

Active changes notification

Author: Prakash Kewalramani
Organization: Evaltech, Inc.
Evaltech Research Group,
Data Warehousing Practice.
Date: 09/24/2007
Email: erg@evaltech.com

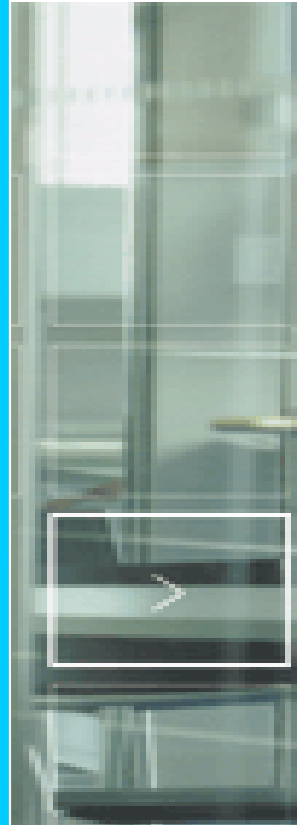


Abstract:

Active notification for clients about changes in the database is an important requirement for advanced interaction between the database and its applications. This notification is based on the subset of database contents. The incremental maintenance of such externally materialized views is an important open problem. In this white paper the solution is described by using knowledge base server concept base. This article includes following topics. Introduction to view maintenance tool, Basic procedure for view maintenance, Change notification for client view, What is API support for change notification?

Intellectual Property / Copyright Material

All text and graphics found in this article are the property of the Evaltech, Inc. and cannot be used or duplicated without the express written permission of the corporation through the Office of Evaltech, Inc.



Summary:

Active notification for clients about changes in the database is an important requirement for advanced interaction between the database and its applications. This notification is based on the subset of database contents. The incremental maintenance of such externally materialized views is an important open problem. In this white paper the solution is described by using knowledge base server concept base.

This article includes following topics.

- ✓ Introduction view maintenance tool
- ✓ Basic procedure for view maintenance
- ✓ Change notification for client view
- ✓ What is API support for change notification?



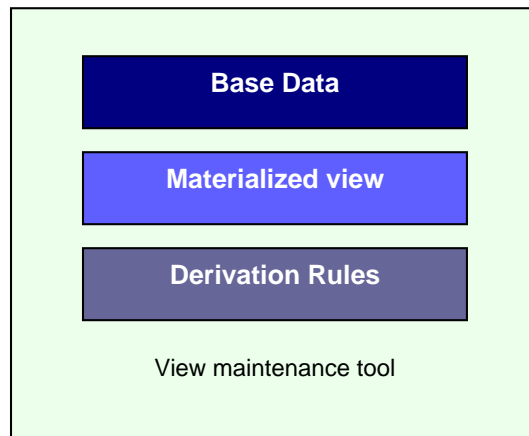
Introduction

When data is derived by set of view definition and redundantly stored to the base data then the view on data is called materialized.

Views can be materialized for reasons of reliability, efficiency and reusability. Materialized views are distinguished on the base of maintaining consistent with the base data over time.

Given a transaction that updates a set of base data, the corresponding changes all materialized views need to be computed by a view maintenance tool. Ideally, this should be done incrementally.

In a typical client-server architecture for database applications the server database management system (DBMS) maintains the state of the base data and provides views derived abstractions. Clients maintain view caches, which contain materialized views either in the same (e.g. clients that themselves have a DBMS component) or in a different representation as the database in client-specific data structures. 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. Following fig shows contains of view maintenance tool.



The maintenance tool has access to the base data and the view definition but not to the materialized view. This occurs quite frequently, namely whenever views are held in client data structures without counterpart inside the database. 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 detection of changes on the client views and 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 we are not aware of combined solutions in this area applicable to externally materialized views.

● The Basic View Maintenance Procedure

For solving the basic maintenance problem for views whose contents is only externally materialized we assume a deductive database with base and intensional relations derived by Data log rules. The solution comprises a two phase **rewriting** of the view definition rules to a set of view maintenance rules, which interact with the rules used for computing the view extension. The **evaluation** can be done using a standard relational query processor.

▶ Rule Rewriting

❖ Phase 1

The transformation follows for each relation r .

Let us assume that,

All possible deletions (overestimate) in a relation r^{del} caused by deletions (insertions) in other relation's occurring (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 occurring 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 re-derived 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.

❖ 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 sub-goals. 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 sub-goals in the body of the maintenance rules.

