Abstract - The traditional concept of a database view is currently under influence on new ideas, in particular the CORBA approach to distributed objects, object-oriented databases, XML and other Web technologies, semistructured data, and autonomous mobile agents. These new trends cause widening the scope of database views by novel features, which are beyond the current theoretical and conceptual paradigms of database systems. New concepts related to database views are wrappers, adaptors, mediators and ontologies. The paper discusses features of object-oriented views, which have to fulfill needs and expectations. The discussion is focused on integrating heterogeneous databases into a federated database. Emphasis is put on relaxing the database view concept from limitations induced by stereotypes inherited from the relational data model, to fulfill nowadays requirements concerning efficient integrating heterogeneous data sources and services.

1 Introduction

A database view defines virtual data that are derived from stored data and are compliant to the assumed data model. Such virtual data have to be queried and updated similarly to stored data. From the user viewpoint there should be no differences concerning retrieval and manipulation of stored and virtual data; this property is known as transparency of views. Virtual transparent database views are considered very important for current and future database technologies. The literature presents many expectations concerning such views, which include their potential as an integration tool for heterogeneous database federations. Views allow the designers to map (heterogeneous, autonomous) local databases according to a common federated schema [ChMe97, DG94, Elia95, JoR98, KDN90, Pae93, SIMB94, S92]. Further expectations concerning views concern a higher level of data independence, known as schema evolution [Bell97b, Brat92, BFK95, K98, LDB97]. Views enable the designers to change a database schema without affecting already written applications.

Views are a standard functionality of SQL-based DBMSs. However, many conceivable mappings between stored and virtual data cannot be expressed due to the limited computational power of SQL. View updating is usually limited to views defined as vertical/horizontal restrictions of stored relations (plus other limitations; Oracle, however, allows limited updating of views defined by joins). Moreover, database applications are written in programming languages (such as C/C++), which are tightly coupled with physical data structures. The level of data independence of such languages is too low to process imaginary data. Additionally, various kinds of semantic, representational and schematic discrepancies among heterogeneous data require other features of views (such as full algorithmic power, access to metadata, reflective capabilities, recursion, and others) which are beyond the intellectual scope of the relational model, relational DBMS, and current theoretical approaches to databases.

Hence the mentioned above qualities of views are actually wishes. They are not available for majority of applications. Most frequently, in addition to views written in SQL and stored in a database, the applications involve proprietary solutions written in lower-level languages. For instance, such an approach is assumed in OMG CORBA [OMG95], where interfaces in IDL can be considered virtual views, but mappings between stored and virtual objects (i.e. adaptors and wrappers) are to be written in a regular programming language.

In Oracle new ideas are implemented; in particular views with a check option (preventing undesirable side effects during view updating) and special options ("do instead of") triggers making it possible to control translations of view updates. Perhaps Oracle is also pioneering the integration of views into fully-fledged language for programming database applications (PL/SQL). There is also an advanced proposal concerning views in the emerging SQL3 standard [SQL99]. The user can define views with a check option, updatable, insertable and recursive. Many very particular details of the proposal (10 pages of specification), unclear compatibility with the specification of other constructs of the language (ca. 1100 pages) and the eclectic nature of the standard cause doubts if the idea is reasonable and implementable. Recently significant research is also done into object-relational mapping through special views, e.g. [BKSW91, FaR97, HaKe95, PMSL94, VeAp95]. The approaches address current need to integrate relational databases with object-oriented programming languages and other tools, e.g. CORBA ORBs.

The development of object views is still in the research stage. To the best of our knowledge, no commercial OODBMS offers views. Some capabilities are introduced in the ODMG standard [ODMG00], but
essentially reduced to macro-definitions (the “define” clause of OQL). The approach has many limitations and open issues [Rupp98]. There are several prototypes, in particular, a prototype for O2 [Souz95], MultiView [Rund94, Rund96, KuRu96] and stored functions in Loqis [SMA90, Subi91]. There are also many other papers (mostly theoretical), in particular, [AbBo91, ABS97, BaKe93, Bell97a, Bert92, BuFa99, Daya90, GeZi94, GBCG97, GPZ88, HeZd90, KiKe95, LaSc91, MaBa91, SAD94]. The proposals are incompatible and tightly bound to a particular data model or formalism; thus comparing and evaluating them is not easy; see [Mots96].

