M. Staudt, C. Quix, M.A. Jeusfeld

View Maintenance and Change Notification for Application Program Views

To appear in Proc. of the ACM Symposium on Applied Computing, Atlanta, 1998.

View Maintenance and Change Notification for Application Program Views

Abstract

Client applications usually hold (derived) subsets of the database contents under their control. The incremental maintenance of such *externally* materialized views is an important open problem. In addition to some necessary changes in the known view maintenance procedures the issue of translating updates through an API and a way for clients to accept such updates have to be defined. This paper presents the properties of an incremental view maintenance procedure which is able to handle externally materialized views and is implemented in the deductive and object-oriented database system ConceptBase. In particular we describe the concepts for supporting direct modifications of data structures representing the materialized views on the client side.

Keywords:
incremental view maintenance, view materialization, API generation, active notification, deductive and object oriented databases

1 Introduction

A view on a data or knowledge base is called *materialized* if its contents - derived by a set of view definition rules - is redundantly stored in addition to the base data. Views can be materialized for reasons of reliability, efficiency by view reuse etc. Materialized views are distinguished from *snapshots* in that they are expected to be maintained consistent with the base data over time. Given a transaction that updates a set of base data, the corresponding changes to all materialized views need to be computed. Ideally, this should be done incrementally, to limit the re-computation overhead and network load. However, present commercial databases allow this only for very simple view definitions corresponding to select-project operations of the relational algebra. Since the early papers by Stonebraker [21], Buneman and Clemons [1], Nicolas [13], and Roussopoulos [16], the incremental maintenance of views has received a lot of attention in database research. Much of the recent work is nicely summarized in [4] who organize their survey mostly according to the amount of information available to the maintenance tool.

Figure 1 shows a typical client-server architecture for database applications. The server database management system (DBMS) maintains the state of the base data, often also a change log for transaction management. Clients support application programs and interactive user interfaces. Clients can maintain view caches which contain materialized views either in the same (e.g. clients that themselves have a DBMS component as in generalized client-server architectures [15]) or in a different representation as the database in client-specific data structures based on some type language.

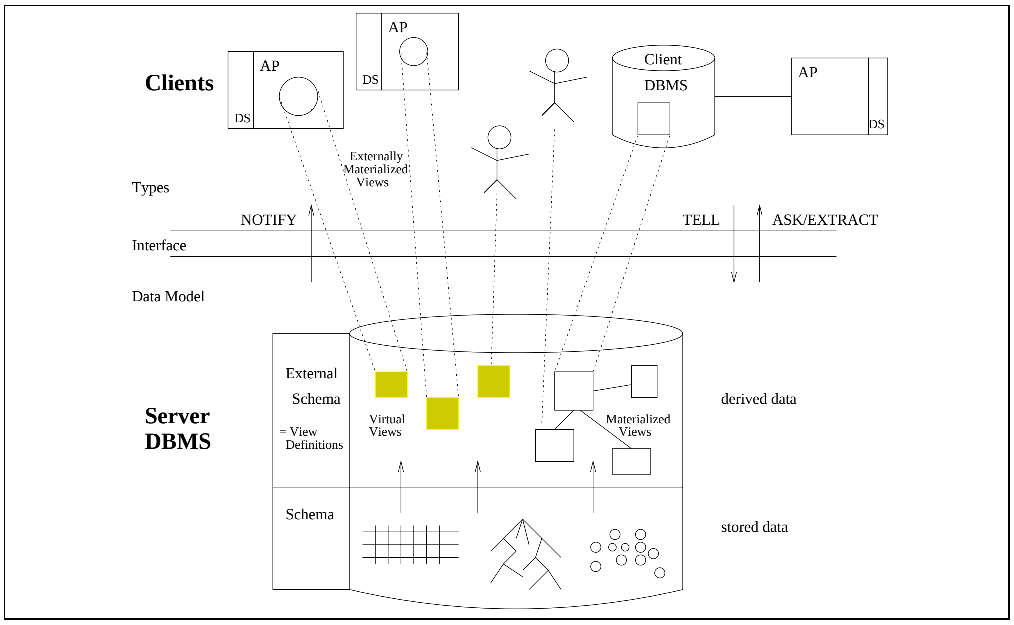


Figure 1: Client-server architecture for database applications

Depending on the software architecture, the server database, the client program, none of them, or both of them, may know the derivation rules by which the materialized view was computed – this is what [4] mean by 'amount of information available'. In the case of *full information*, the base data, the materialized view, and the derivation rule are all available to the view maintenance tool. Among the many conceivable cases of *partial information*, interest has focused on maximizing autonomy on the client side. In a *self-maintainable view*, the view definition (assumed to be available at the client site) is so simple that all the consequences of a base-data change can be locally computed by the client, without accessing the base data [3].

A neglected issue in the literature is the opposite case: *The maintenance tool has access to the base data and the view definition but not to the materialized view.* This may at first sound contradictory to the idea of materialization. However, it does occur quite frequently, namely when views are (*externally*) materialized outside control of the database, for instance, in application programs, graphical user interfaces, or remote autonomous databases.

