but considering that too large a memory forces the process to be constantly swapped in and out, achieving the balance between the amount of memory and the data transfer is expected to eventually improve the performance. However, in some circumstances, dynamic binding may be time consuming.

Interfaces. Two improvements are included in this term. The first one relates to the user interface, which will greatly benefit from the object-oriented approach. Since the objects contain the method to display themselves, many operations that were part of the database client are now written in the object. This makes interfacing easier, which is important for multimedia applications, because they tend to be more and more interactive. Secondly, the interface with the core of the application itself is eased by reducing the

"language mismatch" between the query and the programming language.

In addition to these improvements, the interface between the different levels of design is simplified. The class hierarchy can indeed be viewed as a particular type of net. Each node represents a different level of abstraction of an object. Therefore, unlike the relational systems, the object-oriented approach provides good abstraction capabilities. They allow the views of the users and the implementation model to match more closely. This unifies the three levels of design (conceptual, logical and physical) and simplifies the problems of views, provided that they differ only by the abstraction level of the same objects.

4.2.3 Summary

Object-oriented systems present many advantages from the programmer's standpoint because of their ease and security of use. Object-oriented design and programming, along with the use of class libraries, are an excellent tool for quick and clean software building. Object-oriented databases are particularly well suited to multimedia applications because of their ability to handle large data, to accept new types of data of varying sizes, and to be easily linked to user interfaces.

On the other hand, they suffer from bad performance, lack of standard (there is no agreement, currently, on the definition of an object, of a query and data definition language, or even on a standard object-oriented language), and their young age. Hopefully, from the numerous proposals and systems that are being created by research teams, one or two major products will emerge through a kind of Darwinian selection, and they will set a standard.

4.3 Trends in Database Research

There are very few examples of multimedia databases in the literature. As Sheth notes [55], a large amount of research and development in database domains is oriented towards one goal: constructing extended DBMSs which

allow efficient representation of unstructured data. Among these are general object-oriented DBMSs. There is a very good reason why few laboratories try to implement a true multimedia system: until now and for a long time to come, a DBMS has not been able to control more than one area of disk and more than one filesystem. Multimedia document handling requires that several storage devices be managed by the DBMS.

Even though they do not move directly towards multimedia databases, the database systems evolve at a regular pace: every ten years, researchers come up with a new model. In the '70s, the network model (DBTG) improved on the hierarchical model. The '80s saw the appearance and full development of relational databases. It is not clear now what the model of the '90s will be, but multimedia applications require more that the relational systems can offer. Researchers are taking three main directions. First, they are extending the relation model by suppressing the first normal form constraint (normal forms are presented in Section 4.1.1, page 31). Second, they are extending the DDL and the DML to support new types of data and associated features, but the core system is still relational. Third, they are starting a new DBMS from scratch. This can be a knowledge base, an object-oriented database or many other things. Following, is the description of one representative from each direction.

4.3.1 The AIM project

The AIM project is an IBM initiative to determine and fulfill the requirements of data management for multimedia data [27]. Their approach is based on the idea expressed by Bancilhon that the database world will stay with relational-based data [10]. So they have studied the minimum set of functionalities to add to relational systems as defined by Codd [21] in order to achieve "media polyvalence" (*i.e.* adaptable to many media).

Their first observation is that there is generally one specialised system (in the large acceptance of the term, because it is not always a *database management* system) per type of application programme. Unifying the storage of data for all these types of applications has been achieved by modifying the data model used by the DBMS. Classic relational DBMSs consider that the smallest entity, the atom, is part of a tuple (row of data), which in turn

belongs to a relation (array of data). This hierarchical view of the data organization is not powerful enough to model real-world entities. In newer models, the strong limitation of the first normal form is overcome. Thus a tuple may contain another relation instead of just atoms. Although some systems have been developed with this model, Dadam and Linnemann added two types of data: lists and sets [27]. Their relation with tuples and atoms are the following: a tuple is made of lists or sets, and a list or set is made of either a list, a set or an atom. Using this data model, they can represent any real-world entity, still using the normal forms from the second to the fifth.

From an architectural standpoint, the database uses the client-server model. The client is slightly enhanced compared to that of a pure relational system. In particular, it has to manipulate complex entities. The server, on the other hand, is much more complicated. Basically, it has one additional layer which can handle complex entities and which passes their elements one by one to the lower layer. This last layer is also very different from its pure relational counterpart in that it must solve complex problems such as indexation and storage management for BLOBs (but not only).

With this system, IBM presents a smooth move to newer and more powerful database systems, still maintaining compatibility with older systems. However, they do not seem to give much importance to the concepts of the object-oriented approach, which many people now consider as very well suited to multimedia applications (see Section 4.2). Using object-oriented programmes with such a system enhances the language mismatch (see Section 4.1), and cannot be considered as a long-term solution for multimedia applications.

4.3.2 Relational-Based DBMSs

The area of research involving a new language with slightly changed databases seems to be very attractive if we consider the number of independent projects currently published. The three leaders are Ingres Corporation with its product Postgres, Servio Logic Corporation with GemStone, and HP with Iris. Postgres is similar to the AIM project, for neither of them takes into account the object-oriented paradigm. The main idea is to give attributes either a value or the result of a function. GemStone, on the contrary, is

fully object-oriented, because it is based on Smalltalk. But precisely for this reason, it is extremely slow and those new applications which are resource-demanding cannot feel at ease with such a product. Finally, Iris is the most interesting, because it is the only DBMS which we have found to be built with the thought that multimedia requires several storage devices under the control of only one system, and that it must be able to accept new devices.

The Iris data model uses types and objects, and since these types may contain functions and support inheritance, they correspond to what we call *classes* in the object-oriented vocabulary. Furthermore, they support encapsulation and overloading. Objects are identified with surrogates, which are independent of their classes, their location in memory, their values or states. As we will see later, it is the best type of identification (see Section 7.1.2 for more details). Similarly to Postgres, objects value can be accessed through function, with the difference that functions can be many-valued. The data organization is a very complicated tree with the object at the root. The atomic data types such as *integers*, *real*, or *binary* (unstructured large data) are only a small subset of all the leaves. Most important of all, Iris supports the use of "foreign functions", which are user-defined functions considered as a black box by the system, and which are added at compile time.

The architecture of Iris is somewhat simpler than that of IBM's AIM. It is still based on a client-server architecture, with a server using two separate parts: a *query translator* and a *query interpreter*. While the query translator is self-sufficient, the query interpreter calls the services of the *object manager* and the *cache manager* which in turn calls the *storage manager*. This last element is a true basic SQL interface (working on a table by table basis and not supporting joins), slightly modified to handle surrogates, and it gives direct access to the data which are organized in tables. On the user side, an object SQL allows the user to manipulate true objects and hides the underlying complex decomposition process. A graphical interface is also provided with a browser for object-by-object queries.

As we just saw, Iris fulfills most of the requirements of multimedia data management, and its full object-oriented features are a good step toward language integration with an object-oriented host language. Unfortunately, the lowest level Iris, namely the storage manager, is not able to handle tuples larger than 4 kbytes. This would badly penalise applications using voice (even under compressed format) or images (even small size).

4.3.3 Full Object-Oriented DBMSs: The Example of O_2

We describe here a good example of full object-oriented systems, the elements of which are all built from scratch [31, 48]. O_2 forms a complete database management system with an impressive set of various interfaces, from the user interface builder to the programming languages interfaces (it currently supports its own versions of Basic and C) and several debuggers.

The data model in O_2 is very simple: from top to bottom, each entity is represented by an object. Four special types are provided: tuples, lists, sets, and multimedia objects (which is just another name for unstructured byte streams of arbitrary length). For each of them, there is an appropriate storage algorithm as well as a clustering strategy.

Similarly, because there is no necessity for data representation conversion, the architecture of O_2 is very simple compared to that of the previous systems. It also uses the client-server architecture, with the difference that the client and the server are very akin. The former takes its data from memory, while the latter takes them from the disk. All the data exchange is done by a single element called the object manager, which is made of eight simple independent modules with self-explanatory names: the complex object manager, the message passing manager (transmits messages to the right objects), the transaction manager, the communication manager (dialogues with the clients or the server), a persistent manager acting like a garbage collector, the buffer manager (manages objects' transfers into memory), the clustering manager and the index manager.

The simplicity of this model has great advantages in terms of performance and software maintenance. However, being a full object-oriented system, it gathers all the drawbacks of the object-oriented approach (such as slow execution because of late binding) and loses the advantages brought by relational systems. In particular, this system has difficulties to physically organize many different data on the disk. It makes extensive use of memory transfers to remedy this organizational weakness and loses all the performance benefit of a simple architecture with few data transformations. In addition, using only one storage device, it does not support true multimedia applications.

4.4 Conclusions

We have seen in this chapter that although relational systems are attractive because of their maturity, their limitations make them unsuitable for direct management of multimedia data. The most obvious solution is to have the database clients do what the DBMS cannot do. This includes image compression and decompression, text formatting, VCR control, and many other things. The problems arising from this solution are less technical than related to what Winblad *et al.* call "software construction" [33]. The increasing number of media used in a single application leads to a complexity that has now reached the point where one cannot manage it: in many projects, the cost of the maintainance is an average of four times the cost of the development stage. On the other hand, while object-oriented programming solves many of these problems, it is still difficult to implement an efficient, fully object-oriented database. However, since the object-oriented approach is a natural way to model complex real-world phenomena, it seems to be the best starting point for multimedia applications as a programming methodology, and a relational database seems to be the best tool to keep track of permanent objects.

We then saw, through three examples, that no system is currently satisfactory. The closest to the requirements, as they are defined in Section 5.2, is certainly HP's Iris. In our system, we will try to take the best of each model presented. Clearly, the object-oriented approach is a requisite to multimedia applications. Therefore we require that the database handles objects. But we do not want, such as in O_2 to ignore all the advantages of relational systems, so we will use a relational database as low-level storage controller. Finally, we want to leave an opportunity to add different storage controllers for other storage devices. The simplest way is to present our interface as a library that will be mixed with the code of a base class for those particular objects that need special storage.

Chapter 5

Analysis of Requirements

The last three chapters were devoted to the review of basic concepts in multimedia, databases and object-oriented approach. We have now set the background for the design of our database model. The purpose of this chapter is, in the first place, to understand the complex requirements of bringing together the three concepts reviewed above. We describe in particular the requirements of multimedia applications. This is the result of observations based on our own experience of multimedia. From this point on, we will refer to our system by the names of "system", "interface".