In our opinion, the state-of-the-art concerning utilization of object views as tools for developing heterogeneous database federations should be considered premature at this time. In this paper we try to recognize major issues that has to be considered and discuss various research aspects of object views.

The rest of the paper is organized as follows. In Section 2 we present architectural issues concerning federated databases. Section 3 is devoted to novel concepts: wrappers, mediators and ontologies. Section 4 discusses the potential of object models as abstract canonical ontologies. The section presents also various aspects of object views. Section 5 discusses the view updating problem. Section 6 concludes.

2 Architectural Issues

Fig.1. Architecture of a federated databases

The basic architecture of a federated system is presented in Fig.1. It consists of the following parts:

- Local, autonomous databases supported by local database management systems.
- A local database is accessed by a local application programming interface (API), which exists before integrating the database within the federation.
- A local API is used by a wrapper, which is dedicated to a federated database. The role of a wrapper is to map a local API into a programming interface expected by the federated system.
- A wrapper makes it possible to build a schema mapping between API of a federated system and API of a local system.
- On the side of the federated system there is an API_f of a federated system. API_f is used by global applications to access resources stored in local databases. The access supports transparency: from the point of view of a programmer all local resources are seen as a single centralized database.
- There is some network protocol, which is prepared to connect a global application to local resources.
- Global applications may need an own working database. In particular, it is necessary for storing copies of data sent from local databases.

Fig.2 present a more detailed view on the architecture from the position of an administrator of a local database.

Fig.2. More detailed architecture of a local database

In Fig.2 we have shown an API and a wrapper which adapt the local database to external requests. Assuming autonomicity of a local database the local database administrator (DBA) is an authority having two kinds of functionalities:

- Define a part of the local database, which can be accessed from global applications. The functionality restricts accesses to the local database from the side of
federation. Moreover, it customizes (adapts) the local database to the standard of the federated schema.

- Determine rights of particular users to access particular resources of the local database (R - read, R/W - read and write, etc.).

Taking a bit idealistic view\(^1\), the local DBA has two kinds of facilities to determine the local export schema:

- **Interfaces.** DBA can define many interfaces to the same data. Interfaces are understood as primitive views, which only restrict and rename stored properties.

- **Views.** They enable DBA to define more sophisticated mapping of local database into federated schema. Views having the properties of interfaces (mentioned above) plus selection capabilities are referred to as data-preserving views. In many cases such views are insufficient.

Interfaces and views defined by the local DBA implicitly define a local export schema. Such a schema must be also written explicitly (e.g. by some automatic facility) as a part of a schema repository. The repository has the same role as the interface repository in CORBA. The schema repository is necessary for global applications e.g. to decompose a global query into sub-queries addressing particular local databases.

Another functionality of the local DBA is presented conceptually in the form of an accessibility matrix. It enables the administrator to grant access rights for particular users. In this way each user sees the local database through its own export sub-schema. For some applications this functionality might be omitted, i.e. from the position of the local DBA all users of global applications have the same rights.

The sum of all local exported schemas constitutes the entire federated schema. Analogously, the sum of all exported sub-schemata for a particular user constitutes his/her external schema to the entire federated database.

The general scenario of processing requests (queries) which uses this architecture is the following:

- A global application's request (a query) refers to the global view (the federated schema).
- The request is decomposed into sub-requests addressing particular local object views.
- Each sub-request is mapped by a local view mapping mechanism and wrapped into a request addressing local data.
- Such a request is executed within a local database. The result is mapped through the wrapper, then through the local view and returned to the global application.

- The global application collects results from all sub-requests; then integrates the particular result into a global result.

