SFDC 1016

ACTIVE DATABASE MANAGEMENT SYSTEMS

Umeshwar Dayal*

Computer Corporation of America
4 Cambridge Center
Cambridge, MA 02142 USA

ARPANET: dayal@cca.cca.com

Abstract

Conventional passive database management systems are inadequate for time-constrained applications, because they either do not provide timely response to critical situations or compromise modularity. Active database management systems attempt to provide both modularity and timely response, by allowing event-condition-action rules to be specified declaratively; when events of interest occur, they efficiently evaluate the corresponding conditions, and if these conditions are satisfied, they trigger the corresponding actions. The development of active database management systems requires the solution of a number of research problems in the areas of knowledge modelling, execution modelling, condition monitoring, scheduling, system architecture, and performance evaluation. This paper describes the principal research issues in each of these areas, surveys the approaches being taken in a number of research projects on active DBMSs, and emphasizes the approaches we are taking in the HiPAC (High Performance ACtive DBMS) project.

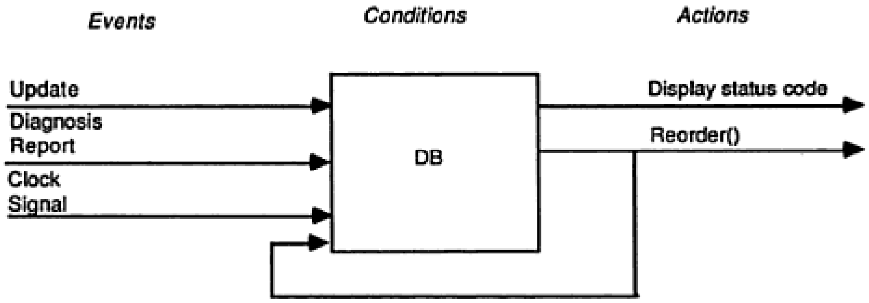
1. INTRODUCTION

Traditional database management systems (DBMSs) are *passive*: they execute queries or transactions only when explicitly requested to do so by a user or application program. Many applications, such as computer integrated manufacturing (CIM), office workflow control, process control, program trading, battle management, and network management, which require timely response to critical situations, are not well served by these passive DBMSs. For these *time-constrained* applications, it is important to monitor conditions defined on states of the database, and then, once these conditions occur, to invoke specified actions, subject to some timing constraints. For example, inventory control in an automated factory requires that the quantity on hand of each item be monitored; if the quantity on hand falls below a threshold for some item, then a reorder procedure may have to be initiated before the end of the working day. A situation assessment application requires that various targets be tracked; if one is discovered to be within a critical distance, then an alert code may have to be displayed on the commander's screen with the highest possible priority. For these applications, the correctness of a result depends not only on the correctness of a computation or on the proper interleaving of operations, but also on the timeliness of the result.

With a passive DBMS, two approaches to meeting the requirements of time-constrained applications are possible, but neither is satisfactory. The first approach is to write a special application program that polls (periodically queries) the database to determine if the situation being monitored has occurred. However, if the program polls too slowly, it runs the risk of missing the response time window; if it polls too frequently, it runs the risk of flooding the system with queries that usually return an empty answer. The second approach is to augment each program that updates the database to check the situation being monitored, and to invoke the action if the situation turns out to be true. Unfortunately, software modularity is now compromised: any modification to the situations being monitored or to the corresponding actions will require

This work was supported by the Defense Advanced Research Projects Agency and by the Rome Air Development Center under Contract No. F30602-87-C-0029. The views and conclusions contained in this report are those of the author and do not necessarily represent the official policies of the Defense Advanced Research Projects Agency, the Rome Air Development Center, or the U.S. Government.

* This paper reports work done by the HiPAC project team, which includes, besides the author, Barbara Blaustein, Alex Buchmann, Upen Chakravarthy, Meichun Hsu, Rivka Ladin, Dennis McCarthy, and Arnon Rosenthal of CCA, and Michail Carey, Miron Livny, and Rajeev Janhari of the Univeristy of Wisconsin, Madison.



EVENT:	*update Quantity_on_Hand (item)*
CONDITION:	*Quantity_on_Hand (item) + Quantity_on_Order (item) < Threshold (item)*
ACTION:	*reorder (item)*

Figure 1.1 Active DBMS Example

modifying every application program that updates the database.

Active DBMSs attempt to provide both modularity and timely response. Situations, actions, and timing requirements are all specified declaratively to the system. The system now monitors the situations, triggers the corresponding actions when the situations become true, and schedules tasks to meet the timing requirements, without user or application intervention.

