Monitoring Database Objects

Tore Risch
Hewlett-Packard Laboratories
1501 Page Mill Rd., Palo Alto, CA 94303

Abstract

A method is described for actively interfacing an Object-Oriented Database Management System (OODBMS) to application programs. The method, called a *database monitor*, observes how values of derived or stored attributes of database objects change over time. Whenever such a value change is observed, the OODBMS invokes *tracking procedures* within running application programs. The OODBMS associates tracking procedures and the object attributes they monitor, and it invokes appropriate tracking procedures when data changes. Use is made of atomic transactions in the OODBMS.

The applicability of monitors is localized both in time and space, so that only a minimal amount of data is monitored during as short a time as possible. Such localization reduces the frequency of tracking procedure invocation, makes it easy to add and remove monitors dynamically, and permits efficient implementation.

To demonstrate these ideas, an implementation is described for the Iris OODBMS [10]. The implementation uses a technique of partial view materialization for efficient implementation.

1 Introduction

Database monitors are computer programs that observe changes in the contents of database objects, e.g. the current price of some commodity or the location of some ship. In addition to stored facts one often wants to monitor derived information, e.g. the highest paid employee in a department or the expenses of a department relative to its sales. Other monitoring applications include real time systems where a process is invoked or where a user is alerted upon pre-specified data changes. The need for similar concepts has been indicated by others [7, 18, 20, 30, 31].

The monitoring mechanism described in this paper is integrated with the Iris object-oriented database system [10]. In Iris, both derived and stored object attributes can be defined using functions expressed by an object-oriented query language, OSQL [3].

A *tracking procedure* is a procedure in a running application program to be invoked if the value of an associated object attribute has changed because of state changes to the database. We use OSQL to specify the derived or stored associated attribute. The DBMS does not transmit monitored data to tracking procedures; it merely signals that the state of derived data has changed by invoking tracking procedures. It is up to the tracking procedures to retrieve data by accessing the database. Possible tasks for tracking procedures include notifying the end user that data have changed, refreshing data browsers, or passing information to other systems. In case the application program caches data retrieved from the DBMS, tracking procedures can be used to inform the application that pieces of the cached data have been invalidated by database updates, so that caches can be refreshed and computations dependent on the old caches can be undone.

In the architecture presented here, the DBMS stores associations between tracking procedures, the attributes they monitor, identifiers of application processes, and the physical workstations in which they reside. After the DBMS has detected changes in monitored data it uses this information to call the appropriate tracking procedures. We then say that the application process has been *notified* or that the DBMS has sent a *notification* to the process.

In conventional database interfaces the DBMS behaves as a 'passive object' in that an application program always must request service from the DBMS. With

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

monitors, the DBMS becomes an 'active object'; a request from an application process to the DBMS may result in procedure invocations in other application processes.

Tracking procedure invocations can be either *local* or *external*. An invocation is *local* if the process containing the invoked tracking procedure also requested the triggering database updates. An invocation is *external* if it is the result of a database update by some other process. External tracking procedure invocations do not occur until after the updates have been committed, while local invocations occur *before* the commit. Thus at the end of an update transaction, the DBMS first invokes local tracking procedures in the updating process and then immediately after the commit, the DBMS invokes external tracking procedures. The local tracking procedures are invoked synchronously, i.e. the system waits for them to return, while the external tracking procedures are invoked asynchronously and the system does not wait for them to return. Since a process may be performing unrelated work when the tracking procedure is invoked asynchronously, the system provides a no-interrupt option. In this mode the DBMS accumulates notifications and the application process can check for them synchronously.

2 Related work

This section discusses related techniques from several research areas. The database field has the notions of alerters, triggers, and integrity constraints, while the programming language field has the idea of active values. These ideas are somewhat related to data driven expert systems.

In [7] it is proposed that alerters be displayed when database updates cause certain boolean conditions to become true. This is a common special case of monitoring a boolean value. The RETRIEVE ALWAYS mechanism of Postgres [30] re-retrieves the results of queries into 'portals' associated with each client whenever there is a data change. Our monitoring mechanism is different from the above methods because the DBMS calls procedures in the client processes instead of sending data. In addition we employ methods, described below, that *localize* the applicability of monitors both in time and space.