This scenario is a bit idealistic. For various reasons, e.g. optimization, it can be modified. The scenario may assume sequential processing of sub-requests: a next sub-request is generated when previous sub-requests have returned results (in order to make the next sub-request more precise, e.g. processing based on semi-joins). It is also possible to send the result of a sub-request addressing local database A to a local database B; then, send a sub-request to B; it addresses data stored in B plus data sent from A. Another method: for the given global request the federated DBMS collects all the data from local databases, which are sufficient and minimal for accomplishing the request. This method implies generating for the given request sufficient and minimal sub-views addressing local databases.

### 3 Wrappers, Mediators and Ontologies

Recently Internet, Web and development of fast and wide computer networks have caused a big pressure to integrate heterogeneous, geographically distributed data, which are not necessarily under the control of some DBMS. New terms are coined to denote various situations related to integrating heterogeneous data sources. **Wrappers or adaptors** (coming from the CORBA and Internet camps) are specialized program modules making it possible to map an internal API to the external API expected by some global application. Usually the mapping rises the level of abstraction, for instance, in CORBA applications the internal API is a collection of some C++ functions while an external API is an IDL interface. Wrappers are considered non-sophisticated programs for changing data representation, names of functions, etc. Moreover, wrappers are usually dealing with data units of low granularity (e.g. tuples rather than tables) and mapping is 1:1, i.e. one "internal" data granule (e.g. a relational tuple plus subordinated tuples) is conceptually mapped into one "external" data granule (e.g. a CORBA object). A wrapper can be considered a "physical bridge" between local databases and global applications.

Another concept is mediator [Wied92]. It comes from the camp of data warehouses and integrating semi-structured data. Till now, however, the concept is intuitive and defined pragmatically rather than semantically or theoretically. A mediator has to resolve (semantic, representational, structural, schematic) incompatibilities among data to be integrated. A similar role is assigned to database views; moreover, there are proposals, which treat mediators as virtual views [BRU96]. Perhaps the conceptual border between database views and mediators is difficult to define, but there are essential pragmatic

---

\(^1\) In many cases the local DBA uses specific facilities, which cannot be classified as interfaces or views.
differences, which allow us to consider mediators as a new quality:

- Mediators can act on irregular data and can produce irregular data, while database views act on formatted data (e.g. tables) and produce formatted data.
- According to (false) stereotypes of the relational model, a database view is to be defined by a single query (e.g. in SQL). However, current query languages have the power much below the algorithmic power. Mediators do not assume such limitations. The designers of mediators can use the full power of algorithmic programming languages as well as advanced higher-level methods, including methods attributed to artificial intelligence.
- Mediators can use any resources from the computer environment (e.g. data written in files) while database views can use only resources stored in a database.
- Database views themselves are stateless: each invocation of a view cannot utilize information from previous invocations of the same view. (Stateful views are referred to as capacity-augmenting views; they are considered yet as a research topic.) Stateful mappings cannot be avoided in many global (federated) applications. Mediators can be stateful with no limitations.
- Database views are self-contained named entities defined in a special syntax of a query language and stored in the database as identifiable persistent units. In contrast, mediators are programs written in a regular programming language and stored as a part of an application program in main memory. Thus, mediators have much lower abstraction level and maintenance potential than database views.

Another term related to integrating heterogeneous data sources is **ontology**. An ontology is a formal theory or description of a being or reality. The term has a long history in philosophy, but it comes to the database world from the world of artificial intelligence. In this context the term has a technical meaning. It is understood as a formal description of everything that the designer or programmer has to know about some particular data/service environment to use it correctly and efficiently in a particular application. Note that ontology is a more general term than a database schema or a database view. From the database perspective an ontology includes formalization of the following issues:

- The meaning, pragmatics and behaviour of data and data services in the (business) reality;
- Static and dynamic constraints among data; cause-effect relationships;
- Associations among data and services (methods, procedures, etc.) acting or utilizing data;
- Peculiarities of data access, maintenance and utilization;
- Performance concerning data accesses and processing;
- Privacy, safety and security issues; transaction processing;
- Exceptions, error messages, extreme cases, null values, side effects, and other exceptional issues;
- Associations and other semantic dependencies among data;
- Issues related to API to data and data services, in particular, syntax, semantics and operational pragmatics of a query language;
- Data naming conventions; meaning of data names in the reality;
- Peculiarities of data representation;
- Metadata information related to data and services.

