

# New Approach to View Updates <sup>1</sup>

Hanna Kozankiewicz <sup>\*</sup>, Jacek Leszczyłowski <sup>\*</sup>, and Kazimierz Subieta <sup>\*,#</sup>

<sup>\*</sup> Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland

<sup>#</sup> Polish-Japanese Institute of Information Technology, Warsaw, Poland

**Abstract.** We present a new approach to virtual, object-oriented, updateable views in object databases. Within this approach it is possible to include into a view definition information on user intents of view updates. These information have a form of procedures that overload default update operations performed on virtual objects. Due to procedures we support full algorithmic power of view definitions, which is essential for many applications. The proposed mechanism is based on the Stack- Based Approach to query languages in which query language is treated as a special kind of programming language and is operated in the similar manner.

## 1 Introduction

A database view is a virtual image of data stored in a database. Views are important for many applications since they can bring many qualities like: security (one can restrict access to data), customization (one can adapt data to particular users requirements), data integration, and so on. The important requirements for view mechanism is transparency i.e., a user formulating a query need not distinguish between stored and virtual data: he/she should apply to both the same data model and query syntax. This requirement of view transparency requires also updating virtual data, which is indistinguishable from updating stored data.

The problem of view updates is well-known since it has been formulated over 30 years ago by Codd [Cod74]. It consists in resolving a translation of updates performed on virtual data into updates performed on stored data. However, a view is a many-to-one mapping between stored and virtual data, so finding this translation can be not trivial. Very often view update may be performed in multiple ways (e.g. when a view returns average salary in the department).

<sup>1</sup>This work is partially supported by the EU 5th Framework project ICONS (Intelligent Content Management System), IST-2001-32429

In classical approaches view updates are performed through some side-effects of a view definition, for instance references returned by view. However, such an approach may lead to warping of user update intents what induced introduction of updateability criteria (a view can be updated if all these criteria are fulfilled). The most advanced approach to view updates are *instead of triggers* implemented in Oracle and MS SQL Server. The idea is to introduce triggers and bind them to update operations performed on virtual objects. In the approach presented in this paper we follow the idea of redefining operations on views but we propose more general and universal mechanism than the mechanism of *instead of triggers*.

In this paper we present a new approach to updateable, virtual views. It is based on the Stack-Based Approach that aims at integrating the concepts of programming and query languages. The approach is abstract and universal and can be applied to any database model e.g. object, object-relational, relational, or XML data model. In the proposed approach a person that define s a view can include into a view definition information on intents of the given view updates. This information has a form of procedures that overload generic operations performed on virtual objects seen through the view.

The rest of the paper is structured as follows. The next section introduces the Stack-Based Approach to query languages and describes Stack-Based Query Language. Section 3 presents the proposed approach to updateable views and Section 4 concludes.

## 2 Stack-Based Approach (SBA)

The Stack-Based Approach [SKL95] is based on assumption that a query languages are similar to programming languages. The approach is based on naming-scoping-binding principle. It means that each name appearing in a query is bound to a proper run-time entity (object, procedure, etc.) according to the scope of its name. A control of scope is managed by means of a special stack, so called Environment Stack<sup>2</sup>. Environment Stack (ES) consists of sections that contain sets of binders – special constructs that relates names with a run-time entities. During query evaluation the sack grows and shrinks according to query nesting. If a query has no side-effects a state of ES after the query evaluation is the same as before the evaluation.

<sup>2</sup>The stack is a counterpart of a call stack known from programming languages and with slightly differences operations on Environment Stack are similar to operations on the call stack

Binding of names on ES follows search from the top rule. Example state of ES with indicated order of binding is depicted in Fig.1.

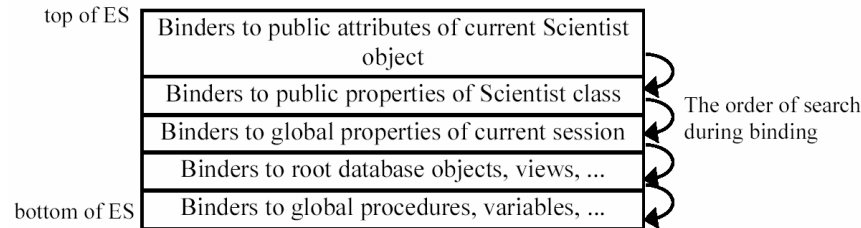


Fig. 1. Example state of Environment Stack.