Although it is widely used, the term "multimedia" has never been clearly defined. It takes, therefore, a different meaning for different people. In some articles, this term is described but never really defined [39]. It seems important to us to define it clearly. This is done in the first part of this chapter. The aim of the second part is to determine what a programmer can expect from our system. Finally, we analyse briefly what capabilities of relational DBMSs we will use, and what features of object-oriented systems are required for our system.

5.1 Definition

Human beings use different vectors of information to exchange their knowledge. Each of these vectors is adapted to one of our senses, and may be presented simultaneously and synchronously with others. A movie, for example,

presents an image and a soundtrack. The soundtrack is itself a compound of two different types of information vehicles: one is the voice and the other one is the music and/or the noise. Similarly, a written report uses typed words, drawings and graphs to shape ideas and knowledge. These are all *examples* of a medium. Let us now give a more formal definition of this term.

We observe first that a *medium* is a vehicle of information for humans. Therefore, it must be presented under a form that can be immediately understood. In the very large definition of Naffah [53], even a stream of bytes, or an electric numeric signal in a cable is a medium. He considers the data, but not the way they are presented to a user. Since for computers there is no difference between the types of data (they are all bytes), a medium is not only a certain type of data but also an output device *and* a method by which the computer "displays" (*i.e.* sends to a user) its data in a form humans can understand. In addition, the computer must acquire the data representing the information and provide an interface to manipulate them. Finally, it must control an appropriate storage device for that type of data. In our final definition, a medium is thus a set containing: an output device, a method of "displaying" its data, an input device, a storage unit and manipulation software that permits the control of the "display" operation and possibly the modification of the data. What uniquely distinguishes two media is the "displaying" method, even though input and output devices may be shared, as well as the storage and the manipulation software.

In a *multimedia* environment, we must have several methods of data conversion and possibly more than one output device (but this is not necessary [2]) and/or a storage unit. According to this definition, multimedia is not only, as some authors tend to write, the collection of text, image and voice. The media are in constant evolution and we should not limit ourselves to a static definition. As we will see in the next section, it is very important that multimedia systems remain open to the addition of a media that did not exist at the time the system was developed.

5.2 Functional Requirements

This database interface is intended for programmers who will implement multimedia applications. Therefore, the requirements are twofold: there is,

first, what we need to implement in order to use multimedia data and, second, what will be provided to the programmer. This section presents these two points in this order.

5.2.1 Multimedia Requirements

Since each medium has some specific requirements, we must first define the kind of media that we may have to use. These are currently text, static and animated graphics, bitmaps (animated or still) and full motion video, voice or sound. More generally, we can classify the media in three categories.

Small structured data. A multimedia system first has to deal with traditional "business" data such as names, currencies, dates and times. These data are generally small and suitable for organized storage in fixed width records. In addition, they are simple data types in the sense that they are not a structure of several atomic data.

Large structured data. Then, the system has to manipulate text and vector graphics. These are data blocks with an internal organization. For example, a book is composed of a title page, a table of contents, parts made of several chapters, and possibly an index, a bibliography, appendices, and a colophon. Each chapter is subdivided into subchapters, sections, subsections and finally paragraphs. In addition, a book often includes illustrations (pure raster image), tables (text organized in rows and columns) and figures (geometric forms and text).

Large unstructured data. Finally, raster graphics and sound are typical examples of large and unorganized data with various sizes that cannot fit into a single record without a great loss of performance: either we have huge records that handle almost any kind of document, or we break a document into small pieces that are stored in linked records. These last media may or may not contain an internal organization which the database is not able to see. For example, an image may be composed of many independent sub-images. Such a decomposition may be useful in animated graphics where, instead of redisplaying a whole new image each time, one decomposes the image into sets of pixels which have their own movement and are not transmitted for each new image.

This description shows that contrary to traditional office applications of the previous decade, the multimedia systems are based on a large variety of data. These data may be either small and atomic, or very large, in which case they may have an internal structure or no structure at all. But this list is not limited and we must keep in mind that one of the major requirements of a multimedia application, and thus of a multimedia database, is to be able to handle new types of data [33], which did not exist when the application was designed. In other words, they must be open to future extensions.

This brief categorisation of media allows us to set three requirements, two of which are derived from the definition of the term "multimedia". The third one, relationships between objects, deserves to be discussed further.

Relationships in Multimedia

In multimedia applications, objects may have complex inter-relationships. The database must be able to allow a document to reference another one, or even part of a document to reference part of another document (or another part in the same document). This is illustrated by the notion of "webs" described by Yankelovitch *et al.* [32]. In their example, relations have no particular meaning. They are the expression of a correlation between two Intermedia documents, which only the user is able to understand because it sometimes refers to a level of comprehension that the machine is unable to reach. The emphasis in future applications must therefore be put on the relationships between the entities that we manipulate. We list the most frequent types of relationships generally required by multimedia objects and applications:

- Some objects may need to be related to other objects for many reasons. A first important type of relationship is the composition: an object may be considered as a set of sub-objects. Therefore the object itself and its sub-objects are tied together with the "element of" and "composed of" relationships. These are particular to emerging applications and previous systems do not allow such relationships to take place easily.[1]

[1] Relational systems cannot have a good representation of such relations because they violate the normal forms. Although it is possible for a relation to be in first normal form

- In an environment where the appearance of elements (the operation of "displaying" them) is one of the most important operations, the temporal relationships between the different retrieval operations of related objects must be implemented in the database. For example, in a document composed of a video, a sound track, and a sequence of raster images, each element presents a relation of *simultaneity* with another one if it is to be "displayed" at the same time; or it presents a relation of *sequentiality* if it must wait for the other one to complete in order to appear. Such temporal relationships are hardly usable in classic systems. They could be represented in some way in the database, so that the objects are retrieved at the right time. In other words, it could be up to the database to make these objects available at the time they will be used.

Requirements of Multimedia Applications

We can now give our complete multimedia requirements:

Storage requirements. Data must be accessed from different storage devices.

Processing requirements. Data may need to be transparently processed, before they are written to their storage devices, or after they are read. Data processing must be dependent on the media that data represent, and not on the physical storage unit which they use.

Relationships. Data need to keep their relationships. The most important of these are the "composed of" and "component of" relationships. Additionally, temporal relationships must be kept, even if the database itself is not totally able to use them to synchronise objects.

---

only, it is not advisable in terms of performance. Regarding the maintainability of the database, breaking the normal forms is also dangerous.

5.2.2 Programming Requirements

The term of "progamming requirements" encompasses all the functions that a programmer will have to use in the database. They are listed in order of importance.

Ease of use. The main point is to provide programmers with an easy access to persistence. This includes reducing the language mismatch between the host programming language and the DML. He/she must also be able to take advantage of a flexible database system that will allow him/her to add other media. Finally, we want an increased security in software production.

Basic operations. These are the core functions of any database system. It should permit data to be exchanged in both directions with the database. This includes creation of data, retrieval, updating and deletion.

Data location. For all these operations, except creation of new data, we need to specify to what data they will apply. Therefore, we first need a way to specifically locate data in storage. The meaning of the term "location" is very large here: for the sake of data independence, we do not want to give a physical location, but an information that the computer can transform into physical address on the storage device. The best way to achieve this is through a language specifying the characteristics of the objects that we want to manipulate. This must be part of the host programming language in order to avoid language mismatch.

Concurrency. In addition to these basic operations, the database system must perform other tasks related to database management. The database must ensures transaction control. The functions for this are "begin transaction", "end transaction", "commit transaction" and "abort transaction". The problem of transaction control is bound to that of concurrency control, which is a research topic onto itself. Since we want to validate our model rather than implement a fully working database system, we will not consider the less necessary functions, such as undoing or redoing a transaction.

Recovery. The DBMS must support recovery control. We saw earlier that there are three levels of failure (see Section 3.2.4). The first two levels

are easily mastered with the use of log files. Media failure, however, needs data to be replicated to another physical location and possibly another device of the same type. We do not study this last issue.

5.3 Achievements

We discuss here the features of the relational database that we will use in order to achieve the goals specified in the previous section, as well as those of the object-oriented paradigm.

5.3.1 The Relational Features

The relational DBMSs are very convenient for many reasons. First of all, the relational database systems are efficient because of the many accelerators and "intelligent processing" (especially regarding queries) due to the sound mathematical foundation of the relational algebra. Every level of the database benefits from this efficiency, and in particular the implementation and physical levels. These are precisely where object-oriented database systems have difficulties. Therefore we will try to keep, as much as possible, these two levels in a relational form.

In addition to this, the relational databases provide all the necessary structure for indexing. Whenever possible, we will thus use relational indexing. As we have stated earlier, images or "meaningless" (at a low level) streams of bytes are difficult to index and we will not consider this case.

Basic operations, and features such as data location, are very well performed with SQL. Thus, we will use this language inside our database system. However, its limitations make it inappropriate for use in a wider context. We propose, therefore, our own way of specifying location, which will be translated then into SQL commands.

Finally, relational DBMSs provide for efficient concurrency control and recovery from failure. But, as we will see in the next chapter, these operations are better implemented outside the relational database system, because they need to manipulate objects rather than tuples.

5.3.2 Object-Oriented Features

We can expect to take advantage of the object-oriented paradigm for flexiblity. The object-oriented approach brings a flexibility in integrating different media that has no equivalent in any other model or programming methodology. This is indeed one of our concerns, as explained above. Therefore, we want to use objects as the natural entities, at least at the application level. The concepts of classes and encapsulation will give us the ability to adapt our system to different media, thus fulfilling the three multimedia requirements.

Traditional programming techniques are considered risky when reusability and maintenance come into play. The object-oriented paradigm is the natural answer to these problems. So we will also be able to answer our main programming requirement, which is the ease of use.

5.3.3 Limitations of the System

Having defined what we expect from our interface, we shall also give an overview of how we limit our first implementation. This allows us to concentrate on useful parts of the system (the core itself) and leave the less necessary features for future work. There are two types of limitations. The first one concerns the database capabilities themselves. The second type concerns some media or object-dependent features which must be implemented within the objects.

Querying capabilities. A first limitation is the lack of high-level queries on large data types. It could be interesting to extract, for example, all the images containing a tree in a database of images, or all sequences from the film *Ninotchka* where Greta Garbo is smiling (luckily for the database, there are only a few). We do not plan to implement such queries in the first version of the system.

Language. Another limitation of the same type regards the query language: we will limit ourselves to a minimal set of language features that allow operations to be correctly performed.

Performance. Finally, we did not regard performance as a goal to achieve. There is, in effect, currently no standard for performance evaluation,

and the only requirement we could have is that processing time be "reasonable". This is meaningless because what is reasonable for one application may be unacceptable to others.

