AD-A037 437 WHARTON SCHOOL OF FINANCE AND COMMERCE PHILADELPHIA P--ETC F/G 9/2
ALERTING IN DATABASE SYSTEMS: CONCEPTS AND TECHNIQUES.(U)
JUN 76 H L MORGAN, O P BUNEMAN N00014-75-C-0462

UNCLASSIFIED 75-12-04 NL

| OF |
AD
A037437

END

DATE
FILMED
5-77

(12)
B.S.

ALERTING IN DATABASE SYSTEMS: CONCEPTS AND TECHNIQUES

O. Peter Buneman
Howard Lee Morgan

75-12-04

Department of Decision Sciences
The Wharton School
University of Pennsylvania
Philadelphia, PA 19174

December 1975
Revised June 1976

Research supported in part by the Office of Naval Research under Contract N00014-75-C-0462.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 75-12-04
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE *(and Subtitle)*: Alerting in Database Systems: Concepts and Techniques.
5. TYPE OF REPORT & PERIOD COVERED: final rept.
6. PERFORMING ORG. REPORT NUMBER: 75-12-04
7. AUTHOR(s): Howard Lee Morgan; O. Peter Buneman
8. CONTRACT OR GRANT NUMBER(s): ONR contract N00014-75-C-0462
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Decision Sciences Department, University of PA/Wharton School, Philadelphia, PA 19104
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Technical report
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Information Systems, Arlington, Virginia 22217
12. REPORT DATE: 6/76 (11 Jun 76)
13. NUMBER OF PAGES: 22 (21 pg)
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*:
15. SECURITY CLASS. *(of this report)*: Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:
16. DISTRIBUTION STATEMENT *(of this Report)*: Unlimited

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*:
18. SUPPLEMENTARY NOTES:
19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*:
Alerting
Interactive database systems

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*:
This paper discusses the subject of "alerters", which can be used in a database management system to provide the same capability of informing a user when a specified state of the world (as reflected in the database) is reached. We feel that the ability to develop and implement alerters efficiently should prove a powerful adjunct to modern database technology. This paper describes our recent research in adding alerting features to database management systems and provides a framework within which this research can be evaluated.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Alerting in Database Systems: Concepts and Techniques

O. Peter Buneman and Howard Lee Morgan

1.0 INTRODUCTION

We usually regard databases and other information systems as passive in that they "speak only when spoken to." In the real world however, information is often spontaneously offered to us by friends and colleagues who, being aware of our interests, draw our attention to changes that may affect us. In a similar vein, this paper discusses the subject of "alerters", which can be used in a database management system to provide the same capability of informing a user when a specified state of the world (as refelected in the database) is reached. We feel that the ability to develop and implement alerters efficiently should prove a powerful adjunct to modern database technology. This paper describes our recent research in adding alerting features to database management systems (DBMS) and provides a framework within which this and similar research efforts can be evaluated.

Some simple examples may help to clarify the concept. Consider the commands:

1. "Report the name of any station at which the temperature falls below 10 degrees Centigrade."

2. "Report the number and owner of any account from which more than $500 is withdrawn."

Such commands make most sense in the case of large shared databases, in which many users are entering updates at the same time that still other users may be querying the database. Rather than require a user repeatedly to enter a query of interest (e.g., 1 or 2 above), we propose that alerters be used. This changes the character of the DBMS from a passive one, where the DBMS only responds to queries, to an active one, in which the system may at any time send a message to alert the user to some condition of interest.

Before delving into our current work, it is instructive to discuss the existing and historical technologies for achieving alerting type effects.

2.0 EXISTING TECHNIQUES

To our knowledge, there is no system which permits the construction of alerters in the sense in which we have described them: as general user-defined programs which monitor online databases. However, some of the ingredients are available in various programming language and database management systems, and it is worth examining them.

1. Exception Reporting.

   Several file management systems permit the construction of programs which take some action, usually issuing a report, and which are activated whenever a record is read in during a sequential pass of the file. For example, RPG[11] and versions of COBOL[1] have a USE statement. This permits a

more or less non-procedural method of constructing file management programs. ASAP[2] allows a program segment to be treated as a "tag", which is stored with the database. These tags are then run whenever any user updates the database. The tags can use a CHANGE function to test for whether a field has been updated. An example would be:

```
DEFINE TAG WATCHACCOUNT
FOR ALL ACCOUNTS WITH CHANGE OF BALANCE='YES' AND BALANCE <0
  PRINT REPORT NEWNEGBALANCE
```

The above are really designed for batch processing systems. To use them as a means of implementing alerters would require frequent, inefficient scans of the database. Moreover, however simple it is to write these programs, they are usually the province of the applications programmer, rather than the end user.

