The problem of lock by value in large data bases

G. Schlageter

Universität Karlsruhe, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, D 75 Karlsruhe, Postfach 6380, Germany

In parallel-accessed database systems a process must be able to lock a subset of the data. In current systems and proposals the only way for locking a set of data is either to lock explicitly each element in the set or to lock a large predefined set, such as a whole file. Though the necessity of a more flexible locking feature is recognised there are no results as to conceptual foundation and implementation. In this paper a concept for lock by value is developed which allows a process to lock a subset of the database just by specifying a condition describing this set. Though the concept looks rather simple some unpleasant problems have to be discussed. A locking system supporting current lock features at the record-level as well as lock by value is proposed.

(Received June 1975)

1. Introduction

To guarantee database integrity and to prevent malfunction of processes in parallel-accessed database systems a process must be able to keep a specified part of the database 'frozen' until it has finished its operation. This problem of access synchronisation is only partially solved in current systems and proposals. Existing solutions are based on *identity lock*, which means that an identified data element *b*, usually a record, is locked by an explicit lock command referring to *b*. As a result of the lock command lock information is stored in the system. Mechanisms of this type have been discussed (Bernstein and Shoshani, 1969; King and Collmeyer, 1973; Schlageter, 1975).

As can easily be seen synchronisation schemes of this type are often too elementary, and more user-oriented features should be available. According to its needs, a process should be able to lock a subset of the database or a 'critical region' without having to know (and to address) each element in this subset. Implementation of these ideas is not straightforward. Principally, it could be based on a lockout mechanism for identity lock, though this could be extremely inefficient for large sets of data elements to be locked. In these cases, for example, a 'lock by value' or 'content lock' feature would be valuable, as will be shown in this paper.

The term '*lock by value*' must be understood from the user's point of view. In contrast to identity lock, lock by value enables the user to lock a set of records just by specifying a condition for the values in particular fields of the records; he is no longer forced to identify and lock each element of the set explicitly. For instance, we would like to lock all records of employees with a special job-code, thereby allowing all other employee records to be accessed without restriction. Lock by value introduces a general form of data dependency of the lock decision. If the record identifier is also considered a data value of the record, identity lock can be seen as a special case of lock by value.

The problem of lock by value has not been discussed in detail, and to the author no suggestion for a solution is known. This paper presents a more general view of the lock problem. A conceptual solution for lock by value is developed, and some special difficulties are discussed.

2. A more general concept of locking in database systems

In available database management systems (DBMS) there are only limited means for a process to request a part of the database for protected or exclusive use. At best we have identity lock on the record level and conventional file protection mechanisms. Lock by value features are not implemented.

Of course, lock by value could be solved with identity lock, simply by an exhaustive search: each record satisfying the given condition is locked, i.e. an entry for that record is created in an allocation table. Obviously, this solution is unsatisfactory; in most cases it would be preferable to lock a whole file, or, what is logically the same, to lock the record type instead of the instances (cf. IBM, 1973).

In the following a more general concept of locking is proposed which includes the current static view (stored lock-information in some way associated with the data) as well as a dynamic or procedural view. The essential point is that in the procedural model of locking we do not have lock information looked up in some table, but evaluation rules applied to the data to determine the lock-status. Thus, in contrast to the classical view of locking, it is not necessary to lock each element of a specified set explicitly before a process can be sure that no other process will change this set.

Take the example of an employee file: a process *P* may need all records of a certain department unchanged during its action. Ideally it would issue a command like 'lock all records with deptcode = sale'. If after that a process *Q* tries to access an employee record, the condition 'deptcode = sale' is evaluated for that record, and *Q* is allowed or not allowed access depending on the result of the evaluation.

Though the static case could be treated as a special case of the procedural concept, one important difference must be noticed: in the static concept the lifetime of a lock of a record depends only on the lock and unlock commands of the processes, whereas the lifetime in the procedural concept depends on the lifetime of the evaluation rules *and* on the state changes of the stored information in the record. As a consequence we have the following interesting property: in current systems and concepts once a set of data is locked it is unaffected by all changes of the database such as addition of new data; in the procedural concept the definition of a set is dynamic, i.e. a record which is added to the database automatically becomes a member of the locked set, if, according to its attribute values, it should belong to this set.

