Chapter 2:

2 Integration of OODBMS Tools and their Concepts

Chapter Snapshot

- Descriptive meaning of buffer management and overview.
- Description of need to effectively manage database buffer.
- Main goal and work organization of my research.
- Description of previous work and literature.
- Chapter wise organization of my thesis
- Summary of the chapter.

2.1 Introduction

An object database (also object-oriented database management system) is a database management system in which information is represented in the form of objects as used in object-oriented programming. When database capabilities are combined with object-oriented programming language capabilities, the result is an object-oriented database management system (OODBMS). OODBMS allow object-oriented programmers to develop the product, store them as objects, and replicate or modify existing objects to make new objects within the OODBMS. Because the database is integrated with the programming language, the programmer can maintain consistency within one environment, in that both the OODBMS and the programming language will use the same model of representation. Relational DBMS projects, by way of contrast, maintain a clearer division between the database model and the application. As the usage of web-based technology increases with the implementation of Intranets and extranets, companies have a vested interest in OODBMS to display their complex data. Using a DBMS that has been specifically designed to store data as objects gives an advantage to those
companies that are geared towards multimedia presentation or organizations that utilize computer-aided design.

Object-oriented data management systems (OODBMS) differ from traditional database systems in several respects. The main difference is that OODBMS offer greater extendibility, the inclusion of user defined behavior, and enhanced reusability of prior work. While traditional database systems are able to cope with and manipulate a fixed set of predefined but parameterizable database types, OODBMS allow the programmer to define new abstract data types. The definition of a data type includes not only its structure but also the behavior of its instances. The definition of new data types is facilitated by inheritance which allows one to adapt previously developed data types to new requirements.

Much work has been done to develop several viable OODBMS. These projects usually concentrate on defining an object and database model and on developing a database management system for these models. Normally these systems offer some low level integration of the database management system with a programming language, in order to allow access and manipulation of the persistent objects contained in some database.

However, OODBMS, like traditional database systems, are not self-contained complete software development tools. They must be combined with other tools such as user interface management systems to produce entire applications, but much less work has been done to explore the combination of OODBMS and software development tools other than programming languages. Several different software development tools are potential candidates for an integration. Thus, the goal of this project is to show advantages and disadvantages of various potential combinations and to explore the integration of OODBMS and GACLs (GACL) in more detail.

2.2 Application model to be supported

Application model to be supported before discussing the main issues we limit the scope of applications we consider. Applications targeted by this project consist of several components:

- User Interface: An application presents itself to the user through its user interface. A user can access the services of the application by issuing commands. Depending on the user interface technology used, commands are induced by various means, as, for example, through written commands or through the manipulation of graphical dialog elements such as buttons or menus. Results of the execution of commands are shown
by the user interface in a textual form or graphically. Our project considers only user interfaces based on a modern graphical user interface paradigm using direct manipulation, bitmapped graphics, multiple windows, and at least a keyboard and a mouse as input devices.

- **Management of persistent data:** Another component of an application is concerned with the manipulation of persistent data. In applications of interest to this project, persistent data is contained in persistent objects which are managed by an object-oriented database management system.

- **Rest of the application:** The last component of an application is called the rest of the application. It includes everything not covered by the user interface component and the management of persistent data. An example are the no persistent collection classes. This component usually also manages any access to operating system services.

Another assumption about the applications concerns the transaction model. The target applications use a simple pessimistic transaction model based on locking. Save points are possible but otherwise no fancy features are assumed.

Prototypical target applications include graphical editors such as a schema editor for a database system or form-based applications such as a bug reporting system.

We now consider how the development of such applications could be facilitated.

### 2.3 Integration of OODBMS and traditional software development tools

First we shall consider tools developed in the context of non-object-oriented, non-database applications and in the context of traditional database systems.