Object-dependent features. The limitations of the second type concern features that most applications will need to redefine, because these are object-dependent (see Chapter 3). We therefore do not propose any model for versioning, integrity checking and security.

To summarise the specifications, we can say that we want to benefit from both the object-oriented paradigm, for the programming part, and the relational experience, for most database features. To do this, we will implement a minimal database interface providing the required operations to check the validity of our model.

5.4 Conclusions

We attempted, in this chapter, to give a real definition (as opposed to description) of the term "multimedia", in order to set the limits of our work. Then we showed on what conditions a multimedia system can be implemented and what constraints it must respect. Finally, we defined the specifications for our system: our goal, the operations performed, and the limits of our work. There is nothing definitive in this and the system is built so that extensions can be easily implemented. We can now turn to the design process itself, with first the model and architecture of our interface.

Chapter 6

Data Model and Architecture

Our goals are to provide the user with an easy way of storing objects, and for this we will follow the requirements set in the previous chapter. The main characteristic of our Multimedia Object-Oriented Database System (more simpiy called "the interface") is that it uses an arbitrary number of storage devices. We will show here how we use a relational database and a BLOB storage element, and how we keep track of the objects using the concept of object identity.

Additionally, objects can use their own methods of storage for some media types. We will not study these methods.

6.1 Data Model

6.1.1 The External Model

The external model corresponds to what the user sees. He/she is only concerned with objects. For example, an object *picture* has three attributes describing the position on the screen, x, y and the contents of the image itself called *image*, plus some methods used for displaying or editing. Such an object is shown in the upper part of figure 6.1.

At this level, we do not want to interfere with the "standard" object-oriented data model. This would be likely to create incompatibilities. There-

fore, we use the pure object-oriented data model, where the definition and manipulations of objects are part of the programming language. This model is characterised as follows:

- The basic entity is the object. There is no other entity manipulated at this level.
- The objects are identified by an *object identifier*. We note, however, that this is not visible by the user.
- The types of the attributes of the objects can be arbitrarily complicated (these are mostly user-defined types), and the associated values may be other objects.
- There is a total media independence: an object can be multimedia (*i.e.* might contain several media).

6.1.2 The Implementation Model

The implementation model consists of the decomposition of an object into its attributes. Each of these attributes keeps track of the object it belongs to. The attributes and values are stored differently, depending on whether they are traditional data types or BLOBs (see figure 6.1):

- Traditional data are stored in records consisting of the OID of the parent object, the name of the attributes, and the data itself. This determines a tuple. It is stored in a relation which depends on the type of the data. In the case of the image of the previous section, these data are the x and y position. Each of these is associated with one record as shown on the figure (middle part).
- BLOBs have additional information corresponding to their physical location on disk. A tuple is formed with the OID, the attribute name, the length of the data, and its physical position on the physical storage. This tuple goes in a special table of the relational table. This is described on the figure by a tuple dedicated to the description of the physical position of the data, and an additional BLOB which is kept apart from these data.

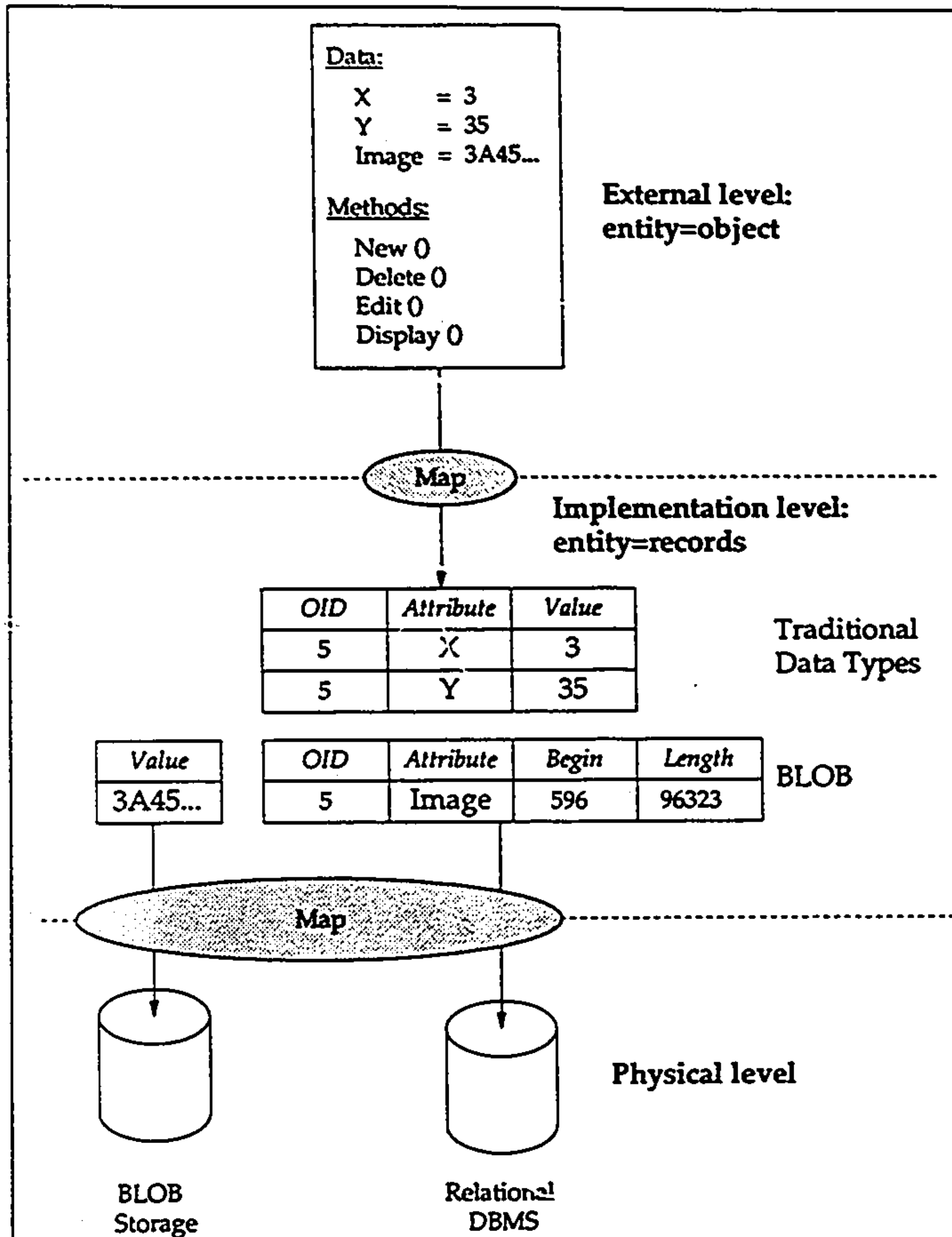

Figure 6.1: Example to illustrate the data model.

6.1.3 The Physical Models

There are currently two physical data models, corresponding to the two types of media: the physical model of the relational database for traditional data, and the BLOB physical model.

The relational database uses its own physical model. We do not try to interfere with it. As a consequence of this, any record-based database system could be used (relational, network, or hierarchical). The adaptation to the DBMS takes place in the query subsystem.

The BLOBs are simply stored as they are. The physical data describing their storage lies in a special table of the relational database.

6.2 Architecture

The system architecture is shown in figure 6.2 and consists of the following components:

- The requirements of integrity are best fulfilled by a centralised database. We need, thus, a **communication module** between the database core and its satellites.
- Then, we have to ensure some database features (see previous chapter). This is done by the **concurrency and transaction controller**.
- Queries are answered by the OID **manager** and the **object location manager**.
- The interrogation of the relational database is done by the **query subsystem**.
- Finally, the control of BLOB storage in handled by the BLOB **storage module**.

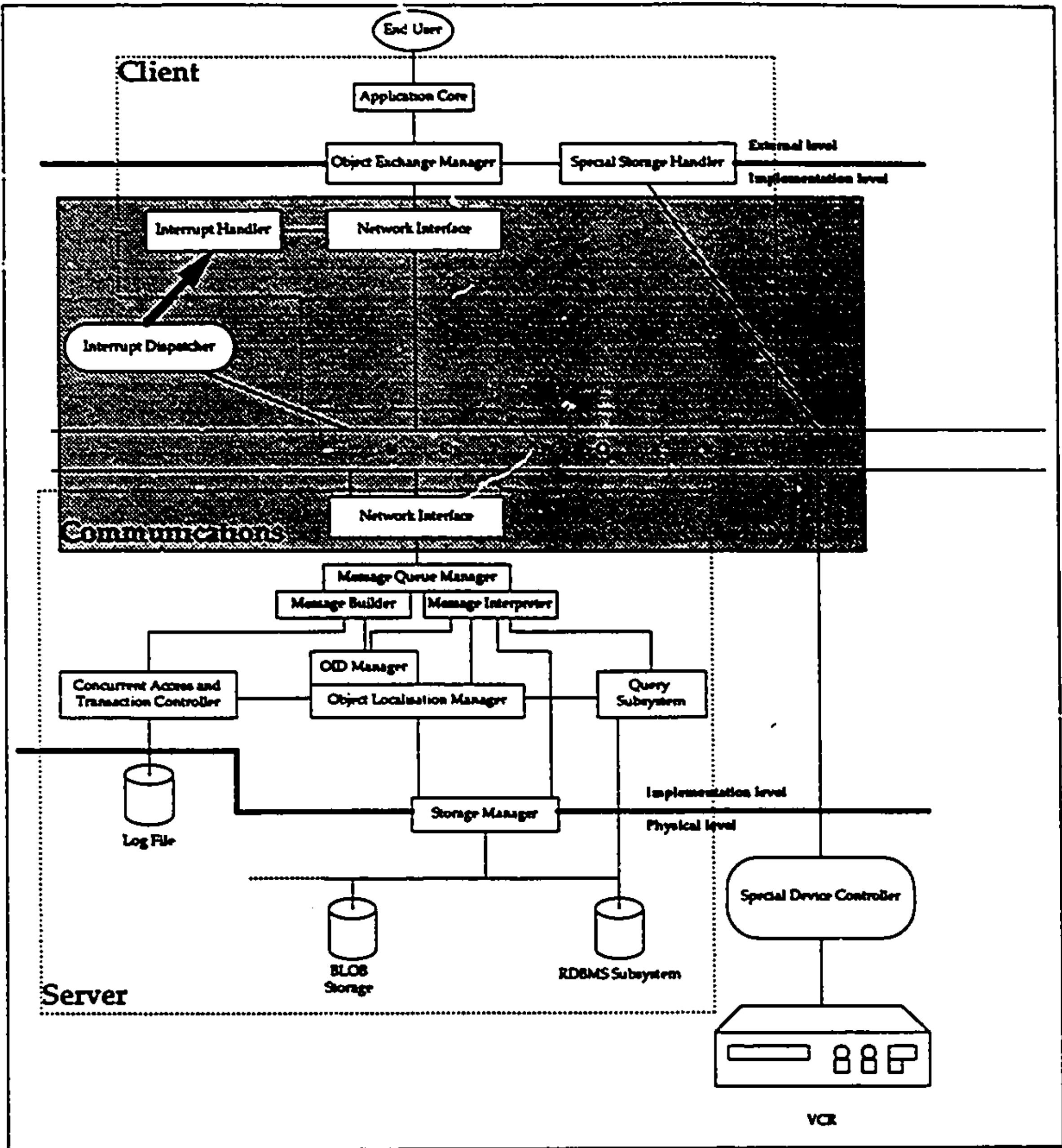

Figure 6.2: Architecture

6.2.1 The Communication Module

Efficient concurrency control is best achieved with a client-server model, in which a server listens for clients wishing to connect, and then creates a private two-way connection with these clients. The clients are the applications, and the server lies within the interface. This architecture is not system-dependent: clients might be connected to the server via a network, or reside on one multi-processor host, in which case we have a real concurrency. Or, they might use the same CPU (with time sharing capabilities) in which case concurrency as such does not really exist.