Error handling is another special case of exception reporting. Hammer[5] has recently discussed the use of alerting type features for preventing semantic errors from creeping into databases.

2. PL/I

A useful feature of the PL/I language is the "ON" condition, which permits the programmer to process interrupts. Together with SIGNAL, which lets the programmer generate interrupts, a method for asynchronous processing is available.

Common uses for the ON condition are for debugging, e.g. for reporting when the value of a variable exceeds some bound, for the trapping of end of file and similar I/O conditions, and for handling machine faults (e.g. arithmetic overflow, division by zero).

Zelkowitz[14] and Morgan[8] have implemented versions of an extended PL/I (both based on the PL/C compiler[10]) which permit general conditions of the form:

```
ON X+7>Y-3 DO;PUT LIST X,Y;END;
```

These implementations pose an interesting version of the binding problem in block structured languages. In the above conditional, is the value of Y to be taken as the value it has at the time the condition is defined, or is it to be taken as the value it has at the time it is invoked? In accordance with the ALGOL conventions, the former is chosen, however, there are situations where the latter choice may appear more natural. We shall side-step this issue by insisting that alerters attach to referents, rather than to the names themselves.

3. PLANNER and CONNIVER

Both these languages[12] allow the programmer to set up antecedent theorems, often called "demons", which are programs that monitor additions and deletions to the "database" (an incore set of lists), and require certain actions to be taken before the addition or deletion can occur. In principle, this is quite close to what we are

proposing, but it is quite different in detail and intent.

In particular, we shall not let an alerter interfere with an update in the sense in which an antecedent theorem can. In effect, much of what we propose (and have implemented) is meant to make demons practicable and efficient for a modern database system (i.e., a large, structured, shared database requiring the use of secondary storage).

4. System R

One of the proposed features of System R[13] is the TRIGGER command. This would permit one to put semantic integrity constraits of the type Hammer has discussed into a DBMS. As far as the authors could determine, this has not been implemented, and it is not expected to be in the early versions of this experimental system.

3.0 DEFINITION OF ALERTER

Morgan[7] has presented a detailed definition of an Event Sequenced Program. Much of this work dealt with resolving conflicts among programs which simultaneously attempt to update the same variable, due to simultaneously triggered alerts. In his system[6], the operation of these programs was conceptually synchronous and, like antecedent theorems, preceded the update. For reasons which shall appear shortly, we now believe that it is best that alerters operate

asynchronously. Also, we propose a different method of coping with the conflicts.

We simply define an alerter as a condition, program pair. To start with, we may consider the condition to be any predicate on the state of the database, and the program to be any general program. Unfortunately, this is both unworkable and inadequate: by making the condition "true," we can specify that the program run continuously. While this may be a pleasing theoretical contruct, it is practically abhorrent. The definition is inadequate in the sense that we may wish to include in the predicate items which are not in the database, e.g., the current time of day. Moreover, we are usually more interested in monitoring changes to a database than its instantaneous state.

The two examples in the introduction show this. In the first case the request is that some response is triggered at the moment "temperature is less than 20 degrees Centigrade" becomes true. In the second, we are asking that the magnitude of change of some value be monitored. In constructing the condition part of the alerter it must be possible to refer to the state of the database both before and after the proposed change which that condition is monitoring. It may also be helpful to have information about both states of the database passed through to the program that triggers is activated by the condition.

The complexity of the alerter predicates is the crucial determinant to the utility of alerting, and is directly related to the spcifcation of the underlying database model. We have distinguished three levels of alerting conditions, described below.

3.1 Simple Alerting

Alerts which can be monitored by dealing with only a single record class (or a single relation in a relational database) are relatively simple to handle. The following predicates are representative of simple alerting.

1. item. Monitor one or more fields of a given record, e.g., temperature in New York.

2. record. Monitor a given record, e.g., any change to the New York City record.

3. Monitor one or more fields in an arbitrary record in a class, e.g., monitor the temperature at any station.

4. Addition of record to a class, e.g., monitor new stations added.

5. Deletion of a record from a class.

6. Any field in any record, e.g. tell me if anyone changes anything in the weather station records.

We should note that one can substitute "tuple" for record, "relation" for record class, and "domain" for field in the above discussion, without loss of meaning. We also note that the primary keys are usually not permitted to change. What for example, would it mean if a user requested an alert on the temperature in New York City, and the key "New York City" were changed?

3.2 Structural Alerting.

These alerter predicates indicate changes in the dynamic structure of the database. They will be explained in terms of DBTG terminology, although they are equally valid for relational databases, provided we regard relations as dynamic objects.

