

Three-Level Object-Oriented Database Architecture Based on Virtual Updateable Views¹

Piotr Habela¹, Krzysztof Stencel^{2,1}, Kazimierz Subieta^{1,3}

¹Polish-Japanese Institute of Information Technology, Warsaw, Poland

²Institute of Informatics, Warsaw University, Warsaw, Poland

³Institute of Computer Science PAS, Warsaw, Poland

{habela, subieta}@pjawst.edu.pl, stencel@mimuw.edu.pl

Abstract. We propose a new architecture for object database access and management. It is based on updateable views which provide universal mappings of stored objects onto virtual ones. The mechanism preserves full transparency of virtual objects either for retrieval and any kind of updating. It provides foundation for three-level database architecture and correspondingly three database development roles: (1) a database programmer defines stored objects, i.e. their state and behavior; (2) a database administrator (DBA) creates views and interfaces which encapsulate stored objects and possibly limit access rights on them; (3) an application programmer or a user receives access and updating grants from DBA in the form of interfaces to views. We present a concrete solution that we are developing as a platform for grid and Web applications. The solution is supported by an intuitive methodology of schema development, determining the perspectives and responsibilities of each participant role.

1 Introduction

The well-know ANSI/SPARC architecture [1] defines three levels which drive the data access and management of a database. These are *internal*, *conceptual* and the *external* levels. The conceptual level is common for all the database environment, while the external level consists of particular views (subschemata) dedicated to particular client applications or particular users. In relational databases the external level is implemented by two kinds of facilities: access privileges to particular resources granted by a database administrator (DBA) to particular users, and SQL views that customize, encapsulate and restrict resources to be accessed. Such an approach has proven to be enough simple and satisfactory for majority of applications of relational databases.

The situation is different for object-oriented and XML-oriented environments, especially in the data/service grid setting (transparent integration of distributed, heterogeneous and redundant data/service resources). Firstly, the database model is much more complex. It includes hierarchies of objects (XML), irregularities in data structures, object-oriented concepts (classes, inheritance, modules, methods, etc.) associations among objects, Web services and other notions. Secondly, the responsibility for the entire environment is more fuzzy than in case of relational systems. Such envi-

¹ This work is supported by European Commission under the 6th FP project e-Gov Bus, IST-4-026727-ST.

ronments may be accessed by many kinds of users, supplied with data and services by various local sites and processed by various applications that appear ad hoc during system operation. This requires much more fine and flexible dynamic security, privacy, licensing and non-repudiation control. Thirdly, the encapsulation and customization must be finer than in the case of relational systems. Different businesses supported by the system may require precise customization and access control of data and services according to the needs, preferences and limitations of particular users.

The above external level qualities are hardly to be achieved by SQL views. An SQL view is defined by a single SQL query, thus it is not powerful enough to express all the required mappings (especially in a case of schematic discrepancies between stored and virtual data). Moreover, SQL views are not sufficiently transparent due to the still unsolved view updating problem. Although INSTEAD OF trigger views (implemented in Oracle and MS SQL Server) can map view updates onto updates of stored data, still the problem remains due to the limited power of SQL. Obviously, SQL does not address data structures richer than relational ones (more powerful SQL-99 is still a rough proposal rather than a feasible, complete and consistent solution).

In this paper we propose a new architecture for object database access which has conceptual and technical advantages over the solutions known from relational systems. The central issue considered is the external level of the ANSI/SPARC architecture and its various relationships with the conceptual level. Our research is motivated by the following requirements and expectations with respect to the architecture:

- Customization and conceptual modeling: data and services resources stored in the database are to be shared by various kinds of users. Each group of users can view the resources according to own habits, terminology, organization, etc.
- Encapsulation and information hiding: an application programmer should be constrained to use in the application code only resources that are explicitly granted for the application by the database system programmer and/or (dynamically) by DBA.
- Security, privacy, licensing, non-repudiation and other regulations that the users have to obey w.r.t. the resources. DBA should possess flexible facilities to grant and restrict dynamically the privileges to particular users and particular resources. The privileges may concern not only retrieval, but also any kind of updating.
- Strong typechecking of application codes. Interfaces that are granted to particular application programmers contain the typing information which should be used to typechecking of the written code. In case of semi-structured data (c.f. XML) some semi-strong typechecking system is expected.
- Orchestration/choreography of services: on the external layer data and service resources can be orchestrated (i.e. composed) in order to achieve some defined business goals. On the conceptual layer various applications can be coordinated to obey some business rules (e.g. licensing). Orchestration can be encapsulated, i.e. a composition of services can be externally perceived as a single service.

The last element of the database architectural puzzle is the three-role database application development model. These roles are the *database designer/programmer*, the *database administrator* and the *application programmer* (or database user). The database designer/programmer develops a conceptual database schema and implements classes/methods to be used by applications. DBA defines user privileges on objects and services, defines views and interfaces restricting external access, and determines

user privileges concerning the interfaces. Views defined by DBA map stored data/services onto virtual ones, limit the visibility of certain features, customize and encapsulate data/services and/or perform service orchestration. Finally, the application programmer builds the code based on the interfaces to views provided by DBA.

Comparing the above architectural issues with the current state-of-the-art of object-oriented and XML databases, we conclude that significant research and development is still to be done. The most widely known ODMG standard [6] assumes data manipulation through language bindings, very similar to the ones used to implement and access objects in CORBA-based middleware [7] (having roots in programming languages rather than in database architectures). The scope of ODMG standardization is obviously too narrow. The standard assumes the support for monolithic applications based on a single database schema. The external architectural level is not mentioned. Hence the security, privacy, licensing and non-repudiation infrastructure has to be built on top of single interfaces to stored objects. There is even a step back in comparison to Java and COM/DCOM object models, which assume multiple interfaces to objects. Moreover, ODMG defines views (a *define* clause of OQL) as a client-side shortcuts (actually, macro-definitions) rather than as a server-side first-class database entities that can be dynamically handled by DBA. This is a step back in comparison to SQL views. View updating mechanism and interfaces to views are not even mentioned. Hence it was not possible to build our proposal within the framework established by the ODMG specification. However, where possible, we attempt to keep the introduced notions aligned with the terminology of ODMG and UML [8] models.

There are not too many papers on the above topics. In [10] related similar issues are discussed. Although we consider a manual derivation of user schemata, the problems of schema closure and interface positioning discussed in [10] may be important for assuring schema consistency. The main difference of our approach is that we do not follow the ODMG architecture and consider a much wider data and services environment. Eventually, our goal is a kind of enhanced Web services based on a virtual repository integrating distributed, heterogeneous and redundant data and service resources. The discussion and language proposals in this paper are also relevant for more traditional centralized object-oriented client-server architecture.

The proposed architecture allows convenient application development over an object database. It is founded on three concepts: (1) the Stack-Based Approach (SBA) [3] to query/programming languages, (2) updateable views, and (3) the model of three roles in the application development. SBA treats a query language as a kind of programming languages and therefore, queries are evaluated using mechanisms which are common in programming languages. SBA introduces an own query language Stack-Based Query Language (SBQL) based on abstract, compositional syntax and formal operational semantics. SBQL can be equipped with a strong or a semi-strong type system. SBA encompasses also powerful recursive tools [4] which allow defining any computable mapping between stored and virtual data (views).

Our approach to updateable views for SBA is presented e.g. in [5]. The idea is to augment the definition of a view with the information on users' intents with respect to updating operations. The first part of the definition of a view is the mapping of the stored objects onto the virtual objects, while the second part contains redefinitions of updating operations on virtual objects. The definition of a view may also contain definitions of subviews and methods. The mapping and the redefinitions are ex-

pressed with SBQL routines, therefore their power is unlimited. SBQL, including virtual updateable object-oriented views, is implemented in several research prototypes. The most recent of them, named ODRA (Object Database for Rapid Application development), serves as a base for implementing the concepts presented here.