In order for applications to go on processing their own data while a query is running in the database, a special communication path uses interrupts. The server uses interrupts to warn the client that it has some data to read. For example, a client may request a very large image (about 1 Mb). Instead of waiting until this image is retrieved from the database, the application may continue its work. When the image is sent to the client over the network by the server, it sends an interrupt so that the client knowns that the requested image is waiting on the line.

The server needs two special modules to build outgoing messages or interprete incoming messages. These messages are taken and put to a cyclic queue controlled by a message queue manager linked with the network. In the previous example, the request for an image coming from a client will be first put in the queue. When the image is retrieved, a message containing the image is built and put in the queue. Finally, when the message queue manager processes this message, it sends the image to its destination (a network address), and then issues an order to the interrupt controller on the client's machine.

6.2.2 The Database System

The Concept of Object Identity

Throughout his/her life, a person acquires a name, has a physical description, a social insurance number, an address, and diplomas resulting from his/her studies. There is nothing that uniquely identifies this person in these characteristics, because all are subject to change. However, even though they may

be different at different stages of life, it is still the same person. Similarly, the attributes and other characteristics of an object can be changed, but the object is still the same in terms of its identity [1].

The Elements of the Database System

The role of the OID *manager* is to ensure that every persistent object can be identified. For example, the new photograph of a person, when introduced first in the database, must acquire a "license number" which will uniquely identify it until it is discarded from the database. This number is given by the OID manager.

The *location manager* identifies the current owner(s) of an object and the current level of transaction for each owner. If there is no owner the object is supposed to be in the database. Taking the example of someone's photograph again, the database may be asked to provide it to some client. In this case, the location manager will try to find a client already working on that picture, and provide the requester with the most up-to-date version of the photograph (if possible, otherwise it sends the address of the current owner). If there is no such application, the relational database is then searched for some record describing the BLOB storage of the picture. If it exists, the photograph is extracted and sent to the original client.

The role of the *concurrent access and transaction controller* is threefold. First, it stores intermediate data in a log file. Second, it is used by the location manager to extract an object if its owner is currently inside a transaction. Finally, it gives information about who owns an object, and warns clients if one of their objects is modified by someone else (optimistic algorithm). For example, client A and client B work on the same photograph, and client A enters a sub-transaction in which it updates the object. B will still see the old picture, until the sub-transaction of A commits. Then, B will be warned that the object has changed and that it should get it from A. Since the concurrency control type (locking or optimistic) is object-dependent, deciding if another user can share an object is not the role of the concurrent access controller and is left to the designer. The concurrent access controller comes into play only once the object has decided that it can be shared by several people.

Finally, the query subsystem is in charge of translating incoming requests into SQL queries and transfering these to the relational DBMS. The requests may be issued either directly (by the message interpreter), or by the core of the database system (one of the elements cited above which needs to get information or modify the contents of the relational database). In our photograph example, a query could be: "What is the physical address of the picture on the BLOB storage device?"

6.3 Data Flow

In this section, we briefly describe the flow of data and messages in the whole system. We will base our description on the schema given in figure 6.2. For the sake of comprehension, we use three different requests, based on our photograph example. We show the path from its origin (the application) to its destination (the last element in the database, or the return of the information to the originating application).

6.3.1 Getting a New OID

When a new photograph (or more generally a new object) is created, it must obtain an identifier. The figure 6.3 shows the elements that come into play in this operation.[1] A message is sent to the server through the network interface to the message queue manager. When the message comes out of the queue, the message interpreter sends the request to the OID manager. A new OID is created, and the message builder makes a message containing this new OID. It is then put on the message queue, and the message queue manager sends it back to the client via the network interface.

[1]This supposes that the object to retrieve has already been identified. Since the object identifier is hidden from the programmer, objects are referred to in the database by their characteristics (attributes values).

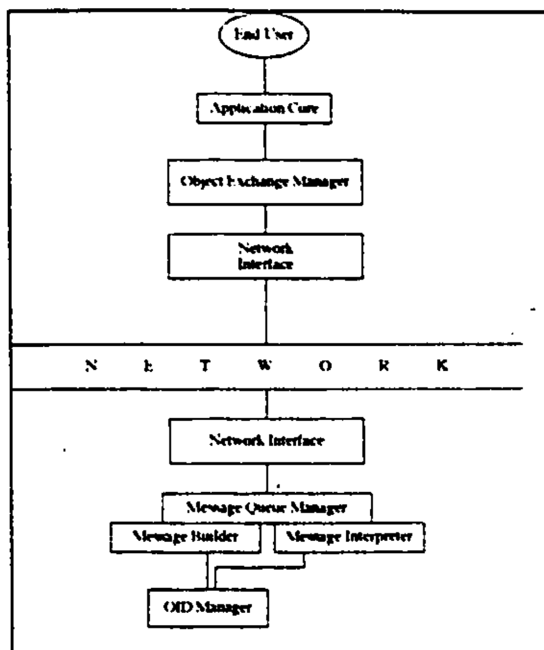

Figure 6.3: Getting a new OID

6.3.2 Getting an Object from the Database

The operation of getting something from the database is more complicated than the previous one, because one must get the most up-to-date data. The initial path of the message requesting, again, a photograph is the same, up to the message interpreter. This element will then forward the message to the object location manager. There are two cases.

- Either the image is not used by any client. It is thus retrieved from the database, using the query subsystem and the storage manager. The picture is then encapsulated in a message by the message builder and returned to the client.
- Or the photograph is currently owned by another client. The message is passed to the concurrent access and transaction controller, which decides if the last possible version of the picture, or at least the address of the client owning it, can be passed to the requester. If true, then the message containing the information is again build by the message builder and sent back. The decision of the concurrent access and transaction controller is based on whether the photograph is locked against reading or not, and whether the owner allows the database to give its address to other clients or not.

All the elements presented in figure 6.2 are used in this dialogue, with the possible exception of the interrupt modules.

6.3.3 Updating an Object

The operation of updating an object is very similar to that of retrieving it from the database. The message follows the same path, up to the object localisation manager. Similarly, this module searches all the current owners of the picture, and gives the control to the concurrent access and transaction controller. It will decide if the update can be done, and possibly warn other users that the picture has changed and that they should get the new version. This is done by sending the appropriate data to the message builder, which will propagate the message to the targets.

When the changes are committed, the concurrent access and transaction controller informs the object location manager of the end of the transaction, and lets it store the modified objects in the database. This last operation uses the query subsystem and the storage manager. As in the previous example, all the elements of the architecture are used.

6.4 Critical Analysis

The proposed data model and the resulting architecture are built in compliance with the requirements of multimedia data management (see Chapter 5. However, although our system inherits from both the relational techniques and the object-oriented paradigm, it does not inherit all the associated avdvantages. In this section, we analyse the most important shortcomings of our system. These are: the need for mapping modules, the lack of a recovery system and the problems caused by the client-server model. We describe these topics and brielfy answer them.

6.4.1 Mapping Between Levels

The object-oriented approach in databases is supposed to suppress the need for mapping between levels. However, our data model and architecture prevent us from taking advantage of this. It occurred to us that we had basically to chose between a relational database with mapping functions and the use of a single BLOB storage for all the data without any mapping. This last solution is not necessarily faster (because relational databases are known for being very well optimized), and it prevents any possibility of easily implementing features such as searching.

Regarding the mapping of data, one may also note that the data specifying the position of a BLOB in the BLOB storage device is not known until the object is actually stored. Therefore, the first mapping is incompletely done and the data are completed when the second mapping occurs (refer to figure 6.2 for the location of the mapping modules). This is not actually a drawback because it causes no penalty whatsoever, but it deserves to be pointed out.

6.4.2 Recovery

There is currently no recovery system implemented. The main reason for this is that we preferred to implement essential features first, such as concurrency control and OID management. It did not seem useful at first because normal programmes are not supposed to crash (we do not consider recovering from media failure). Moreover, unless we make a special effort to create a complete recovery system, some crashes cannot be recovered from. An example is provided by the edition of a video sequence. because of the way the video is stored (on a VCR), convenient archiving of intermediate video data is extremely difficult. More research can be carried out in the domain of recovery in multimedia databases.

6.4.3 The Client-Server Model

The client-server model causes two kinds of problems:

- First, the machine running the server can be extremely busy, depending on the number of clients. Such a strain on one machine only requires that it be dedicated to this task only, and even that does not prevent the server to be a bottleneck in the execution of a user request. The only gain is that it reduces the maintenance of the integrity inside the database to few operations.

- Second, in a distributed system, each database will be running its own server, and this is difficult to handle in terms of integrity. The area of distributed databases is vast and provides many interesting topics for further research.

6.5 Conclusion

Based on the requirements expressed in the previous chapter, we have designed a data model and an achitecture for a system capable of using both a relational database system for traditional data types, and a BLOB storage device for other data types. Its mapping modules, as well as the use of object

identity, allow the user to manipulate exclusively objects in a simple way. In other words, he/she is only concerned with objects. The problems that we have found in our data model could be answered by research in related areas. However, they do not prevent the system from working efficiently in its field of operation, which can be further extended with new features.