1. Change in ownership or membership of sets. E.g. a doctor acquires a new patient.

2. Change in aggregate properties of sets. E.g., a doctor gets more than 20 new patients.

3. Change in fields with different record classes. E.g., a doctor gets a patient with chicken pox (either by the doctor acquiring the patient, or an existing patient acquiring chicken pox).

These alerters are more complicated to implement since they may involve interactions between more than one record class.

3.3 Complex Alerting

We include here predicates which we would like to be able to handle, but which require still more global views of the database system and its user interactions.

1. Statistical alerters. E.g., let me know when the average temperature in the Northeast drops by more than 10 degrees. Clearly, one can come up with efficient methods for handling simple statistical functions (with averages by maintaining a special counter which is updated appropriately). But the more difficult the function, the less likely that such a routine exists, and the more likely that a scan of the database may be required.

2. Transaction spanning alerters. E.g., let me know when a bank balance drops by more than 10,000 in any 24 hour period. This requires a constant rolling history for 24 hours.

3. Pattern recognition. Monitoring for the occurence of specific patterns in the database. This is closest to CONNIVER's original intentions for using demons. Again, efficient implementation requires a lot of problem specific knowledge.

4. Time based alerting. This is similar to tranasction spanning, in that one often does not require instant alerting, but rather alerting that doesn't come too late. For example, one may wish to be informed within one week if a particular check is cashed.

4.0 DATABASES VIEWED AS MESSAGE PASSING SYSTEMS

It is usual for a data managment system to provide the programmer with a set of subroutines which form the interface between his program and secondary storage. Because sharability of databases is central to our concept of alerting, we need to view the data managment system in a different way. The database itself consists of a separately running job which serves many users and communicates with them by sending and receiving messages. Similarly, any user wishing to interrogate, or to modify, the database, does so by sending it the appropriate message.

The distinction between message passing and subroutine calling may seem unnecesary, especially since messages will certainly be formatted and sent by special-purpose subroutines. However, the distinction is important at two levels. The programmer must be aware that communication with the database is not instantaneous and that, as far as possible, his programs should be designed to work asynchronously. At a conceptual level, the environment in which messages are processed is subject to external influences; whereas subroutines are usually thought to operate in a more predictable environment, and to have a degree of control over the data which is considerably greater than in a message passing system. It is precisely because the environment is subject to unpredictable changes that alerting is a useful concept. In addition, the software to handle message passing in computer networks is simpler than that required to set up a virtual (synchronous) channel. Hence, this approach may have advantages in a distributed processing system.

As an example, the LDEMON system which we shall shortly describe, incorporates a miniature operating system. This is capable of servicing several users in addition to the database manager who has privileged control over the system. For each user, a two-way communication system is set up in which messages are queued in sharable files. One such user is a decision aiding system known as DAISY[9] which makes heavy use of alerting. Any transaction with the database -- queries, updates and setting alerts -- is passed to LDEMON through these files. DAISY also periodically reads its mail from LDEMON which may contain the response to a query, an alert message, or an error message indicating that a previous communication from DAISY was incomprehensible. In the second or two between sending a message to LDEMON and receiving the answer, a DAISY user can, and usually does, continue to work on some other aspect of his decision making. In short, neither system is in control; neither can, in normal operation, disrupt the other; but LDEMON at least is obliged by contract to answer messages sent to it.

We expect that with the growth of computer networks and distributed databases, this message passing philosophy will become more prevalent. To use it properly will call for the further development of high level languages for communicating with databases. In order to scan a file sequentially for a record with given properties, it is impracticable to send and receive a message for each record. Rather, the specifications for this search should be transmitted as one message, and the message (as already happens in LDEMON) should then be expanded by the database into the appropriate program.

5.0 THE LDEMON SYSTEM

Over the past year, the authors, together with Stan Cohen, have implemented a system which supports simple alerting and the type of message passing facilities just described. LDEMON is written in LISP and will accept such messages as:

```
ALERT "Frost warning" (STATION)
  IFJUST TEMPERATURE <10
    THEN SEND ("Frost warning for ",NAME)
  END
```

The function of this alerter should be obvious once it is understood that STATION refers to a class of records and that TEMPERATURE and NAME are fields of that class.

The basis of LDEMON is a set of programs for constructing and manipulating records. Record classes may be dynamically defined, and the LISP functions for accessing their fields are automatically constructed. In addition, procedures are provided for adding, deleting and modifying records. This system is consistent with a relational model and a (not especially efficient) set of programs have been added to the system which provide for the usual relational constructs.