The paper is organized as follows. In Section 2 we describe the overall architecture. Section 3 presents the notion of a module as we use it. Sections 4, 5 and 6 discuss the three roles in the database development (database programmer, DBA, application programmer). Section 7 concludes.

2 Architecture Elements

The principal value of database management systems is that they provide data persistence mechanisms separable from applications. However, the variability of users and applications brings important new requirements concerning the schema management.

Offering DBMS content to various applications makes it necessary to introduce a DBMS-managed “middle layer”. The features essential for such a layer are: transparency, ease of management and modification, no conceptual limitations comparing to stored data, more than one transformation step allowed to get the desired form of data. In such architecture, Fig.1, the following user roles and responsibilities are needed.

Database programmer creates internal and conceptual schema of the data upon previously created design, taking into account business goals. His/her task is to implement data classes, their methods, attributes and association, and services. The database programmer can use the implementation inheritance. There may be a number of developers working independently on particular database parts.

Database administrator defines external schemata for particular users with respect to the resources provided by database programmers. The database administrator creates updateable views built over the data store. He/she can create two kinds of views. *Predefined views* preserve the semantics of stored objects but may limit access to certain attributes, methods and generic operations. Such views can be automatically generated by some tool or simply written by the administrator. *Custom views* are created solely by the administrator. Such views can arbitrarily change the semantics of the stored object and introduce a different privilege model than predefined views.

Such views play in fact the role of interfaces restricting and/or refining the access to stored objects. Different views, designed for different users' categories, may be attached to the same object at the same time.

After these views are defined the database administrator grants privileges on them to particular users (application programmers). This views can be reused (views can be built on views) and/or inherited (a view can inherit from a number of classes).

Application programmer (database user) is a programmer, who uses the database, dealing with the user schema assigned. He/she knows the interfaces of views granted by the database administrator. Such an interface (view) serves two purposes. First, it keeps the conceptual model of the database for a particular application programmer. Second, it is used during the static type check of the application code.

From the above we may notice that there are two tasks for the middle layer in our approach. The first one is to transform data schema to the form required by certain usage. The second one is to define access privileges for different groups of users.