Formally, an ontology is defined as a specification of conceptualization [GrOl94], i.e. a description (like a formal specification of a program) of the concepts and relationships that can exist in a particular data/service environment. The postulated "formal specification" is an idealistic (and obviously utopian) assumption promoted by AI theoreticians. As a matter of fact, current data/service environments are too complex to specify them by a formal logic (or even by any mathematical framework). In reality some elements of an ontology can be defined formally (for instance, schema and meta-schema), some half-formally (for example, the semantics of object interfaces), and some informally (for example, mappings between objects and the business domain). Nevertheless, ontology is a useful concept, because it allows us to denote the real scope of issues that the designers of heterogeneous systems have to deal with, without binding them to current limited database models and theories.

Different data servers have different ontologies. Even if the servers are supported by the same DBMS, the differences may concern semantics of data in the business domain, structuralization of data, representation of data, etc. Having a common ontology enables collaborators to work together with minimal risk of misunderstanding each other [Wied94].

The task of designers of federated applications based on heterogeneous data/service sources can be subdivided into the following steps:

- Understanding ontologies of all participating data/service sources and describe it (formally, semi-formally or informally);
- Developing a common ontology which will satisfy the needs of future global applications, and on the other

---

2 Papers concerning mapping between heterogeneous ontologies for autonomous mobile agents (considered by AI) remain early theoretical ideas discussed by the database camp in late 70-ties. From that time the database domain has made important steps to full awareness of the problem. It seems not to be fully recognized yet by the AI camp.
hand, will correspond to all participating data/service sources; such a common ontology is called canonical.

- Developing a formal mapping between ontologies of participating databases and a canonical ontology.

Any low-level or unspecified details of ontologies decrease the potential of common understanding. Hence the requirement that a canonical ontology has to be defined on a high abstraction level, which hides implementation details of participating ontologies. This point of view is taken by OMG CORBA, which uses some variant of an object model as a basis for a canonical ontology.

In OMG CORBA the mapping between a particular ontology and a canonical ontology is determined through wrappers and adaptors written in regular programming languages. The CORBA object model, although powerful, is not sufficient from the database perspective. A more powerful object model, which includes collections, relationships and other features of databases, is proposed by ODMG [ODMG00]. Still, the ODMG object model is considered limited for many applications. Moreover, it is recently under some critics concerning the insufficient level of precision and consistency (see papers by the authors of the IRO DB project, papers by Suad Alagic, and other sources).

Developing a mapping between a particular ontology and a canonical ontology is frequently a difficult task. For various types of discrepancies between data and services the mapping cannot be reduced to defining views in some query language. The mapping requires deep understanding of formal properties of data and services, as well as understanding of the meaning of data in the business domain. This goal requires query/programming languages and tools, which will be either powerful (universal), abstract, and simple to use. In the following we discuss the potential of object views to fulfill this role.

4 Object Models as Abstract Canonical Ontologies

The object-orientedness is currently the major paradigm of many domains of computer technologies, including the analysis and design of software systems, programming languages, interoperability, component-based software, databases, visual programming interfaces, and others. Thus the natural expectation that the object-oriented paradigm is a viable candidate for developing a canonical ontology for heterogeneous database federations. There are also expectations that the research into object views will result into a powerful and simple tool for determining mappings between ontologies.