At first sight there is some similarity to data dependent check in security control (Conway *et al.*, 1972; Hoffmann, 1970; Tsichritzis, 1973). Consider again the example of the employee file: to restrict a particular type of user from ever seeing values of salary in excess of $5,000 the data records have to be checked before being passed to the user program for processing. The necessary check-routines are defined in the database schema, together with the definition of the user types.

Though the fundamental necessity of data dependent checks is common to both security control and lock by value, the underlying problems are different by their nature; locking of data to control simultaneous updating is an entirely separate function from security control. The details will become clear in the following discussion of the concept of lock by value.

3. A functional model of a locking system

We consider a database that comprises records of various types. A record is composed of a number of fields containing attribute values; an internal unique identifier is associated to each record. For this discussion the identifier can be regarded as an attribute, too. A record type is described by $r\langle a_1, \ldots, a_n\rangle$, where r is the name of the record type and a_i is an attribute name. Within larger contexts an attribute name must be qualified: $r.a_i$.

We informally introduce the operations

lock($N|C$): 'lock all records satisfying selection criterion C' N is a name for C.

unlock(N): 'unlock the set of records specified by the selection criterion named N'

We consider *selection criterions* of any form, but which involve only attributes of one record type, such that the truth value of the criterion can be evaluated for each record just by inspecting the attribute values of this record. (Some of the record's attributes may be defined as virtual data). We call this type of selection criterion '*elementary boolean condition*' (EBC).

A set that comprises records of various types must be describable as a disjunction of EBCs, e.g.

$$r_i.a_j < 10 \vee r_k.a_l > 70 \wedge r_k.a_m = 20 \ .$$

A criterion of this form must be split up into the EBCs, each of which can be treated separately. We choose a factored form for EBCs on the user's level, such that the above example is formulated as

$$\mathbf{lock}(N|r_i{:}a_j < 10);$$
$$\mathbf{lock}(M|r_k{:}a_l > 70 \wedge a_m = 20) \ .$$

By declaring C in **lock**($N|C$) to be a list of EBCs we can make sure that all elements of a set of EBCs are established at a time, or no one.

We define $C(r)$ to be the truth value of the EBC C when applied to the attribute values of record r, and $S(C)$ to be the set of records r specified by $C{:}S(C) = \{r|C(r) = \mathbf{true}\}$.

To support reading as well as writing processes, read lock and write lock should be introduced; for the purposes of the lock by value discussion we only consider write lock, but the generalisation is straight-forward.

We now consider a *locking system* as part of the DBMS through which all access to data must pass, unless the process does not want any protection and only wants to read data. As a first rough scheme we assume that before delivering data to a process P the locking system checks whether the data satisfy an EBC established by P, and whether they do not satisfy an EBC of another process. If access to data must be denied, the requesting process is blocked, and deadlock analysis is started. As will be seen the function of the locking system must be discussed carefully to avoid pitfalls for database integrity.

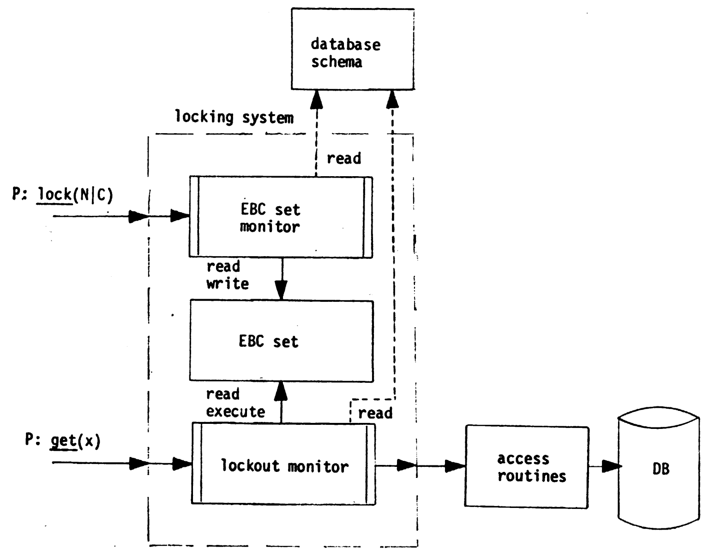

Fig. 1 Organisation of the locking system

Fig. 1 shows the principal organisation of the locking system. Central part is the *EBC set*. The *EBC set monitor* is responsible for inserting and deleting EBCs in the EBC set. The *lockout monitor* takes care of the appropriate application of the EBCs to the data at every access. The two monitors must be synchronised as to EBC set access, e.g. by defining them as the local procedures of one monitor as defined by Hoare (1974).