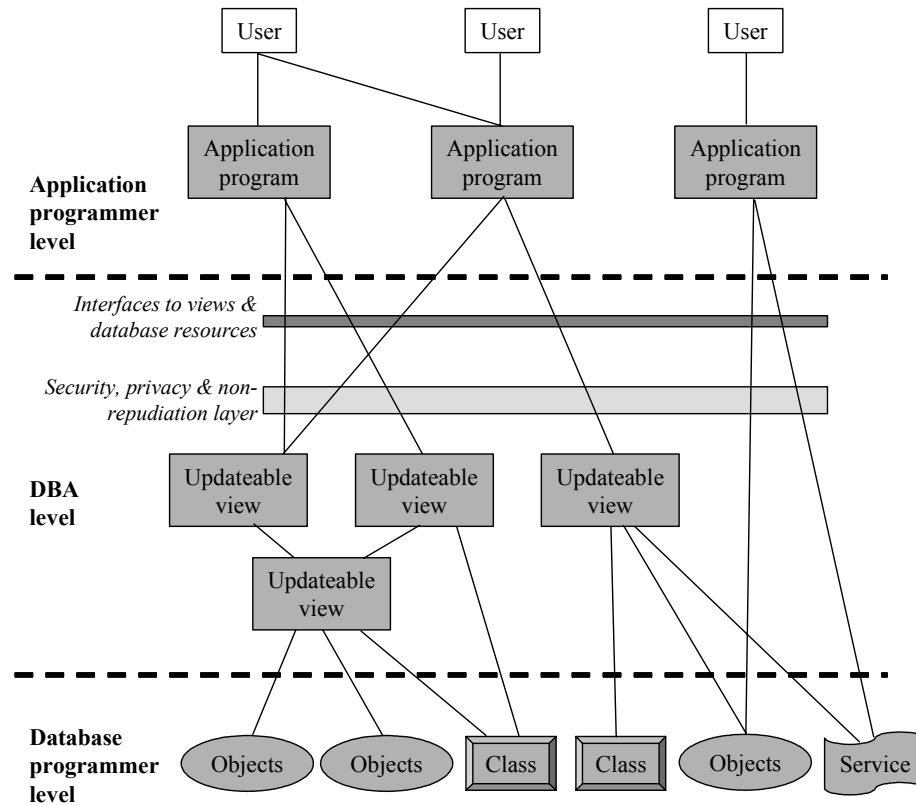


Fig. 1. A conceptual view of the three level architecture

3 Modules: Organizing Schemata and Storage

The data management based on a single, monolithic *database* unit, as assumed by ODMG, we consider problematic for advanced applications. This is confirmed by the fact that commercial ODMG-compliant ODBMSs designed to deal with large data amounts introduce additional levels of storage objects (see e.g. [12]). Thus we have introduced *modules* which partition database contents accordingly to purpose and ownership. This means that the features would be declared for particular modules rather for database as a whole. Since modularization is also useful for the code and related metadata (e.g. classes and interfaces), our modules have two purposes: organizing storage and organizing metadata.

Module declaration results in the creation of a dedicated space for it, both in regular database storage (allows a module to serve as a container of objects) and in the

metadata repository (to store module-defined interface definitions). Appropriately, an import clause between modules can serve two purposes. Considering class and interface definitions, it establishes code dependency. At the same time, for stored objects (defined by module's structural features), it specifies side-effects (that is, non-local data dependencies). Depending on particular needs, modules can be purely code-oriented (no features declarations allowing regular object storage) or storage-oriented (no locally defined classes or interfaces). A developer may also decide to mix those responsibilities to reduce the number of modules.

Modules are decentralized in the sense that only interface and feature definitions of each module are located in the common metadata repository. Class implementation including method code and view definitions are located in data store, together with regular objects stored in a given module.

4 Database Programmer

A database programmer creates a database schema. He/she defines collections of objects and their invariants, like the type, default values of attributes, integrity constraints, implementations of methods, and so on. The allowed generic operations (insert, update, delete and retrieve) might also be limited on this level. Here is an example of a module created by a database programmer.

```

module Studio {
    movie [0..*] : Movie insert, delete;
    actor [0..*] : Person insert, delete;
    director [0..*] : Person insert, delete;
class Person {
    export {
        name : string retrieve, update;
        starsIn [0..*] : ref Movie reverse stars retrieve, insert;
        directs[0..*] : ref Movie reverse director retrieve, insert;
        avgPay : double retrieve;
        phoneNo : string retrieve, update;
        setPay (m : Movie) : void;
    }
    name : string;
virtual starsIn [0..*] : ref Person
    { (movie where exists (contract where contractor=this and role="actor") as si; }
      retrieve { return deref(si); }
      insert newMovie { /* add new contract for this actor to newMovie */ };
virtual directs [0..*] : ref Person
    { (movie where exists (contract where contractor=this and role="director")as di; }
      retrieve { return deref(di); }
      insert newMovie { /* add new contract for this actor to newMovie */ };

```

```

virtual avgPay : double
    { avg((starsIn.Movie.contract where contractor =this).pay) as ap }
    { retrieve { return ap; } };
phoneNo : string;
setPay(m : Movie) : void { /* the implementation code comes here */ }
} // class Person

class Movie {
export{
    title: string retrieve, update;
    star[0..*] : ref Person reverse starsIn retrieve, insert, delete;
    director : ref Person reverse directs retrieve, update;
    contract[0..*] : struct Contract { contractor : ref Person; pay : double; role:string}
                                     retrieve, insert;
}

    title : string;
    contract[0..*] : struct Contract {contractor : ref Person; pay : double; role:string};

virtual star [0..*] : ref Person
    { (contract.contractor.Contract where role = "actor") as s; }
    retrieve { return deref(s); }
    insert newActor { /* add new contract with newActor */ }
    delete { /* delete the contract with this actor */ };

virtual director : ref Person
    { (contract.contractor.Contract where role = "director") as d; }
    retrieve { return deref(d); }
    update newDirector { /* set the director contract with newDirector*/ }

} // class Movie
} // module Studio

