On the Integrity of Data Bases and Resource Locking

Rudolf Bayer, Technische Universität München

Abstract

The problem of providing operational integrity of data bases as opposed to operating systems is discussed. Techniques of resource locking, mainly individual object locking and predicate locking, are surveyed, improved, and unified. An efficient on-line transitive closure algorithm for deadlock discovery is presented and analyzed. Several strategies for preventing indefinite delay of transactions are proposed. Phantoms and the need for predicate locking are surveyed and reconsidered. Several strategies for handling phantoms are proposed: one without predicate locking and two in which predicate locking is needed for writing transactions only, and in which individual object locking sufficies for pure readers.

I. INTRODUCTION

Providing data base integrity means to guarantee the correctness of the data (more precisely their accuracy, consistency, and timeliness) through

1) the proper operation of the hardware,
2) the proper operation of the software, as well as
3) the proper use of the system.

This paper only covers part of the software aspect of integrity. The problem of guarding data bases against hardware failures has been covered extensively by M.V. Wilkes [Wil 72]. Proper use of the system is mainly concerned with quality control in data acquisition and with prevention of accidental or mieschievous misuse, i.e. with the security of computer systems.

As opposed to many other computing environments, data bases give rise to especially high integrity requirements for at least the following reasons:

1) Longevity: Even rare errors will in the long run lead to a certain contamination and degradation of the quality of a data base. Completely purging erroneous data and all their consequences from a data base is difficult.
2) Limited repeatability: Even if data or processing errors are discovered, it may be impossible or useless to rectify the situation due to time constraints, unavailability of the correct source data, unavailability of a correct system state preceding the fault.
3) The need for immediate and permanent availability: This prevents a practice often used elsewhere, namely running a program and then checking by careful inspection and analysis whether the result is or at least "looks" right, correcting and rerunning the program otherwise.
4) Multiaccess: Data bases are manipulated by many users with probably quite different quality standards. It is infeasible to completely entrust the quality control to these users and difficult to track the source and the proliteration of errors.

II. SEMANTIC AND OPERATIONAL INTEGRITY

We wish to distinguish between semantic and operational integrity of data bases:

By semantic integrity we mean the compliance of the data base contents with constraints derived from our knowledge about the meaning of the data. Semantic integrity might be enforced by allowing on certain data only a limited set of precisely specified meaningful operations, by adopting a set of programming and interaction conventions, by dynamically checking the results of updates, or by proving for each program manipulating the data base, that the semantic integrity constraints are satisfied.

Little is known about how to describe, to enforce, and to implement such semantic integrity constraints. Still we believe, that semantic integrity is of a much more basic nature than operational integrity, and that a better understanding of semantic integrity would greatly

help the solution of other integrity problems as well. An approach has been described in [Bay 74] to obtain semantic integrity via the definition of "aggregates" which limit the processing of data to the use of a set of carefully designed operations directly associated with the data.

Operational Integrity: For the purpose of this discussion let a "transaction" [EGLT 74] be the unit of processing for scheduling purposes and for external data base manipulation. A transaction is a sequence of more primitive "actions". Most work to date concerned with integrity has been limited to those integrity problems arising from the activity of the operating system:

1) the effort to schedule transactions to be processed in parallel as far as possible [EGLT 74], [Eve 74], [KiC 73], [CBT 74],
2) the need to acquire resources, in particular sets of data objects or individual data objects (also called "records" in [CBT 74] and "entities" in [EGLT 74], for exclusive or shared use by a transaction and to lock those resources accordingly,
3) the induced problems of deadlock among locking transactions, of deadlock discovery, of deadlock prevention, and of preemption of resources from transactions to resolve deadlocks.