Some DBMSs, e.g. System R [1], use database procedures called 'triggers' that are invoked upon updates of either user-specified base relations or by other actions [8, 16]. The ECA model [8] is a generalization of triggers in which the programmer can declare a precondition for the trigger. Triggers can be chained so that one update triggers another. This is useful for maintaining integrity constraints. However, the complicated cascading of triggers can make the database structure tangled and difficult to understand and maintain. Moreover, improper use of cascading triggers may slow down database updates considerably. In contrast, the invocation mechanism described here is non-procedural. The user specifies a database query whose value states are to be monitored, leaving the system to keep track of actions that may change a monitored state. In addition, the system creates an optimized mechanism to invoke tracking procedures when appropriate.

The technique called *access-oriented programming*, or 'active values', used in object-oriented programming languages such as Loops [28] is similar to traditional, procedural database triggers. But unlike DBMSs, active values are restricted to a single-user environment. Active values typically set properties or display some value. It is important that active values can be dynamically added to and removed from running systems [21], so that their use does not interfere with the logic of the rest of the system.

Declarative integrity constraint methods [4, 11, 18, 20, 23, 27, 29] are related to the monitor model in that they monitor boolean conditions, typically cancelling the updating transaction if the boolean conditions are unsatisfied at commit time. Tracking procedures, by contrast, are local to the application program process, are active during limited time periods, are triggered by value changes of any data type, can perform many tasks, and can be invoked both locally inside the updating transaction and asynchronously in processes outside the transaction. The tracking procedure may cancel the transaction only when invoked locally.

Database integrity constraints typically have performance problems [2]. The reason for this is that they may be very expensive to check for every update. Several techniques have been proposed to attack this problem, e.g. limit the assertions so that only those relevant to the updates of the particular file are checked [11, 19]; symbolically simplify the constraints checked per file [7, 11]; maintain intermediate aggregation data [4, 15]; exploit semantic constraint simplification [23]; simplify and check all constraints after many updates [12]; distribute constraints over several processors [24]; or automatically generate consistency maintenance code from declarative integrity constraints [18, 19, 20, 23]. These techniques are applicable to the monitor model as well. The monitor model avoids the efficiency problems of global integrity constraints, by being localized both in time and in the amount of data monitored, and by allowing asynchronous execution outside updating transactions.

Cactis [13] is an OODBMS to support software environments in which each data item is time stamped when updated. This allows the system to know what to recompile when the user retrieves an object. By contrast, our method checks if monitored data have actually changed, in which case the tracking procedures are invoked.

We may also contrast the present work with some data driven expert system tools like Syntel [9, 25, 26] and OPS5 [6]. Syntel is a tight integration of a database, an expert system tool and a user interface,

specialized for financial applications. Syntel monitors a moderate amount of virtual memory data retrieved from persistent data and user input. In Syntel all shared data is passive, and there is no feedback from the persistent data to Syntel. By contrast, the method presented here monitors interactions between multiple users and their active and shared persistent large database independent of applications and user interfaces. The methods presented here may be used to extend the scope of data-driven systems like Syntel or forward chaining, rule-based systems like OPS5, making it possible to design systems where several expert systems cooperate.

3 An example

The following example of an Iris database will be used later to illustrate the monitor model. Assume a database of 'Person' and 'Department' objects with the attributes Name, Income, and Dept for objects of type Person and the attributes DeptName and Manager for objects of type Department.

The objects types can be defined by the OSQL statements:

```
create Type Person;
create Type Department;
```

The attributes of the above types are accessed using *extensional* Iris functions with OSQL definitions:

```
create Function Name(Person) -> String;
create Function Income(Person) -> Integer;
create Function Dept(Person) -> Department;
create Function DeptName(Department) -> String;
create Function Manager(Department) -> Person;
```

For example, given an object identifier (i.e. a surrogate), p, for a person, we may access its income and department name by the OSQL queries:

```
select Income(p);
select DeptName(Dept(p));
```

We may also set the value of an attribute by OSQL statements, e.g.:

```
set Income(p) = 41000;
```

These statements define the object types Person and Department. Iris also allows different *views* of objects by defining *intensional* Iris functions. For example, we may want to work with a view of the Person object with the attributes Name, Income, DeptName (as a string), MgrName (as a string), EarnsMoreThanManager (as a Boolean), and ColleaguesEarningMore (as a set of tuples of their names and incomes).