LDEMON allows a special class of programs, called demons. These may contain conditionals of the form IFJUST <condition> THEN <actions>; meaning that <actions> are to be performed whenever <condition> has just become true. Changes which cause a particular state of the database to occur can therefore be monitored. Often, it is more desirable to monitor for some property of the change itself, rather

than for the resulting state of the database. A demon's condtion may also contain references to the state of the database which just preceded a change. Thus,

IF (OLD BALANCE - BALANCE) > 500 THEN . . .

can be used to monitor withdrawals from an account. These conditions may be predicates in a combination of fields from some record class and may be used to effect all forms of simple alerting described above. In particular,

IFJUST (BALANCE < 100) AND (ACCOUNT# = 1234) THEN . . .

has the effect of monitoring a given field of a given record.

The distinction between alerters and demons lies in what kinds of action they are allowed to take. For a demon, the action part may be anything including a program which may itself modify the database. For reasons given below, this is unacceptable. Moreover, with an incorrect program, a demon may "hang" the database indefinitely. The action of an alerter is limited to sending messages in various formats. Let us make clear again that a user sends messages to LDEMON which operates as a separate job. In addition to alerters, a user may send messages to LDEMON which interrogate or modify the database. All these messages are translated into suitable LISP programs: in particular, an alerter message is checked by LDEMON for consistency with the database and is then translated into a demon which is added to the system. This means that we can no longer look upon the whole process as being synchronous: alerter messages are effectively issued after the update has taken place. Most importantly, we cannot use an alerter in LDEMON for preventing an update.

The fact that LDEMON is limited to sending and receiving messages in restricted formats does not mean that these messages are just displayed at the user's terminal. From a suitable version of the DAISY system mentioned above, it is possible to send LDEMON a message such as

```
ALERT "overdrawn" (ACCOUNT)
    IFJUST BALANCE < 0 THEN
        RUN accountcheck(ACCOUNT#)
    END
```

The program "accountcheck" resides in the DAISY system and not in LDEMON; and the instruction RUN is interpreted by LDEMON as an intruction to send back a message in a form which will cause the execution of this program with the appropriate arguments.

One of the most important aspects of alerting is efficiency. LDEMON operates by monitoring transactions with the database rather than by any form or search. This means that the overhead for keeping an alerter in the system depends on the frequency of transactions with the database but is independent of its size. The following is an informal description of what happens in LDEMON when an alerter is added to the system and subsequently activated.

5.1 Adding An Alerter To The System

1. On receiving an alerter message, the alerter is "stamped" with the identification of the user who sent it. It is then checked for consistency with the database "schema". That is, field names must be known to the system, and related to the

proper record types. Any inconsistency causes an error message to be returned.

2. The action portion is decoded, and user specific commands (such as SEND) are checked. At this time, in the LISP based system, a new function is constructed which will actually send the appropriate response if the alerter is triggered and the condition is met.

3. The alerter is indexed under the (*field, record class*) pair for each field name which appears in the condition.

4. At this point, a "ALERT SET" message is sent back to the sending user.

5.2 Triggering An Alerter

1. When a modifying update is to be performed, a copy of the old values are made in a temporary area.

2. The update is completed and a copy of the updated values are also placed in temporary store.

3. Either by comparing the two records in temporary store, or by examination of the updating transaction, a list of affected (changed) fields is constructed.

4. Any alerter indexed under the appropriate record class and field name is activated; i.e., its condition is evaluated with respect to the records in temporary storage and, if satisfied, the function which sends the appropriate messages is invoked.

6.0 CONFLICTS

Much of Morgan's work[6] was dedicated to resolving conflicts among the "event sequenced programs", which, in the present terminlogy would be the actions. Since alerters cannot interfere with updates, and can only have the "side effect" of sending a message, there is no conflict problem as far as the database system is concerned.

At a more global level, however, we may still run into the conflict. For example, suppose that two people independently receive an alert indicating that an account is not properly balanced. One tries to fix this with a credit entry, and the other with a debit, causing it to again go out of balance. As a result, they each get called again and . . .

We propose that in any shared database, a user sending an update has some implicit knowledge about the items being updated. If that knowledge does not correspond with the values actually in the database, the update *should not be permitted*. In other words, rather than locking and unlocking the record (or perhaps a large number of records for complex alerting), we permit the user to send a

conditional update which is effective only if the record to be updated is the same as when the user last inspected it. (similar to one of Dijkstra's "guarded commands"). If the condition is not satisfied, the update is rejected, and the user must reread the record.

System R[13] permits the users to define transactions, which involve essentially a stream of commands, all of which must be executed if any are to be. This is the type of feature which we are proposing to use for resolving "conflicts" of this nature.