With each EBC in the EBC set the identification of the owner process which established the EBC must be associated. It should be pointed out that an EBC can be activated at run time only, according to the **lock**-operation, i.e. the passing of the EBC from the process to the EBC set monitor principally takes place at run time. There are various ways for implementing this passing of an EBC, among other aspects depending on the binding time rules in the DBMS.

We thus have the following interpretation of the introduced operations:

process P:
⋮
lock($N|C$) 'insert EBC C named N with owner process P into the EBC set'
⋮
unlock(N) 'delete EBC with owner process P and name N in the EBC set'.
⋮

4. The problem of overlapping conditions

In the given rough functional scheme of the locking system no restrictions were introduced for the EBC set. Yet, there is one unpleasant problem: process-interference because of overlapping EBCs. The protected sets specified by different processes need not be disjointed; for example: process A locks all employees with a special job-code, process B all employees with a special department-code. We therefore have the following problem: when B establishes its condition C_B, A may already have accessed a record $x \in S(C_B)$.B is not aware of this. A can produce changes in $S(C_B)$, whereas B assumes its set to be frozen. Note however that simultaneous access of two processes to a record satisfying two EBCs is not possible. Though the risk of malfunction or integrity loss due to 'initial overlap' is not great, it will not be acceptable in general.

Before looking for reasonable solutions we state the following:

Lemma 1:
Correct operation of two concurrent processes A and B with the EBCs C_A respectively C_B is guaranteed, if no record $r \in S(C_A) \cap S(C_B)$ is modified by the processes.

Observation 1:
To have an EBC-specified set frozen for a process it is not necessary that this set be pairwise disjointed with each other EBC-specified set.

Observation 2:
Lemma 1 is not a necessary condition, though it seems to be the most stringent condition that can be given on this abstract level.

Suppose we could guarantee, by some yet unknown mechanism, that an EBC X is established only if no record $r \in S(X)$ is already being modified by another process; then a variety of EBCs overlapping in many ways could exist in the EBC set. The function of the lockout monitor would be the following: Process A with EBC C_A requests access to record x;

lockout monitor action: 1. $x \in S(C_A)$?
2. $x \notin S(C_P)$?, for all EBCs $C_P, P \neq A$.

Unfortunately it seems that the implementation of the assumed mechanism would result in an exorbitant overhead. We have to develop less powerful but reasonably implementable solutions:

Solution 1:
For each record type all EBCs must have the same owner process.

Solution 2:
An EBC is established only if it does not introduce the possibility of overlap. Principally, two EBCs U and V for record type r do not overlap, if $U(r') \wedge V(r') = \textbf{false}$ for all possible instances r' of r. If this cannot be shown, the establishment of the EBC must be deferred, i.e. the process must be blocked.

Since the EBCs cannot overlap the function of the lockout monitor is simplified:

Process A with EBC C_A requests access to record x;
lockout monitor action: $x \in S(C_A)$?

On the other hand the function of the EBC set monitor becomes more complicated; on every **lock**($N|C$) it has to test for possible overlap, and if there is the possibility of overlap, it has to block the process and to initiate deadlock analysis.

5. Lock by value and identity lock

Processes often cannot a-priori specify their 'critical sets' by an EBC, or, if they did, the locked set would be unnecessarily large. These processes must be allowed to apply identity lock.

Conceptually this situation need not be treated separately, because at least after a record is found an appropriate EBC can be formulated. Yet, for efficiency reasons it is preferable to treat identity lock separately in a conventional way (Bernstein and Shoshani, 1969; King and Collmeyer, 1973; Schlageter, 1975), though, of course, also under the control of the locking system. The lockout monitor now has the following functions:

1. Process requesting identity lock for record x;
   lockout monitor action:
   1. x not in allocation table?
   2. $x \notin S(C)$? for all EBCs C of the specified record type.
2. Process A, with EBC C_A, requesting access to record x;
   lockout monitor action: $x \in S(C_A)$?

Under this scheme we again have possible initial overlap: on establishing an EBC we do not know whether records of the specified type are already locked by identity lock. This can be solved simply by a counter for each record type indicating the number of records locked by identity lock.