Such view of a Person object can be defined by the following Iris functions:

```
create function Name(Person) -> String;
                              /* extensional */
create function Income(Person) -> Integer;
                              /* extensional */
create function DeptName(Person p) -> String
  as select DeptName(Dept(p));
                              /* intensional */
create function MgrName(Person p) -> String
  as select Name(Manager(Dept(p)));
                              /* intensional */
create function EarnsMoreThanManager(Person p)
                                     -> Boolean
  as select where
     Income(Manager(Dept(p))) < Income(p);
                              /* intensional */
create function ColleaguesEarningMore(Person p)
                         -> <Person, Integer>
  as select each Person coll, Integer inc
  where inc = Income(coll) and
        inc > Income(p) and
        Dept(coll) = Dept(p);
                              /* intensional */
```

Thus an application may have the above view of the Person object, not even knowing how attributes are physically stored.

Iris functions can also have more than one argument, e.g. to retrieve those persons earning more than a given income in a given department:

```
create function HighInc(Department d, Integer i)
                                     -> Boolean
  as select each Person p
     where Income(p) > i and Dept(p) = d;
```

4 Design overview

We will now continue by discussing some critical aspects of the architecture:

- Efficiency is gained by minimizing the amount of monitored data and by allowing monitors to be active only while applications need them.
- We also gain efficiency by separately performing the expensive process of compiling monitor definitions, and storing the analyzed definitions in the database for later use.
- Application processes may receive asynchronous notifications while performing an unrelated task. Such processes will be interrupted by the DBMS to invoke the tracking procedures asynchronously. The DBMS does not wait for such asynchronous calls to return, to maximize system performance.
- Notifications of external tracking procedures must be delayed until just after transactions are committed.
- The DBMS must handle distributed notifications to client processes executing on separate machines.
- Finally, we must efficiently detect when values of monitored queries have changed.

4.1 Localizing Monitors

The monitor model handles common situations occurring in a serverized, shared, object-oriented DBMS such as Iris. Efficient execution is obtained from a relatively simple implementation.

One common situation involves monitoring values of properties of a given object. The properties may be either stored or derived from other properties. Often, monitoring is needed only during the execution of particular sections of an application process.

For example, we may wish to monitor the values of attributes `Income`, `ColleaguesEarningMore`, and `EarnsMoreThanManager` of a Person object view, p, invoking the tracking procedures `Mp1`, `Mp2`, and `Mp3`, respectively, when any of these values changes.

We provide *restricted monitors* for efficient execution of common situations, as follows:

- The monitors can be *time localized* so they are active only during a limited time, e.g. when a segment of an application program requires them to be active. We must therefore provide for efficient activation and deactivation of monitors.
- The monitors can be *client localized* by associating monitors with client processes. The system deactivates monitors when no client needs them.
- The monitors can be *object localized* by monitoring values of particular, or *focused* database objects, e.g. properties of specific persons. During the execution of an application program the monitor's focus may change from one object to another, i.e. the functions will be monitored for different arguments. Therefore the system should be reasonably efficient in refocusing from one object to another.
- The monitors can be *property localized* so that not every property of a database object is monitored. In the example, only the attributes Income, ColleaguesEarningMore, and EarnsMoreThanManager are monitored, not, e.g. Manager and Name. This restriction is important, since an object view may have many attributes.

In Iris, all the above localizations are supported.

4.2 Monitor Definition and Activation

Time localization requires efficient monitor activation and deactivation. This is achieved by separating monitor *activation* and *deactivation* from monitor *creation* and *deletion*, thereby performing as much work as early as possible at monitor creation time.

Monitor creation analyzes the monitored query and dynamically builds persistent tables and indexes to support efficient tracking. Monitor creation is therefore relatively slow. Monitors remain in the database until they are explicitly deleted. Monitors are generic in that they may be subsequently activated for different sets of arguments; we say that there are many *monitor instances* for a given monitor definition.

An application program typically focuses on different database objects during different time periods. The system thus has to support efficient changes in the set of focused objects. In Iris this amounts to an efficient method to vary arguments of monitored Iris functions. To refocus a monitor from one object to another we simply deactivate the monitor for one object and then activate it for another object.

Iris also supports overloaded functions. These currently cannot be monitored.