Chapter 7

Implementation of MOODS

After making hypotheses and setting the requirements for a multimedia object-oriented interface on top of a relational database system, we have described the data model and the architecture of such an interface. In this chapter, we first present the implementation which we have carried out, using these specifications. We then evaluate our system with respect to the goals defined earlier. In order to emphasize the difference between the theoretical system discussed in the previous chapters and its actual implementation, we will call it MOODS (Multimedia Object-Oriented Dabatase System), instead of "the system" or "the interface".

7.1 Implementation Results

During the implementation, we had to make a number of choices. We do not intend, here, to present them and give a justification or explanation: the ultimate verdict is the adequacy of MOODS to the requirements of multimedia applications. Thus, we will only describe the main points of the implementation, without discussing them in further detail.

First, we present the programming environment, hardware and software. Our overview then encompasses three fields: object identity, concurrency control and transactions. Finally, in the light of our modelling and architectural choices, we give the design of the relational database tables, and an overview of the communication system.

7.1.1 Programming Environment

For our implementation, we have a hardware environment compatible with the data model and architecture as explained in the previous chapter. The workstations are P.C.'s (H.-P. Vectra RS25C with and Intel 80386 CPU running at 25 MHz), and a Sun 3/150 (using a Motorola 68020 CPU at 16.7 MHz). The Sun workstation was used as a client only. The workstations are interconnected via an Ethernet cable supporting 10 Mbit/s.

From a software point of view, the machines run different versions of UNIX. The Sun uses 4.3 BSD, and the P.C.'s run the SCO version of System V. Although the Sun has its own relational database (Oracle), we used Ingres on a P.C. The programming was all done in C++ 2.0 provided by HCR, except one part: the relational database supports Embedded SQL with the C language only. Therefore, we concentrated all the operations on the relational database in one source file which is entirely written in C. We did not have a C++compiler for the Sun, which explains why we did not use it as a server.

7.1.2 Object Identity

Keeping track of the location of objects is one way of permitting them to be related to each other. This is allowed by object identification. Among the existing possibilities, there is one flawless way of identifying objects. This method is based on a strong separation of the class of an object, its state (*i.e.* the value of its instance variables) and its identity. The identities of objects will thus be represented by arbitrary numbers, unique throughout the system, independent of the classes of the objects.

The role of generating new object identifiers is given to the OID manager. Each time the server starts, it takes an OID value in the database. This is the value of the last identifier used in the previous run. Each time the server is shut down, it writes back the last value it has used. Each time an object is created, the identifier counter is incremented. When an object is deleted, there is no attempt to save its identifier in order to give it to another object. Thus, we ensure absolute uniqueness of identifiers and total independence to the class or state of an object. This is the best of the various systems we have considered. Some programming tricks were necessary to transparently

control the use of OIDs, and they seem efficient enough. For example, it is not possible to exactly copy an object into the same object (unless a malevolent programmer intentionally evades the protection), because the OID would thus refer to two different objects.

7.1.3 Concurrency Control

The main component of the architecture which uses the OID is the concurrency control and transaction manager (see figure 6.2). Since concurrency must be handled differently depending on the applications, or, rather, depending on the objects, we provided a framework to allow objects to control concurrency without relying too much on the work of the programmer. The basic idea is that since concurrency is object-dependent, we want to allow each object to control its concurrency management strategy.

When a client needs an object, it requests (indirectly) the object location manager. Either the object, which is being looked for, is in the database, in which case it is normally retrieved, assembled, and sent to the client, or it is already possessed by another client, in which case the concurrency is managed by the client itself (the object is extracted from the client if possible).

Problems arise when one of the applications saves its changes to the object while other applications are working on a copy of the original object. In this case, the other applications are informed of the change and they must acquire the new version of the object. Let us take the example of the photograph and assume that Ted is editing the top half. When another user, Paul, request this photograph, he gets the latest version that Ted's object can give (or none if Ted wants exclusive use of the picture). Ted then continues his work and saves it. At this point, Paul is informed that the object "photograph" has changed, and that he must re-read it. Whether Paul can get a copy of the photograph or not, whether he has the right to modify it, and thus overwrite Ted's work, or not is entirely dependent on the object and on the application.

The role of the concurrency control is to know what is the latest version of an object, who owns it, and to suppose that all the objects are working on an optimistic basis (*i.e.* that anyone can modify its objects and overwrite someone else's work, unless the object itself does not allow several people to rewrite it).

7.1.4 Transactions

The problem of transactions is slightly more complicated, because we wanted nested transactions to be possible. Instead of trying to allocate a private workspace for each level of transaction and for each client, which would quickly consume large chunks of memory, we simply use a log file to store a "before" and an "after" image of the updated objects. We have seen that media are storage specific; consequently, this log file is physically at the same place as the actual data. The problem is that if the data are refered to by their OID, the same OID can give two different objects (the current version and the initial one). Therefore, we use temporary OIDs: an object within a transaction has its OID changed. But a translation table in the transaction controller makes these changes transparent to the rest of the programme. In addition, the location manager "believes" that for the client who is currently running the transaction, the requested object is owned by the transaction controller. The location manager then acts exactly as if the transaction controller were another client of the database. This allows a great simplicity in these two elements, and complete transparency to the client.

For instance, taking our previous example agiain, let us give the photograph the OID number 7. When its first owner, Ted, enters a transaction, a new photograph with OID 8 is created. The transaction manager maps, then, all requests for object 7 into requests for object 8, until Ted's transaction commits. Then, object 7 is discarded, and the OID of object 8 is changed to 7. All clients owning object 8 are warned of these changes.

7.1.5 Database Design

The Relational Database

In the relational database, the tables are designed as follows. Attributes are gathered in tables according to their types. For instance, all integers are stored in the same table, independently of the classes of the objects they belong to. To allow easy querying, they are stored with the name of the attribute and the OIDs of their objects. Sometimes, the table may be so large that it penalises searching. In such a case, it is split into several smaller tables and a hash function on the OID and/or the attribute name easily reduces the

field of the search. However, since we did not have measurements for deciding the limits of the size of the tables, we set it to a very large value, so that splitting never happens.

The only problem regards the storage of strings. They are indeed of various length, and it did not seem reasonable to us to reserve the maximum length for all of them. We have, therefore, three tables supporting strings whose maximum length is either 30, 100 or 1900.[1]

BLOB storage

The physical organization of data is done by the operating system. But in order to minimize its role (we cannot completely bypass it), we have created a very large file (about 10 Mbytes) where we store our binary data, and only these data. We actually use the relational database to store the following information: the OID of the object owning the data, the name of the attribute and the position and length of the BLOB. We have chosen what A. Tanenbaum considers the simplest and most efficient allocation algorithm [58]: it simply consists of filling in the first fitting hole. The relational database keeps a list of the existing holes (which comes from the garbage collector embedded in the BLOB storage manager).

7.1.6 Communications

The basic element of communication in UNIX is the *socket*. Used in networking, it provides an endpoint for communications at the transport level. This endpoint is identified by its host address and the service port number. The transport service used is the DARPA "Transmission Control Protocol" (called TCP) because it is very reliable and best suited to end-to-end communications. It is built on top of IP, the Internet Protocol, which can be configured to use various buffer sizes, or to specify the behaviour of the connected parts

[1]These values were chosen based on the fact that the strings most often used in databases are, probably, the names of people or shorter reference strings, the name of files with their access path, and "short" remarks (a 25×80 character screen contains 2000 characters, and an average paragraph in this thesis contains less than about 200 characters). The last limit was given by the implementation of Ingres.

when the other side fails to show up. Since setting particular values cannot be justified without extensive studies, we did not change them. However, because of the particular organization of the server, we requested non-blocking operations in both directions.

At the application level, we have implemented the client-server protocol. The server always listens to clients wishing to connect. When one is detected, a circuit is created that links the client and the server through a bidirectional communication line. The client then issues requests and has the option of waiting for the answer, or going on processing other data while waiting to be interrupted by the server when it has successfully sent its data. This is particularly useful when large objects must be retrieved without penalising the execution of commands at the user level. Finally, a client terminates the session by sending a message to the server, which immediately closes its side of the communication line.

The server and its clients exchange information with messages. A message is composed of three parts (see figure 7.1):

The length indicator. This is a four byte number containing the total length of the message, including the length indicator itself.

The message header. It contains a code identifying the message type. This code gives the receiver some information about what it is supposed to do with this message. Whether the message is a response to a request or a request is not taken into consideration. Then, there is a format indicator which indicates the format of arguments.

The arguments. The format indicator currently allows six types of arguments: no argument at all, the argument is an OID, the argument is an attribute, the argument is an object, the argument is a list of objects, or the argument is special. The interpretation of this last case is left to the receiver. There is no theoretical limit to the size of the argument part.

7.1.7 Programmer's Entry Points

The programmer can manipulate the contents and the behaviour of the database through the "entry points" provided by the interface. These are all

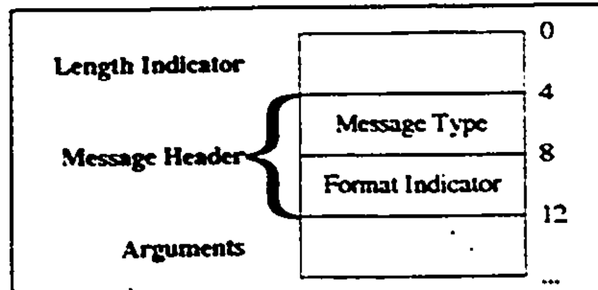

Figure 7.1: Message Structure

gathered in one class called `persistent`. The basic idea is that any object inheriting from this class is automatically part of the database contents (even temporarily). This greatly simplifies the programmer's work: inheritance is specified once in the class definition. Detailed definition of the class `persistent` can be found in Appendix B.

The private methods are numerous in the class `persistent` because an important part of the processing is hidden from the programmer. At a high level, it consits of automatically requesting an OID for newly created objects and of managing requests for retrieval by other clients. At a lower level, it can manage its own network data exchange with the server or with other clients.

The programmer has therefore very few specific methods to deal with. These are mainly: (1) get an object from the database according to given characteristics or according to a given OID, (2) store an object in the database and (3) permanently erase an object from the database. Transactions are mainly controlled by two functions which respectively begin and end a level of transaction.

7.2 Evaluation

We have described the implementation of MOODS. In this section, we shall review the most salient benefits of this implementation. These are:

- Transparency and ease of use for the programmer.

- Extensibility of various features.
- Fulfillment of the multimedia requirements.

In addition to these three points, we will also discuss some of the shortcomings of the system.