In all of these cases, the clients could rightly expect to be notified of what incremental updates should be done on their views. Notification requires an active behavior of the database concerning a) detection of changes on the client views and b) providing suited messages enabling the client to perform the necessary updates on its data structures. Although research on active databases also deals with simple actions that produce external effects outside the database system, e.g. system operations or mailing mechanisms, and there are even limited approaches to handle the maintenance problem for internally materialized views through E(vent)C(ondition)A(ction) rule processing [2], we are not aware of combined solutions in this area applicable to externally materialized views.

Therefore, for our specific but nevertheless widespread maintenance problem we need both enhanced API support *and* changes in the basic view maintenance methods. Both issues are adressed in the following. Section 2 summarizes the basic view maintenance procedure presented in [18] which is based on Datalog$^\neg$, i.e. allows deductive rules with stratified negation, and has been implemented in the deductive and object-oriented database system ConceptBase [6]. This algorithm combines the commonly agreed way of maintaining views in case of full information with magic-set rewriting and only requires availability of view definition, base relations and base data updates. In Section 3 we present the view language for the O-Telos data model and describe how modifications of data structures representing the externally materialized views on the client side can be supported by the system. Section 4 covers implementation aspects and Section 5 shortly reports application experiences in maintaining so called integrity views.

2 The Basic View Maintenance Procedure

In a deductive database context view definitions are considered as sets of Datalog rules of the form $p(\vec{X}) :- l_1(\vec{X_1}), \ldots, l_n(\vec{X_n})$ defining a dedicated intensional relation (resp. view predicate) p whose contents can be computed from (*extensional*) base relations and from other *intensional* relations l_i that themselves are derived by rules. Datalog$^\neg$ extends Datalog with stratified negation in the body of rules, i.e. no cycles involving negation are allowed in the predicate dependency graph. The maintenance problem to be solved here implies that the rules defining the view predicate and the base data (including deleted or inserted tuples of a given transaction) are available but not the old state (extension) of p. However, at least parts of it are definitely required for determining the net effects an update has on the view.

After many approaches have been experimented with, a consensus seems to emerge that a fairly general class of view definitions with recursion, negation, and aggregate functions for the case of full information can be incrementally maintained by the following rough algorithm [9, 5]:

1. estimate the consequences of deletions by forward reasoning
2. prune those derived deletions for which alternative derivations exist
3. add the consequences of insertions

Some special cases allow for faster special-case counting algorithms, some complex cases (e.g. duplicates) require additional treatment. While much of the formal results have been developed for Datalog, results have also been transfered to active relational databases [2], multi-databases and data warehouses [23, 11].

We remain in the Datalog$^\neg$ framework and solve the problem of maintaining views that are only externally materialized by a two phase rewriting of the view definition rules:

- We first change the procedure for the case of full information slightly so that it yields a purely declarative version of the view maintenance rules.
- In a second step the view maintenance rules are rewritten in a magic-set like way [22] in order to reach interaction with the set of query rules that serves for initially computing the view extension.

Evaluation of both the final set of maintenance rules and the set of query rules (also assumed to be rewritten by a magic-set transformation and used for rederiving necessary parts of the view) can then be done using a standard bottom-up fixpoint procedure which typically constitutes a deductive/relational query processor.

2.1 Rule Rewriting

In [18] both phases of rewriting the view definition into maintenance rules are described in detail. Below we only summarize the main ideas for transforming the view definition rules.

Rewriting Phase 1 The transformation follows the same idea as [9, 5]: for each relation r we derive all possible deletions (overestimate) in a relation r^{del} caused by deletions (insertions) in other relations occuring (negatively) in the body of a rule defining r. Each tuple t in r^{del} corresponds to one derivation path for t that is cut off. The provisional new state r^{new} of r consists of those tuples of r not occuring in r^{del}. Since always one remaining derivation path for t is sufficient to still belong to r in the new state the overestimate has to be reduced to those tuples that definitely have no justification. The rederived tuples for r are contained in a relation r^{red} and are put back into r^{new}. Finally, insertions (r^{ins}) for r have to be processed and the net effects r^{plus} and r^{minus} can be computed by eliminating idle insertions and phantom deletions. In contrast to [9, 5] we encode the complete procedure by a pure set of declarative rules instead of mixing rules with procedural steps or employing meta predicates.

Rewriting Phase 2 Since the rewritten rules produced in phase 1 manage the task of computing differentials of views only if the state of *all* intensional relations is known as it was before a considered set of base data updates took place, the second phase links the maintenance rules with the query rules for computing the view contents. We assume that the latter are transformed with the supplementary Magic-Set method. This method introduces so called magic predicates into the rule body that trigger firing of the rule as well as propagate constants to other subgoals. Whenever a query q has to be answered a magic predicate m_q (specifying the bound arguments of q) initiates the evaluation process. For view maintenance rules this role can be played by the predicates r^{del} and r^{ins} describing the base data updates and successively allow other rules to fire. The transformation in addition ensures that whenever tuples of r with a certain partial binding of variables are needed for evaluating the body of a maintenance rule the corresponding query rules for r are triggered and derive these tuples. This is done by producing magic predicates m_r with these bindings gained from provisional results of evaluating preceding subgoals in the body of the maintenance rules.

2.2 View Maintenance by Rule Evaluation

Algorithm 1 shows the view maintenance procedure that takes the transformed rule sets and computes the changes for views of interest by selectively rederiving only the relevant parts of the old (intensional) database state.

