# Implementation of Federated Databases through Updatable Views

Hanna Kozankiewicz[1], Krzysztof Stencel[2], Kazimierz Subieta[1,3]

[1] Institute of Computer Sciences of the Polish Academy of Sciences, Warsaw, Poland
{hanka,subieta}@ipipan.waw.pl
[2] Institute of Informatics Warsaw University, Warsaw, Poland
stencel@mimuw.edu.pl
[3] Polish-Japanese Institute of Information Technology, Warsaw, Poland

**Abstract.** We present a new approach to the grid technology that is based on federated databases and updatable views. Views are used in two ways: (1) as wrappers of local servers that adopt local schemata to the federated database requirements; (2) as a facility for data integration and transformation into a canonical form according to the federated database schema. Views deliver virtual updatable objects to global clients. These objects can be associated with methods that present the procedural part of remote services, like in Web Services. The fundamental quality of the approach is transparency of servers: the user perceives the distributed environment of objects and services as an integrated virtual whole. The approach is based on a very simple and universal architecture and on the stack-based approach, which treats query languages as a kind of programming language.

## 1 Introduction

Grid technology provides a new information processing culture that integrates many local services into a big virtual service, summing the resources belonging to particular services. The grid user is interested in services rather than in service providers. For instance, if the grid integrates thousands of small bookstores, the customer is looking for a cheapest required book and can buy it without interesting in concrete bookstores offering this book (see Amazon's Z-shops). The service providers transparency has to be supported by many technical forms of transparency that facilitate designers and programmers of grid applications.

In this paper we focus on data-intensive applications of the grid technology where distribution of data implies distributed and parallel computing. From this point of view, the technology can be perceived as continuation of distributed/federated databases – the topic that has been developed for many years. The domain of federated databases worked out many concepts that are very close to the current grid research, such as various forms of transparency, heterogeneity, canonical data models, transaction procession within distributed federations, metamodels for federations, etc. In our opinion the concepts, ideas and solutions worked out by the distributed/federated databases domain must sooner or later be absorbed by the field of grid technology.

The key issue behind data integration in grid is *transparency* i.e., abstraction from secondary features of distributed resources. There are many forms of transparency like location, access, concurrency, implementation, scaling, fragmentation, replication, indexing, or failure transparency. Due to transparency (implemented at the middleware level) some complex features of distributed data/service environment need not to be taken into account in the application code. Thus, transparency much amplifies programmers' productivity and supports flexibility and maintainability of software.

In this paper we present a new approach to implementation of several forms of transparency based on universal architecture for integration of local, autonomous databases. Data and services of a particular server are made visible for global applications through a *wrapper* that virtually maps the data/services to some assumed *canonical object model*. Then, all contributing services are integrated into the virtual whole by means of a *updatable views*. In this way all participating local data and services are seen through a single federated schema.

The novelty of the presented approach is that we propose to use updatable object-oriented views with full computational power. Views are defined in the query language SBQL that is integrated with imperative constructs (e.g. updating) and abstractions (functions, methods, procedures). The idea of using views for integration of distributed/federated databases is not new (see e.g. [Bell97, Subi00, Hal01]) and there are some prototype implementations [KLN+04, KW96]. However, to the best of our knowledge the issue is still challenging because of updatability and practical universality of view definitions. In our recent research [KLS03] we have developed and implemented object-oriented virtual views that have full algorithmic power and are updatable with no anomalies and limitations. Our views support full transparency of virtual objects, i.e. the programmer is unable to distinguish stored and virtual objects. Such views are considered as a general facility for integrating distributed and heterogeneous resources, including methods acting on virtual objects. Views deliver virtual data to the clients of a federation. Therefore, such views offer similar facilities as Web Services or a middleware based on CORBA. The advantage of our approach is that it offers a very simple architecture that is much more flexible and universal than the technologies mentioned above.

The rest of the paper is structured as follows. In Section 2 we describe Stack-Based Approach and its query language SBQL. In Section 3 we sketch the approach to updatable views. In Section 4 we explain the ideas of our grid approach. In Section 5 we present an example application of the approach. Section 6 concludes.

## 2 Stack-Based Approach

Our grid mechanism is based on the Stack-Based Approach (SBA) to query languages [SKL95]. SBA aims at integrating the concepts of programming languages and query languages. Therefore, queries are evaluated using mechanisms that are common in programming languages, such as stacks.

### 2.1 SBA Data Store

In SBA each object has the following properties: a unique internal identifier, an external name, and a value. Basing on the value we distinguish three kinds of objects:

- atomic object where the value is an atomic value (number, string, blob, etc.);
- link object with the value being a pointer to another object;
- complex object where the value is a set of objects (the definition is recursive and allows one to model nested objects with no limitations on nesting levels).