The following functions are provided for monitor creation and activation:

- `DefineMonitor(Fn)`, creates a new monitor; Iris data structures are created to support subsequent activation of the created monitor. Any Iris query can be monitored by first defining it as an Iris function and then defining a monitor for that function.
- `ActivateMonitor(Fn,Arglist,Proc)`, activates a monitor for a given Iris function, set of arguments, and tracking procedure within the current application process. The system will subsequently invoke the tracking procedure on changes to monitored object values.
- `DeactivateMonitor(Fn,Arglist)`, deactivates a monitor for the given Iris function and arguments in the current process.
- `DeleteMonitor(Fn)`, deletes a monitor definition permanently.

`DefineMonitor` creates a new table for each monitor definition called the *instance client table*. The instance client table is later updated by `ActivateMonitor` to store the network address and process identifier of each application process monitoring a given instance. The system also assigns a unique *tracking procedure identifier* to each tracking procedure in each process and stores it in the instance client table. Thus, by looking up the instance client table, the system will know which tracking procedures in which processes in which machines to notify when the instance is changed.

4.3 Notifying clients

After the database has been updated, one or more tracking procedures may be invoked in case their monitored function values were changed. The database server must detect value changes of monitored Iris functions, and notify client processes to invoke their tracking procedures.

The system only sends notifications after it has detected change to monitored data.

The actual time to check if updates make notifications necessary may vary. The most common case is that monitor notifications are checked only at the end of a transaction, when the database is in a consistent state. Therefore, in the current implementation, checks for notifications are made at transaction commit. However, the system provides an option for the application

program to check for notifications even before a transaction is committed, by explicitly calling a procedure, `CheckMonitors`.

If `CheckMonitors` is called before a transaction is committed, and there have been updates to values monitored by that process, the system will invoke local tracking procedures of the application process. However, notifications of monitors activated by *other* processes will always be delayed until just after the transaction is committed. The system works this way because transactions hide updates from other clients until they are committed. (If 'dirty reads' were allowed we could eventually relax this delay.)

Another option is to defer checking monitors altogether. Instead, `CheckMonitors` is called explicitly after any number of transactions have been committed, or perhaps in some separately running process. This option has not been implemented.

Local tracking procedures are invoked because of local database updates. By contrast, external tracking procedures are invoked because of database updates by other processes. The external invocation may happen at any time, even when the application process is doing unrelated work. The notified client process is interrupted asynchronously when the notification arrives, in order to invoke the tracking procedure.

It is important to have the option of ignoring external monitor notifications, e.g. while processing a critical section of code. When external notifications are accepted, we say the application process is in *listening mode*. The system provides a procedure to toggle between listening mode and *not* listening mode.

When an application process is not in listening mode its monitor notifications will be collected by the system. Tracking procedure invocations will then be delayed until the procedure `CheckMonitors` is explicitly called or the process is toggled to listening mode.

4.4 Distributed Notification

Since the application process may run on a workstation other than the Iris server, we need a mechanism to handle the network communication involved in notifying a client. Such a communication architecture has been designed, by using the Network File System protocol for remote procedure calls. Figure 1 illustrates this architecture. [1]

The basic idea is that every workstation running an Iris client must also run a *monitor server* that takes care of receiving notifications and interrupting application processes as appropriate.

Given a notification, a remote procedure call, `NotifyMonitor(ProcId,MonId)`, is made to the affected monitor server, informing the monitor server that a notification has occurred to the monitor identified by `Monid` of the process identified by `ProcId` .

The monitor server saves the tracking procedure identifiers for a given process in a main memory data structure. It then interrupts the appropriate application process, signalling that some of its tracking procedures have been activated. In case the notified process has been terminated the system deletes the tracking procedure identifiers of that process from the monitor server and deactivates the monitor. When control returns back to the Iris server, the DBMS knows that the application process has been notified, but it is up to the application process to do something when the notification interrupt arrives.

If the application process is not in listening mode the interrupt signals are ignored. A later call to `CheckMonitors` first does a remote procedure call, `GetNotifications(ProcId,MonIds)`, to the monitor server to access the tracking procedure identifiers activated for the application process identifier. `GetNotifications` then deletes the tracking procedure identifiers from the global data structure of the monitor server.

Given these tracking procedure identifiers, `CheckMonitors` knows which application process procedure to invoke for the activated monitors.

