

SBQL Views – Prototype of Updateable Views¹

Hanna Kozankiewicz[#], Kazimierz Subieta^{#, &}

[#]) Institute of Computer Sciences of the Polish Academy of Sciences, Warsaw, Poland

[&]) Polish-Japanese Institute of Information Technology, Warsaw, Poland
{hanka, subieta}@ipipan.waw.pl

Abstract. In this paper we describe a prototype implementation of updateable views called “SBQL Views”. The prototype follows a novel approach to view updates based on the Stack-Based Approach to object-oriented and XML-oriented query languages. The SBQL Views prototype is implemented on top of the SBQL query language for the XML DOM model. The novelty of the approach consists in augmenting a view definition by descriptions of update operations that can be performed on virtual objects. The paper describes the main ideas of SBQL Views and shortly compares the implemented approach to relational views based on *instead-of* triggers, a corresponding solution used in Oracle and MS SQL Server.

1. Introduction

A view is a mapping of stored objects into virtual ones that are adapted to particular application requirements. The motivations of introducing views into databases can be various – from security issues like information hiding, through supporting database schema evolution, to transparent integration of heterogeneous distributed resources. The qualities that the views can provide were already discussed in [KLS03].

For many years usage of views was limited because of problems with their updates. If a view is a mapping of stored objects into virtual ones, then updates of virtual objects must be mapped into updates of stored objects. The problem is that there is no generic and automatic way to find such a mapping which is at the same time always coherent with the user intents. For instant, if one wants to update some aggregated value (e.g. the average salary of employees) delivered by a view, then there are plenty of methods to perform such an operation on stored objects and probably only one of them can satisfy the user. Any predefined or automatic mapping may lead to warping of user update intents (the problem is discussed e.g. in [KLPS02]).

Although the view updating problem has been identified many years ago and there are many papers on the topic (see e.g. [Bert92, GPZ88, KiKe95, LaSc91, Rund96, SLT91]), the existing implementations support updates of views to a limited extent (e.g. [AAC+99, ABS96, KK95, Lac01, Rund92]). As far as we know, only one solution for relational views in Oracle and MS SQL Server (referred to as *instead-of* trig-

¹ Supported by the European Commission 5th Framework project ICONS (Intelligent Content Management System); no. IST-2001-32429.

gers views) handles the problem properly. No similar solution exists for object-oriented and XML-oriented databases. Typical solutions focus the attention on identifying cases when a view cannot be updated, instead of proposing methods to support such updates. In [KLS03] we presented an approach that is an alternative to *instead-of* triggers and addresses a general object/XML-oriented model. The approach assumes that generic updates of virtual objects are overwritten by calls of procedures that augment the view definition. The view definer can determine through the procedures any view updating intention.

In this paper we present an implementation of the idea. We also show advantages of our approach over *instead-of* trigger views. The prototype is called “SBQL Views”. It is implemented on top of the SBQL query language for the XML DOM model. Other repositories, in particular object-oriented databases, can be easily adopted through corresponding wrappers.

Both SBQL and SBQL Views are based on the Stack-Based Approach (SBA) to query languages. SBA is a novel approach to object-oriented query languages that is a come back to the classical concepts of programming languages, such as the environmental stack and the naming-scoping-binding paradigm. In contrast to other approaches to object-oriented query languages SBA addresses all the primitives that are necessary to define updateable views properly, in particular, generic imperative constructs (updating, inserting, deleting, etc.), functions and procedures, parameter passing, and others.

The structure of the paper is the following. In the next section we shortly describe SBA that is a basis of our approach to view updates. Next, we sketch ideas of the approach to updates of views. In Section 4 we present basic assumptions for the prototype and its architecture. Section 5 contains description of the prototype. In Section 6 we compare our approach with the *instead-of* trigger mechanism. Section 7 concludes.