In this model blocking of processes can occur for the following reasons:

(*a*) a process wants an EBC to be established, but the EBC monitor refuses for overlapping reasons,
(*b*) a process wants to lock a record by identity lock which either is already locked by another process or satisfies another process's EBC.

In all these cases the process (or processes) responsible for the blocking is known, and an arc can be created in the process graph which is maintained for deadlock detection. The problem of deadlock does not reveal really new aspects in this model.

6. An example: Lock by value and classifications

Objects of the same type are often classified according to some characteristics, and a name is associated to each class. In various systems a class can be regarded as the lock-unit of the next level below the file. Classifications are explicitly defined in CODASYL DBTG (1971) by sets: the member records of a set occurrence form a class which is named by the owner record of the set occurrence. In the relational model classifications are not explicitly modelled.

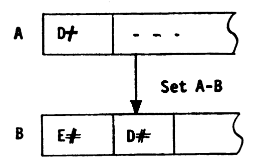

Fig. 2 Example of a CODASYL set

In *CODASYL-based systems* a process may want to lock a whole set occurrence without being forced to identify and lock each member record. This could be done by lock by value. We assume that record type A, the department record, is the owner of a set A-B, and record B, the employee record, is the member of this set, as shown in **Fig. 2.**

The process executes

$$\textbf{lock}(A);$$
$$a := A.D\#;$$
$$\textbf{lock}(N|B{:}D\# = a);$$

Clearly, **lock**(A) is an identity lock referring to the current of A, while **lock**($N|B{:}D\# = a$) establishes a condition in the EBC set, such that each B-record which is a member of the locked department record is inaccessible for any other process. $D\#$ need not be physically contained in the B-record, though this is the most efficient solution; $D\#$ might be defined as a virtual item such that on every access to a B-record $D\#$ is taken from the corresponding owner record A.

In an extension of the CODASYL proposal the above problem of locking a set occurrence could be formulated in the following way (Schlageter, 1974):

declaration division of user program:

cr name is *KB*; **contains current of** *A*, **members of** *A* **in set** *A-B*.
⋮
procedure division:
⋮
$N = \textbf{lock}(KB)$.
⋮

In the declaration division the type of a critical region KB is defined. By issuing $N = \textbf{lock}(KB)$ an instance of the critical region type KB is created and named N; the current record of type A and its members of type B in set A-B are locked. The user need not have any knowledge about the size or other details of the locked portion of the database.

In the *relational model* (Codd, 1970) classifications are considered another attribute of the objects. In the relation

$$\text{REL}\,\langle E\#, \ldots, D\#\rangle$$

the domain $D\#$ identifies the different classes. Tuples with the same value for $D\#$ are in the same class. In order to lock a special class, lock by value would be appropriate. However, it is an open question whether lock operators of the discussed form are desirable features of end user languages, and to which degree automatic lock can substitute them without losing too much efficiency.

7. Conclusion

Although the concept of lock by value is a valuable tool for access synchronisation in databases it turns out to be rather difficult to implement. A functional model for lock by value was discussed, and it was shown that a reasonably efficient implementation should be possible with a restricted concept. Finally a locking system was outlined that includes conventional locking features based on identity lock as well as capabilities of lock by value. The problem of deadlock detection does not become more difficult than in systems with identity lock.

References

Bernstein, A. J., and Shoshani, A. (1969). Synchronization in a Parallel Accessed Data Base, *CACM*, Vol. 12, p. 604.

CODASYL (1971). Data Base Task Group Report, April.

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks, *CACM*, Vol. 13, p. 377.

Conway, R. W., Maxwell, W. L., and Morgan, H. L. (1972). On the Implementation of Security Measures in Information Systems, *CACM*, Vol. 15, p. 211.

Hoare, C. A. R. (1974). Monitors: An Operating System Structuring Concept, *CACM*, Vol. 17, p. 549.

Hoffmann, L. (1970). *The Formulary Model for Access Control and Privacy in Computer Systems*, SLAC Report No. 117, Stanford University, May.

Information Management System/360 (IMS), Version 2, IBM-Form GH 20-0765-4 (System Description).

King, P. F., and Collmeyer, A. J. (1973). Database Sharing—an Efficient Mechanism for Supporting Concurrent Processes, *AFIPS NCC* 1973, p. 271.