7.2.1 Transparency to the Programmer

The class `persistent` gives the programmer some simple methods of accessing the database. In addition, the use of encapsulation allows complex processing to be hidden from him/her and thus facilitates the transparency. The consequences are twofold: (1) any implementation detail regarding OID and OID management is hidden from the programmer, and (2) the operations "search" and "update" are very easy. We shall see with an example the simplicity of this class.

For instance, if we want to create a permanent object of a class named `photograph`, the class needs first to inherit from the class `persistent`. Then, the use of the functions `new()` and `delete()` automatically triggers the necessary operations for OID management and memory allocation for the picture to take place.

The functions "search" and "update" the photograph are also very simple. All the objects of the class `persistent` have a function `save()` and a function `search()`. In our case, the function `save()` must be overriden in the class `photograph`, so that it decomposes a photograph into a set of attribute-value couples: $(x, 3)$, $(y, 35)$ and $(Image, 3A45...)$. The rest of the operation (initiating a transaction to save the whole object or nothing, sending the attributes one by one) is done by the class `persistent` without further intervention by the user. The search of an object based on the values of some of its attributes is operated in exactly the same way.

7.2.2 Extensibility and Flexibility

Extensibility occurs in several places thanks to the override capabilities of C++. To illustrate this, we show how various concurrency control schemes can take place in the same application.

The default behaviour of the concurrent access controller is to let anyone update objects without control, provided that the other owners of the objects are warned. Let us suppose that our application is an image editor. We do not want other people to edit a picture while someone is already working on it. However, we may want to grant them read access only. Someone may also want to deny any kind of access while he is holding the object: for instance, two scenes of the same video cannot be displayed at the same time.

We experimentally implemented a concurrency control scheme in which the objects themselves decide what kind of access they allow. The modifications inside the database system were extremely simple: basically, a few types of messages to add. The rest of the work was not difficult, because it only consisted of implementing in each object a new (short) method of asking the user if he/she wants the object to be shared, and possibly modified by someone else. The overall modifications did not take more than one hour, of which most was application design rather than database work. This showed us that: (1) concurrency control can be adapted in many various ways; that was until now impossible with traditional databases; (2) changing the database behaviour is extremely quick and simple.

7.2.3 Multimedia Requirements

We have already seen an example of extensibility. Media extension is provided inside the database as well as outside, by overriding some methods. This encompasses the processing of data, the storage management and the relationships between objects.

For example, assuming that we want to add video images to our media list, a simple class `Video` providing control of the hardware is sufficient. Because traditional data are necessary in this type of objects, we are sure that all the features in our system will also apply to `Video` objects: concurrency control, transaction management, OID management are only examples. Regarding pre- or post-processing of data, overriding the `save()` and `retrieve()` is the best and simplest implementation.

7.2.4 Shortcomings

Our goal was not to build a complete database system, but only to prove that our model and architecture are a good basis for such a system. Therefore, MOODS lacks some features that would be needed in a more complete version. We discuss some of the more interesting points for future enhancements. They all require a lot of research and development time.

Granularity The concurrency and transaction controller is not able to act on entities other than objects. We may want, in some occasions, to lock a set of objects, or just some attributes of an object (for example all the photographs of John Smith, or only the bitmap of a photograph, but not its position on the screen). The reason we did not implement this is due to the ability of the application designer to create a large object encompassing a set of smaller objects: a class `collection` could be used to gather several photographs. A lock on one object of the class `collection` would also lock all the sub-objects. A second reason for not implementing variable granularity, is that it is better to use the concurrency control scheme explained above, in which each object is responsible for releasing the part of information it wants. This is, in our opinion, the only good way of implementing flexible concurrency control.

Dynamic class definition. The server is not able to know what a class consists of. To change this would require that a class defined in the application be also defined in the server. Only dynamic class definition would allow this. Unfortunately, C++does not support this and, for technical reasons, it is extremely difficult to emulate.

Query language. The query language that we used is too simple to be useful. It simply consists of filling the attributes values with the required characteristics. If a characteristic is not a selection criterion, it is given the special "null" value. This greatly limits searches such as: "What are the photographs of John Smith between 1980 and 1989?" The difficulty of an extended query language lies in the need for the database to store the extension of each class, and thus to perform costly joins for each search. In addition, it requires that the class of the result be known in advance which may not be convenient in some cases.

Temporal relationships. At the lower level, the database is not able to understand the concept of temporal relationships (sequentiality, simultaneity). This could be implemented with some extensions in the design of the tables in the relational database, as well as with synchronisation modules in the server and the application. The problem of multimedia synchronisation is much too comlex to be treated here. However, at a higher level, objects can use this concept and store these relationships in the relational database under a decomposed form.

Execution time. This is could be an important shortcoming in some cases. Some applications require extremely short responses, while others allow for reponse times of a few tens of a second. Our system is not currently able to give quick responses. There are two reasons for this. First, the interface with Ingres spends most of its time in controls and warnings, because neither the database provided by Ingres nor the data coming from the rest of the system can be entirely trusted. This part prevents the program from often crashing and allows other parts to be tested. Second, the communications through the network leaves much to be desired. For example, in one run, the version of TCP/IP provided by SCO needed one minute to read a line in a configuration file. Given such random behaviour, optimization is not our first concern.

7.3 Conclusions

This database system is presented to the programmer as a C library and set of C++ classes which he/she can use to implement persistence of objects on different storage devices. Because of the inheritance mechanism and because of encapsulation, the complexity of the underlying operations is hidden from the programmer, so that he/she only sees a few objects (at most five of them). This greatly simplifies his/her task. Additionally, everything out of his/her sight can be changed (or, less drastically, bugs can be removed) without incidence on the rest of the application.

The implementation which we have built solves most of the problems while complying with our initial goals, which were to make a database as easy as possible for a programmer to use, while not penalising such ease

of use with restrictions that would make our system uninteresting. This interface does not have all the features which we would like it to have, in order for it to be a true multimedia object-oriented database system, but it convinced us that it is a good basis for such a complete database system.

Conclusions

In this thesis, we investigated the areas of multimedia, the object-oriented approach and relational databases in order to provide programmers with an easy way to integrate these concepts. This system proposes a library and a set of classes to be used with the C or C++ language. It supports different types of data, in particular binary large objects, and can handle various storage devices. It remains open to any type of extension, which can thus be easily implemented.

The first three chapters of this thesis were dedicated to the study of relational technology, object-oriented methodology, and multimedia databases. This showed us that relational database systems present many advantages due to their long experience and their sound mathematical basis, but that they cannot be used for other data than traditional business or office application data. These typically include numbers, names and dates. On the other hand, object-oriented database systems are a perfect tool for the manipulation of multimedia data. It is an ideal method of software production and programming, but persistency is difficult to achieve. We have thus presented the current trends of the research in multimedia databases. There exist, basically, three methods: using extensions of the relational model, extending the data manipulation or data definition language to support new types of data (object-oriented or not), and building new databases from scratch based on the object-oriented model only.

Before designing our own model and architecture, we conducted a study of the requirements of multimedia applications based on a new definition of the term "multimedia", the first consequence of which is to require that applications be open to extensions towards new media. We classified multimedia data in three categories: small and structured data, large structured

data, and large unstructured data. This allowed us to specify three types of constraints for our system: storage must support different physical devices, data must be processed adequatelly, and relationships between objects must be kept in the database.

We were then able to present our data model, based on the separation of data between different storage devices, according to the media they represent. The architecture allowed us to fully benefit from the features brought to us by the underlying relational DBMS, while permitting us to use other devices as well. The main idea is that several features of our software require a centralised control: object identification, concurrency control, transaction management, are examples of these. We have used, therefore, the client-server architecture in which the server is responsible for doing all the centralised work, while the clients can concentrate on executing the end-user application. This model and architecture are independent from the hardware, the programming environment (language, operating system, network, etc.), and even from the database. Instead of the relational database, any record-based DBMS could be used. The use of a relational system is simply better in terms of performance that any other.

To validate our model and architecture, we have implemented our interface using C++ and Ingres SQL. This allowed us to determine what were the good points in our design, and what should be changed or added. Among the advantages, we can stress the ease of use of the interface by application programmers. This was one of our major goals and it was well achieved. Another good point is the flexibility left to the application programmers for deciding what type of concurrency control they wish to apply to their objects. More important still, our model is designed in such a way that adding a new storage media in the system can be achieved easily. This is mainly due do the independence of the implementation model from the physical model. In addition, the object-oriented paradigm allows hidden processing of data such as compression or decompression of information. Thus, most of the requirements for multimedia data handling specified in the course of our study, have been met.

The weaknesses of this interface are due to both the model and the implementation language. We give here some areas that deserve to be treated further (some of these points have already been introduced in previous chapters). Many of these are interesting areas for future research:

- First of all, the current use of C or C++ prevents us from adding methods at run-time, or modifying the classes. For example, without access to the source code of the server, one is not able to implement a new physical layer for an additional media. This is a strong limit to the extensibility of the database. The limitation is caused less by our model and architecture, than by the language that we have chosen for our implementation.

- Secondly, there is no way of expressing time constraints on the data. It would be interesting if an application requests, from the database, that an object be delivered at a specific time. Muitimedia data synchronisation is an exciting topic in which lots of research is involved.

- Thirdly, a query language is necessary. It must handle objects as general entities and atomic data, but should not try to implement entities such as lists, sets or anything too specific. Additionally, it must provide the users with some sort of global security control that cannot be managed by individual objects.

Other topics can be fruitfully used as research areas. When we presented our architecture, we saw that queries cannot be carried on non-relational data types. One could search for a method to easily allow objects to support queries on any of their attributes, with a homogeneous syntax and vocabulary, so that it could then be integrated in a larger data manipulation language. A second idea comes from the need that both the server and the client have to execute a programme through the network. Currently, UNIX supports remote procedure calls (RPCs). More difficult, is the execution of the method of an object, without having any knowledge of what the object exactly is, just by specifying its identifier. This would solve almost all the problems we encoutered during the implementation stage.

To sum up, we have developed a data model and an architecture which proved to be media polyvalent. The implementation of this system, in accordance with the specifications, fulfilled most of the requirements of this study. This is a good basis for further refinements and research in the field of multimedia databases.

Appendix A

Tables

This appendix is not intended to give an extensive view of the software sources. They are too large to be listed here (approximately 200 pages), and are not good examples of well-written programmes. We present in these three appendices the most important data structures, which are the tables used in the relational database (Appendix A), the classes that we have defined (Appendix B), and the list of supported messages with their formats (Appendix C).

A.1 Table Chart

This table contains one-byte values.

Attribute	*Type*	*Comment*
OID	integer4	Object identifier
Field	char(64)	Limited by C to 64
Value	integer1	1 byte of data

A.2 Table Shortt

This table contains two-byte values.