Algorithm 1 (View maintenance procedure)

Input: 1. A set of query rules Q
2. A set of maintenance rules M
3. A set of (view) relation names $V = \{v_1, \ldots, v_n\}$
4. A set of base data changes

Output: changes to the views in V

```
begin
for $r \in IDB \cup SUP$ do
  initialize $ENV[r] := \emptyset$ od;
for $r \in EDB$ do
  initialize $ENV[r^{del}]$ and $ENV[r^{ins}]$ with the base data changes;
  initialize $ENV[r]$ with the old extension of $r$ od;
for i := 1 to m do
  repeat
    $OLD$ := $ENV$;
    $ENV$ := $EVAL(ENV, M^{(i)})$;
    $ENV$ := $EVAL(ENV, Q)$
  until $ENV = OLD$;
  for $r \in IDB$ with $S(r) = i$ do
    $ENV[r^{ins}]$ := $ENV[r^{plus}]$;
    $ENV[r^{del}]$ := $ENV[r^{minus}]$ od od;
MOD := $[(ENV[v_1^{plus}], ENV[v_1^{minus}]), \ldots, (ENV[v_n^{plus}], ENV[v_n^{minus}])]$;
return(MOD)
end
```

The algorithm employs the notion of an *environment* ENV as a mapping from predicate symbols to sets of tuples that represent a partial database state. This state is the subset of the overall extensional and intensional database contents which is unavoidable needed for maintaining the views of interest. While EDB and IDB denote the extensional and intensional predicates - corresponding to stored and derived relations respectively - SUP contains all delta, supplementary and magic predicates introduced for M and Q during the rewriting phases. Each $p \in EDB \cup IDB$ has a unique stratum number $S(p)$ ranging between 0 and some constant m. In addition we define $M^{(n)}$ as the set of rules $r \in M$ such that r is based on an original rule defining a predicate $p \in EDB \cup IDB$ with $S(p) = n$.

EVAL is assumed to be a fixpoint evaluator that works on a stratified set of rules and an environment with initializations for the relations involved. It respects the changes of the environment produced during the preceding evaluation in a semi-naive manner and returns its with additional tuples inserted for certain relations. When called for evaluating the maintenance rules *EVAL* only has to process the subset $M^{(i)}$ for a given stratum i of the original rule set. The second call with Q, however, is exactly as for evaluating an arbitrary query but now on a partially materialized intensional database state. The new tuples for the magic predicates generated in the first call of *EVAL* denote queries that have to be answered in order to continue the maintenance process. These queries may have to rederive tuples for predicates from lower strata that were not rederived before since they were not needed. Therefore, *EVAL* is called with the complete set of query rules Q.

3 From View Specifications to Client Data Structures

Detecting changes for view (relations) inside the database is only the first step for providing clients with respective change notifications. The second step consists of API support that enables the client to receive *and* interpret notification methods from the database server in an appropriate way, namely by updating the view contents directly on its own data structures. The context of our solution is built by the deductive and object oriented database system ConceptBase. We first present some basic features of its O-Telos data model and based on that show how to bridge the gap between the relational world and client data structures.

3.1 The DOOD system ConceptBase and its O-Telos Data Model

Leaving the pure deductive database context we now describe how the presented maintenance procedure serves for maintaining complex views of application programs working on top of the deductive and object-oriented data base (= DOOD) server ConceptBase [6]. ConceptBase is mainly intended for meta data management and supports the O-Telos object model, a derivative of the knowledge representation language Telos [12]. The advantages of O-Telos are its very simple basic concepts, its straightforward notion of objects and their properties, its flexibility and genericity. The third aspect in particular allows to easily customize O-Telos to particular application needs which often take the form of multiple overlapping views related to each other by other views or integrity constraints [14] and forms the basis for enabling arbitrary (meta) layers of models.

An O-Telos object base is semantically equivalent to a deductive database (Datalog with negation) which includes a predefined set of rules and integrity constraints coding the object structure. The surface language syntax is frame based. Rules and integrity constraints are included as first-order formulas defined over a basic set of predicates describing the abstraction principles of instantiation ($In(x, y)$ = "x is instance of y"), specialization ($Isa(c, d)$ =

```
Class Module with
  attribute
    import:Procedure;
    .....
    comment:String
end

Class OperatingSystem with
  attribute
    version: String
    .....
end

Class Agent with
  .....
end

Class Procedure with
   attribute
     defined_in:Module;
     depend_on: OperatingSystem;
     args:ArgSpec;
     .....
     comment: String
end

Class Program with
  attribute
    contains: Module;
    owner: Agent
    .....
end
```

Figure 2: Example schema: Programs, modules and procedures

"d is superclass of c") and attribution ($A(x\ l\ y)$ = "y occurs as l-attribute value for x").

A deductive object base is a triple $DOB = (OB, R, IC)$ where OB is the extensional object base, R and IC contain deduction rules and integrity constraints. The axioms are represented as predefined formulas in R and IC. Let ID and LAB be sets of identifiers, and labels resp. An **extensional O-Telos object base** is a finite subset

$$OB \subseteq \{P(o, x, l, y) \mid o, x, y \in ID, l \in LAB\}.$$

The elements of OB are called **objects** with identifier o, source and destination components x and y and name l.