2. Stack-Based Approach

The Stack-Based Approach to query languages has been proposed in [SKL95, SBMS95] and then developed in many next papers, reports and theses. It assumes that query languages are special programming languages. Therefore, evaluation of queries is based on similar mechanisms as expression evaluation and procedure calls in programming languages. The core of the approach is an ENVIRONMENT STACK (ENVS) that manages binding of names occurring in queries. ENVS consists of sections that determine “environments”. There are several kinds of environments, in particular, an environment of a database, of a user session, of a currently invoked procedure/method, of a currently processed object, and so on. Each environmental section contains entities called “binders”. A binder relates a name that may occur in a query/program with a run-time (or database) program entity (usually an object identifier). When a query interpreter binds a name in a query, it looks – from the top of ENVS through succeeding ENVS sections – for the closest binder with a proper name. This name binding obeys scoping rules similar to scoping rules in programming languages. SBA uses also an auxiliary query result stack (QRES) for storing temporary and final result of (sub)queries.

The Stack-Based Approach is explained through operational semantics (abstract implementation) of a query language, which is called “Stack-Based Query Language” (SBQL). SBQL is based on abstract syntax and orthogonal definition of operators thus it has the flavor of other theoretical approaches to query languages, such as relational/object algebras, calculi, etc. SBQL defines atomic queries that can be literals or names of variables e.g. 2, “Smith”, a , $radius$. More complex queries can be built by combining simpler queries using unary and binary operators; for example, *Employee where (Name = “Doe”), 2+2, exists Employee, Employee.(Name, Salary)*, etc.

SBQL distinguishes two kinds of operators: algebraic and non-algebraic. The main difference is that evaluation of algebraic operators does not involve ENVIS. In a query $q_1 \Delta q_2$, where Δ is an algebraic operator, queries q_1, q_2 are evaluated independently and their results are combined according to the semantics of operator Δ . Evaluation of a query $q_1 \theta q_2$, where θ is a non-algebraic operator, looks differently. In this case the query q_2 is evaluated in the context determined by q_1 . The evaluation looks as follows: first q_1 is evaluated; next, for each element e of the result of q_1 the query q_2 is evaluated. Before each such evaluation ENVIS is augmented with a section containing the local environment of e . Finally, all q_2 results are combined into the final result in the way depending on the operator θ . In the following we describe evaluation of algebraic and non-algebraic operators more precisely, because understanding of them is crucial to understanding our solutions in the SBQL Views prototype.

SBQL also supports procedures and functional procedures with no restrictions on their computational complexity. Procedures can be defined with or without parameters, can have local environment, and can have side-effects.

We note that in the traditional setting database views are essentially (usually limited) stateless functional procedures (e.g. SQL views). View updating is defined (as a rule) via side effects of procedures, e.g. via references to database objects returned by a procedure invocation. This must lead to a lot of limitations and anomalies, which are then fixed by severe restrictions on possible view updates. In our approach we have rejected such an idea. Our view definitions are more complex programming entities than a single functional procedure. The approach is described in the following section.

3. Approach to View Updates

The detailed description of the approach can be found in [KLPS02, KLS03]. We assume that a definition of view consists of a definition of virtual objects (as usual) and additional information describing intents of updates of virtual objects. The information has form of procedures that overwrite generic updating operations that can be performed on the view. We identify four generic operations that can be performed on virtual objects: update of a value of the given object, insertion of a new object into the given object, deletion of the given object, and dereference that returns a value of the given object. In case of a view update these operations are overridden by calls of procedures from the view definition which have fixed names, correspondingly, *on_update*, *on_insert*, *on_delete*, and *on_retrieve*.

A view definition consists of three main parts: (1) mapping between stored and virtual objects, (2) declarations of the above-mentioned overwriting procedures, (3) definitions of subviews (attributes of virtual objects). The first part is a functional procedure that returns entities called *seeds* that ambiguously identify virtual objects and are parameters for the second and third parts of the definition. The view definer decides which operations should be defined for a given view. If an operation is not defined it is forbidden. A view can also contain other elements like definition of procedures, objects (for stateful views), classes, etc. A simple view definition (for database containing data of computer components) might look as follows:

```

create view CheapComponentNameDef {
    virtual objects CheapComponentName {
        return (Component where price < 100) as p; }
    on_retrieve do { return capitalize( p . name ); }
    on_delete do { if UserHasDeletePermission() then delete p; }
    on_update new_name do {
        if UserHasUpdatePermission() then p . name := new_name; }
}