Attribute	*Type*	*Comment*
OID	integer4	Object identifier
Field	char(64)	Limited by C to 64
Value	integer2	2 bytes of data

A.3 Table Integert

This table contains four-byte values.

Attribute	*Type*	*Comment*
OID	integer4	Object identifier
Field	char(64)	Limited by C to 64
Value	integer4	4 bytes of data

A.4 Table Floatt

This table contains double precision floating point values.

Attribute	*Type*	*Comment*
OID	integer4	Object identifier
Field	char(64)	Limited by C to 64
Value	float	8 bytes of data

A.5 Table StringtS

This table contains short strings (up to 30 characters).

Attribute	*Type*	*Comment*
OID	integer4	Object identifier
Field	char(64)	Limited by C to 64
Value	varchar(30)	Short string

A.6 Table StringtM

This table contains short strings (up to 100 characters).

Attribute	*Type*	*Comment*
OID	integer4	Object identifier
Field	char(64)	Limited by C to 64
Value	varchar(100)	Medium string

A.7 Table StringtL

This table contains short strings (up to 1900 characters).

Attribute	*Type*	*Comment*
OID	integer4	Object identifier
Field	char(64)	Limited by C to 64
Value	varchar(1900)	Large string

A.8 Table Datet

This table contains dates and time. Their are stored with one millisecond precision, but we did not find a way to have Ingres accept milliseconds on input.

Attribute	*Type*	*Comment*
OID	integer4	Object identifier
Field	char(64)	Limited by C to 64
Value	date	

A.9 Table Blobt

This table contains the list of blocks allocated for blobs.

Attribute	*Type*	*Comment*
OID	integer4	Object identifier
Field	char(64)	Limited by C to 64
Begin	integer4	Address of the beginning of a block
End	integer4	Address of the end of a block
Length	integer4	Length of the block

A.10 Table Freet

This table contains the list of free blocks.

Attribute	*Type*	*Comment*
Begin	integer4	Address of the beginning of a block
End	integer4	Address of the end of a block
Length	integer4	Length of the block

A.11 Table UsedOid

This table contains the last used object identifier.

Attribute	*Type*	*Comment*
OID	integer4	Object identifier

Appendix B

Classes

We did not list here the contents of all the classes of the programme, but only the most relevant among them. Each class is defined in a particular header, except the classes List and ListItem which are defined in the same header.

B.1 Class Bullet

```
class Bullet : public ListItem {
    private:
        Connection  * connection;
        Direction   direction;
        Message     * message;
        void        (* post_processing) (Operation *,
                                         Bullet *);

        int         position;
    public:
        Bullet      (Connection &, Direction, Message &,
                     void (*) (Operation *, Bullet *));
        ~Bullet     () {}
        Boolean     Shoot (Operation *);
        Message     & GetMessage () {
                        return * message;
```

```
                    }
        Connection  & GetConnection () {
                        return * connection;
                    }
};
```

B.2 Class Client

```
class Client : public Socket {
    private:
        struct sockaddr_in  server;
    public:
        Client              (char * service,
                             char * hostname = NULL);
        void                Connect ();
};
```

B.3 Class Connect

```
class Connection : public Socket {
    private:
        struct sockaddr_in  peer;
        Transaction         * transaction;
    public:
        Connection   (int);
        ~Connection  () {}
        int          operator == (Connection &);
        int          operator != (Connection & connection_) {
                         return ! (* this == connection_);
                     }
        void         SetTransaction (Transaction &);
        Transaction  * GetTransaction ();
};
```

B.4 Class ConnectionLoc

```
class ConnectionLoc : public ListItem {
    private:
        Connection              * connection;
        Transaction             * transaction;
        static ConnectionLoc    * current;
    public:
        ConnectionLoc   (Connection &);
        ~ConnectionLoc  () {}
        ConnectionLoc   * Search (List &, Connection &);
        ConnectionLoc   * Search (List & list_) {
                            return Search (list_,
                                           * this->connection);
                        }
        ConnectionLoc   * SearchAgain (Connection &);
        ConnectionLoc   * SearchAgain () {
                            return SearchAgain
                                   (* this->connection);
                        }
        Connection      & GetConnection () {
                            return * connection;
                        }
        Transaction     & GetTransaction () {
                            return * transaction;
                        }
};
```

B.5 Class List

```
class List {
    private:
        ListItem    * head,
                    * tail,
                    * current;
```

```
        int         number;
    public:
        List        ();
        ~List       () {}
        ListItem    & Head () {
                        return * head;
                    }
        ListItem    & Tail () {
                        return*  tail;
                    }
        int         Number () {
                        return number;
                    }
        void        For_all (void f(ListItem &));
        ListItem    & Search (Boolean f(ListItem &));
        ListItem    & Search_again (Boolean f(ListItem &));
        void        Add (ListItem &);
        void        Remove (ListItem &);
};
```

B.6 Class ListItem

```
class ListItem {
    friend class    List;
    protected:
        ListItem    * previous,
                    * next;
    public:
        ListItem    ();
};
```

B.7 Class Location

```
class Location : public ListItem {
```

```
    private:
        int             oid;
        List            connection_list;
        static Location * current,
                        * previous;
    public:
        Location    (int, Connection &);
        ~Location   () {}
        Location    * Search (List &, int);
        Location    * Search (List &, Connection &);
        Location    * Search (List &, int, ConnectionLoc * &);
        Location    * SearchAgain (int);
        Location    * SearchAgain (Connection &);
        Location    * SearchAgain (int, ConnectionLoc * &);
        List        & GetConnectionList () {
                        return connection_list;
                    }
        int         GetOid () {
                        return oid;
                    }
};
```

B.8 Class Message

```
class Message {
    private:
        Commands    command;
        char        * content;
        int         length;     // Total length of message
    public:
        Message     (Commands = UndefinedCommand,
                     char * = NULL, int = 0);
        ~Message    () {}
        int         Send (Connection &, int = 0);
        int         Send (Client &, int = 0);
```

```
        int         Receive (Connection &, int = 0);
        int         Receive (Client &, int = 0);
        Commands    GetCommand () {
                        return command;
                    }
        char        * GetContent () {
                        return content;
                    }
        int         GetLength () {
                        return length;
                    }
};
```

B.9 Class OidMapItem

```
class OidMapItem : public ListItem {
    private:
        int                     true_oid,
                                temporary_oid;
        static OidMapItem       * current,
                                * previous;
    public:
        OidMapItem              (int, int);
        ~OidMapItem             () {}
        int                     * SearchTrue (List &, int);
        int                     * SearchTemporary (List &, int);
        int                     * SearchAgainTrue (int);
        int                     * SearchAgainTemporary (int);
        int                     GetTrue () {
                                    return true_oid;
                                }
        int                     GetTemporary () {
                                    return temporary_oid;
                                }
};
```

B.10 Class Operation

This class implements the main loop of the server-side of the interface. In addition to the functions represented here in the private segment, all the-functions relative to the interpretation of a message have been also implemented. The basically correspond to the public operations presented in the class permanent.

```
class   Operation {
    private:
        Server        * server;
        int           current_oid,
                      old_oid;
        List          * queue,
                      * object_location;
        Boolean       shutdown;
        void          Interprete (Bullet *);
        void          Wait (Bullet *);
    public:
        Operation     (PGM_PARAM&);
        ~Operation    ();
        void          Run ();
};
```

B.11 Class Permanent

```
class Permanent {
    private:
        int           oid;
    public:
        Permanent     ();
        ~Permanent    ();
        void          Save (char *, char);
        void          Save (char *, short);
        void          Save (char *, int);
```

```
        void        Save (char *, Date &);
        void        Save (char *, double);
        void        Save (char *, char *, int = UseStrlen);
        void        * Get (int);
        void        SearchValue (char *, char);
        void        SearchValue (char *, short);
        void        SearchValue (char *, int);
        void        SearchValue (char *, Date &);
        void        SearchValue (char *, double);
        void        SearchValue (char *, char *,
                                 int = UseStrlen);
        int         Search (int *);
        void        BeginTransaction ();
        void        EndTransaction ();
        int         Commit ();
        void        AbortTransaction ();
        void        Remove (int);
};
```

B.12 Class Server

```
class   Server : public Socket {
    private:
        struct sockaddr_in  server;
    public:
        Server              (char *, int = QUEUE_SIZE);
        Connection          * Connect ();
};
```

B.13 Class Server

```
class   Socket {
    protected:
        int         file_descriptor;
```

```
    public:
        Socket       (int = AF_INET, int = SOCK_STREAM,
                      int = 0, Boolean = TRUE);
        ~Socket      ();
        int          Write (char *, int,
                             int = REPLACE_FILE_DESC);
        int          Read (char *, int,
                            int = REPLACE_FILE_DESC);
        Boolean      ReadyToRead (int = REPLACE_FILE_DESC);
        void         SetBlocking ();
        void         SetNonBlocking ();
};
```

B.14 Class Transaction

```
class   Transaction {
    private:
        List            * oid_map;
        Transaction     * parent_transaction;
        Connection      * connection;
    public:
        Transaction     (Transaction &, Connection &);
        ~Transaction    ();
        void            Commit ();
        void            Terminate ();        // Implicit commit.
        void            Abort ();
        Boolean         InTransaction (int);
        int             Map (int);
        void            Remove (int);
};
```

Appendix C

Messages

We give here the list of codes supported by the message interpreter (an element which is part of the Operation class). The names are self-explanatory.

```
enum    Commands    { NewOid,
                      ReplyNewOid,
                      Delete,
                      GetOid,
                      SaveOid,
                      SearchSpec,
                      DoSearch,
                      FoundOid,
                      ReplyOid,
                      SendTo,
                      GetFrom,
                      TransactionFailed,
                      BeginTansaction,
                      EndTransaction,
                      CommitTransaction,
                      AbortTransaction,
                      Terminate,     // Terminates the server.
                      UndefinedCommand };    // Not used.
```

Bibliography

[1] Abnous, Razmik and Setrag Khoshafian. *Object-Oriented Concepts, Languages, Databases, User Interfaces.* Addison-Wesley, Reading, Massachusetts, 1990.

[2] Acksyn, Robert, Donald McCracken, and Elise Yoder. KMS: A distributed hypermedia system for managing knowledge in organizations. *Communications of the A.C.M.*, volume 31, number 7, pages 820–835, July 1988.

[3] Aho, A. V., C. Beeri, and J. D. Ullman. The theory of join in relational databases. In *Proceedings of the 18th I.E.E.E. Symposium on Foundations of Computer Science*, pages 107–113, October 1977.