Active capabilities in DBMSs can be traced back to the ON conditions of CODASYL [CODA73]. Triggers were proposed for System R [ESWA75, ESWA76] as a mechanism for enforcing integrity constraints ("assertions"). The use of triggers for maintaining materialized views, snapshots, and derived attribute values, and some algorithms for implementing them have been described in [BUNE79, KOEN81, ROUS82, MORG83, BLAK86, HUDS86, LIND86, HANS87]. The term "active database" was used in [MORG83] to describe a system that supports automatic update of views and derived data as base data are updated. Simple triggers (where the triggering conditions involve only a single relation) are supported by some current commercial relational DBMSs (e.g., [DARN87]). Time triggers, where the triggering condition is a point in time (e.g., at 2:00:00 on 5/1/1988), have been described for office system applications in [ZLOO82, BARB85]. In [STON82, STON85], Stonebraker points out the utility of production (i.e., situation-action) rules as a unifying mechanism for integrity control, access control, and view processing, and for supporting inference via forward and backward chaining. Of course, AI systems have long used production rules [FORG77], actors [HEWI75], daemons, active objects [BOBR83], and procedural attachment to slots of frames [MINS75, BOBR77, KEE85] as "active" knowledge representation and inference mechanisms. However, these representations and their implementations assume small numbers of objects (rules, facts) stored in main (or virtual) memory, not in large databases on secondary storage. Also, they typically assume a single thread of execution, and hence do not provide any concurrency control over shared objects (as DBMSs do).

Recent work on active database management systems is aimed at embedding rules in a DBMS [STON85, DAYA88a,b, KOTZ88, RASC88, SELL88].*

This paper describes work in progress on HiPAC, an active, object-oriented database management system under investigation at CCA [DAYA88a,b]. Central to HiPAC is the concept of *event-condition-action (ECA) rules*, which can be used to generalize many of the DBMS functions previously implemented by special purpose mechanisms. The *event* part of an ECA rule specifies database operations, temporal events, or signals from arbitrary processes; the *condition* part specifies a database query; and the *action* part specifies a program. When the event occurs (is *signalled*), the condition is evaluated; if the condition is *satisfied* (i.e., if the query returns a non-empty answer), the action is executed (see Figure 1.1). Note that the action part may

* Note that there is also a large, separate, body of work on embedding *deductive rules* (usually expressed as Horn clause logic programs) in database systems. This work is aimed at enhancing the expressive power of the query language to include some form of recursion [BANC86], rather than at improving the timeliness of the DBMS's response to critical situations.

include database operations as well as external operations (e.g., the reorder procedure in Figure 1 may update the Quantity_On_Order of the item in question).

Effective support for ECA rules in a DBMS requires research on the following major topics:

- Knowledge Model: extend conventional data models to express ECA rules and their associated execution and timing requirements in a database,
- Execution Model: extend conventional transaction models to specify the correct interleaving of system-triggered actions in addition to user- or application-initiated transactions,
- Condition Monitoring: develop techniques for efficiently evaluating sets of dynamic, overlapping conditions,
- Scheduling: develop algorithms for scheduling tasks to satisfy concurrency and timing constraints,
- Architecture: define the functional components of an active DBMS, and their interactions with one another and with the underlying operating system,
- Performance Evaluation: construct a testbed for evaluating alternative architectures and algorithms for condition monitoring and scheduling.

Sections 3-8 of this paper will discuss the important issues in each of these areas, and the approaches under investigation in the HiPAC project. Interested readers are referred to [DAYA88b] for more details. To understand the requirements for modelling and implementing ECA rules, we examine in Section 2 the various applications of these rules in an active DBMS.

2. APPLICATIONS OF ECA RULES

Rules can be useful both to external applications and as a convenient mechanism for implementing a DBMS's functions beyond simple storage, retrieval, and update of data. Some examples of DBMS functionality that can be implemented in a unified way using rules are described in this section.

Alerters: Many application domains have the need for tracking changes in the database and taking action if some condition over the database is met. Wasteful polling by the application can be avoided if the DBMS monitors events of interest (e.g., changes to the relevant object classes or instances), evaluates the condition only when a potentially impacting change occurs, and initiates the corresponding action. This is the canonical application of ECA rules.

In addition to database operations, the events that cause rules to fire may be clock signals (e.g., the balance in all bank accounts should be checked *at 5 p.m. every day*) or any user- or application-generated signal (e.g., a failure signal from a diagnostic routine on a hardware component).

The conditions to be monitored may be complex, and may be defined not only on single data values or individual database states, but also on sets of data objects (e.g., the total of employees' salaries exceeds the departmental budget), transitions between states (e.g., the new position of the target is closer than the old position), trends and historical data (e.g., the output of the sensor increased monotonically over the last hour).

For some applications, in order to provide timely response to critical events, it is important to evaluate the condition immediately after the event, and to execute the action part immediately after the condition evaluation. In this *immediate* mode of execution, the processing of the remaining steps of the original transaction (which caused the event to occur) is suspended until the fired rule has been completely processed. Long delays can result in completing the processing of the original transaction, especially if the action part of the rule causes the cascaded firing of other rules. Response times and concurrency can be improved if the condition evaluation or action execution are *detached* from the original transaction (i.e., run in a separate transaction). For example, in a situation assessment application, the transactions that append position reports of ships into the database should be committed independently from any triggered transactions that evaluate distances between the ships and potential targets, and from transactions that cause alerts or countermeasures to be initiated. Similarly, in our inventory control example, it may be desirable to delay the reordering action to the end of the day, just in case a previous order is filled during the day and the quantity on hand goes back above the threshold.