▶ View Maintenance by Rule Evaluation

The view maintenance procedure takes the transformed rule sets and computes the changes for views of interest by selectively re-deriving only the relevant parts of the old (intensional) database state. The basic idea is to process the maintenance rules stratum-by-stratum, perform a bottom-up evaluation on each layer and switch to the complete set of query rules whenever no further tuples can be derived. The algorithm employs a fix-point procedure that works on a stratified set of rules and an environment representing a subset of the overall database contents and steadily increasing by new relevant tuples. The fix-point procedure respects the changes of the environment produced during the preceding evaluation in a semi-naive manner and returns it with additional tuples inserted for certain relations. The new tuples for the magic predicates generated from the maintenance rules denote queries to be answered by switching to the query rules in order to continue the maintenance process.



Change Notification for Client Views

Based on detected changes for view (relations) inside the database the second step for providing notifications consists of API support that enables the client to receive and interpret notification messages in an appropriate way, namely by updating the view contents on its own data structures.



Views in ConceptBase

Following is the explanation of how the presented maintenance procedure serves for maintaining complex views of application programs working on top of the knowledge base management system ConceptBase.

ConceptBase is mainly intended for Meta data management and supports the O-Telos object model. An O-Telos object base is semantically equivalent to a deductive database, which includes a predefined set of rules and integrity constraints coding the object structure. The surface language syntax is frame based.

Based on the idea of so called query classes which serve for representing queries as special classes with necessary and sufficient membership conditions, views in ConceptBase are defined through an extended nested frame syntax.

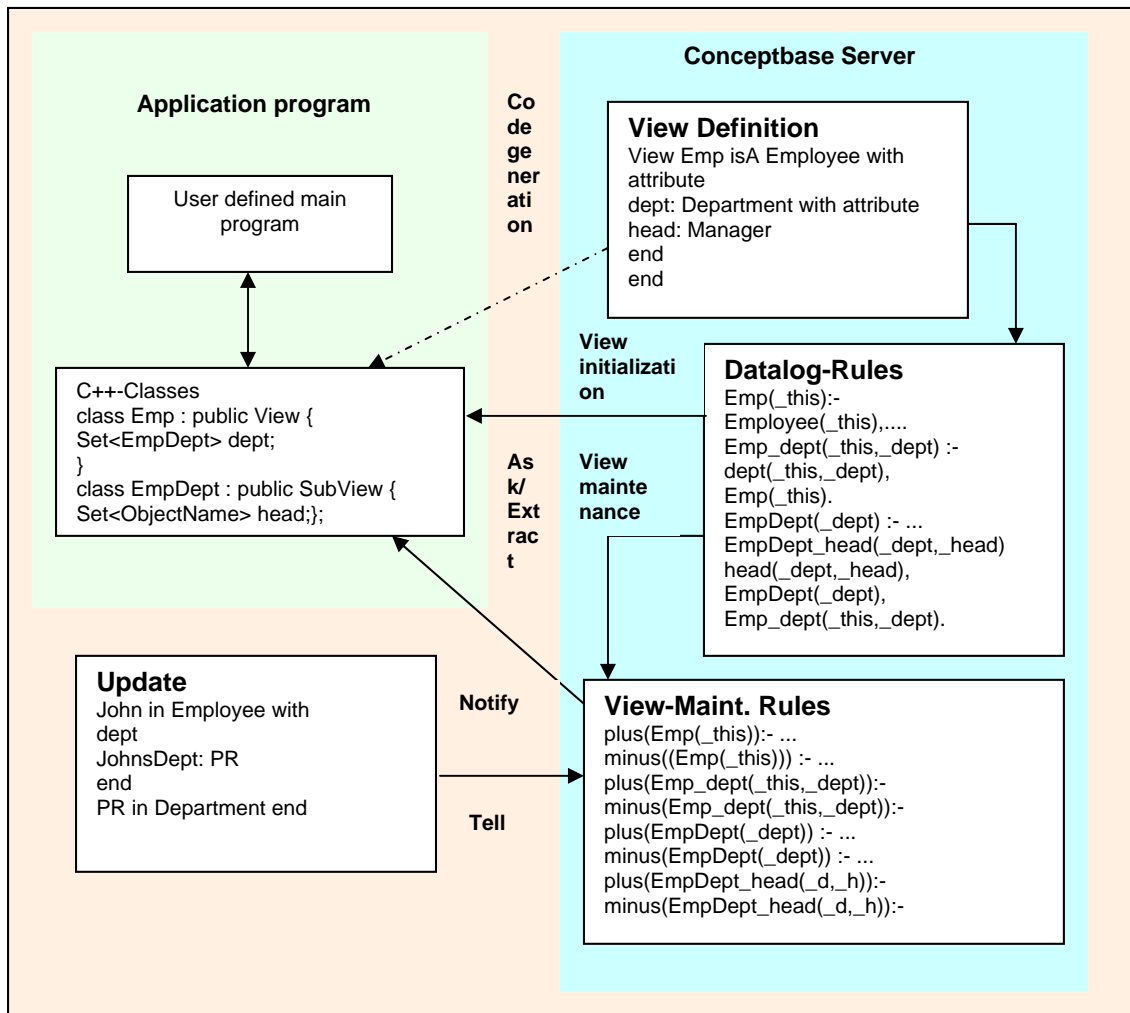
A simple schema of an employee database where employee's work for departments (attribute dept) which are led by managers (attribute head) could be the following:

Class Employee with attribute dept: Department end	Class Department with attribute head: Manager end
--	---

Class Manager end

Attributes in O-Telos are not mandatory in general and may be set-valued. Hence, not all employees e.g. must belong to a department and more than one manager may lead a department. Rules and integrity constraints can be specified in a many-sorted first-order language based on literals $In(x, c)$ (describing instantiation relationships) with shorthand $(x \text{ in } c)$, $I \text{ sa } (c, d)$ (for generalizations) with shorthand $(c \text{ I_sa } d)$ and $A(x, m, y)$ ("x has value y for attribute m") with shorthand $(x \text{ m } y)$. Variables are quanti-fied over classes, which is interpreted as instantiation relationships.

For illustration we can e.g. attach a constraint to class Employee stating that no employee is allowed to earn more money than his boss:



```
Class Employee with
...
constraint
salaryIC: $ forall m/Manager s1,s2/Integer
(this boss m) and (this salary s1)
and (m salary s2) ==> (s1 < s2) $
end
```

The same combination of frame based and logical language is used for specifying views on an O-Telos knowledge base. In order to represent the ternary relationship between employees, the departments they are working for together with the corresponding leaders, we can define a complex view Emp as follows,

```
View Emp isA Employee with
attribute
dept: Department with
attribute
head: Manager
constraint
: $...$
end
constraint
: $...$
end
```

Even those employees not working for a department and even those departments that do not have a leader should be-long to Emp. The example is very simple and we will not discuss details of the view language that allows annotating arbitrary constraints, path expressions, parameterization etc. within view definitions. Above shows the transformations that are necessary to ensure that the data in Emp can be extracted and maintained on the client side.

In order to process the view definition inside the database it is mapped to a set of Data log-rules: The view is decom-posed into a main view (derived relation Emp) specifying the membership of all complex view objects in class Employee and possible additional restrictions given in the constraint clause. A set of relational sub-views results both from the attribute clauses, and from inner sub-frames.

The derived relations Emp_dept (specifying employees together with their departments), EmpDept (containing all relevant departments satisfying the constraint of the inner sub-frame) and EmpDept_head (linking departments with their leaders).

Sub-views for attribute clauses are introduced due to the non-mandatory property of attributes to avoid any form of null values. Based on this rule set the rewriting procedure sketched above produces maintenance rules that for a given base data update compute the differentials for the derived relations gained for Emp.