[4] Alexandridis, Nikitas A. Adaptable software and hardware: Problems and solutions. *Computer*, volume 19, number 2, pages 29–39, February 1986.

[5] American National Standards Institute. The database language SQL. Technical report, ANSI, 1986. Document ANSI X3.135.

[6] Andrews, Timothy and Craig Harris. Combining language and database advances in an object-oriented development environment. In *A.C.M. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, October 1987.

[7] Atkinson, Malcolm, François Bancilhon, David J. DeWitt, Klaus Dittrich, David Maier, and Stanley B. Zdonik. The object-oriented database system manifesto. In Kim, Won H., J.-M. Nicolas, and S. Nishio, editors, *Proceedings of the First International Conference on Deductive*

and Object-Oriented Databases, pages 40–57, Amsterdam, 1989. Elsevier Science Publishers B. V. (North-Holland).

[8] Ballou, Nathaniel, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Hyoung-Joo Kim, Won H. Kim, and Darrell Woelk. Data model issues for object-oriented applications. *A.C.M. Transactions on Office Information Systems*, volume 5, number 1, January 1987.

[9] Ballou, Nathaniel, Jorge F. Garza, Won H. Kim, and Darrell Woelk. Architecture of the ORION next-generation database system. *I.E.E.E. Transactions on Knowledge and Data Engineering*, volume 2, number 1, pages 109–124, March 1990.

[10] Bancilhon, François. Object-oriented database systems. In *Proceedings of the 7th A.C.M. Symposium on the Principles of Database Systems*, pages 152–162, March 1988.

[11] Banerjee, Jay, Kyung-Chang Kim, and Won H. Kim. Queries in object-oriented databases. In IEEE88 [44], pages 31–38.

[12] Beech, D., H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, D. H. Fishman, C. G. Hoch, W. Kent, Peter. Lyngbæk, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. IRIS: An object-oriented database management system. *A.C.M. Transactions on Office Information Systems*, volume 5, number 1, pages 48–69, January 1987.

[13] Bertino, Elisa and Lorenzo Martino. Object-oriented database management systems: Concepts and issues. *Computer*, pages 33–47, April 1991.

[14] Björnerstedt, Anders and Christer Hulten. *Version Control in an Object-Oriented Architecture*, chapter 18, pages 451–485. In Lochovsky and Kim [49], 1989. Published jointly with A.C.M. Press, New-York, N. Y.

[15] Blumberger, Robert E. and David H. Walters. A PC-based multimedia document manager. *I.E.E.E. Journal on Selected Areas in Communications*, volume 7, number 2, pages 283–289, February 1989.

[16] Booch, Grady. *Software Engineering with* ADA. Benjamin Cummings, Menlo Park, California, May 1983.

[17] Carey, Michael J., David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and file management in the EXODUS extensible database system. In VLDB86 [61], pages 91–100.

[18] Chen, P. P. The entity-relationship model: Toward a unified view of data. *A.C.M.-T.O.D.S.*, volume 1, number 1, pages 9–36, January 1976.

[19] Chignell, Mark, Setrag Khoshafian, Kamran Parsaye, and Harry Wong. *Intelligent Databases.* John Wiley & Sons, New York, N.,Y., 1989.

[20] Clark, C. M., C. L. Gallo, W. B. Harding, and H. Tang. Object storage hierarchy management. *I.B.M. Systems Journal*, volume 29, number 3, pages 384–397, September 1990.

[21] Codd, E. F. A relational model of data for large shared data banks. *Communications of the A.C.M.*, volume 13, number 6, pages 377–387, June 1970.

[22] Codd, E. F. Further normalization of the data base relational model. In *Data Base Systems*, volume 6 of *Current Computer Science Symposia Series.* Prentice-Hall, Englewood Cliffs, N. J., 1972.

[23] Codd, E. F. Recent investigations into relational data base systems. In *Proceedings of the I.F.I.P. Congress*, August 1974.

[24] Copeland, George P. and Setrag Khoshafian. A decomposition storage model. In *Proceedings of the A.C.M.-S.I.G.M.O.D., International Conference on the Management of Data*, 1986.

[25] Copeland, George P. and Setrag Khoshafian. Object identity. In *A.C.M. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, September 1986.

[26] Copeland, George P., Setrag Khoshafian, and Patrick Valduriez. Implementation techniques of complex objects. In VLDB86 [61], pages 101–109.

[27] Dadam, P. and V. Linnemann. Advanced information management (AIM): Advanced database technology for integrated applications. *I.B.M. Systems Journal*, volume 28, number 4, pages 661–681, December 1989.

[28] Dale, Alfred, Kyung-Chang Kim, and Won H. Kim. *Indexing Techniques for Object-Oriented Databases*, chapter 15, pages 371–394. In Lochovsky and Kim [49], 1989. Published jointly with A.C.M. Press, New-York, N.Y.

[29] Date, C. J. *An Introduction to Database Systems*, volume 1. Addison-Wesley, Reading, Massachusetts, fourth edition, November 1987.

[30] D.B.T.G. Report of the CODASYL data base task group. *A.C.M.*, April 1971.

[31] Deux, O. et al. The story of O_2. *I.E.E.E. Transactions on Knowledge and Data Engineering*, volume 2, number 1, pages 91–108, March 1990.

[32] Drucker, Steven M., Bernard J. Haan, Norman K. Meyrowitz, and Nicole Yankelovich. Intermedia: The concept and the construction of a seamless information environment. *Computer*, volume 21, number 1, pages 81–96, January 1988.

[33] Edwards, Samuel D., David R. King, and Ann L. Winblad. *Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, November 1987.

[34] Eliot, J. and B. Moss. *Object-Orientation as a Catalyst for Language-Database Integration*, chapter 24, pages 583–592. In Lochovsky and Kim [49], 1989. Published jointly with A.C.M. Press, New-York, N.Y.

[35] Elmasri, Ramez and Shamkant B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Redwood City, California, 1989.

[36] Fagin, R. Multivalued dependencies and a new normal form for relational databases. *A.C.M.-T.O.D.S.*, volume 2, number 3, pages 279–296, September 1977.

[37] Fagin, R. Normal forms and relational database operators. In *Proceedings of the A.C.M.-S.I.G.M.O.D. Symposium on Principles of Database Systems*, May 1979.

[38] Falquet, Gilles, Marc Junet, and Michel Leonard. ECRINS/86: An extended entity-relationship data base management system and its semantic query language. In VLDB86 [61], pages 259–266.

[39] Ghafoor, Arif and Thomas D. C. Little. Network considerations for distributed multimedia object composition and communication. *I.E.E.E. Network Magazine*, pages 384–397, November 1990.

[40] Hasan, Waqar, Peter Lyngbæk, and Kevin Wilkinson. The IRIS architecture and implementation. *I.E.E.E. Transactions on Knowledge and Data Engineering*, volume 2, number 1, pages 63–75, March 1990.

[41] Hawryskiewycz, I. T. *Database Analysis and Design.* Science Research Associates, Chicago, Illinois, 1984.

[42] Hirohima, Michael, Lawrence A. Rowe, and Michael Stonebraker. The implementation of POSTGRES. *I.E.E.E. Transactions on Knowledge and Data Engineering*, volume 2, number 1, pages 125–142, March 1990.

[43] Hodges, Parker. A relational successor? *Datamation*, volume 35, number 21, pages 47–50, November 1989.

[44] *Proceedings of the I.E.E.E. Conference on Data Engineering*, February 1988.

[45] Ishihara, Kozo, Katsumi Tanaka, and Masatoshi Yoshikawa. Schema virtualization in object-oriented databases. In IEEE88 [44], pages 23–30.

[46] Klug, A. and D. C. Tsichritzis, editors. *The* ANSI/X3/SPARC DBMS *Framework.* American Federation of Information Processing Societies, 1978.

[47] Korson, Kim and John D. McGregor. Object-oriented design. *Communications of the A.C.M.*, volume 33, number 9, pages 39–60, September 1990.

[48] Lécluse, Christophe, Philippe Richard, and Fernando Velez. O_2, an object-oriented data model. In *Proceedings of the A.C.M.-S.I.G.M.O.D., International Conference on the Management of Data*, pages 424–433, June 1988.

[49] Lochovsky, Frederic and Won H. Kim, editors. *Object-Oriented Concepts, Databases, and Applications.* Frontier Series. Addison-Wesley, Reading, Massachusetts, 1989. Published jointly with A.C.M. Press, New-York, N. Y.

[50] Maier, David, Allen Otis, Alan Purdy, and Jacob Stein. Development of an object-oriented DBMS. In *A.C.M. Proceedings of the Conference on Object-Oriented: Systems, Languages and Applications*, pages 472–482, September 1986.

[51] Maier, David and Jacob Stein. Development and implementation of an object-oriented DBMS. In Shriver, B. and P. Wegner, editors, *research Directions in Object-Oriented Programming*, pages 355–392. M.I.T. Press, Cambridge, Massachusetts, 1987.

[52] Maier, David and Stanley B. Zdonik, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Mateo, California, 1990.

[53] Naffah, Najah. Multimedia applications. *Computer Communications*, 1990.

[54] Nierstrasz, O. M. and D. C. Tsichritzis. *Directions in Object-Oriented Research*, chapter 20, pages 523–536. In Lochovsky and Kim [49], 1989. Published jointly with A.C.M. Press, New-York, N.Y.

[55] Sheth, Amit. Managing and integrating unstructured and structured data: Problems of representation, features, and abstraction. In IEEE88 [44], pages 598–599.

[56] Skarra, Andrea H. and Stanley B. Zdonik. *Concurrency Control and Object-Oriented Databases*, chapter 16, pages 395–421. In Lochovsky and Kim [49], 1989. Published jointly with A.C.M. Press, New-York, N.Y.

[57] Stonebraker, Michael. Future trends in database systems. In IEEE88 [44], pages 222–231.

[58] Tanenbaum, A. *Structured Computer Organization.* Prentice-Hall, Englewood Cliffs, N. J., 1984.

[59] Thatte, S. Persistent memory: Merging AI knowledge and databases. *Texas Instrument Engineering Journal*, pages 151–159, January 1986.

[60] Ullman, J. D. Database theory: Past and future. In *Proceedings of the A.C.M.-P.O.D.S.*, March 1987.

[61] *Proceedings of the Twelfth International Conference on Very Large Data Bases*, August 1986.

[62] Zaniolo, Carlo. The database language GEM. In *Proceedings of the A.C.M.-S.I.G.M.O.D. International Conference on the Management of Data*, May 1983.