III. OPERATING SYSTEMS AND OPERATIONAL INTEGRITY

As opposed to semantic integrity there is at least a brute force, straightforward solution for operational integrity, namely to avoid parallelism between transactions completely and to sequence in time the execution of transactions. This is unsatisfactory for many reasons, and better solutions have been developed for use in operating systems. We will survey these solutions briefly and indicate, why they are not satisfactory for data base applications. As usual in this field we use "process" as the analogon for "transaction". The list of techniques is adopted from G.C. Everest [Eve 74]:

Presequence Processes: Processes potentially competing for resources must be presequenced and must execute one after the other. For data base transactions it is often not known a priori, which data resources will be needed. This means that any two transactions will be potentially competing and must be sequenced. As a consequence, no parallelism is

possible and we have the unsatisfactory brute force method mentioned before. Still presequencing transactions, e.g. through time-stamping, may be useful for other purposes, like preventing indefinite delay of transactions by introducing an aging mechanism to increase the priorities of transactions.

Preempt Processes: This technique relies on discovering deadlocks after they have occurred. It then terminates (or backs up to an earlier state) one of the processes involved in the deadlock, the resources locked by that process are freed. As we shall see, this technique plays an important role in data base locking, too, butthere its application is much more difficult due to the large number of transactions and resources involved. This makes deadlock discovery and preemption quite complicated and expensive.

Preorder all System Resources: The processes are then required to claim their resources according to such a total order. It has been shown, that more general than linear orders, e.g. hierarchical orders, are sufficient to support a deadlock-free locking strategy [Ram 74]. In data bases the resources are data objects, which often do not have such an natural order. Furthermore a process might not be able to claim resources according to such an order, since his needed resources might be data dependent [EGLT 74], [CBT 74].

Preclaim needed Resources: Before starting to execute, a process has to claim all the resources it will ever need. Typically they are specified on the control cards preceding a job or job-step, and the process is not started until the operating system has granted to it all the requested resources. This is probably the most common technique for assigning non-sharable resources.

In a data base environment this technique requires considerable modifications to become feasible. Claiming resources may itself be a complicated and lengthy task requiring searching through large areas of a data base. These searches should run concurrently if possible.

Deadlock Prevention Algorithms: They often rely on too special properties of resources - like Habermann's banker's algorithm [Hab 69] - or on too special models of computation - like Schroff's algorithm [Sch 74]- to be generally applicable here.

IV. THE CHAMBERLIN, BOYCE, TRAIGER METHOD

In [CBT 74] a technique is proposed to provide operational integrity for data bases. The technique can be considered as a modification and combination of several methods described in section III. Integrity of the data base must be guaranteed at the beginning and again at the end of a transaction, it may be - and generally must be - violated by the single actions. Due to the potential interference of two or more transactions executing in parallel, transactions must lock certain parts of the data base for exclusive or shared use. The scheme proposed in [CBT 74] therefore requires each transaction to lock all its resources (parts of a data base, e.g. individual records or fields of records) during a so-called "seize phase" before starting the "execution phase". During the seize phase the data base must not be modified by the seizing transaction and therefore

1) preemption of locked resources from a transaction still in its seize phase is feasible, and
2) backing a transaction in its seize phase up to wait for the preempted resource is rather easy.

Once a transaction has started its execution phase, it is not allowed to claim more resources, thus no backup will be necessary. At the end of an execution phase a transaction must free all its resources before starting a new seize phase.

The seize phase may be a rather complicated task, thus seize phases of transactions should be run in parallel. This raises the deadlock problem again as usual: Let t_1, t_2 be two transactions. t_2 trying to seize resource r_1 already locked by t_1 must wait until r_1 is freed by t_1. But since resources are not locked in any particular order, t_1 may wish to lock first r_1, then r_2. If t_1 successfully seizes r_1 and t_2 successfully seizes r_2, then a deadlock has occurred. Such deadlocks must be discovered and a resource must be preempted from a transaction involved in the deadlock, say r_2 from t_2, causing t_2 to wait for t_1 on r_2.

In [CBT 74] an aging mechanism is attached to transactions to avoid deadlock due to indefinite delay of transactions. It is then shown in [CBT 74] that the scheme described is deadlock-free in the sense, that each transaction will eventually be processed. This requires, of course, the proper algorithms for discovery of deadlocks between transactions in their seize phases, for preemption or resources, and for backing up trans-

actions to certain points within their seize phases.