```

The view returns names of cheap computer components (the price is smaller than 100). The view defines: the dereference operation that returns the capitalized name of a given component, the protected operation of deletion, and the protected operation of update that changes name of the component to the new value. We can call this view e.g. in the following request:

(CheapComponentName **as** cn **where** cn = "FN5600") := "GeForce FX5600"

Note the implicit call of *on_retrieve* when the interpreter performs the dereference of *cn* before comparison '=', and *on_update* when it performs ':='. The example illustrates the most important features of our view mechanism i.e. transparency. The users operate on virtual objects in the same manner as they operate on stored objects and in fact, they do not have to be aware that a given object does not really exist and is only generated by the view.

View update process. A query interpreter must distinguish updates on virtual data and updates on stored data. Thus we have introduced the notion of a virtual identifier. It is a counterpart of an identifier of a stored object. A virtual identifier, besides information on the seed of a virtual object, contains information on the definition of view that has generated the given virtual object.

When the user performs an update operation on a virtual object, a query interpreter detects that it deals with a virtual object due to its virtual identifier. Thus instead of the generic update operation the interpreter calls the corresponding procedure defined within the view definition. The interpreter knows which operation is to be called due to view definition identifier included into the virtual identifier.

4. Basic Assumption and Architecture of the Prototype

The main goal of the implementation was to check whether the proposed approach to view updates is feasible, easy in implementation and optimizable. The implementa-

tion fully proved that the approach is consistent and efficient. We have made the following assumption for the prototype:

- The proposed approach to view updates has to be abstract and universal. It has to be applied to general object-oriented or XML-oriented models.
- The prototype should be able to operate on multiple (possibly heterogeneous) data resources at the same time.
- Besides developing implementation for a general object model we have to create a view mechanism for XML repositories. As far as we know, views for XQuery [XQuery03] are still an issue.

The SBQL Views prototype is currently implemented as standalone Java application, but it can also be used as an API to Java. Data from any database is seen by the prototype via wrapper that transforms the data to the form based on the SBA canonical data model M0. M0 supports nested object and relationships between them. In the model we have distinguished three types of objects: atomic objects, link objects (that models relationships between objects), and compound objects (that consist of other objects). Currently, there is only one working wrapper for data stored in the XML DOM format. Other wrappers are considered in the future.

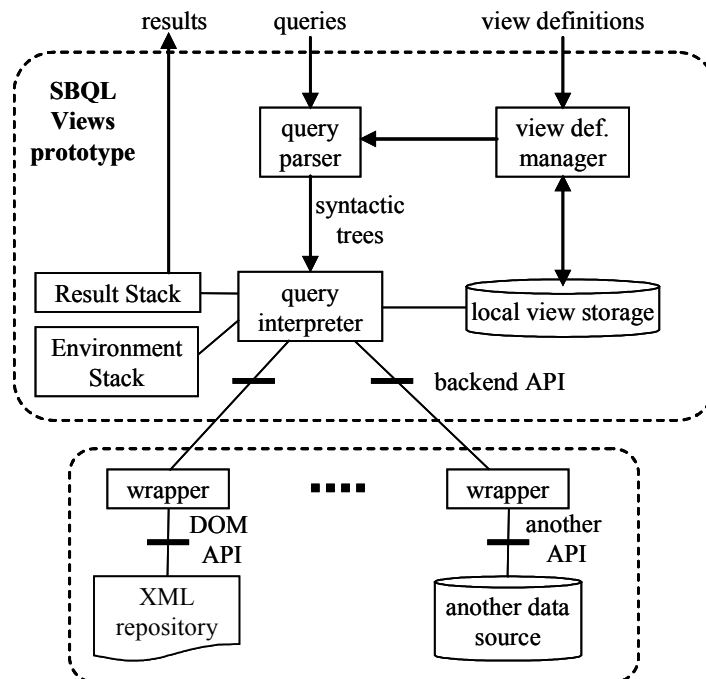


Fig. 1. Architecture of the SBQL Views prototype

The architecture of the SBQL Views prototype is depicted in Fig.1. The users ask queries via a graphical interface. A query is parsed by a parser of SBQL queries and then its syntactic tree is evaluated by the query interpreter. The interpreter uses the

environment stack and the result stack. Finally, its result is displayed to the user. Independently of queries, an administrator can introduce view definitions at any time. They are parsed and stored as complex objects within the local storage. The query interpreter makes use of them during processing of queries.