## 2.1 Stack-Based Query Language

SBA has its own query language Stack-Based Query Language (SBQL). It is based on principles of compositionality and orthogonality of operators. In the language complex queries are built from simpler queries by combining them using unary operators (like **sum**, **count**) or binary operators (like **where**, **+**, **join**, **dot**). The simplest SBQL queries (atomic queries) are names and literals e.g. 5, "Smith", b, Scientist.

In SBQL binary operators are divided into two groups: algebraic operators and non-algebraic operators. In a query  $q_1 \Delta q_2$  that contains an algebraic operator  $\Delta$  the order of evaluation of  $q_1$  and  $q_2$  does not matter. Both queries  $q_1$  and  $q_2$  are evaluated independently and their results are combined into the final result according to the semantics of the operator  $\Delta$ . Algebraic operators include string comparisons, Boolean and numerical operators, aggregate functions, set, bag and sequence operators and comparisons, the Cartesian product, etc.

The concept of non-algebraic operators is a core of SBA. In opposite to algebraic operators, non-algebraic operators modify the state of ES. In a query  $q_1 \Theta q_2$  where  $\Theta$  is a non-algebraic operator, the query  $q_2$  is evaluated in the context determined by the query  $q_1$ . For each element  $r$  of the result returned by  $q_1$ , the subquery  $q_2$  is evaluated. Before each such evaluation ES is augmented with a new scope determined by  $r$  (the new scope can be formed of several sections). After the evaluation the stack is popped to the previous state. A partial result of the evaluation is a combination of  $r$  and the result returned by  $q_2$  for that row; the kind of combination depends on  $\Theta$ . Finally, these partial results are merged into the final result. Non-algebraic operators include

selection ( $q_1$  **where**  $q_2$ ), navigation ( $q_1.q_2$ ), dependent join ( $q_1$  **join**  $q_2$ ), quantifiers ( $\forall q_1(q_2)$  and  $\exists q_1(q_2)$ ), etc.

## Examples of queries

In this subsection we present example SBQL queries. Our example database schema is depicted in Fig.2. The database contains information on scientists and their publications.



Fig. 2. Example database.

SBQL queries can be as follows:

- Get information on all papers published in 2003:  
Paper **where** year = "2003"
- Get names of all Ph.D. students:  
(Scientist **where** position = "Ph.D. student") . name
- Get titles of all papers written by Smith:  
(Scientist **where** name = "Smith") . publication . Paper . title

## Procedures in SBQL

SBA supports procedures that can be defined with or without parameters, can have local environment, side-effects, can call other procedures within their bodies, and can be recursive. Here, we define an SBQL procedure that returns coauthors of a given author:

```

procedure getCoauthorsOf( sci ){
  return distinct (((Scientist where name = sci) . publication
    . Paper . author) where name  $\neq$  sci);
}

```

The procedure can be called in the following way:

1. Get all scientists that have ever written an article with Smith:  
getCoauthorsOf( "Smith" )
2. Check if Smith has written any paper with Black:  
"Black" ∈ getCoauthorsOf( "Smith" ). name

In classical approaches views are essentially functional procedures and view updates are performed using references returned by this functions. In this paper we reject this approach and present an alternative idea.

### 3 Approach to Updateable Views

In this section we describe our approach to updateable views. It is in detail described in [KPLS02]. In our approach view definition consists of the following parts:

- (a) part that preserves information on stored objects. This part has form of functional procedure that returns *seeds*. A seed is an entity that within the view context unambiguously determines virtual object.
- (b) definition of operations that can be performed on virtual objects. These information have a form of procedures that overload generic operations that are to be performed on virtual objects.
- (c) optionally, definitions of sub-views.
- (d) optionally, some supplementary elements like functions, associations, etc.