An SBA store consists of: the structure of objects/subobjects (as defined above), identifiers of root objects (starting points for queries), and constraints.

### 2.2 Environment Stack and Name Binding

SBA is based on the programming languages' naming-scoping-binding principle. Each name occurring in a query/program is bound to a proper run-time database/program entity according to the name scope. Scopes for names are managed by means of an Environment Stack (ES). The stack supports the abstraction principle what means that a programmer can consider a piece of code to be independent of the context of its use.

ES consists of sections, which contain entities called binders. Binders relate names with run-time objects and are used during binding names. A binder is a pair ($n$, $i$), written as $n(i)$, where $n$ is an object name, and $i$ is an object identifier.

New sections on ES are built by means of a special function *nested* that works in the following way:

- For the identifier of a link object the function returns the binder of the object the link points to.
- For a binder the function returns that binder.
- For a complex object the function returns binders its sub-objects.
- For structures $nested(\text{struct}\{x_1, x_2, ...\}) = nested(x_1) \cup nested(x_2) \cup ...$
- For other elements the result of *nested* is empty.

The process of name binding is the following. When the query interpreter wants to bind name $n$ occurring in a query it searches ES for a binder with the name $n$ that is the closest to the top of ES. The process of binding respects scoping rules what means that some sections of ES can be not visible during the binding. The name binding can return multiple binders and this way we can handle collections.

In this paper w present examples basing on a simple database that keeps information on Books in a form Book(title, author, ISBN, price). Example ES states during evaluation of a query *Book* **where** *author* = **"**Niklaus Wirth" are illustrated in Fig. 1.

In the initial and final states the ES contains only base section with binders to root objects. During evaluation the query the operator **where** puts on ES a new section with the environment (internal objects) of the particular Book object. This section is created by means of the function *nested* which for a book $i_d$ returns binders: *author*($i_w$), *title*($i_x$), *price*($i_y$), *ISBN*($i_z$). After the query evaluation ES returns to the previous state.

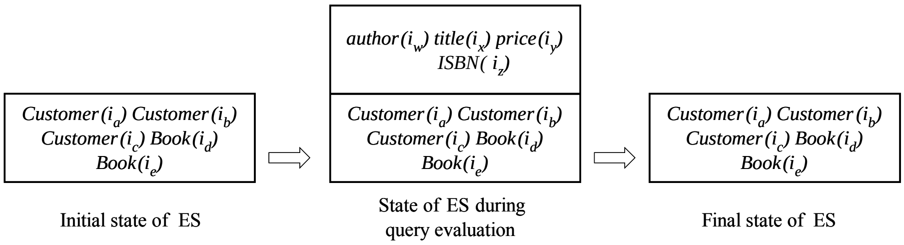


**Fig. 1.** States of ES during evaluation of query *Book* ***where*** *author* = **“**Niklaus Wirth”

### 2.3 Stack-Based Query Language

SBA has its own query language – Stack-Based Query Language [SKL95] that is based on the principle of compositionality what means that more complex queries can be built of simpler ones. In SBQL queries are defined as follows:

1. A name or a literal is a query; e.g., 2, “Niklaus Wirth”, *Book*, *author*
2. $\sigma$q, where $\sigma$ is an unary operator and q is a query, is a query; e.g., *count*(*Book*), *sin*(*x*).
3. $q_1$ $\tau$ $q_2$, where $\tau$ is a binary operator, is a query; e.g., 2+2, *Book.title*, *Customer* **where** <condition>

Due to the compositionality discipline that recursively defines semantics of complex queries on semantics of its components, up to primitive queries (literals and names), the whole semantics of SBQL is precise, minimal, easy to implement and optimize, and easy to understand by the users.

SBQL can be extended to support procedures. They can be defined with or without parameters, can have local environment, can have side-effects, and can call other procedures (including recursive calls).

## 2 View Mechanism

A view is a mapping of stored data into virtual ones. In the classical approach (SQL) a view is a function, similar to programming languages’ functions. View updating means that a function result is treated as an l-value in updating statements. Such an approach, however, appeared to be inconsistent due to problems with finding unequivocal mapping of updates on virtual data into updates on stored data. In our approach to view updates [KLS03] a view definer can include into a view definition information on update intents for the given view. It allows us to eliminate ambiguities connected with multiple ways of performing a view update and the risk of warping user intention, which is a well-known problem.