Object database models introduce many notions thus the formalization of their features (data structures, query languages, views, etc.) is more complex than in the relational case. There are many proposals: the ODMG object model and approaches to formalization of OQL, non-normalized relations (NF2), a variety of object algebras, F-logic, comprehensions, monoid calculus, and other, see e.g. [AHV95, AlAr93, BNPS92, BRU96, Clue98, DaDe91, FeMa95, GKGH+97, HeSc91, KiLa89, LaSc93, LMSV+93, ShZd90]. There is little agreement which model is adequate for investigations concerning object views.

In general, capabilities of view definitions can be a subject of various assumptions. The most comfortable case is when views are provided as an additional facility for retrieval. In this case the user can use the new possibility or not, depending on the problem, familiarity with the new option, his/her customs, etc. On the opposite extreme we have a non-comfortable case, when views are to be used for integrating hundreds heterogeneous databases, according to some federated ontology, which includes data model, programming interfaces, naming conventions, representation of values, semantics of values in the business domain, exchange formats, etc. In such a case the capabilities of view definitions are to be comparable to the capabilities of universal programming languages, including access to metadata, reflection, recursion, full algorithmic power, advanced view updating, complex data structures, and many other features. For instance, resolving schematic discrepancies among heterogeneous databases [KLK91] requires access to metadata and reflection. Similarly, dynamic invocations within CORBA environments require accesses to an interface repository, which is a recursive structure. Assuming it will be seen through database views, it should be possible to use recursive views, queries with transitive closures or other equivalent facilities.

A basic misconception concerning object views is that proponents of ideas do not realize that the second, non-comfortable extreme presents a very wide and unexplored terra incognita. The typical paper at the beginning contains promises addressing the non-comfortable extreme. Then, there is a concrete syntax and semantics, which is frequently advanced and complex, but obviously close to the first extreme: a view definition is reduced to a single query in some limited query language (much below the algorithmic power), with a lot of additional options, perhaps important, but secondary. This style of research papers cause that the progress in the domain is difficult to evaluate and cause frustrations [DVB93].

Generality of object-oriented views requires many features:

- Restricting and renaming attributes in virtual objects (cf. SQL);
- Possibility to define virtual attributes;
- Views joining information from two or more (stored or virtual) objects;
- Views with parameters, recursive, and higher-order; views defined by complex algorithms, with the
possibility of access to metadata and reflective capabilities.

- Updatable views supporting the transparency principle. The correct translation of view updates is a big problem, which till now has no proper treatment. Majority of approaches to object-oriented queries is unable to correctly formulate this problem, because updating operations are not expressible in proposed formal frameworks, e.g. an object algebra.
- Integration of notions such as strong typing, classes, encapsulation, inheritance, methods, etc.
- Development of methods to avoid performance penalty due to view processing.

To be more close to the non-comfortable extreme of view definition capabilities there is a need to extend the scope in which object views are typically set. Object views should follow the paradigms of programming languages rather than traditional database concepts, such as relational/object algebras or calculi. Assuming queries are programming languages’ expressions, the concept of programming languages that is analogous to database views is a functional procedure (i.e. a procedure returning an output) [SKL95, Subi97]. If a first-class, dynamically bound procedure is stored in a databases and returns a collection, then an invocation of the procedure is semantically equivalent to execution of a query. Such a procedure will be defined by queries and can be called within queries.

The goal of the research into such object views is to integrate the above idea with query languages into a homogeneous, semantically simple, consistent and universal system.

### 4.1 Performance of Object Views

The majority of approaches to view objects assume (sometimes implicitly) that virtual objects are eventually calculated when the corresponding view is used in a query. For many applications such an approach will result in very bad performance. Relational systems use a query modification technique, which practically avoids any performance/storage overhead related to processing views. Query modification assumes macro-substitution of view invocations [SuPI00] and then optimizing resulting queries by rewriting [PISu99, PIKr99, Plod00]. A big advantage of query modification is that in many cases it enables using other very efficient techniques, for example indices and other access support data structures.

Another optimization technique, known as predicate move-around [LMS94], is similar to query modification, but works in otherwise direction. The technique assumes that a predicate occurring in a query invoking a view modifies the view definition. The technique can be efficient when a view is defined by a complex algorithm (thus query modification is not applicable).