Schlageter, G. (1974). A Concept for Supporting Concurrent Processes in Database Systems (in German). *Forschungsbericht 24 des Instituts für Angewandte Informatik und Formale Beschreibungsverfahren*, (H. A. Maurer, and W. Stucky, eds.), Universität Karlsruhe, Dec.

Schlageter, G. (1975). Access Synchronization and Deadlock-Analysis in Database Systems: An Implementation-Oriented Approach *Information Systems*. Vol. 1, p. 97.

Tsichritzis, D. (1973). Reliability, In: *Advanced Course on Software Engineering* (F. L. Bauer, ed.), Lecture Notes in Economics and Mathematical Systems 81, Springer-Verlag, Berlin—Heidelberg—New York.

Book review

Data Structures by M. Elson, 1975; 307 pages. (*Science Research Associates Limited, Global Book Resources Limited* £7·65)

This posthumous publication is a tutorial treatment of the material commonly covered under this heading in computer science courses, and which, as the discursive bibliography acknowledges, is covered in a rather more formal manner in several other texts. It begins with a preface to the instructor which, though explaining the reasoning behind many of the techniques of presentation adopted, says little to characterise the student at whom they are aimed beyond making the assumption that 'his computer science and mathematics backgrounds may be less than ideal'. I would recommend this book to computer science undergraduates, but it is too academic for those on non-degree-level courses, and the pace is too slow for postgraduates.

The data structures and algorithms presented in the bulk of the text are not in any of the standard languages but in an informal notation having (for the initiated) strong similarities with certain features (arguably some of the less attractive ones, too!) of PL/I and JOSS. Later in the book, however, LISP and SNOBOL 4 are introduced, but their use is confined to their respective chapters, 4 and 7, which largely reproduce material from the author's *Concepts of Programming Languages*, (Chicago: Science Research Associates, 1973). The LISP is LISP 1.5, mainly in the form of M-expressions; but while CONS[CAR[x];CDR[x]] may be an improvement on the S-expression (CONS(CAR X)(CDR X)) the McCarthy conditional $[p_1 \rightarrow e_1; p_2 \rightarrow e_2]$ is hardly an improvement on (COND(P1 E1) (P2 E2)). Personally I would have preferred respectively, cons(head(x), tail(x)) and **if** p_1 **then** e_1 **else if** p_2 **then** e_2. The chapter on SNOBOL 4 is both adequately comprehensive and admirably concise, but compared with that on LISP, it sheds little light upon the underlying data structures which are, after all, the main topic of the book. This is of course one of the strong points of SNOBOL 4!

The text is liberally sprinkled with exercises, and answers to a selected subset of these are appended. The index is adequate but the table of contents would have been more useful had it included at least the main subheadings, especially in view of the enigmatic titles of the first three chapters.

The order of presentation of the material follows the philosophy that 'complexity is a function not of the number of parts, but of the number of moving parts'. Thus Chapter 1, Static structures, covers vectors, arrays (including cross-sections of arrays and triangular arrays), records (structures to the PL/I programmer) and arrays of records. Chapter 2, Semistatic structures, covers self-describing records (c.f. COBOL's OCCURS DEPENDING), array variability (truncating, extending, deleting and adding dimensions), stacks, queues and (only in their non-restricted form) deques. Chapter 3, Dynamic structures, covers linked lists (including circularly linked lists and those with shared elements), general trees, binary trees, graphs (superficially), and plexes. Expression evaluation, decision trees and critical paths are used as applications.

Chapter 5, Storage management, treats an area which is sometimes taken for granted in a course of this sort. The analogy between the worst-fit storage allocation strategy (which tends to make all free blocks the same size) and a policy of taxing the rich until they become the poor is typical of the more endearing features of the authors' approach in the book as a whole. Chapter 6, Strings, contrasts the vector approach with the list approach and suggests physical implementations which look particularly appropriate to an IBM-360-type architecture. Chapter 8, Data sorting, includes the standard techniques of binary search, selection sort, bubble sort, insertion sort, tree sort, tournament sort, heap sort, quicksort, distribution sort and sorting by merging. Finally Chapter 9, Data searching, covers linear searching, searching sorted data, and (in considerable detail) hash addressing techniques.

D. J. Cairns (London)