Therefore, in our approach a view definition consists of two main parts. The first part describes a mapping between stored and virtual objects. The second part describes operations that can be performed on virtual objects. A view definition can contain other elements like definition of subviews, procedures, internal state variables, etc.

The first part is a procedure that returns entities called *seeds* that ambiguously identify virtual objects. Seeds are also parameters for the procedures describing operations on virtual objects that are defined in the second part of the view definition. In the approach we identified four generic operations on virtual objects:

- Deletion of a given virtual object.
- Update that changes value of a given virtual object. The operation has a parameter that is a new value of the object.
- Insertion of a new object into a given virtual object. The operation has a parameter that is an object to be inserted into the given virtual object.
- Dereference that returns a value of the given virtual object.

If an operation is not defined, it is forbidden. Internally, each operation on a virtual object has a fixed name – respectively *on_delete*, *on_update*, *on_insert*, and *on_retrieve*. The names of parameters of operations *on_update* and *on_insert* can be freely chosen by the view definer. The name of a view definition is different from the name of virtual objects – we introduced a naming convention where name of the view definition is written with a suffix "Def".

Here, we present an example definition of a view returning books that have never been bought in a bookstore:

```
create view worstSellingBookTitleDef {
        virtual objects worstSellingBookTitle {
                        return (Books where count( bought_by ) = 0) as b; }
        on_retrieve do { return b . title;}
        on_update new_title do { b . title := new_title; }
}
```

The view defines the following operations: dereference that returns title of the book, and update that changes the title of the book to the new value. We can call the view:

( *worstSellingBookTitle* **as** *wsb* **where** *wsb* = "XMl" ) := "XML"

Mind, that in the query we call operation *on_retrieve* when comparison "=" is performed and operation *on_update* when ":=" is performed.

In our approach views can be nested, with no limitations concerning number of nesting levels. We follow the principle of relativity what means that at each level of nesting hierarchy a view is defined using the same syntax, semantics, and pragmatic. Subviews of the given view are seen as subobjects of a given virtual object.

# 3 Novel Approach to Integration of Federated Databases

There are many aspects of the grid technology that have to be considered for real applications, in particular, transparency, security, interoperability, efficiency, computational universality, etc. Here we concentrate only on the transparency and computational universality aspects, presenting a general architecture of a grid application and some discussion concerning adaptation of SBA to this architecture.

## 4.1 Architecture

The heart of our approach to federated databases is *the global virtual object and service store* (we will use also a shorter term i.e. the *global virtual store*). It stores addresses of local servers and has some computation power. It involves a query engine with ES. *Global clients* are applications that send requests and queries to the global virtual store. Global virtual store is stateless thus can be replicated by many clients.

The *global schema* is a collection of definitions of data and services provided by the global virtual store. Application programmers are the main users of these definitions (also the grid organizer uses it; see below). The programmers make use of them, while they are creating global clients. The global schema is agreed upon by a consortium (of course it may be a single company) that funds the creation of the grid.

The grid offers services and data of *local servers* to these global clients. The *local schema* defines data and services inside a local server. The syntax and semantics of these schemata as well as the natures of the data and services can be very distinct at each local server. They can be written in e.g. OMG IDL, WSDL and ODL. However, this schema is invisible to the grid.

The first step of the integration of a local server into the grid is done by the administrator of this local server. She has to define the *contributory schema* which must conform to the global schema (we will describe this conformance later in this paper). It is the description of the data and services contributed by the local server to the grid. The local server's administrator also defines *contributory views* that constitute the mapping of the data and of the local server to the data and services digestible to the grid.

The second step of the integration of local servers into the grid is the creation of *global views*. These views are stored inside the global virtual store. The interface of them is defined by the global schema. They map the data and services provided by the local servers (in the contributory schema) to the data and services available to the global clients (in the global schema).

Such a grid would be useless if we had not had updatable views. The global clients not only query the global virtual store but also request its updates. The same concerns the grid that not only queries the contributory views but also updates data of it.

Note that the view in SBA are just complex objects, thus they can contain methods and procedures as well as the local variables that store the state of these views. Therefore these views can offer the same facilities as CORBA or Web Services. There is no limitations, because SBQL has the power of universal programming languages.

The *communication protocol* is the collection of routines used in the definition of the global views. It contains the functions to check e.g. the state (up or down) of a local server and the access time to a local server.

The global views are defined by the *grid designer*, which is a person, a team or software that generates these views upon the contributory schemata, the global schema and the integration schema. The *integration schema* contains additional information how the data and services of local servers are to be integrated into the grid. The integration schema does not duplicate the definitions from the contributory schemata. It holds only the items that cannot be included in the contributory schemata, e.g. the way to integrate pieces of a fragmented object the relationships among local servers that they are not aware of. The integration schema is used during the creation of views of the global virtual store.