The SBQL Views prototype can at the same time work with multiple (possibly heterogeneous) data sources. Each of them is managed by a special connection object that delivers objects stored in the given location.

5. Description of the Prototype

The prototype was built on the prototype implementation of SBQL for the XML DOM model [PH02]. The language has been extended with imperative and procedural features. In the following we describe basic modules of our prototype.

5.1. Basic modules of SBQL View prototype

Parser of Queries. In the prototype the user asks queries using a graphical interface. Queries are translated into query syntax trees that are input for query interpreter. A parser of queries that is used in SBQL Views prototype was generated by CUP Parser Generator for Java [CUP] basing on the grammar of SBQL. The parser generated by CUP uses a scanner that was generated by JFlex (the Fast Scanner Generator for Java) [JFlex]. This scanner, basing on an SBQL lexical structure, delivers basic tokens of queries like operators or literals.

Query Interpreter. Query in the form of query syntax tree is delivered to query interpreter. The interpreter evaluates a query using the Environment Stack and the Result Stacks in accordance with SBA. Here, we shortly describe algorithms of query evaluation within the prototype.

1. If query q is an atomic query i.e. literal or name:

```

procedure eval (q ) { // q is an atomic query
    ...
    case q is a literal l:
        push( QRES, l );
    case q is a name:
        push( QRES, bind( n ) );
    ...

```

where operation *bind* binds given name n on ENVS.

2. If query q involves an algebraic operator Δ :

```

//q has the form  $\Delta(q1)$  or  $q1 \Delta q2$ 
case q is  $\Delta( q1 )$  : {
    var result_of_q1; // auxiliary variable

```

```

eval( q1 );
result_of_q1 := top( QRES ); pop( QRES );
push( QRES, Δ (result_of_q1) ); };
case q is q1 Δ q2: {
  var result_of_q1, result_of_q2; // auxiliary variables
  eval( q1 );
  eval( q2 );
  result_of_q2:= top( QRES ); pop( QRES );
  result_of_q1:= top( QRES ); pop( QRES );
  push( QRES, result_of_q1 Δ result_of_q2);};
...

```

3. If query q involves a non-algebraic operator θ (where, dot, join, quantifiers, etc.) the situation is a bit more complex. In this case we use an auxiliary function *nested* that for an argument r returns the following results:
- If r is a single identifier of a complex object then *nested* returns binders to subobjects of given object.
 - If r is an identifier of a link object then *nested* returns binders to the object the link points to.
 - If r is a binder, then *nested* returns $\{r\}$ (a set consisting of the binder).
 - If r is a structure **struct** $\{r_1, r_2, r_3, \dots\}$, then *nested* returns the sum of the results returned by *nested* for r_1, r_2, r_3, \dots
 - If r is a virtual identifier then *nested* returns binders to subviews of given view if there are any.
 - For other r *nested* returns an empty set.

Non-algebraic operators are handled in the following way:

```

case q is q1  $\theta$  q2 and  $\theta$  is a non-algebraic operator : {
  var intermediate_result, tmp_result, final_result; // aux. variables
  eval( q1 );
  for each e in top( QRES ) do {
    if e is virtual_identifier then
      push( ENVVS, binder to seed from virtual identifier);
    push( ENVVS, nested( e ) );
    eval( q2 );
    tmp_result:= merge $_{\theta}$  ( e, top( QRES ) );
    intermediate_result:= intermediate_result  $\cup$  { tmp_result };
    pop( QRES ); pop( ENVVS );
    if e is virtual_identifier then pop( ENVVS );
  };
  final_result := aggregate $_{\theta}$  ( intermediate_result );
  pop( QRES );
  push( QRES, final_result ); };
}

```

To support views we change a bit the function *nested*, which for a virtual identifier returns binders of subviews of the given view. We also modified the procedure *eval* for non-algebraic operators acting on a virtual identifier: we push a new section with the seed of the given virtual object on top of ENVs. In this way we send the seed parameter to procedures *on_update*, *on_delete*, *on_insert*, and to sub-views of the given view.