If the application process is in listening mode, it catches all notification interrupts and then calls `CheckMonitors` immediately to asynchronously invoke notified tracking procedures.

If one or more notification interrupts arrive while another tracking procedure is running, the system will call `CheckMonitors` again *after* the first tracking procedure has finished executing. By this sequentialization, programmers need not worry about recursive notifications of tracking procedures.

4.5 Change detection

The purpose of the change detection algorithm is to determine if the values of a monitored Iris function have changed since the last time its monitoring clients were notified. This process is initialized by notifying the monitoring client at the end of the transaction in which the monitor was activated. Then the system determines if the value of the monitored function has changed since the previous notification.

The current algorithm is rather simple, assuming that only a few objects are monitored per function and that the values of monitored properties are not very voluminous. It also assumes that the definitions of Iris functions are simple and fast to evaluate, a reasonable assumption if the functions are used to do simple derivations such as those in the example.

The algorithm is designed to minimize the overhead of database operations when no monitor is created or active. It also minimizes the notification traffic; in particular, notification occurs only when there is actually a value change and then only at monitor checking time.

[1] The inter process communication in the current implementation is slightly more complicated than illustrated in Figure 1., because the code to detect changes is actually implemented as part of the client interface to Iris.

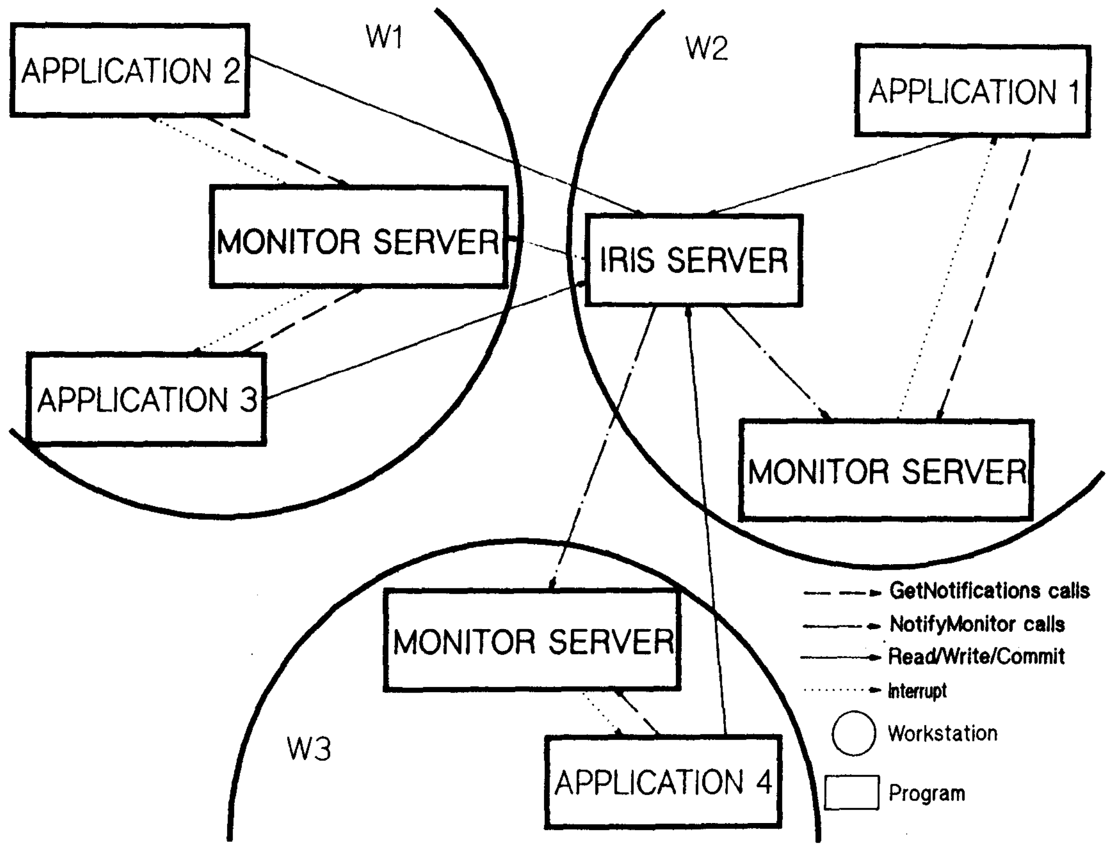

Figure 1: Inter process communication

The algorithm can be viewed as a variant of view materialization [5, 15, 17] whereby only the monitored instances of an Iris function are materialized, rather than the complete function traditionally materialized; thus we call the method *partial view materialization.* When an Iris function is defined, the system will also access-path-optimize the query to retrieve function values given that all arguments are known. If there has been an update invalidating some partial view materialization, the algorithm re-executes the access-path-optimized monitored function call for each monitored instance and compares the new values with the materialized values. If they are different, the new values are cached and the monitoring application processes are notified.