The literals mentioned above can be derived from the general relation P as follows:

$$\forall o, x, y\ P(o, x, in, y) \Rightarrow In(x, y)$$

$$\forall o, c, d\ P(o, c, isa, d) \Rightarrow Isa(c, d)$$

$$\forall o, x, l, y, p, c, m, d\ P(o, x, l, y) \wedge P(p, c, m, d) \wedge In(o, p) \Rightarrow A(x, m, y)$$

In [7, 6] it is shown how the still very general relations can be refined by partial evaluation resulting in the definition of a so called internal object base. Hence, each literal $In(x, C)$ and $A(x, l, y)$ for some constants C and l can consistently be rewritten as $In.C(x)$ and $A.l(x, y)$, respectively.

The P-tuple representation of objects introduced above directly induces a semantic net like view on an object base by representing individuals as nodes and all other objects as edges of a graph. Due to the object identity of instantiations, specializations and attributions there may also occur edges between other edges in the graph.

In the following we will use an example schema describing the contents of a software database. Figure 2 shows an excerpt of this schema:

Programs consist of modules. Procedures are defined in modules, in part they are operating system specific. Modules also import procedures that are defined in other modules.

3.2 Views in O-Telos

The view concept for O-Telos is based on the idea of so called *query classes* [19] which serve for representing queries as special classes with necessary *and* sufficient membership conditions. A query has always one or more root classes, i.e. generalizations, which restrict the possible answer objects to their common instances. We distinguish between two different types of attributes, namely attributes from the schema - they serve for projecting and restricting attributes of interest - and dedicated new (computed) attributes of the query that specify additional derived properties of the answer objects due to their membership in the query class. The following sample query derives simple dependencies (computed attribute `needs`) between modules and other modules from which they are importing procedures.

```
QueryClass ImportModule isA Module with
  attribute
     needs: Module
  constraint
    c: $ exists p/Procedure ('this imports p) and
                             (p defined_in 'needs) $
end
```

This query class also gives an idea how the first-order formulas which are used to specify O-Telos integrity constraints and deduction rules look like. The atomic predicates like `('this imports p)` are infix variants of the predicates above, e.g. $A.imports(this, p)$. Arguments with a preceding prime are implicit variables either refering to the answer objects themselves (`'this`) - then the variables range is given by the root classes - or to their attributes (`'needs`) - then the variable takes over the attribute's class as range[1].

As in most other systems the O-Telos view language follows the basic constructs of the query language syntax[2]. For the example scenario above a simple (recursive) view specifies general module dependencies due to transitive import relationships:

```
View ModuleDependency isA Module with
  attribute
    based_on:Module
  constraint
    :$ exists proc/Procedure
         ('this import proc) and
         (proc defined_in 'based_on)
        or exists m/ModuleDependency
         ('this based_on m) and
         (m based_on 'based_on) $
end
```

[1]In both cases here the range is `Module`.

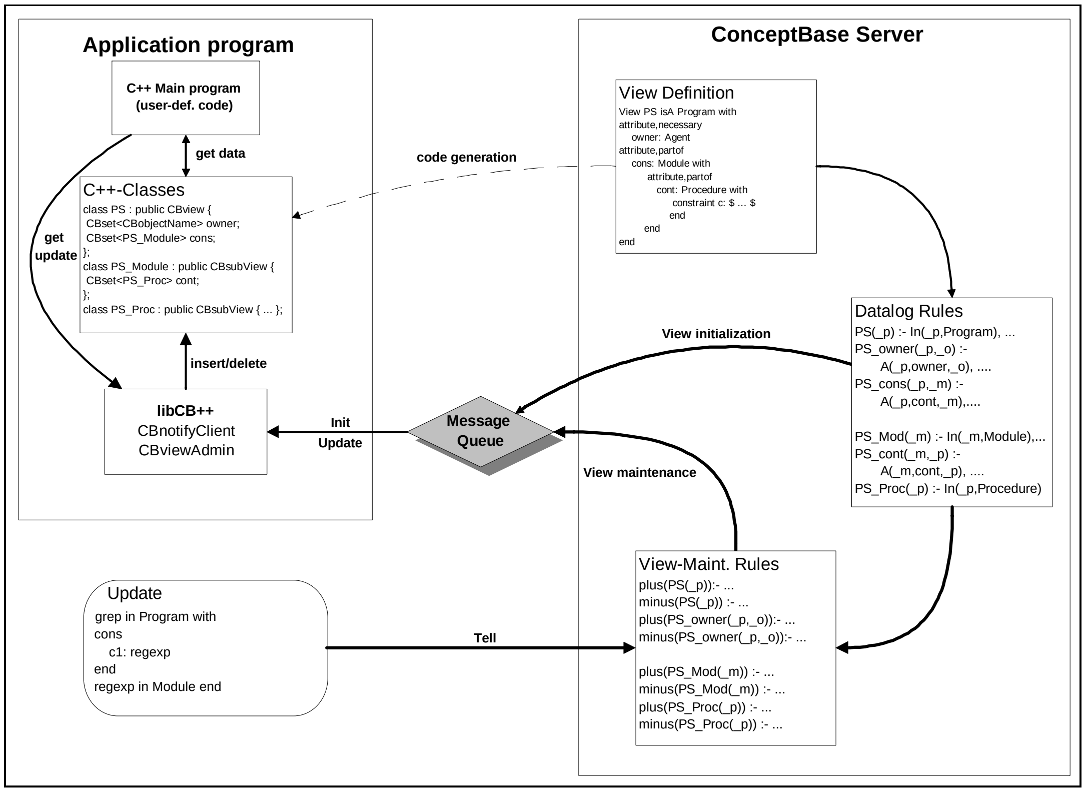
[2]For the dual alternative approach that generates queries from type definitions see [8].