7.0 WAND

In addition to alerting within the relational context, as described above, we have been implementing these concepts in the DBTG framework. WAND (Wharton Alerting Network Database)[3] was originally designed as a research vehicle for testing concepts which could be applied to DBTG databases. In addition to the standard DBTG features, there is an interactive "navigation" aid, and plans for a dynamic restructuring capability[4]. For alerting, it will behave in the same fashion as LDEMON.

The modifications are to the MODIFY modules of the system, and the STORE functions. R. Cortes, along with one of us(DPB) is currently implementing both simple and structural alerting features in this system.

8.0 CONCLUSIONS

We have shown that a reasonably large class of alerters (simple and structural) can be implemented, with a computing time overhead which is independent of the number of records in a large database. The extra time required to handle alerting along with normal database updating is dependent only on the number and complexity of the alerter conditions themselves.

An experimental system with these features has been implemented in LISP and running for over a year. A similar system, working with DBTG databases, is to be completed later this summer. Both of these are in use and the concept of alerting is being tested.

We feel that alerting features are a natural extension to database technology, and that they should be planned for in any major database implementations.

9.0 REFERENCES

1. COBOL Language Reference Manual. IBM Corporation.

2. Conway, R., W. Maxwell and H. Morgan. "A Technique for File Surveillance." Proceedings of IFIP Congress 74. North-Holland Publishing Company.

3. Gerritsen, R., J. Ribiero and R. Cortes. WAND User's Guide. Working Paper No. 76-01-03. Department of Decision Sciences. The Wharton School.

4. Gerritsen, R. and H. Morgan. "Dynamic Restructuring of Databases using Generation Data Structures." Proceedings of ACM '76. ACM.

5. Hammer, M. "Error Detection in Database Systems." Proceedings of the 1976 National Computer Conference. AFIPS Press. (June 1976).

6. Morgan, H. "An Interrupt Based Organization for Management Information Systems." Comm. ACM 13, 12 (December 1970).

7. Morgan, H. L., "Event sequenced programming." Technical Report No. 119. Dept. of Operations Research. Cornell University. Ithaca, NY.

8. Morgan, H. L., "A generalized interrupt processor for PL/1." Internal memorandum. Caltech. 1972.

9. Morgan, H. L. "DAISY: An Applications Perspective." Proceedings of the Wharton/ONR Conference on Decision Support Systems. (to appear 1976).

10. Morgan, H. L. and R. Wagner. "The Design of a High Performance Compiler for PL/1." Proceedings of the 1971 Spring Joint Computer Conference. AFIPS Press.

11. RPG Language Specification. IBM corporation C24 333700.

12. Sussman, G. and McDermott, D. "Why Conniving is Better than Planning." MIT Artificial Intelligence memo 255A (April 1972).

13. Astrahan, M.M. et al. "System R: Relational Approach to Database Managment" ACM Transactions on Database systems. 1, 2 (June 1976).

14. Zelkowitz, Marvin. "Reversible Execution as a Program Debugging Tool." Ph.D. Thesis. Cornell University. 1972.

DISTRIBUTION LIST

Department of the Navy - Office of Naval Research

Data Base Management Systems Project

Defense Documentation Center (12)
Cameron Station
Alexandria, VA 22314

Office of Naval Research (2)
Information Systems Program
Code 437
Arlington, VA 22217

Office of Naval Research (6)
Arlington, VA 22217

Office of Naval Research
Code 102IP Branch Office, Boston
495 Summer Street
Boston, MA 02210

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, Illinois 60605

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106

New York Area Office
715 Broadway - 5th Floor
New York, NY 10003

Naval Research Laboratory (6)
Technical Information Division
Code 2627
Washington, DC 20375

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, DC 20380

Office of Naval Research
Code 455
Arlington, VA 22217

Office of Naval Research
Code 458
Arlington, VA 22217

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner
Naval Ship Research and
Development Center
Computation & Mathematics Dept.
Bethesda, MD 20084

Captain Grace M. Hopper
NAICOM/MIS Planning Branch
(OP-916D)
Office of Chief of Naval Operations
Washington, DC 20350

Mr. Kim B. Thompson
Technical Director
Information Systems Division
(OP-911G)
Office of Chief of Naval Operations
Washington, DC 20350

Bureau of Library and
Information Science Research
Rutgers - The State University
189 College Avenue
New Brunswick, NJ 08903
Attn: Dr. Henry Voos

Professor Omar Wing
Columbia University
Dept of Electrical Engineering
and Computer Science
New York, NY 10027