The change detection algorithm has four steps:

- A local *update detection* procedure is called for *every* Iris operation that updates the value of a stored Iris function [10], independent of whether the function is monitored or not. For every transaction, the system keeps track of Iris functions that have been updated. This procedure holds no locks and is very fast, associating with each transaction a virtual memory table of update-detected Iris functions. The size of this table is limited by the number of Iris functions updated by the transaction.
- At monitor checking time, for every update-detected Iris function, a monitor detection test is made to see if the updated functions participated in some monitor for some set of arguments. This is done by referring to a table, `SupportsMonitors`, that is updated by the system when monitors are created. This table maintains relationships between updatable Iris functions and their associated monitored Iris functions. For example these are some value sets of `SupportsMonitors`:

```
SupportsMonitors(Income)  =
      {Income, EarnsMoreThanManager,
       ColleaguesEarningMore}
SupportsMonitors(Dept)    =
      {EarnsMoreThanManager,
       ColleaguesEarningMore}
SupportsMonitors(Manager) =
      {EarnsMoreThanManager}
```

  `SupportsMonitors` is of limited size and can be kept in main memory.
- For every Iris function that was *monitor-detected*, a more expensive *instance detection* is done to test if its value changed. This detection is much slower than the previous ones, requiring re-execution of the pre-defined (and access-path-optimized) monitored Iris function in order to materialize the monitored instances and compare the new and old materialized values.

  The materialized values are stored in a system-generated table called the *instance cache table.* For example, the monitor for `ColleaguesEarningMore` will generate an instance cache table

```
_MCA_ColleaguesEarningMore(Person)
                -> <Person, Integer>
```

- For every *instance detection*, a notification is sent to the client processes that had activated that instance. The process identifier is found in the *instance client table* that maps each monitored instance to a list of its client processes and tracking procedures. Three levels of identification are needed:
  - A *client identifier* identifies the name (network address) of the workstation running each application process.
  - A *process identifier* identifies each (Unix) process per workstation.
  - A *tracking procedure identifier* identifies each tracking procedure in each process.

4.6 Fetching notified data

The notification mechanism does not transmit any data to the client. Instead the application process may decide to re-retrieve monitored instances. This can be done using the standard Iris query language. However, since the partial view materialization algorithm also caches monitored instances, we make use of the *instance cache table* when fetching notified data. This is implemented by an interface between the notification mechanism and the Iris query language, so that queries are modified to make use of the instance cache table.

For example, assume that the application program makes the OSQL query:

```
select ColleaguesEarningMore(p);
```

and that `ColleaguesEarningMore` is monitored for Person p. The system will modify the above query to access the instance cache table:

```
select _MCA_ColleaguesEarningMore(p);
```

thus avoiding any access path search. That is `ColleaguesEarningMore` is a derived function defined in terms of the stored Iris functions `Income` and `Dept`. It is not required to compute the derived value again since it is already cached. This simple query modification can be made in the client process without accessing the Iris server.

When a tracking procedure is invoked, the system passes the system-created tracking procedure identifier and a flag indicating whether the tracking procedure is invoked locally or externally. By passing the tracking procedure identifier, it is possible to write generic tracking procedures that can be used for many monitor definitions. A special primitive is provided to fetch the monitored data given such a tracking procedure identifier, using the instance cache table.

5 Summary and Discussion

We have proposed and implemented a method to interface application programs to the Iris object-oriented database system by using *monitors* that activate application program tracking procedures whenever values of derived or stored attributes of objects change. The attributes are specified declaratively, using an object-oriented query language. Monitors are localized both in space and time by limiting the amount of monitored data to be object attribute values, and by being active only while application processes need them.