As usual we require all answer attributes to be filled for an answer object of a query, i.e. the attributes and their restrictions are in particular *necessary* conditions for the class membership. For the view language which is intended to allow the specification of arbitrary subnets of an O-Telos object base this restriction is given up. Therefore, even modules that are not based on other modules due to missing `import` relationships belong to the view above. Attributes in O-Telos schema classes are not mandatory in general and may be set-valued. In so far views are closer oriented at the schema definition part of O-Telos. As in schema definitions the necessity of attributes can, however, still be enforced by explicitly assigning them to a predefined category `necessary`.

Another restriction of the query language lies in the implict binary relationships expressible using the frame syntax: we can only extract objects in a direct relationship to other objects. If such a direct relationship does not exist we can construct it artificially through new computed attributes. In our example taking the simple view `ModuleDependency` all modules are mapped directly to the other modules whether they really import directly or not. What also becomes obvious is that we loose the concrete relationships between the modules but only keep the first and the last of an "import path". Therefore, whenever we want to represent relationships between objects of arbitrary arity we have to use the nested frame syntax offered for O-Telos views. Still staying with the example we could be interested in extracting a complete program architecture, i.e. starting from programs, their modules and the procedure defined in the modules. Note, that the relationship between modules and procedures is directed towards modules in the schema, however should be inverted within the view. This can be reached defining the following *complex* view `PS`:

```
View PS isA Program with
  attribute
    contains: Module with
                attribute
                  consists: Procedure
                              constraint
                               :$ ('consists defined_in 'contains)
                                                ...
                               $
                            end
                constraint
                  :$ ... $
               end
    owner: Agent
  constraint
    :$ ... $
end
```

Even programs not containing modules and modules contained in a program but without defined procedures will belong to this view. In addition the owner of each program is included in the view. The `constraint` part within the deepest subframe (with root `Procedure`) states the inverse relationship between the schema attribute `defined_in` of `Procedure` and the computed view attribute `consists`. Further restrictions at this level could be specified to relate the occuring procedures to the program at the most outer level: e.g. we might select only those procedures whose creation date is the same than the last



```
class PS : public CBview {
 CBset<CBobjectName> owner;
 CBset<PS_Module> cons;
};
class PS_Module : public CBsubView {
 CBset<PS_Proc> cont;
};
class PS_Proc : public CBsubView { ... };
```

```
View PS isA Program with
attribute,necessary
    owner: Agent
attribute,partof
    cons: Module with
        attribute,partof
           cont: Procedure with
                constraint c: $ ... $
               end
        end
end
```

```
PS(_p) :- In(_p,Program), ...
PS_owner(_p,_o) :-
     A(_p,owner,_o), ....
PS_cons(_p,_m) :-
     A(_p,cont,_m),....

PS_Mod(_m) :- In(_m,Module),...
PS_cont(_m,_p) :-
     A(_m,cont,_p), ....
PS_Proc(_p) :- In(_p,Procedure)
```

```
plus(PS(_p)):- ...
minus(PS(_p)) :- ...
plus(PS_owner(_p,_o)):- ...
minus(PS_owner(_p,_o)):- ...

plus(PS_Mod(_m)) :- ...
minus(PS_Mod(_m)) :- ...
plus(PS_Proc(_p)) :- ...
minus(PS_Proc(_p)) :- ...
```

```
 grep in Program with
cons
    c1: regexp
end
regexp in Module end
```

Figure 3: View processing in ConceptBase

modification of the program in order to identify which parts of it are new. Similarly, the constraint parts of the other levels could specify relationships to the levels above.

The example view is still very simple and we will not discuss details of the view language that allows to annotate arbitrary constraints, path expressions, parametrization etc. within view definitions [17].

Figure 3 now shows the transformations that are necessary to ensure that the data in views like `PS` can be extracted and maintained on the client side.

3.3 View processing in ConceptBase

In order to process the view definition inside the database it is mapped to a set of Datalog$^\neg$ rules: The view is decomposed into a main view (derived relation PS) specifying the membership of all complex view objects in class `Program` and possible additional restrictions given in the `constraint` clause. A set of relational subviews results both from the attribute clauses and from inner subframes. For the example they are given by derived relations PS_cont (specifying programs together with their modules), PS_owner (collecting owners of a program), PS_Mod (containing all relevant modules satisfying the constraint of the inner subframe) and PS_Mod_cons (linking modules with their procedures). Note, that subviews for attribute clauses are introduced due to the non-mandatory property of attributes in order to avoid any form of null values.