In the approach we assume that a view definer has full control over the operations that can be performed on virtual objects. We identified four such generic operations: update of a given virtual object value, insertion of a new objects into the object, deletion of the given object, and dereference. Each of these operations have assigned fixed names; correspondingly: `on_update`, `on_insert`, `on_delete`, and `on_retrieve`. (Additionally, operations `on_update` and `on_insert` have parameters and names of these parameters are freely chosen by a view definer.) If any of these operations is not defined it means it is forbidden. The view definition has the following form:

```

create view ViewNameDef {
  virtual objects ViewName { ... } // part that preserves information
                                     // on stored objects

  on_retrieve do { ... }
  on_update rvalue do { ... }
  on_delete do { ... } // part that contains information
  on_insert object_ref do { ... } // on operation on virtual objects
  /* the rest of the definition */
}

```

### Virtual Identifiers and View Update Processing

A system has to distinguish virtual objects from stored objects. Therefore, we introduced a notion of virtual identifiers. It is a triple:

*<flag "I am virtual", view definition id, seed>*

Now, the system knows that the updating concerns a view (due to the flag "I am virtual"), which view (due to the identifier of the view), and which virtual object (due to the seed). Thus, the mechanism has all the information necessary to substitute the view updating operation by the proper updating procedure written by a view definer.

The scenario of a view update looks in the following way. First, a user executes a query that invokes a view. The query returns a set of virtual identifiers. When a query interpreter tries to perform an update operation, it detects that it deals with virtual objects. Therefore, it takes the corresponding procedure from a view definition and executes it. After this execution the control is passed back to the user program.

### Further Features

In the presented approach view can have parameters which are passed to the view body in the similar manner as parameters of procedures (i.e., through ES). Currently, we provide two methods of parameters passing: *call-by-value* and *call-by-reference*; however, there can be easily implemented some other techniques of parameters passing.

Within the presented approach a view definer can nest views (with unlimited number of hierarchy levels). All sub-views are seen as sub-objects of a given virtual object. View nesting requires extension of virtual identifiers that has to contain

information on ancestors (in nesting) of a given view. Presented approach also supports recurrence.

### 3.1 Example

For the example database from Fig.2 we define a view that keeps information on Ph.D. students. For each Ph.D. student the view returns a virtual object which contains a pair: information on name of a given student and his/her salary. The example view definition may look like this:

```

create view PhDStudentDef {
  virtual objects PhDStudent {
    return (Scientist where position = "Ph.D. Student") as s }
  on_delete do { if hasDeletePermission() then delete s; }
  create view NameDef {
    virtual objects Name { return s . name as name; }
    on_retrieve do { return deref( name ); }
  }
  create view SalaryDef {
    virtual objects Salary { return s . salary as sal; }
    on_retrieve do { return sal; }
    on_update new_val do {
      if( sal > new_val ) print "Cannot decrease salary"
      else sal := new_val ; }
  }
}

```

Example call of the view can be:

1. Fire Ph.D. student Black:  
`delete PhDStudent where name = "Black"`
2. Change salary of Smith to 2000:  
`(PhDStudent where name = "Smith") . salary := 2000`
3. Change name of Ph.D. student from "Smith" to "Black":  
`(PhDStudent where name = "Smith") . name := "Black"`

The call is incorrect, because the view does not implement update operation for names of Ph.D. students.

## 4 Summary

In this paper we presented a novel approach to updateable views. It is based on the Stack-Based Approach to query languages. The approach is abstract and universal what makes it relevant to a general data model. Within this approach to view a user can define powerful views: with parameters, local environment, nested, and recursive. Currently, we are finishing implementation of the mechanism for XML native databases based on DOM model and later we are going to extend to cover more complex data models.

## References

- [Cod74] E. F. Codd. Recent investigations in relational data base systems. In *Proceedings of IFIP Congress 74, Stockholm, Sweden*, p. 1017–1021, 1974.
- [KPLS02] Hanna Kozankiewicz, Jacek Płodzień, Jacek Leszczyłowski, and Kazimierz Subieta. Updateable Object Views. Technical Report 950, ICS PAS, Warsaw, Poland, 2002.
- [SKL95] Kazimierz Subieta, Yahiko Kambayashi, and Jacek Leszczyłowski. Procedures in Object-Oriented Query Languages. In *Proceedings of 21<sup>st</sup> International Conference on Very Large Data Bases (VLDB), Zurich, Switzerland*, pages 182–193. Morgan Kaufmann, 1995.