The syntax of the query language accepted by the SBQL Views query interpreter is presented in the following section.

Input data. SBQL Views prototype supports data from arbitrary sources if only there is implemented an appropriate wrapper. In the current version the prototype do not support classes, dynamic roles and inheritance, but it is clear how these features are to be implemented within SBA and therefore, it seems that they can be easily added to the prototype.

The backend of the SBQL Views prototype operates on data in the SBA M0 canonical model. M0 distinguished three main types of data: atomic objects, link objects, and compound objects. Views and procedures are treated as kinds of atomic or complex objects. All these kinds of objects are specified as interfaces. Hence, in order to use our prototype with data in the format different from XML, one has to develop a set of classes that implements these interfaces for the target storage (in particular, there must be implemented basic operations for each kind of object (*update*, *delete*, *insert*, *create*) and a set of classes that manages connection with the given storage.

5.2 Functionality of the SBQL Views prototype

In this section we describe functionality of SBQL Views prototype and the syntax of the query language that is accepted by the prototype.

Functionality related to query language. In our prototype user can ask queries and view their results. The graphical interface of the query language module is depicted in Fig. 2.

Here, we present a simplified syntax of queries accepted by the SBQL Views prototype. The syntax is the following:

```

query ::= literal | name
query ::= unary_operator query
query ::= query binary_operator query
query ::= query non-algebraic_operator query
non-algebraic_operator ::= where | . | depjoin
query ::= for any query holds query
query ::= for all query holds query
query ::= (query)
query ::= query as name
query ::= query group as name

```

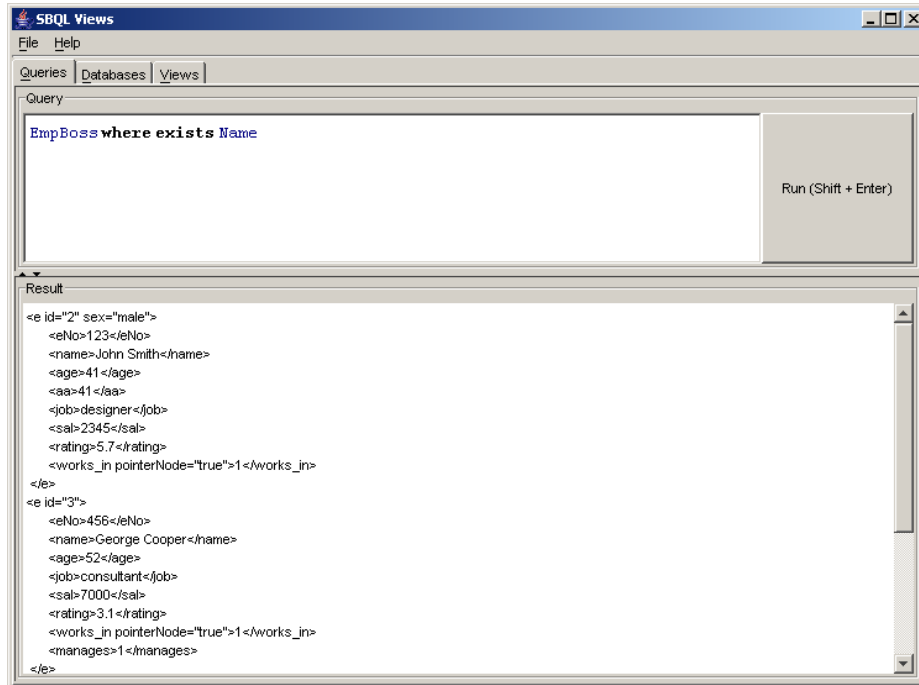


Fig. 2. Query evaluation window

We have also implemented the following imperative operations:

- assignment statement that allows to change the value of an object. The statement has form $l\text{-value} := r\text{-value}$. The statement assigns $r\text{-value}$ (possibly after dereferencing) to the object with object identifier returned as $l\text{-value}$. The syntax of the update statement in our prototype is the following:

imperative_statement ::= query := query

- delete statement that allows to delete arbitrary object(s). The statement deletes all objects with identifiers that are returned by $query$. The syntax of the delete statement is the following:

imperative_statement ::= **del** query

- create statement that allows to create new variable(s). In the statement, first $query$ is evaluated and next, there are created as many variables with name $name$ as there is a multiplicity of the result of $query$. Flag $where$ determines where newly created variable(s) should be placed – flag $permanent$ means that it will be created as root objects of our database whereas flag $local$ means that object should be created in local environment (of the procedure, function, etc.) The syntax of the create statement for our prototype is the following:

imperative_statement ::= **create** where name (query)
 where ::= permanent | local

- insert statement that inserts object(s) into other object. The operation has two parameters - the first one is a query that returns an identifier of destination object and the second one is a query that returns set of objects to be inserted into the destination object. The syntax of the insert statement is the following:

imperative_statement ::= **insert**(query, query)

All these imperative statements can be performed on stored objects as well as on virtual objects. Therefore, when the user calls one of these imperative statements, the query interpreter must be check whether the statement involves real or virtual object. If it deals with virtual objects the prototype instead of generic operation calls the corresponding procedure from body of view that has generated the given virtual object.

In the SBQL Views prototype we have also introduced conditional statement *if* and the loop *for each*:

conditional_statement ::= **if** query **then** query
conditional_statement ::= **if** query **then** imperative_statement
loop ::= **for each** query **do** statement
statement ::= conditional_statement | imperative_statement

In the SBQL Views prototype the user can define procedures and functional procedures. All procedures (thus also views) can be recursive. We have implemented two classical methods of parameters passing: call-by-value and call-by-reference, with queries as parameters. The syntax is the following:

```

procedure      ::= proc_header proc_body
proc_header    ::= proc name | proc name ( param_list )
param_list     ::= param | param , param_list
param          ::= query | ref query
proc_body      ::= { statement_list }
statement_list ::= imperative_statement | imperative_statement; statement_list
imperative_statement ::= return query

```

When a procedure is defined a corresponding binder is created and it is placed in base sections of the ENVS. Base sections contain, in particular, binders to database objects, binders to temporary objects of a user session, and binders to views

Finally, the user can define views using the following syntax:

```

view           ::= view_header view_body
view_header    ::= view name | view name ( param_list )
view_body      ::= { virtual_objects_def view_operation_list subviews_list }
virtual_objects_def ::= create virtual name { proc_body }
view_operation_list ::= view_operation | view_operation view_operation_list
subviews_list  ::= view | view subviews_list

```

The syntax of *view_operation* is the same as the syntax of procedures except fixed names (for procedures that describes operation on view). When a view is defined, two binders are created and placed in the base section of the ENVS: the first one is a binder to the view definition, and the second one is a binder to a procedure that defines virtual objects.

Functionality related to managing data sources. In the prototype we can:

- mount a new source of data – when a new source of data is mounted there are placed corresponding binders in the base section of ENV5 which allows to navigate over data stored in that source.
- unmount any source of data – when a source of data is unmounted the corresponding binders are deleted from the based section of ENV5 (therefore, data from given source cannot be returned by any query).
- re-read any source of data from given location.

All these features are available through SBQL Views graphical interface.