After this view decomposition we can compute the extents of a view as follows: The resulting Datalog$^\neg$ program basically contains rules whose dependency graph is not connected somehow to the main predicate. As mentioned above the program is transformed using the Magic-Set method with respect to a certain view predicate. This transformation includes additional steps which ensure that during evaluation of the program all relevant tupels (and only them) are derived even for the independent relations [17]. After computing the extension of all involved view relation the final step of view extraction is an NF^2-like[3] processing of the relations separately specifying the main view and the subviews. Starting with the subviews for the innermost subframes of the view definition their extensions are joined by a (left) *outerjoin* operation (as e.g. discussed in [10]) which collects all objects and their properties together. The step from subframe to the enclosing frame of the next outer level walks along with a *nest* operation performed on the objects properties such that the contents of the subview has the form of a binary NF^2 relation consisting of pairs linking each object with tuples for their property sets.

The maintenance of a view like `PS` also starts from its Datalog$^\neg$ representation and applies the rewriting procedure sketched above to produce maintenance rules that (evaluated by the algorithm in Section 2) for a given base data update compute the differentials for all derived relations gained for `PS`. As during the initial view computation the independence of certain relations of the view might lead to the detection of potential updates that are irrelevant for the whole view. Consider e.g. a new module inserted in the database. In principle it could become member of the PS_Mod relation introduced for the subframe at level 2. However, if no existing program package is using this module the update does not cause a change to the complex view. Therefore, as above the rewriting is complemented by adding declarative rules which filter all such irrelevant differentials [17]. The relationship between the differentials and the data structure modifications is explained below. We do not extend the view differentiation to NF^2-like expressions.

3.4 API Support for Change Notification

Concerning the view representation on the client side a corresponding C++ class is generated for `PS` that manages for each of its attributes sets of pointers to other C++ classes representing subviews obtained from subframes in the view definition. The client program can be built around these classes.

The C++ classes are instantiated when the view extension is extracted from the object base for the first time. The computed NF^2 relation for the main view corresponds nicely to the object-oriented class structure on the client side such that a mapping becomes straightforward.

[3]NF^2 (= Non-First-Normal-Form) data models can be understood as predecessors of object oriented models with a clean semantics and sound algebraic evaluation mechanisms

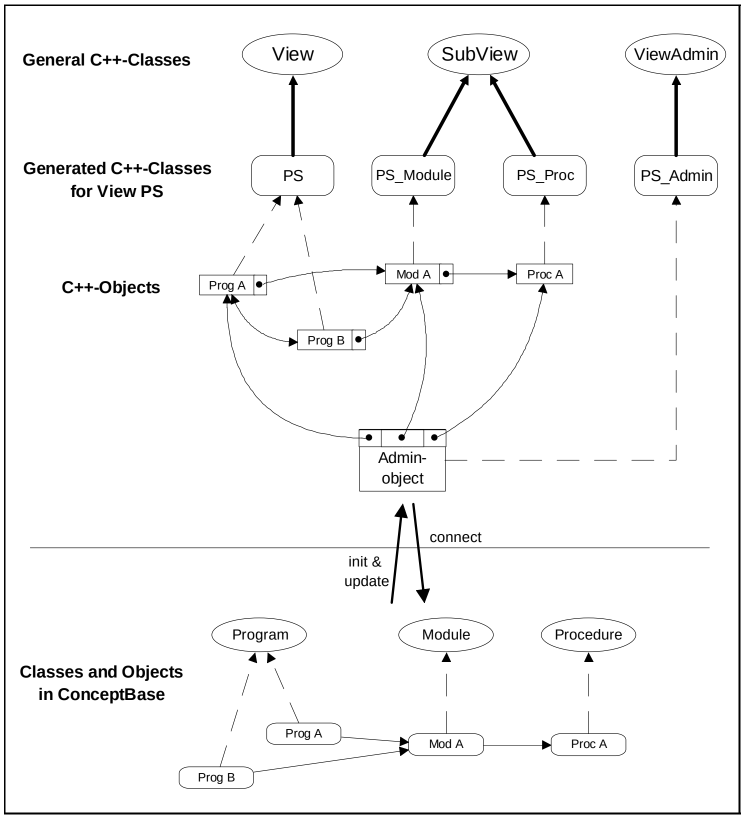


Figure 4: C++ materialization and management of O-Telos objects

For the change notification and update step we decided to directly start with the relational differentials as output of the maintenance process. All differentials to relational subviews introduced for attributes simply denote attribute links that have to be inserted or deleted. Due to the transformation into deductive rules it is guaranteed that whenever an object itself has to be deleted from or inserted into the view it results directly into a relational update for all affected views, namely the main view or one of those subviews stemming from subframes within the view definition. Hence, the differentials of the relational views can easily be mapped to the necessary elementary operations on the data structure (create/delete objects or pointer links between them).

Both the initialization and the incremental update of C++ objects have to be supported by appropriate methods included in the definition of the generated classes. Figure 4 illustrates the relationship between O-Telos objects in the database and their occurence in the materialized view on the client side. For the view `PS` an equally named C++ class represents the main part of the view definition as subclass of a general class `CBview` while all occuring subframes are mapped to subclasses of `CBsubView` (`PS_Module` and `PS_Proc` in our example). Another general class `CBobjectName` collects simple references in the view definition to O-Telos classes without further properties, as e.g. the destination `Agent` of attribute `owner`.