The time localization strategy requires fast methods to turn monitors on and off. Therefore we separate the (slow) process of defining monitors from their activation and deactivation. Thus, monitor definition involves creation and update of database tables as well as optimizations for later monitor activations. A monitor that is created but not activated generates little overhead to database operations.

The application may run on a client workstation other than the database server. The system keeps track of which application processes on which workstations monitor which data, and automatically notifies application processes when monitored data change. Notification is done by invoking tracking procedures; external invocations may interrupt application processes and execute asynchronously, without the DBMS waiting for the tracking procedures to terminate.

Monitors may improve the concurrency of object-oriented database systems by allowing for very short transactions instead of long transactions, and letting the notification mechanism broadcast notifications to affected clients when database updates are committed.

The monitor model is also useful for the construction of systems of data-driven cooperative processes by letting the DBMS activate processes given data updates, rather than letting the cooperative processes directly call each other. In this way concurrent process invocations can be stated declaratively.

It is important that monitors execute efficiently and do not slow down the database server for applications not using them. The current implementation is efficient enough for some typical uses of database objects, where attribute definitions are specified using simple (thus fast) queries. More research is needed to extend the technique to handle complex queries.

The present change detection strategy could be combined with other strategies, such as those used for integrity constraint checking [4, 7, 11, 12, 15, 18, 19, 20, 23, 24]. Sometimes the system should not need not save values to determine that a change has occurred. For example if a monitored query specifies the sum of a set of values of which only one has changed, we may deduce that the sum has changed without doing any materialization.

The methods presented here are also applicable to relational DBMSs, by regarding Iris functions as relational views. The method proposed by Wiederhold [32] can be used to define derived objects by relational views. (Iris has many additional object features, including a persistent hierarchical type system, and persistent object identifiers.)

The monitor model could be generalized to handle integrity constraints, which can be regarded as universal monitors activated continuously for all applications. The tracking procedure for an integrity constraint would always be locally invoked whenever the constraint is violated and it would always cancel the transaction. However, more research is needed to get good performance for such integrity constraint monitors, since they would not be localized and since they could not use asynchronous tracking procedure invocations.

ACKNOWLEDGEMENTS:

I wish to thank Dan Fishman, Bill Kent, David Beech, and Waqar Hasan for helpful comments on earlier versions of this paper.

References

[1] M.Astrahan et al: System R: A Relational Approach to Database Management, *ACM Transactions on Database Systems*, 1 (2), June 1976.

[2] D.Badal, G.Popek: Cost Performance Analysis of Semantic Integrity Validation Methods, *Proc. ACM SIGMOD Conf.*, 1979, pp109-115.

[3] D.Beech: A Foundation for Evolution from Relational to Object Databases, *Advances in Database Technology - EDBT '88*, Lecture Notes in Computer Science, Springer-Verlag, 1988, pp251-270.

[4] P.Bernstein, B.Blaustein, E.Clarke: Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data, *Proc. 6th VLDB Conf.*, 1980, pp126-136.

[5] J.A.Blakeley, P.A.Larson, F.W.Tompa: Efficiently Updating Materialized Views, *Proc. SIGMOD*, Washington D.C., 1986, pp61-71.

[6] L.Brownston, R.Farell, E.Kant, N.Martin: *Programming Expert Systems in OPS5*, Addison-Wesley, Reading, MA, 1985.

[7] O.P.Buneman, E.K.Clemons: Efficiently Monitoring Relational Databases, *ACM Transactions on Database Systems* 4, 3 (sept. 1979), pp368-382.

[8] U.Dayal, A.P.Buchmann, D.R.McCarthy: *Rules Are Objects Too: A Knowledge Model For An Active, Object-Oriented Database System,* Advances in Object-Oriented Database Systems, 2nd Intl. Workshop on Object-Oriented Database Systems, Sept. 1988, pp129-143.

[9] R.O.Duda, P.E.Hart, R.Reboh, J.Reiter, T.Risch: Syntel: Using a functional language for financial

risk assessment, *IEEE Expert* 2, 3 (Fall 1987), pp18-31

[10] D.H.Fishman, D.Beech, H.P.Cate, E.C.Chow, T.Connors, J.D.Davis, N.Derrett, C.G.Hoch, W.Kent, P.Lyngbaek, B.Mahbod, M.A.Neimat, T.A.Ryan, M.C.Shan: Iris: An Object-Oriented Database Management System, *ACM TOOIS*, 5, 1 (Jan 1987), pp48-69

[11] M.Hammer, D.McLeod: A Framework for Database Semantic Integrity, *Proc. 2nd Int. Conf. on Software Engineering*, San Fransisco, 1976, pp498-504.