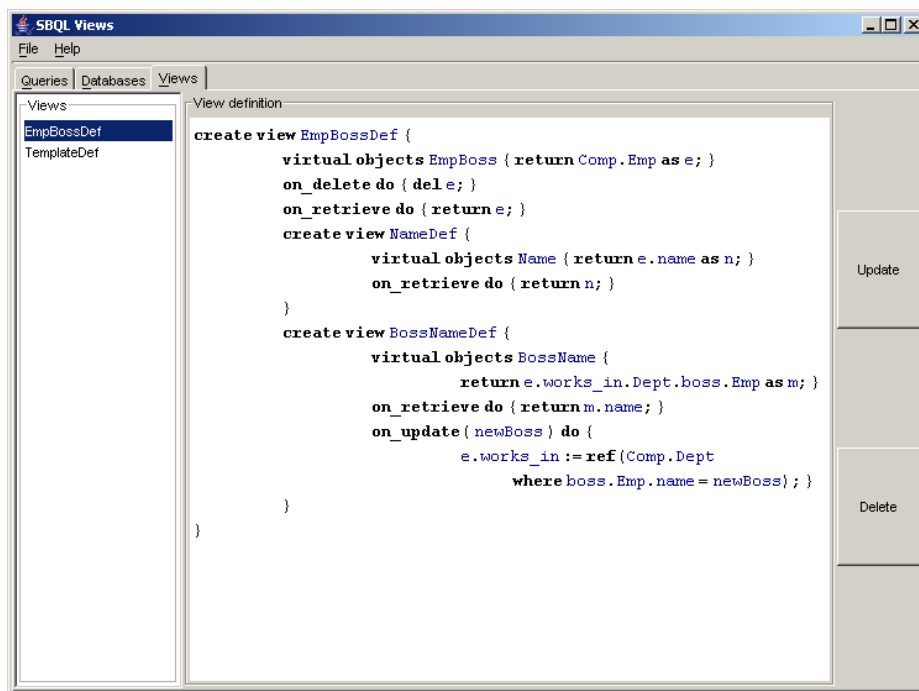


Fig. 3. View management window

Functionality related to managing views. In the prototype we can:

- browse views
- update views
- delete views

A screenshot illustrating these functionalities is shown in Fig.3.

6. Comparison to Instead-of Triggers

The approach to updates of views implemented in the SBQL Views prototype is similar to the *instead-of* triggers views implemented in Oracle and MS SQL Server. The idea of such triggers consists in associating with a view definition a special trigger that detects update operations on a virtual table and then calls specially designed procedures instead of generic update operations. Our approach to view updates has however advantages over the mechanism based on *instead-of* triggers. We enumerate several of them:

- Our views can address any kind of object-oriented, object-relational, or XML-oriented databases rather than relational databases only. We can easily introduce to our mechanism classes and static inheritance, object roles and dynamic inheritance, methods and message passing, polymorphism, encapsulation, etc.
- In our approach a mapping between stored and virtual data can be defined through arbitrary procedures having full algorithmic power. In SQL a view is defined by a single query, but because SQL is not algorithmically complete some more complex views are impossible to express.
- In our approach we have introduced the possibility of redefinition of dereference operation (*on_retrieve* procedure) what can be very useful for many applications. In this way the user can take control not only on updating operations but also on retrieval ones, which may be important e.g. for security. There is no such a possibility within the *instead-of* triggers views.
- We need not to introduce the *event* or *exception* notion, because the idea is based on overwriting of generic operations. Events or exceptions requires further notions such as event registers and event handling blocks, which complicate implementation, cause problems with optimizations and synchronization, and could be difficult for programmers.
- In SQL when a view updating operation is called the view must be fully materialized (because the *OLD* variable must be properly filled in). Full materialization may cause performance problems. In our approach, due to the seed notion, we do not need to perform such materialization. In particular, we do not have to materialize sub-views if it is not necessary in the given situation. We can also easily apply to our views the query modification technique [SuPI01].
- In SQL a definition of view and a definition of the corresponding trigger are separate entities. In our approach the definition of a view contains definition of operations; thus they constitute a conceptual whole. This supports encapsulation and is easier to manage.
- In SQL all views are stateless. Our approach has no such limitation: we can introduce to our view definition (to its compiled version) any objects and then manage them by procedures defined within the view. Stateful views are very important for many applications, e.g. for integration heterogeneous resources (storing the state of distributed transactions, storing the state of remote object activation, etc.).

Summing up, we think that these advantages prove that our approach is at least worth-attention alternative for the *instead-of* triggers.

7. Conclusions

In this paper we have presented the implementation results of the work that is continuation of the research presented last year at ADBIS [KLS03]. We have shown the main concepts of SBQL Views - a prototype of a novel approach to view updates. The implementation proved that proposed approach is feasible, consistent, easy in implementation and really operating. Our implementation currently addresses XML data, but as we have shown that it is possible to adopt it to any data source if there exists a wrapper supporting a corresponding data format. In this paper we have also shown advantages of our approach over *instead-of* triggers.