It is now clear, that the scheme proposed in [CBT 74] is a shrewd modification and combination of the following:

1) Try to preclaim needed resources.
2) If 1) would lead to deadlock, preempt resources.
3) Superimpose a presequencing scheme for transactions - e.g. through timestamping - to enforce an aging mechanism and to avoid deadlock due to indefinite delay of transactions.

V. SOME MODIFICATIONS AND AN ON-LINE TRANSITIVE CLOSURE ALGORITHM

The deadlock discovery algorithm mentioned as useful in [CBT 74] is not really applicable, since it requires that a transaction t_1 may wait for at most one other transaction to release resources. In the CBT-scheme, however, t_1 may be waiting for resources to be released by arbitrarily many transactions $t_{w_1}, t_{w_2}, \ldots t_{w_k}$ as the result of arbitrarily many preemptions of resources from t_1:

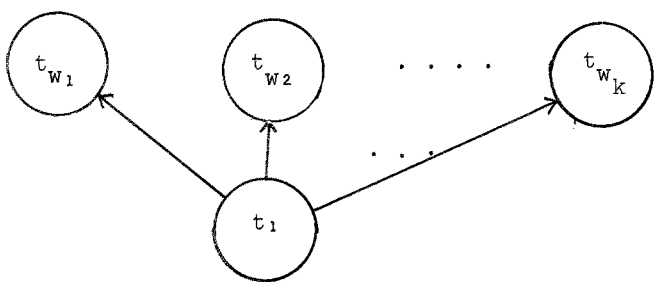

Fig. 1: Transaction t_1 waiting for other transactions.

The resource state of a transaction t_i is determined by the set

$$A_i = \{r_{i_1}, \ldots, r_{i_{q_i}}\}$$

of resources which it has so far acquired, and the set of request pairs

$$B_i = \{(r_{i_1}', t_{i_1}), \ldots, (r_{i_{p_i}}', t_{i_{p_i}})\}$$

where (r_{i_j}', t_{i_j}) indicates that resource r_{i_j}' is desired from transaction t_{i_j}. Any transaction t_i for which B_i is non-empty is in a wait state.

We may then define the wait relation $w \subseteq T \times T$ where T is the set of transactions, such that

$$(t_i, t_j) \in w \text{ iff } \exists\, r : (r, t_j) \in B_i.$$

We say that t_i is <u>waiting for</u> t_j (to release r). t_i may be waiting for several transactions as noted above, and for several resources from the same transaction.

The wait graph G_w is the directed graph

$$G_w = (T, w).$$

Deadlock discovery amounts to finding cycles in G_w or, equivalently, to finding pairs (t, t) in the transitive (but not reflexive) closure w^+ of w. Thus deadlock exists iff $\exists t \in T : (t,t) \in w^+$.

Maintaining w is trivial, since something like the B_i's will have to be maintained in any case. Calculating w^+ from w is, on the other hand, quite expensive, the best known algorithms requiring $O(n^3)$ [War 62] or $O(n \cdot m)$ [Bay 74] steps, where n is the number of nodes in G_w and m the number of arcs.

It would be sufficient, however, to have a good "on-line" transitive closure algorithm since w^+ need only be partly modified as arcs are added to and deleted from w.

More precisely, "on-line" transitive closure algorithm means an algorithm solving the following problem:

Given w, w^+, calculate
w', w'^+, where
$w' = w \cup \{(t_i, t_j)\}$ or
$w' = w \setminus \{(t_i, t_j)\}$.

Although it is quite simple to add an arbitrary arc and calculate w'^+ from w^+, it seems in the general case notoriously difficult to delete an arbitrary arc and calculate w'^+ from w^+. No better alternative seems to be known than calculating w'^+ from scratch, i.e. starting with w' and ignoring the fact that we already have w^+.