Other optimization techniques, known as materialized views, assume various forms of caching results returned by view invocations.

### 4.2 Virtual vs. Materialized

In principle, views are virtual, i.e. they exist only as a definition of a mapping between stored data and imaginary data. For several reasons (optimization, security, accessibility, parallelization, etc.) many authors have proposed materialized views. They assume that the result returned by view invocation is calculated in advance. A disadvantage of materialized views concerns updating of stored data, which implies updating materialized views. To cope with this issue various techniques are proposed. The majority of them assume incremental updating, i.e. updating the minimal part of a materialized view, which is to be influenced by updating.

To accomplish this kind of algorithm the view definition must satisfy the distributivity property. It can be roughly formulated as:

\[
v(s_1 \oplus s_2) = v(s_1) \oplus v(s_2)
\]

where \(v(s)\) is the result returned by the view \(v\) for database state \(s\); \(s_1, s_2\) are any data states that the view \(v\) concerns; and \(\oplus\) is a state composition operator (e.g. union of sets). The distributivity property makes it possible to keep a 1:1 relationships between database objects and elements of a materialized view.

An interesting variant of a materialized view is when it stores not newly calculated data, but references to stored data. For instance, this kind of view can be defined in Loqis as a function:

```plaintext
function RichEmp
    begin
        return EMP where SAL > 2000 end;
```

The function returns references (OIDs) to object of employees earning more than 2000. Such a view shares (to some extent) advantages of virtual and materialized views. In particular, updating of any attribute (except \(SAL\) in \(EMP\) objects causes no changes in the materialized view. Actions on the view are necessary in the case of deleting and inserting \(EMP\) objects and in the case of updating the \(SAL\) attribute in any object. The idea can be generalized to materialized views which return not only references to objects, but also references to attributes and methods.

### 4.3 Views Defined by Complex Algorithms

Traditionally, a database view is to be defined by a single query. For complex mappings between local and federated ontologies this assumption leads to severe limitations, because the power of all known query languages (including SQL and OQL) is much below the algorithmic power. In Loqis this limitation is avoided. For example, the \(WellPaid\) function has a parameter \(JobPar\) and a local variable \(a\):
function WellPaid( JobPar )
begin
    real a := avg(EMP.SAL);
    return EMP where JOB = JobPar and SAL > 2*a;
end;

Get companies where well-paid clerks used to work previously:

WellPaid( "clerk ").PREV_JOB.COMPANY

Although the presented "algorithm" is very simple, in general there are no limitations. A wider discussion concerning various features of object views implemented in Loqis (views defined by joins, virtual attributes, virtual links, etc.) can be found in [SuPl00].

4.4 Object Views as Database Modules

The relational model has introduced some ambiguity concerning the term "view". Essentially, database design methodologies as well as the ANSI/SPARC architecture regard a view as some part of the database schema, or (more precisely) as some mapping between a database schema and an external (user) schema. In contrast, an SQL view is essentially a programming function (like functions or functional procedures in Pascal or C) which maps stored tables into a virtual table.

In general the SQL approach to views is limited. A view should be defined as a module, which stores several named functions (a la SQL views). Such a module should follow the maintenance semantics of SQL views, i.e. it can be stored in a database and deleted from the database. A module being a view can store other run-time entities, in particular:

- Common private functions;
- Public and private classes necessary for the view definition; methods within these classes;
- Constraints and rules; for instance, rules for view updating;
- Public and private persistent data; for instance, data necessary for capacity-augmenting views.

5 Updatable Views

The majority of research and development devoted to heterogeneous database federations assume that global applications are based purely on retrieval from local databases. Some applications, however, strongly depend on updating. In case of heterogeneous database federations this updating is to be done through local views and wrappers.