#### 2.3.1 User interface toolkits

A user interface toolkit is a library of data types and procedures which implements common user interface elements such as menus, scrollbars or radio buttons. A programmer can use these data types in his or her program to construct a user interface for the application. The library usually also contains procedures to access events from input devices as mouse and keyboard. Examples of user interface toolkits are the Macintosh Toolbox [App185] and the Ext Toolkit for the X Window System [Youn89]. The problems with toolkits are: (1) they usually are not programmed to take
Figure 2.1: Seeheim Model

Advantage of object-oriented programming techniques, (2) the level of abstraction is too low, and (3) they do not provide an architecture for an application. We therefore decided against integrating OODBMS with user interface toolkits based on conventional programming paradigms.

2.3.2 User interface management systems based on the Seeheim architecture

A user interface management system is a system in which a user interface for a given application can be specified declaratively. It furthermore automatically manages the processing of user events as far as the user interface is concerned on the basis of the user interface specification [Olso87]. Often a graphical editor is used for the specification of user interfaces. A programmer can assemble the layout of a window from predefined display and data entry fields like text fields, radio buttons, check boxes, graphical output areas, popup menus, buttons and menus attached to a menu bar. The user interface constructed with such a system dispatches user events to the items of a layout. The dialog items process related events on their own or call attached user defined actions. User interface management systems are usually employed in the context of applications which don't use a database system for the management of persistent data.

Many if not most of the user interface management systems are based on the Seeheim application architecture [Gree851. In the Seeheim model the user interface is clearly separated from the other parts of the application. The model assumes that the user interface and the other parts of an application are only loosely coupled and that communication between the user interface and the other parts may be accomplished by a low bandwidth connection [Myer89].
One could indeed run the user interface in one process and the other parts of the application in another process. The user interface is further divided into a presentation component, a dialog control component and an interface to the other parts of the application (Fig. 3.1). The dialog control component dispatches and processes user events, the presentation component is responsible for drawing the user interface, and the interface to the other parts of the application connects the user interface to the other parts of the application, i.e., to the so-called rest of the application along with the persistent data management part.

Communication between the user interface and the other parts of the application is accomplished through shared variables, message passing or a procedural interface.

One advantage of this architecture is that a new user interface can be used without altering the rest of the application as long as the interface between them is not changed. Thus the user interface can easily be modified to take advantage of new user interface paradigms or new user interface techniques without the necessity of changing the other parts of an application. Separating the user interface from the other parts of an application also facilitates the incremental development of the user interface because the other parts of the application are not affected by these nonfunctional changes.

Another advantage of this architecture is that the non-user interface parts of an application are relieved of the burden of processing every single user input. The user interface of an application is thus relatively autonomous, i.e., much of the user input can be processed locally in the user interface without the necessity of accessing other parts of the application. For example, when a user fills in a form the user interface processes user inputs until the user has committed the changes. Only then is the application sent the final content of the form.

A final advantage is the declarative nature of these tools. The programmer need no longer worry about the sequence of events and can concentrate her or his efforts on other parts of the application.

Unfortunately this architecture has several shortcomings:

- A dialog can only be assembled from a predefined set of dialog items. While few systems based on a textual specification languages allow the extension of the set of dialog items, for example Apollo Computers Open Dialogue [Comp86], even fewer systems with a graphical user interface editor provide an extension feature. Even then,
the definition of new dialog items is normally difficult. Thus it is either not possible or at best difficult to enhance the user interface with domain specific interaction techniques.

- Only form-based user interfaces are supported, but only a fraction of the applications which fit the application model we want to support have a form-based user interface.
- The basic assumption of the low bandwidth communication between the user interface and the rest of the application does not hold for many applications, especially graphical editors. In these applications communication between the user interface and the rest of the application is very frequent because the rest of the application must often be consulted to provide the semantically correct feedback, for example during dragging or resizing graphical objects. Some systems, such as Higgs [Huds88], try to provide this kind of feedback in the context of a user interface management system based on the Seeheim architecture. Unfortunately however, much of the declarative nature of other user interface management systems is lost. If the Higgs specific terms are stripped off the system resembles a somewhat modified model-view controller architecture where the model is constructed of active values triggered by user or application events.
- An investigation [Ross87] also showed that many designers have difficulty separating the user interface from the other parts of the application.
- The division of an application into a user interface and other parts gives the application developer no further advice for the architecture of the other parts of the application.
- In these systems, reusability is provided through the user interface management system and not through the inheritance mechanism usually used in object-oriented systems.
- These tools do not support the development of application specific aspects not related to the user interface by high level tools.