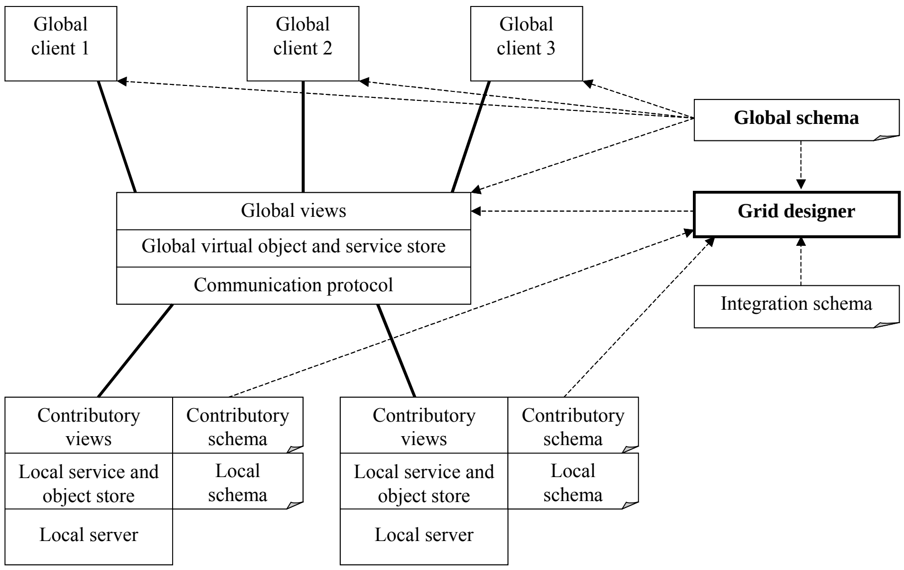


**Fig. 2** Architecture of the GRID

The architecture of GRID is depicted in Fig. 2. In the figure solid lines represent runtime relationships i.e., queries, requests, and answers. Global users ask global view for resources and the global view request for some resources from local servers.

In the figure dashed arrows illustrate association that are used during development of the grid software. The programmers of global applications (global clients) use the global schema. The global views conform to the global schema. The grid designer uses the contributory schemata, the global schema and the integration schema in order to develop a global view.

### 4.2 Adaptation of the Approach to SBA

In the approach global views have to have access to information stored at local servers. Therefore, the global virtual store keeps the addresses of local servers in the form:

(*oid*, *server's name*, *server's address*)

In this triple *oid* is the internal identifier inside the global virtual store, *server's name* is a human-readable name (e.g. Cracow) and *the server's address* is the description of the physical location of the server (e.g. URI or URL). We call such a triple a *server link object*. A server link object looks like a link object. However, when someone navigates into it, it behaves like a complex object.

Let us assume that we have a server link object ($i_C$, *Cracow*, *IPCracow*). Conceptually when identifier $i_C$ is processed (e.g. when query *Cracow.Book. title* is evaluated), ES is augmented with all root objects of the local server located at *IPCracow*. More formally the value of *nested*($i_C$) is equal to the set of all the root objects stored by the server identified by $i_C$ (in this case server *IPCracow*). This process should be the subject of query optimization, to avoid sending to the global virtual store too many information. Conceptually all the root objects migrate onto the global virtual store's ES. Physically however local query engines can evaluate certain parts of queries (like *Book . title* in this case).

When object references fall onto the top of the global store's ES, they are no longer sufficient to identify objects. The local object identifiers are unique only inside local servers. Thus when a local object identifier $i_O$ finds its way onto the global store's ES, it is implicitly converted to *global object identifier* i.e. pair ($i_{LS}$, $i_O$) where $i_{LS}$ is the identifier of the server link object.

Therefore, the binders on the global store's ES can hold the identifiers of server link objects, the global object identifiers and possibly references of necessary local objects of the global virtual store.

## 5 Example Application

In this example we show that our approach can deal with replicas of data. We assume that we deal with multiple bookstores (respectively, located in Cracow, Warsaw, and Radom) and each of them sells books. We would like to create one virtual bookstore that offers books from all locations. Additionally, the local server in Cracow is a replica of the server in Warsaw.

In the integration schema we have the following information:

1. ISBN is a unique identifier of a book.
2. There are no duplicates in databases in Warsaw and Radom.
3. The local servers in Cracow and in Warsaw contain the same information.
4. Data should be retrieved from the server that has shorter access time.
5. Data in Cracow cannot be modified as they are replica of data in Warsaw.
6. After updating data in Warsaw data in Cracow are automatically updated.