```

The module *Studio* defines two classes, including stored and virtual data (attributes). The information on the associations between movies and persons is stored in the (multi-valued) attribute *contract*. This information is further used to establish virtual links *avgPay*, *starsIn*, *directs*, *star* and *director*. Note that the definitions of this virtual attributes are not accessible outside. The *export* clause controls the external access them. Any external entity will perceive these virtual attributes and concrete data in exactly the same way (similarly as in case of methods, whose implementation is invisible outside of a class). This module declares also some stored objects (*director*, *actor* and *movie*). Therefore, *Studio* module serves as both metadata and data storage.

5 Database Administrator

The database administrator defines the data and methods visible for application programmers. He/she uses the modules provided by database programmers and builds appropriate views over these modules. These views are organized into modules which

are then granted to application programmers. Here is an example module defined by a database administrator over the module *Studio* (see Section 4). It provides database access for person with low credentials (i.e. they can add a director contract to a movie, but they cannot set the pay).

```

module CartoonStudio {
  import Studio;

  virtual director[0..*] : Person = Studio.director;
  virtual cartoon : Movie = (Studio.movie where count(stars) = 0);

  export Person {
    name : string retrieve;
    directs[0..*] : ref Movie reverse director retrieve, insert;
  }

  export Movie {
    title: string retrieve;
    director : ref Person reverse directs update;
  }
} // module CartoonStudio

```

Here, the database administrator has limited the movies only to cartoons. Other movies are not visible. Similarly, only the directors (and not stars) are accessible. The interfaces of provided objects are also restricted, i.e. only the attributes *name*, *title* and the associations *directs* and *director* are available. The only update allowed by this module is to change the director of a cartoon. This example uses so called predefined views, offering simplified syntax applicable for access restriction or data selection.

The following example shows custom views, used here to redefine generic operations to include logging the information on all the changes and retrievals performed.

```

module PersonalData {
  import Studio;

  virtual person[0..*] { (Studio.actor union Studio.director) as p; }
  {
    virtual name : string { p.name as pfn; }
    retrieve {
      create persistent Log (Username & “retrieved data on” & p);
      return deref(pfn);
    }
    update newName {
      create persistent Log (Username & “changed the name of” & p);
      return pfn := newName;
    };
  }
} // virtual person
} // module PersonalData

```

In this module a custom view which provides virtual objects *person* is defined. A single subview *name* is declared. Its generic operations are redefined so as to create a log entry for each update and retrieval. This is an important gain if we compare it to relational databases where triggers fired on retrieval are not available.

6 Application Programmer

The database administrator grants privileges on certain modules to application programmers and/or users. If an application programmer is granted a module, he/she can use all its components (class/export definitions and provided objects). This access is transparent i.e. it does not depend whether an object is virtual or stored. The application programmer simply uses attributes, associations and method of provided objects.

Let us assume that the task of an application programmer is to code a user interface which will display the data on cartoon and directors and allow assigning directors to movies. The user interface could consist of two lists to select names of directors (the list *directorList*) and titles of cartoons (the list *cartoonList*) and a button “Assign” (referenced as *assignButton*). This example shows code of the GUI event handler for event *buttonPress* of this button.

```
module CartoonGUI {  
    import CartoonStudio; import GUI;  
    assignButton : GUI.Button {  
        proc whenButtonPressed {  
            CartoonStudio.Person p =  
                (CartoonStudio.director where name = directorList.currentSelection;);  
            CartoonStudio.Movie m =  
                (CartoonStudio.cartoon where title = cartoonList.currentSelection;);  
            m.director := p;        }  
        }  
    }  
}
```

Note that only the features of the module *CartoonStudio* are used here. The full transparency of virtual data has been achieved, as an application programmer needs not be aware of using virtual data (*CartoonStudio.director*, *CartoonStudio.cartoon*, *m.director* are all virtual!).