Updatable views, which follow the transparency principle (i.e. the user does not know whether she is working with stored or virtual data) are still in infancy. Typical theoretical paradigms (relational algebras, relational calculi and formal logics) do not deal with updating operations. Thus updating is to be introduced by some ad hoc enhancements to these formalisms, which - as a rule - are not sufficiently general and not well understood. In effect, view updating was limited to particular cases. Another disadvantage of the relational research into view updating concerns some naive (implicit) assumptions that the transformation of view updates has to be done in some auto-magical way, e.g. by utilizing functional dependencies and other properties of relations stored in the database. In general, transformations of view updates have to be controlled by the view definer by special options dedicated to this purpose. This is eventually accomplished in Oracle-8, but the solution is not sufficiently general yet and has no theoretical or conceptual background.

Object databases open new horizon for the view updating problem. Some part of the object databases camp follows more general theoretical paradigms based on the semantics of programming languages rather than (obsolete and inefficient) paradigms based on algebras, calculi or logic. From the position of the programming languages’ semantics (including operational and denotational semantics) updating operators are regular operators, thus the research concerning view updating can be based on a stable fundament.

Concerning updating of object views we can distinguish two cases:

a) Data can be updated by generic operations, i.e. assignment, deleting and inserting.

b) Data cannot be updated by generic operations. Any updating of data objects has to be done by methods (following the narrow understanding of encapsulation).

The case b) implies no special conceptual problems with view updating. In this case the view definer has to define a view together with a class that virtual objects will belong to. The class stores all the methods (written or adopted by the view definer) necessary for updating virtual objects. During writing the methods the view definer has full access to stored data objects, thus can update them with no limitations. There is some danger of inconsistent updates, thus one research problem concerns automatic discovering of them. Another problem concerns connecting the objects generated by a view invocation to the methods stored within the class. Assuming views are dynamically bound and methods are statically bound this can imply technical difficulties.

In majority of cases we cannot avoid generic updating operations. However, the case a) is more challenging for researchers. The simplest solution we obtain assuming that a view invocation returns references to stored objects rather than new objects. These references can be used as l-values in assignment and deleting statements. (Inserting is problematic.) The case is generalized in Loqis, where a

---

3 The understanding of the concept of encapsulation can be different. Originally (as proposed by D.Parnas) it means subdividing object's properties into public and private, independently of the property's kind (c.f. Modula-2 and C++). Many authors are trying to promote the narrow view, where only some methods are public; other properties are private.
view invocation can also return structures built upon references to attributes. This allows one to update views defined by complex queries, including queries defined by joins. Such an updating, however, may lead to undesirable side effects, thus additional syntactic and semantic features are necessary to prevent them.

The most general case (currently under investigations) is based on virtual object identifiers. They are returned by view invocations. Virtual identifiers are complex structures storing all the information about the "origin" of the virtual object. Virtual identifiers make it possible to overload generic view updating operations by specific methods defined by the view definer [SuMi89]. During writing the methods the view definer has full access to stored data objects, thus can update them with no limitations (similarly to the case b)). The overloading methods are fully transparent for the view user: she is using generic updating operations with no differences. Although conceptually the method seems to be viable and implementable, there are many details (optimization, in particular), which so far are not sufficiently investigated.

6 Conclusion

The paper is presents a research agenda concerning objects views considered as a tool for mapping heterogeneous ontologies into a federated ontology. We have tried to show that many research efforts devoted to this topic present too narrow understanding of the problem, thus the produced results we consider premature. The complexity of database ontologies requires novel approaches, which are going far beyond the traditional paradigms known from the relational model. These approaches should assume a rich and abstract canonical object model, in order to cover all conceptual issues that may occur in local databases to be integrated. Moreover, object views should be defined as complex database modules, having full algorithmic power and dealing with complex data structures and abstractions.

7 References


[Daya90] Dayal, U. Queries and Views in an Object-Oriented Data Model. 2nd DBPL Workshop, Morgan Kaufmann, 80-102, 1990


4 Acronyms of journals and conferences according to the M. Ley DBLP Bibliography, http://www.informatik.uni-trier.de/~ley/db/index.html