In the case of a textual specification a programmer has to learn a new special language.

Recently several research groups started to explore the integration of user interface generators and user interface management systems with object oriented database systems. Facekit [King89] and LOOKS [Alta89] [Plat89] combine an OODBMS and a user interface management system. They exhibit most of the previously mentioned advantages and disadvantages. Usually a nested object representation for complex objects is derived from the database schema and it is possible to call methods of the displayed persistent objects. In LOOKS the user interface is called through
Global procedures from the database application thus deviating from the programming paradigm used in the non-user interface related parts of an application. Neither system supports the construction of graphical editors.

Another interesting research project [Fly11901 tries to unify persistent data management and user interface specification management. User interface specifications are also considered as persistent objects. They store both a description of the display of the persistent object and rules applied to process user events. If a persistent object has to be displayed and manipulated, a specification object is activated. The specification and the persistent object are used to generate a display object, which displays the persistent data and controls the processing of user events. Every persistent object can be combined with several different specification objects. Because the system supports inheritance, specification objects can be easily reused to construct new ones. And because specification objects are persistent objects any tool provided by the database system to modify persistent objects can be used to alter the content of these objects during program execution, giving the system much flexibility.

After considering advantages and disadvantages of these various systems we decided not to investigate the integration of a user interface management system and an OODBMS although this approach could be very reasonable for applications using a form-based user interface. But we feel that it would be advantageous to retain in our approach at least some of the declarative flavor of the user interface management systems.

2.3.3 4th generation languages

In the context of advanced database systems based on traditional data models such as 4th Dimension from Acius [ACWSG], the high level tool support covers the interactive definition of database schemas, the specification of the layout of windows, and the formulation of queries and reports through graphical editors. Systems as these are called 4th generation languages [Olso87]. This actually is a misnomer because the languages in which the rest of the application is programmed do not differ from third generation languages except for the more or less elegant integration of a persistent data type. The user interface and the management of persistent data are normally tightly coupled in 4th generation languages. After specifying the database schema, a default user interface is often inferred from the schema and one can then enter new data and issue queries whose results are displayed through the default user interface. A programmer can adapt the default user interface to his or her own taste or can define completely different ones.
Typically, dialog items such as text entry fields are bound directly to fields of a record and the system automatically propagates to the database data entered in a panel. It is usually not possible to relax this tight coupling of user interface and database or to have dialog items for data computed from persistent data or for data from which persistent data is inferred.

4th generation languages have the following advantages:

- User interface design and user event processing are simplified.
- Not only the user interface but also the management of persistent data is supported through high level tools.

And the following disadvantages:

- Dialogs are assembled from a predefined, usually not extendible set of dialog items.
- Only form-based user interfaces are supported.
- The database and user interface are normally tightly coupled, i.e., it is often not possible to have special fields to display data derived from the persistent data. We think that the tight coupling is not reasonable for many applications we would like to support.
- These systems do not support the development of the application domain specific aspects of an application not related to the user interface or persistent data management through higher level tools than the programming language.
- If a program needs facilities which cannot be produced with these tools a programmer must use the low level programming language which is part of the whole system. There is no intermediate level of support between the high level tools and the low level programming language.
- 4th generation languages usually provide only a basic transaction mechanism built into the language. They don't support different transaction patterns through higher level abstractions.