7 Conclusions

In this paper we have shown, how object interface specifications, class implementations, view definitions and user profiles could be combined to achieve the necessary transparent schema management with no limitations on the generality of the mapping of virtual data and updating operation.

The database design and maintenance architecture outlined distinguishes three main roles of its participants: the database programmer, the database administrator

and the application programmer. Their cooperation facilitates the management of user privileges and directs the process of database application development. The proposed architecture clearly separates concerns of those three roles involved. It resembles the model-view-controller design framework with the database programmer as the “model”, the database administrator as the “controller” and the application programmer as the “views”. Analogously to the MVC, in the proposed architecture the most of coordinative work is assigned to the database administrator (“controller”) who orchestrates the efforts of both kinds of programmers.

The general framework could be similar to ANSI/SPARC in case of relational DBMSs. However, relational views are too limited in terms of virtual data/operation mapping. Moreover, a number of ODBMS-specific issues needs to be addressed. Our main aim was to employ the updateable view mechanism to serve customized user schemata and to prepare them to be a foundation for user privilege management.

Several related issues were not fully addressed here. Although we assume a full administrator’s control over the user schema construction, some tool assistance for assuring their consistency and controlling dependencies could be desirable. Also the privilege control issue has only been sketched here. User classification, priorities and metadata updating restrictions require additional research.

Currently in our prototype ODBMS we have implemented the updateable view mechanism. The current work includes interface definitions management and its alignment with views and classes. Next, the user schema management functionality, as outlined here, will be realized. For our future research we plan the integration of the dynamic object role support into the schema and an extension of view mechanism to support parameterized views.

References

1. D.C. Tsichritzis, A. Klug (eds.): The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems, *Information Systems* 3, 1978.
2. C.J.Date, H.Darwen. A Guide to SQL Standard. Addison-Wesley Professional, 1996.
3. K.Subieta, Y.Kambayashi, and J.Leszczylowski. Procedures in Object-Oriented Query Languages. Proc. VLDB Conf., Morgan Kaufmann, 182-193, 1995.
4. T. Pieciukiewicz, K. Subieta: *Recursive Query Processing in SBQL*, ICS PAS Report 979, November 2004.
5. H. Kozankiewicz, J. Leszczylowski, K. Subieta: New Approach to View Updates. Proc. of the VLDB Workshop Emerging Database Research in Eastern Europe. Berlin, 2003.
6. R.G.G. Cattell et al: The Object Data Standard: ODMG 3.0. Morgan Kaufmann 2000.
7. Object Management Group: OMG CORBA™/IIOP™ Specifications. http://www.omg.org/technology/documents/corba_spec_catalog.htm, 2002.
8. Object Management Group: Unified Modeling Language (UML), version 1.5, <http://www.omg.org/technology/documents/formal/uml.htm>, 2003.
9. C.J. Date. Encapsulation Is a Red Herring. Intelligent Enterprise’s Database on line, Programming & Design, <http://www.dbpd.com/vault/9809date.html>, 1998
10. M. Torres, J. Samos: A Language to Define External Schemas in ODMG Databases. in Journal of Object Technology, vol. 3, no. 10, November-December 2004, pp. 181-192.
11. K. Subieta. Theory and Construction of Object-Oriented Query Languages. Polish-Japanese Institute of Information Technology Editors, Warsaw 2004, 522 pages.
12. Objectivity. Objectivity for Java Programmer’s Guide. Release 8.0. Objectivity, Inc. 2003.