The general methods for initializing and updating the view materialization are defined

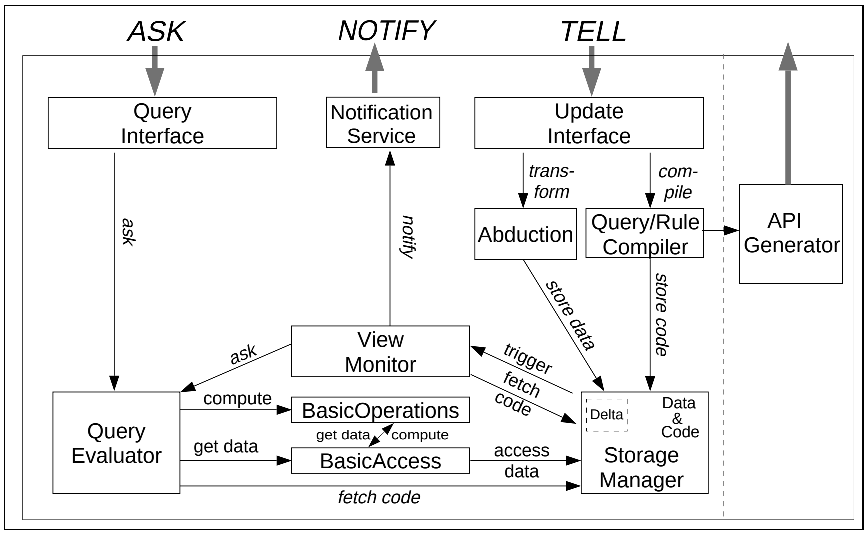


Figure 5: The ConceptBase server architecture

within a C++ class `CBviewAdmin` which again is specialized for each concrete view (`PS_Admin`). At the instance level we have a 1 to 1 correspondence between O-Telos and C++ objects where pointers replace references by name in the O-Telos database. The management (i.e. initialization and maintenance) of the materialized view `PS` is supported by instances of `PS_Admin` that contain information about the structure of each view object and its components and support communication with the server.

4 Implementation in ConceptBase

The basic view maintenance procedure sketched in section 2 as well as the view mapping described in the previous section were realized as additional components of the ConceptBase system. Since the API generation for C++ itself as presented above necessarily already covered several implementation issues we concentrate on architectural and communication aspects.

4.1 General Architecture

The ConceptBase server architecture [6] is shown in figure 5. Within this architecture, the rewriting steps sketched above are part of the ConceptBase `Query/Rule-Compiler` component while algorithm 1 constitutes the `ViewMonitor`. Textual and graphical user interface tools are linked to the system as clients over the Internet. Notifications about changes in the database which are of interest to such clients are provided through the `Notification Service`. This server component receives information about changes in view relations from the `ViewMonitor`, performs the transformation to a more complex format based on the original view definition and issues the corresponding messages to the client.

Another component `API Generator` generates the interface (including data structures and corresponding methods for data extraction and update) between server and client with respect to a certain view definition.

4.2 API Generation and Communication Interface

The whole set of generated classes and methods is naturally embedded into the library package of ConceptBase's C++ programming interface which provides basic communication routines, encapsulates the message protocol and makes a parser for O-Telos frame syntax available on the client side. Figure 6 shows an excerpt of the C++ class hierarchy constituting this library: Based on a class `IpcClient` realizing low-level TCP/IP socket communication the specialization `CBclient` provides all necessary methods (e.g. establishing and terminating the connection or updating and extracting data) the server offers to its clients. This support is extended by a further specialization `CBnotifyclient` which creates an additional message channel for notifications from the server to each client. A justification for this dedicated channel is that usually notifications are expected to be issued asynchronously to possible other operations the client performs on the server. Notifications sent out from the server are buffered on the client side and can be accessed by method `getNotificationMessage`. In contrast to polling style interaction where either the client has to compute view changes by himself (and only initiates recomputation on the server side) or at least has to check whether new messages for him arrived on the server, this approach is really event driven but of course there still remains the problem of synchronisation between "normal" program execution and initiating message processing as a task for the programmer. Finally, the general view administration class `CBviewAdmin` supports direct changes on the client data structures (method `processUpdateMessage` based on `getNotificationMessage`) by appropriate interpretation of the changes derived for the view relations during the view maintenance process on the server side. The initial extrac-

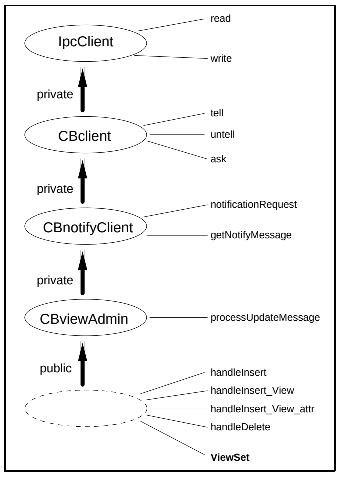


Figure 6: Hierarchy of communication classes

tion of the view extension sketched above can be done using the inherited method `ask` of superclass `CBclient`.

5 Application Experiences

Experience with using ConceptBase in design applications show that the usage of so called *integrity views* [20] is preferable to the usage of ordinary integrity constraints. The contents of integrity views precisely describe configurations of data that lead to a consistency violation and may be employed for a suitable representation of this violation on the client side. The problem of integrity checking can then be understood as special view maintenance problem and be handled with the same mechanism as sketched above. A second advantage of integrity views over traditional constraints is that it enables different levels of integrity at different times in the usage process of the database.