For our purpose, we need a highly simplified version of the on-line algorithm for the transitive closure only. By closer inspection one observes, that we need to delete sinks of G_w and the arcs leading into sinks of G_w only. This is the decisive property which makes the difficult general problem tractable in our special case. To get w'^+ from w^+ now simply amounts to deleting or zeroeing out a column from the Boolean matrix describing w^+.

We will now develop such an on-line transitive closure algorithm in more detail. We assume that transactions will wait in queue q(r) for an already locked resource r. The first transaction on a queue has successfully locked (or seized) the resource, it may be in its seize or execution phase. All other transactions on the queue are waiting (or blocked). We indicate this as in Fig. 2.

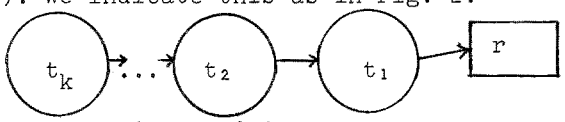

Fig. 2: Transactions waiting for resource r.

t_1 has locked r,

t_{i+1} is waiting for t_i to release (or free) r; i=1,2,...,k-1,

when t_i eventually releases r (and no preemptions have occurred in the meantime), then t_{i+1} will seize r.

Let us first consider the state transition diagram of a transaction (Fig. 3) and the operations relevant to that diagram, which a transaction may perform:

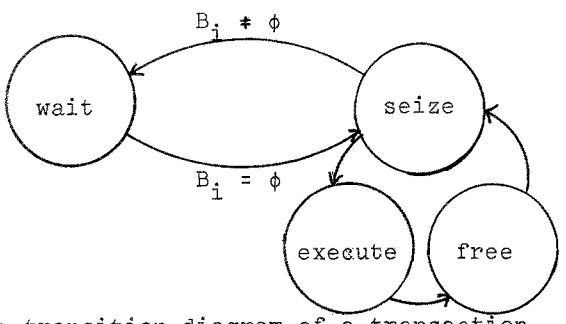

Fig. 3: The state transition diagram of a transaction.

A transaction t_i can perform the following operations involving resource r and another transaction t_k:

Seize r: $A_i := A_i \cup \{r\}$; update q(r);

$\forall r \in A_i$: Free r: $A_i := \phi$;

$\forall r \in A_i$ do if t_k is next in queue for r
then begin (r,t_i) must be in B_k;
$B_k := B_k \setminus \{(r,t_i)\}$;
$A_k := A_k \cup \{r\}$;
update q(r);
if $B_k = \phi$ then make t_k
continue to seize
end;
update w and w^+;

Seize unsuccessfully: t_i is still in the seize state,
let t_k be last in queue for r:
Case 1: no deadlock arises, if t_i is queued behind t_k in q(r):
$B_i := \{(r,t_k)\}$;
put t_i into wait state;
update w and w^+;
Case 2: A deadlock would arise, if t_i were queued as in Case 1. This deadlock is discovered by tentatively, but not definitely queueing t_i as in Case 1, updating w^+ and checking, whether w^+ contains cycles. In this case t_i might have to preempt r from t_k. t_k must be in wait state, since we have a cycle:
In this situation t_i should move forward in q(r) until it can be inserted and no deadlock arises; update q(r) accordingly.
Let t_ℓ be the first transaction in q(r) (starting from t_k) such that inserting t_i between t_ℓ and $t_{\ell-1}$ causes no deadlock, then we have Case 2a. If there is no such t_ℓ then we have Case 2b.

Note: In [CBT 74] t_i is always inserted as close to the head of the queue as possible. This strategy favors the younger transactions and must rely heavily on an aging mechanism to prevent indefinite delay. The processing costs of this aging mechanism are not analyzed.