The example global view is below:

```
create view myBookDef {
  virtual objects myBook {
      int timeTo Warsaw := 1000000; int timeToCracow := 1000000;
      if alive(Warsaw) then timeToWarsaw := checkAccessTime(Warsaw);
      if alive(Cracow) then timeToCracow := checkAccessTime(Cracow);
      if min(bag(timeToWarsaw, timeToCracow)) > 100 then {
            exception(AccessTimeToHigh);
            return ∅;
      }
      return  ((Radom . Book as b) ∪
            if timeToWarsaw < timeToCracow  then (Warsaw . Book as b)
                                            else  (Cracow . Book as b));
  }
  on_retrieve do {
      return b.(deref(ISBN) as ISBN, deref(title) as title,
             deref(author) as author, deref(price) as price)
  }
  on_delete do {
      if server(b) = Cracow then
         delete (Warsaw . Book where ISBN  = b.ISBN);
      else delete b;
  }
  on_insert newObj do {
      if server(b) = Cracow then
         insert newObj into ( Warsaw . Book where ISBN = b.ISBN);
      else insert newObj into b;
  }
  on_update newTitle do
      if server(b) = Cracow then
         (Warsaw . Book where ISBN = b.ISBN) . title := newTitle;
      else b . title := newTitle;
}}
```

Note, that in this example there are implicitly used some routines of the communication protocol i.e., navigation (“**.**”), deletion (keyword **delete**), and update (“:=”). All these operations are called as if they are performed locally, in fact they are performed on data from remote servers. In means that inside global virtual store these operations have to be implemented as elements of the communication protocol.

Apart from the core protocol routines (like navigation, insert, delete, and update), we used additional routines of communication protocol. Their names are underlined. Functions <u>*alive*</u> and <u>*checkAccessTime*</u> are used to determine which local servers (Cracow or Warsaw) to use when constructing the seeds. The routine <u>*server*</u> acts on a pair ($i_{LS}$, $i_O$) returned by query *b* and then delivers the name *server’s name* from the triple ($i_{LS}$, *server’s name*, *server’s address*) representing the corresponding local server.

Clients can call the view in the following query:

(*myBook* **where** *title* =”Solaris” ) . ( *author, price* )

that returns an author and a price of the book with the title “Solaris”. This example shows that our mechanism supports service providers transparency – the user does not need to know the origin of the data.

## 5 Conclusion

We have presented a novel approach to implementation of federated databases based on updatable views. The approach fulfills some fundamental requirements for grid applications like transparency, interoperability and computational universality. The approach is based on a powerful query language SBQL which ensures high abstraction level. The advantage is decreasing development time of grid applications and simpler adaptation to changing requirements. The presented view mechanism is flexible and allows for describing any mapping between local databases and a federated database. Since our views can also map methods, we offer facilities as powerful as CORBA or Web Services. We have implemented SBQL and updatable views for XML repositories based on the DOM model. Our nearest plans assume implementation of our approach to the grid technology in the prototype ODRA, an object-oriented database platform build from scratch by our team (ca. 20 researchers). Currently in ODRA we have finished implementation of SBQL and start to implement other functionalities (imperative constructs, procedures, views, distribution protocols, etc.) that are necessary to make our idea sufficiently ready for testing and prototype applications.

## References


[Bell97] Z. Bellahsene: Extending a View Mechanism to Support Schema Evolution in Federated Database Systems. Proc. of DEXA 1997, 573-582

[Hal01] A. Y. Halevy: Answering queries using views: A survey. VLDB J. 10(4): 270-294 (2001)

[KLN+04] S. Kambhampati, E. Lambrecht, U. Nambiar, Z. Nie, G. Senthil: Optimizing Recursive Information Gathering Plans in EMERAC. J. Intell. Inf. Syst. 22(2): 119-153 (2004)

[KLS03] H.Kozankiewicz, J. Leszczyłowski, K. Subieta. *Updateable XML Views.* Proc. of ADBIS’03, Springer LNCS 2798, 2003, 385-399

[KW96] Chung T. Kwok, Daniel S. Weld: Planning to Gather Information. AAAI/IAAI, Vol. 1 1996: 32-39

[SKL95] K.Subieta, Y.Kambayashi, J.Leszczyłowski. Procedures in Object-Oriented Query Languages. Proc. of 21-st VLDB Conf., 1995, 182-193

[Subi00] K.Subieta. Mapping Heterogeneous Ontologies through Object Views. Proc. of EFIS, IOS Press, 1-10, 2000