When conditions or actions are detached, parameter bindings from the original transaction may have to be propagated to them. For example, the identifier of the item whose quantity on hand was updated must be passed to avoid checking the condition for *all* items in the inventory. Of course, if the action is detached from the condition evaluation, it may need to re-evaluate the condition for the identified item when it executes, because the state of the database may have changed between the transactions.

Whether the conditions and the actions are immediately coupled or detached, they often have to be executed within tight timing constraints, once the firing event has occurred. For example, a program trading system has to spot price differences in different markets and has to take the corresponding actions (placing buy and sell orders) under timing constraints. Considerable losses can occur if these constraints cannot be declared and the actions are executed arbitrarily late. Instead of hard deadlines (which are very difficult to satisfy in a database system), soft constraints such as value functions and relative task priorities/urgencies may be specified.

Finally, a mechanism for activating and deactivating individual rules or sets of rules is often needed. For example, once an item in the inventory has been placed on order, it may be necessary to deactivate the rule that evaluates the threshold condition, until the order is filled (lest the rule keep firing at every update, causing multiple orders to be written). The selective activation and deactivation of rules is also useful in providing a context mechanism to restrict the number of rules that must be searched. For example, once the threshold for an item has been crossed, a different set of rules might be applicable: if the item is critical (i.e., it could cause unacceptable production delays), then requests for it must be filled (until it is depleted), but all managers of projects requiring the item must be notified every day of the current quantity on hand. These rules need not be activated, however, until the threshold has been crossed. Similarly, the set of rules being evaluated while a plane is taxiing must be deactivated the moment it becomes airborne, and a different context activated.

Storage server for rule-based inferencing: Large-scale rule-based expert systems require the storage and retrieval of a vast number of rules and facts. AI systems, however, typically have an inference engine component that cycles sequentially through all the rules, and hence would be inefficient and inadequate for large applications [FORG77, WATE78]. Structuring the rules and facts around contexts is important both for performance and for improving understandability, and hence maintainability, of the knowledge base.

Additional performance improvements result from using database storage, indexing, and retrieval techniques for rules and facts, and smart query processing techniques for evaluating rules (some of these techniques are described in [SELL88, TZVI88]), instead of relying solely on the RETE structures prevalent in AI systems [FORG79].

Most expert systems cannot handle asynchronous updates: they allow the database to be updated only at the end of an inference cycle, thus restricting concurrency and delaying response to critical events. Using database concurrency control mechanisms will obviate this restriction.

Concurrency control also provides an alternative to conflict resolution policies. Because AI systems are single-threaded, conflict resolution is needed to select one rule to fire out of all the candidates that can be fired in an inference cycle. In an active DBMS, *all* firable rules can be executed concurrently and serializably; the serialization order may be influenced by specifying priorities or timing constraints.

Finally, the combination of the event, context, and concurrency control mechanisms supports a more structured problem solving paradigm than that prevalent in existing expert systems. Instead of cycling sequentially through a collection of rules, the problem solving process is initiated by signalling an event (e.g., patient arrives), and, thereafter, proceeds by firing rules as and when their events are signalled. The context mechanism allows the rule base to be structured in accordance with the phases of the problem solving strategy (e.g., diagnose ailment; prescribe treatment); switching between contexts is accomplished by activating and deactivating rules or by signalling special events. Concurrency control allows all this activity to run concurrently with other activity (including updates) over the database and knowledge base (e.g., detected new symptom, consulted different specialist's knowledge base).

Constraint management: One of the early applications of triggers was the definition and enforcement of integrity constraints. Examples are the assertions of SEQUEL/System R [ESWA75, 76] and the constraint equations of [MORG83]. ECA rules provide more flexibility than simple triggers in specifying the events that initiate constraint checking and the actions to be performed if some constraints are violated.

Some constraints need to be evaluated immediately after an update event occurs (e.g., value within range). Others need to be deferred to the end of the transaction in which the update event occurs (e.g., customer's accounts should balance after a transfer of funds from one to another). Also, when constraint evaluation is deferred to the end of a transaction, it is usually necessary to fire the integrity checking rule only once, no matter how many times the update event occurred during the transaction. This means that the mechanism for passing bindings between the event and the condition evaluation must aggregate the effects of the multiple occurrences of the update event. These options exist in most proposed constraint mechanisms (e.g., [ESWA75, 76], [MORG83], [KOTZ88], [CASA88]).

More powerful capabilities are often needed when trying to enforce consistency constraints in some "non-traditional" application environments. For example, in a CAD environment, the need has been identified for delayed (detached) evaluation of constraints (e.g., at the end of a design phase instead of after every transaction); for evaluation of constraints on explicit user request; and for specifying context-dependent actions when a constraint is violated (e.g., an update that causes some physical law to be violated must be disallowed, but an update that causes some contractual constraint may require notification to the designer and logging for future negotiations). A wish-list of rule handling capabilities in design environments can be found in

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase®
Smarter legal research.