The proposed approach to updateable views will be a basis for further research conducted within our group. We are now start implementation of a data intensive grid with integration of distributed heterogeneous resources through updateable views. SBQL views are now ported to another platform ODRA in the .NET environment. Another interesting research line connected with our updateable views concerns the novel approach to aspect-oriented programming (AOP).

References

- [AAC+99] S.Abiteboul, B. Amman, S. Cluet, A. Eyal, L. Mignet, T. Milo. Active Views for Electronic Commerce. Proc. of VLDB Conf., 1999, 138-149.
- [ABS96] S. Amer-Yahia, P. Breche, C. Souza dos Santos. Object Views and Updates, BDA'96, France
- [Bert92] E.Bertino. A View Mechanism for Object-Oriented Databases. Advances in DB-Technology, Proc. of EDBT Conf., Springer LNCS 580, 1992, 136-151
- [CUP] <http://www.cs.princeton.edu/~appel/modern/java/CUP/>
- [DOM] <http://www.w3.org/DOM/>
- [GPZ88] G.Gottlob, P.Paolini, R.Zicari. Properties and Update Semantics of Consistent Views. ACM Transactions on Database Systems, 13(4), 1988
- [JFlex] <http://jflex.de/>
- [KiKe95] W.Kim, W.Kelley. On View Support in Object-Oriented Database Systems. In Modern Database Systems, Addison-Wesley, 108-129, 1995
- [KK95] W. Kim, W. Kelley: On View Support in Object-Oriented Database Systems. Modern Database Systems 1995: 108-129
- [KLPS02] H. Kozankiewicz, J. Leszczyłowski, J. Płodzień, and K. Subieta. Updateable Object Views. Institute of Computer Science of PAS, Report 950, 2002
- [KLS03] H. Kozankiewicz, J. Leszczyłowski, and K. Subieta. Updatable XML

- Views, ADBIS 2003, Dresden, Germany
- [Lac01] Z. Lacroix. "Retrieving and Extracting Web data with Search Views and an XML Engine". Workshop on Data Integration over the Web, in conjunction with the 13th CAiSE Conf., Switzerland, 2001.
- [LaSc91] C.Laasch, M.H.Scholl, M.Tresch. Updatable Views in Object-Oriented Databases. Proc. of 2nd DOOD Conf., Springer LNCS 566, 1991
- [PH02] R. Hryniów, T. Pieciukiewicz. A Stack-Based XML Query Language. Master's thesis, Polish-Japanese Institute of Information Technology, 2002 (in Polish).
- [Rund92] Elke A. Rundensteiner: Multiview: A Methodology for Supporting Multiple Views in Object-Oriented Databases. VLDB 1992: 187-198
- [Rund96] E.A.Rundensteiner. Object-Oriented View Technology: Challenges and Promises, Proc. of Intl. Symposium on Cooperative Database Systems for Advanced Applications, Kyoto, Japan, 1996.
- [SBMS95] K. Subieta, C. Beeri, F. Matthes, and J. W. Schmidt. A Stack Based Approach to Query Languages. Proc. of 2nd Intl. East-West Database Workshop, Klagenfurt, Austria, 1994, Springer Workshops in Computing, 1995.
- [SKL95] K. Subieta, Y. Kambayashi, and J. Leszczyłowski. Procedures in Object-Oriented Query Languages. VLDB Conf., 182-193, 1995
- [SuPI01] K.Subieta, J.Łodzień. Object Views and Query Modification, (in) "Databases and Information Systems" (eds. J. Barzdins, A. Caplinskas), Kluwer Academic Publishers, pp. 3-14, 2001
- [SLT91] M.H.Scholl, C.Laasch, M.Tresch. Updatable Views in Object-Oriented Databases. Proc. 2-nd DOOD Conf. Springer LNCS 566, 1991
- [Xerces] <http://xml.apache.org/xerces2-j/index.html>
- [XQuery03] XQuery 1.0: An XML Query Language, ed. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Siméon, W3C Working Draft 12 November 2003