The consideration of the advantages and disadvantages of 4th generation languages led us to the decision that we don't want to develop a 4th generation language using an OODBMS instead of a relational database system. But it would be interesting to have a system with similar tools supporting applications with more complex user interfaces less tightly coupled to a database.
2.3.4 Generic database browsers

Another less ambitious approach to combining a database and a user interface are generic database browsers such as the database browser of Sun simplify [Sun 881. With this browser every database has an application independent user interface and every database can be examined and modified.

An example of such a system in the context of OODBMS is the KIVIEW browser [Laen89]. It provides a standard user interface for every database. It is also possible to customize the user interface. KIVIEW therefore provides functionality similar to the systems mentioned 3.2 or 3.3. Because such a generic database browser is valuable tool, we would like our approach to provide an application independent standard user interface. Having seen several approaches we turn now to our own approach of the integration of OODBMS and other software development tools.

2.4 Architectural Issues

This section outlines various OODBMS architectural concepts and design alternatives. In addition, the particular design alternative chosen for this thesis is also explained and justified.

2.4.1 Client and Server Layers

OODBMS systems typically have the notion of a client and server. The client runs the sists of the language run-time system, and the OODBMS run-time system necessary to communicate with the server. The server implements efficient stable storage for objects using the secondary storage, employing recovery, concurrency control, and other database protocols (e.g. versioning, indexing, etc.).Client programs mostly access objects sequentially (one at a time). One object at the time is ‘visited’ by de-referencing object pointers and thus ‘navigating the object graph’.

The exception occurs when ad-hoc queries are used, in which case operations are performed on sets of objects. When an object is accessed by the client, the OODBMS run-time provides the object (by issuing a request to the server) and verifies that the attempted operation on the object is allowed. Depending on the concurrency control protocols used, a read or write lock may be needed to complete the operation. When the object arrives from the server the suspended client computation resumes.
2.4.2 Network Models

There are two standard network models for OODBMSs: client/server; and peer-to-peer:

**Client/server:** In general this model is designed to run a client process on a small private workstation and communicate with one or a set of machines (with large disks) acting as a server. Figure 2.2 (a) shows a typical client/server network configuration. The advantage of this model is that client workstations do not need to perform much processing and thus can be relatively ‘thin’ machines. However, the disadvantage is that servers need to service requests from all clients and thus may become a bottleneck when the number of clients is large. Examples of client/server OODBMSs are Exodus [Carey et al. 1986], Object Store [Lamb et al. 1991], and O2 [Deux 1991].

**Peer-to-peer:** In this model every workstation on the network has a server process and any number of client processes running. The storage of the data is distributed across the machines. Figure 2.2 (b) shows a typical peer-to-peer network configuration. The advantage of this approach is that local data (data primarily used by a local client) can be stored on local servers and thus reduces access costs and decentralizes data storage. Decentralized data storage means the server is less likely to be a bottleneck. The disadvantage of this ashore [Carey et al. 1994] and Platypus [He et al. 2000].

This thesis examines performance issues surrounding disk IO optimization for a stand-alone single node of the peer-to-peer network model. This way we remove the network and remote caching behavior of the general peer-to-peer network model and can thus focus our attention on reducing the effects of disk IO. However the techniques developed in this thesis, although designed for this more restricted model,

2.4.3 Granularity of Caching

In an OODBMS, main memory caches are used extensively to reduce network and IO costs. It is for this reason that we have decided to focus on main memory caches, however it is important to note that as the performance discrepancy between main memory and upper level caches widens, more work needs to be done at the higher level of caching. This important topic is a good candidate for future work. An important distinction between caching 2 strategies is the grain at which data is cached. The granularity issue can be viewed in terms of three alternatives:
Object Grain: Data is stored and evicted at the object grain. The advantage of this approach is that useful objects can be extracted from a page and stored in the cache, while less useful objects can be discarded. This leads to better cache utilization, which is particularly important when the cache size is very small maintaining data at such fine grain is high. Objects from a loaded page need to be copied one at a time into the object cache. The Thor system uses this approach [Liskov et al. 1996].