In a commercial application of ConceptBase [14], the first stages of a business reegineering process were supported by a set of more than 80 integrity views, defined on a given meta model that is used to guide the modelling process. Some of them represent hard integrity constraints that definitely have to be obeyed by the current database state. Others represent temporary disagreements in user perceptions or actual opportunities for process improvement. Thus, while all of them need to be monitored continuously, they are reacted to at different periods in the analysis process; Corrections of violations in one view may indirectly correct other violations, or it may actually cause new ones. In such a setting, non-incremental view maintenance may become very expensive. Since most of the views are externally materialized, the approach developed in this paper has therefore been a critical success factor for the usefulness of design database support in such a setting. We are currently studying the question how much certain techniques of automatic constraint repair (view updates) which consider the maintenance of multiple views simultaneously could further improve the analysis process. However, we recognized that users are often quite happy to observe the interplay between different views because it helps them to understand the application problem better; therefore, we are quite cautious with respect to automating too much.

6 Conclusions

We described an approach for notifying changes to externally materialized views held in application programs. Besides a basic view maintenance procedure that works without knowing the view materialization and only selectively rederives its *relevant* parts inside the database, it includes full API support based on C++ data structures. The deductive object manager ConceptBase served as implementation platform. We reported first experiences in practical applications of ConceptBase. Ongoing work comprises completing the implementation and building a mechanism for management of multiple perspectives within modelling activities on top of the basic view maintenance and notification procedures. The presented solution for C++ can be easily adopted by other object-oriented programming languages in particular Java. Since Java is a dynamic, interpreted language, application programs are able to load new classes and methods at runtime. The current procedure of code generation can be extended to send ready-to-run Java code to the client application such that views may be defined and materialized interactively. In addition to the management of data the server is then also responsible for delivering program code suited in a specific state of the application. This covers e.g. different representations of the views contents available to the users of the application and offered for choosing or switching.

References

[1] O. P. Buneman and E. K. Clemons. Efficiently monitoring relational databases. *TODS*, 4(3):368–382, September 1979.

[2] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. 17th Intl. Conf. on Very Large Data Bases*, pages 577–589, 1991.

[3] A. Gupta, H.V. Jagadish, and I.S. Mumick. Data integration using self-maintainable views. In *Int. Conf. on Extending Database Technology*, pages 140 –144, 1996.

[4] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering*, 18(2), June 1995.

[5] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Conference on Management of Data*, 1993.

[6] M. Jarke, R. Gallersdoerfer, M. Jeusfeld, M. Staudt, and S. Eherer. ConceptBase - a deductive object base for meta data management. *Journal of Intelligent Information Systems*, 4(2):167–192, 1995.

[7] M. Jeusfeld. *Update control in deductive object bases* (in German). Infix-Verlag, St.Augustin, Germany, 1992.

[8] M. Jeusfeld. Generating queries from complex type definitions. In *Proceedings of 1st Workshop KRDB'94*, Saarbruecken, Germany, 1994.

[9] V. Kuechenhoff. On the efficient computation of the difference between consecutive database states. In *Intl. Conf. on Deductive and Object-Oriented Databases*, 1991.

[10] B.S. Lee and G. Wiederhold. Outer joins and filters for instantiating objects from relational databases through views. *TKDE*, 6(1):108–119, 1994.

[11] J.J. Lu, Moerkotte G., J. Schue, and V.S. Subrahmanian. Efficient maintenance of materialized mediated views. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 340–351, 1995.

[12] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, Oktober 1990.

[13] J. M. Nicolas and K. Yazdanian. An outline of BDGEN: A deductive DBMS. In *Proceedings of the tri-annual IFIP Conf 83, Mason(ed), N-H*, 1983.

[14] H.W. Nissen, M.A. Jeusfeld, M. Jarke, G.V. Zemanek, and H. Huber. Managing multiple requirements perspectives with metamodels. *IEEE Software*, March 1996.

[15] N. Roussopoulos and A. Delis. Modern client–server DBMS architectures. In *Proceedings of the ACM SIGMOD conference*, pages 52–61. ACM, September 1991.

[16] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS. *IEEE Computer*, 19(12), December 1986.

[17] M. Staudt. *View management in client-server systems* (in German). Infix-Verlag, St.Augustin, Germany, 1997.

[18] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *Proc. 22nd International Conference on Very Large Databases (VLDB'96)*, Bombay, India, September 1996. Morgan Kaufmann.

[19] M. Staudt, M. Jarke, M. Jeufeld, and H. Nissen. Query classes. In S. Tsur S. Ceri, K. Tanaka, editor, *Proceedings of the 3rd International Conference on Deductive and Object-Oriented Databases (DOOD-93)*, pages 283–295. Springer, December 1993.

[20] M. Staudt, H.W. Nissen, and M.A. Jeusfeld. Query by class, rule and concept. *Applied Intelligence*, 4(2):133–156, 1994.

[21] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proceedings of the ACM SIGMOD conference*, San Jose, CA, June 1975.

[22] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 2: The New Technologies.* Computer Science Press, Rockville, MD, 1989.

[23] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *1995 ACM SIGMOD International Conference on Management of Data*, 1995.