● API Support for Change Notification

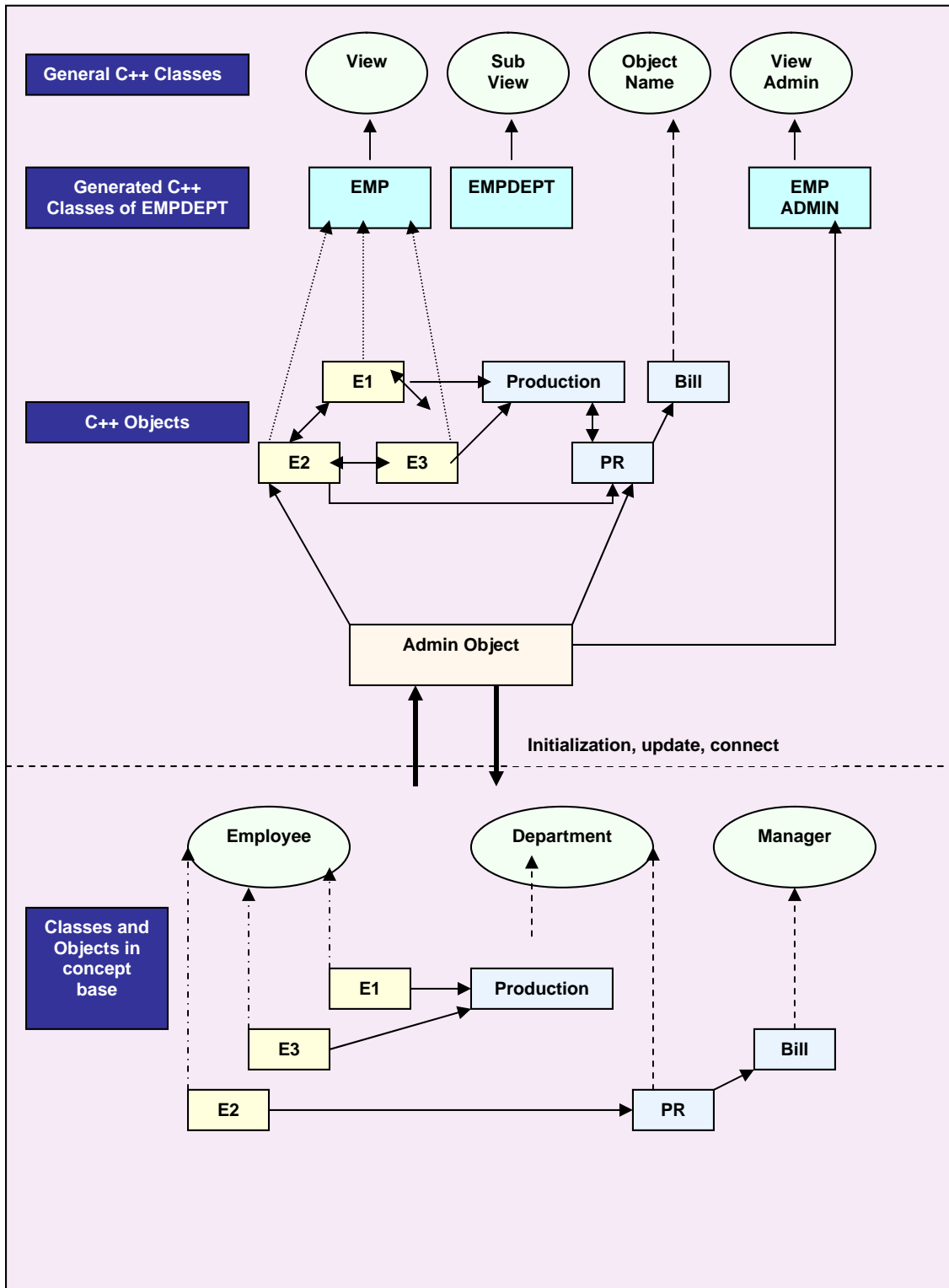
Concerning the view representation on the client side a corresponding C++ class is generated for Emp that manages for each of its attributes sets of pointers to other C++ classes representing sub-views obtained from sub-frames in the view definition. The client program can be built around these classes. The C++ classes are instantiated when the view ex-tension is extracted from the object base for the first time. On the server side the extraction is an NF2-like processing of the relations separately specifying the main view and the sub-views. Starting with the sub-views for the innermost sub-frames of the view definition their extensions are joined by an (left) outer-join operation, which collects all objects and their properties together. The step from sub-frame to the en-closing frame of the next outer level walks along with a nest operation performed on the objects properties such that the contents of the sub-view has the form of a binary NF2 relation consisting of pairs linking each object with tuples for their property sets. The mapping of the NF2 relation for the main view to the object-oriented class structure on the client side is straightforward.

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 sub-views 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 sub-views stemming from sub-frames 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.

Above figure illustrates the relationship between O-Telos objects in the database and their occurrence in the materialized view on the client side. For the view Emp an equally named C++ class represents the main part of the view definition

Following figure shows C++ vs. O-Telos objects



As subclass of a general class View while all occurring sub-frames are mapped to subclasses of SubView (only EmpDept in our example). Another general class ObjectName collects simple references in the view definition to O-Telos classes without further properties, as e.g. the destination Manager of attribute head. The general methods for initializing and updating the view materialization are defined within a C++ class ViewAdmin which again is specialized for each concrete view (EmpAdmin). 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 Emp is supported by instances of EmpAdmin that contain information about the structure of each view object and its components and support communication with the server.

```
class CBclient : private IpcClient {
public:
CBclient(char *host, int port);
CBanswer* tell(char *);
CBanswer* ask(char *query,char* format);
CBanswer* enrollMe(char *user,char *tool);
CBanswer* cancelMe();
// ...
};
class CBnotifyClient : private CBclient {
public:
CBnotifyClient(char* host, int port);
CBanswer* getNotificationMessage(int TimeOut);
};
class ViewAdmin : private CBnotifyClient {
public:
ViewAdmin(char *host, int port);
int processUpdateMessage(int TimeOut);
}
}
```

The whole set of generated classes and methods is naturally embedded into the library package of ConceptBase's C++ programming interface. Figure 3 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 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. This dedicated channel enables notifications to be issued asynchronously wrt. other operations on the server. Notifications sent out from the server can be accessed by method etNotificationMessage. In contrast to polling style interaction where either the client has to compute view changes by himself (and only initiates re-computation 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 synchronization between “normal” program execution and initiating message processing as a task for the programmer. Finally, the general view administration class ViewAdmin 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.