Page Grain: Data is stored and evicted at the page grain. The main advantage of this approach is the low cost of buffer management. Low costs arise from the fact that pages are fixed size and the fact that disk to memory transfer is at the page grain. It is cheaper to manage fixed sized ages than it is to manage variable sized objects. Loading and caching data at the same grain (page grain) removes the need to perform memory copies. In object grained caching, objects need to be extracted and copied into the object cache upon loading. The disadvantage of this approach is that if objects are not well clustered, the cache can contain many useless objects, leading to high memory wastage. Systems that use page grained caching include: Platypus [He et al. 2000], EXODUS [Carey et al. 1986], and O2 [Deux 1991] and Object Store [Lamb et al. 1991].

Dual Grain: In this approach main memory is divided into multiple buffers. Each buffer can be either page or object grained. This approach has the advantage that well clustered pages can be left in the page buffer and the badly clustered pages can have their objects copied into the object buffer. Kemper and Kossmann [1994] show that dual grained caching often outperforms page grained.

Caching when using the OO7 benchmark [Carey et al. 1993]. However, they use naive clustering strategies that do not offer high quality clustering. We believe page grained caching will outperform dual grained caching when high quality clustering algorithms are used. Dual grain caching is also used in SHORE [Carey et al. 1994].

In this thesis we choose to explore page grained caching. The reason for this choice is three fold: the popularity of the page grained caching (Object Store [Lamb et al. 1991], EXODUS [Carey et al. 1986], O2 [Deux 1991] and Platypus [He et al. 2000]); the good performance of page grained caching when the system is well clustered; and our aim of exploring the effects of clustering on OODBMS performance. We believe page grained caching
outperforms both object and dual grained caching when the system is well clustered. This is because when the system is well clustered, the savings made on reduced cache maintenance costs outweigh any cache space wastage costs incurred.

### 2.4.4 Data Transfer Grain

The granularity of data transferred between client and server caches has a large effect on system performance. There are typically two grains at which OODBMSs transfer data:

- **Object server architecture**: In this architecture the unit of data transfer is groups of one or more objects. The advantage of this design is that only those objects that are needed are transferred to the client. The disadvantage is that the cost of transfer per object is high. In the case of transferring a single object, the network latency is incurred for just one object, thus making the transfer cost per object high. When a group of objects is transferred, the CPU cost of assembling the desired objects into one unit of transfer can place a high per object transfer cost on the server. This technique is employed by Thor [Liskov et al. 1996].

- **Page server architecture**: The server sends pages of data to the clients. This approach incurs a low per object cost since one instance of network latency is incurred for all the objects residing in the page or pages. In addition, no processing cost is incurred for grouping objects for sending (as needed by the object server architecture). However, when objects are not well clustered, many useless objects may be transferred to the client. Many object-oriented databases use this approach: EXODUS [Carey et al. 1986], ObjectStore [Lamb et al. 1991], and O2 [Deux 1991] and Platypus [He et al. 2000].

In this thesis we focus on a single stand-alone node of the peer-to-peer network model in which clients and servers share the same main memory cache. In such a model, data transfer between clients and server is done through the shared main memory cache and thus network latencies arising from either object grained or page grained data transfer are non-existent. However, the techniques developed in this thesis are best suited for use in page server architectures.

### 2.5 Conclusion and Summery:

In this Chapter we are trying to explain some basic and important concepts of OODBMS, that I will be use in my research work. After reviewing the different OODBMS concepts my main focus on exploring different architectural issues which I explained above like client and server
layer, data transfer grain etc. We use the peer to peer model for our work. In this model every workstation on the network has a server process and any number of client processes running. The storage of the data is distributed across the machines. The advantage of this approach is that local data (data primarily used by a local client) can be stored on local servers and thus reduces access costs and decentralizes data storage. Decentralized data storage means the server is less likely to be a bottleneck. This thesis examines performance issues surrounding disk IO optimization for a stand-alone single node of the peer-to-peer network model. This way we remove the network and remote caching behavior of the general peer-to-peer network model and can thus focus our attention on reducing the effects of disk IO.