[12] A.Hsu, T.Imielinski: Integrity Checking for Multiple Updates, *Proc. ACM SIGMOD Conf.*, Austin, 1985, pp152-168.

[13] S.E.Hudson, R.King: Object-Oriented Database Support for Software Environments, *Proc. ACM SIGMOD Conf.*, San Fransisco, 1987, pp491-502.

[14] W.Kim, D.S.Reiner, D.S.Batory: *Query Processing in Database Systems*, Springer-Verlag, New York, 1985.

[15] S.Koenig, R.Paige: A transformational framework for the automatic control of derived data, *7th VLDB Conf.*, 1981, pp306-318.

[16] A.M.Kotz, K.R.Dittrich, J.A.Mulle: Supporting Semantic Rules by a Generalized Event/Trigger Mechanism, *Advances in Database Technology - EDBT '88*, Lecture Notes in Computer Science, Springer-Verlag, 1988, pp76-91.

[17] B.Lindsay, L.Haas, C.Mohan, H.Pirahesh, P.Wilms: A Snapshot Differential Refresh Algorithm, *Proc. SIGMOD*, Washington D.C., 1986, pp53-60.

[18] M.Morgenstern: Active Databases as a Paradigm for Enhanced Computing Environments, *9th VLDB Conf.*, Florence, 1983, pp34-42.

[19] M.Morgenstern: CONSTRAINT EQUATIONS: Declarative Expression of Constraints With Automatic Enforcement, *Proc. 10th VLDB Conf.*, Singapore, 1984, pp291-300.

[20] M.Morgenstern: The Role of Constraints in Databases, Expert Systems, and Knowledge Representation, in L.Kerschberg ed.: *Expert Database Systems*, Benjamin Cummings, Menlo Park, CA, 1986.

[21] K.Osterbye: Active Objects: An Access Oriented Framework for Object-Oriented Languages, *Journal of Object-Oriented programming*, Vol. 1, No 2, (June/July 1988), pp6-10.

[22] J.Park, A.Segev: Using Common Subexpressions to Optimize Multiple Queries, *Proc. 4th Intl. Conf. on Data Engineering*, Los Angeles, 1988, pp311-318.

[23] X.Qian, G.Wiederhold: Knowledge-based Integrity Constraint Validation, *Proc. 12th VLDB Conf.*, Kyoto, Japan, 1986, pp3-12.

[24] X.Qian: Distribution Design of Integrity Constraints, *Proc. 2nd Intl. Conf. on Expert Database Systems*, Virginia, 1988, pp75-84.

[25] R.Reboh, T.Risch: Syntel: Knowledge programming using functional representations, *Proc. AAAI-86* (Philadelphia, PA, Aug), Morgan Kaufman, Los Altos, CA, 1986, pp1003-1007.

[26] T.Risch, R.Reboh, P.Hart, R.Duda: A Functional Approach to Integrating Database and Expert Systems, *Communications of the ACM* 31, 12 (Dec. 1988), pp1424-1437.

[27] A.Shepherd, L.Kerschberg: PRISM: A knowledge based system for semantic integrity specification and enforcement in database systems, *Proc. ACM SIGMOD Conf.*, Boston, MA, 1984, pp307-315.

[28] M.J.Stefik, D.G.Bobrow, K.M.Kahn: Integrating access-oriented programming into a multiparadigm environment, *IEEE Software* 3, 10 (Jan. 1986), pp10-18.

[29] M.Stonebraker: Implementation of Integrity Constraints and Views by Query Modification, *Proc. ACM SIGMOD Conf.*, San Jose CA, May 1975.

[30] M.Stonebraker: The Design of POSTGRES, *Proc. ACM SIGMOD Conf.*, Washington, D.C., 1986, pp340-355.

[31] M.Stonebraker: Future Trends in Data Base Systems, *Proc. 4th Intl. Conf. on Data Engineering*, Los Angeles, 1988, pp222-231.

[32] Wiederhold, G.: Views, objects, and databases, *IEEE Computer* 19(12) 1986, pp.37-44.