Case 2a: $B_\ell := \left(B_\ell \setminus \{(r,t_{\ell-1})\}\right) \cup \{(r,t_i)\}$;
$B_i := \{(r,t_{\ell-1})\}$;
update q(r);
t_i goes into wait state;
update w and w^+;

Case 2b: t_1 cannot be executing, otherwise t_i would queue behind t_1 according to Case 2a. Therefore t_1 is seizing or waiting. We make t_i preempt r from t_1, i.e. we queue t_i in front of t_1;
if $B_1 = \phi$, then make t_1 wait;
$B_1 := B_1 \cup \{(r,t_i)\}$;
$A_1 := A_1 \setminus \{r\}$;
$A_i := A_i \cup \{r\}$;
update w and w^+;

Necessary Changes to w and w^+ and Analysis of their Complexity

For the following analysis we assume that w^+ is represented in an n×n Boolean matrix K with the meaning

$$K[i,j] \equiv (t_i,t_j) \in w^+.$$

Operation	Description of Operation	Complexity of the Change to w^+
Seize r:	No change to w or w^+	0
$\forall r \in A_i$: Free r:	Since t_i frees all its resources at the end of its execution phase, we can remove all arcs (t_k,t_i) from w, and delete or zero out column i of K.	O(n)
	For the analysis of the following operations we need two auxiliary procedures first. Let t_i be in its seize state. To insert an arc (t_i,t_k) into w and to update w^+ accordingly we need the procedure INSERT1.	

Operation	Description of Operation	Complexity of the Change to w^+
	To insert (t_ℓ,t_i) we need the procedure INSERT2.	
INSERT1 (t_i,t_k):	Comment t_i is in seize state;	
	$w := w \cup \{(t_i,t_k)\}$;	constant
	$\forall t_j_{(t_j,t_i)\in w^+} \; \forall t_\ell_{(t_k,t_\ell)\in w^+} : w^+ := w^+ \cup \{(t_j,t_\ell)\}$;	$O(n^2)$ at worst, $O(m)$ average, see lager analysis
	$\forall t_\ell_{(t_k,t_\ell)\in w^+} : w^+ := w^+ \cup \{(t_i,t_\ell)\}$;	$O(n)$
	$\forall t_j_{(t_j,t_i)\in w^+} : w^+ := w^+ \cup \{(t_j,t_k)\}$;	$O(n)$
	$w^+ := w^+ \cup \{(t_i,t_k)\}$;	constant
INSERT2 (t_ℓ,t_i):	Comment t_i is in seize state;	
	$w := w \cup \{(t_\ell,t_i)\}$;	constant
	$\forall t_j_{(t_j,t_\ell)\in w^+} : w^+ := w^+ \cup \{(t_j,t_i)\}$;	$O(n)$
	$w^+ := w^+ \cup \{(t_\ell,t_i)\}$;	constant
	Note: Since t_i is in the seize state, there is no t such that $(t_i,t)\in w^+$. Consequently no cycle in w^+, and therefore no deadlock can arise due to the operation INSERT2 (t_ℓ,t_i).	
Seize r unsuccessfully:	As before, let t_k be last in queue for r:	

	for j := k step -1 until 1 do
	begin tentatively INSERT1 (t_i,t_j);
	if no deadlock then
	begin ℓ := j+1;
	exit to perform Case 2a
	end end;
	perform Case 2b;
		for each deadlock $O(n^2)$ or $O(n+m)$ resp.

Operation	Description of Operation	Complexity of the Change to w^+
	Case 2a: make last INSERT1 operation definite;	at worst $O(n^2)$ or $O(n+m)$
	if $\ell \neq k+1$ then begin INSERT2 (t_ℓ, t_i);	$O(n)$
	if $\exists r' \neq r: (r', t_{\ell-1}) \in B_\ell$ then	search of B_ℓ
	else $w := w \setminus \{(t_\ell, t_{\ell-1})\}$ end;	
	Case 2b: INSERT2 (t_1, t_i);	$O(n)$

Analysis of INSERT1:

Adding a single arc to w, according to INSERT1, say (t_i, t_k), requires oring row k of the Boolean matrix K to all rows j with $(t_j, t_i) \in w^+$. At worst this part of INSERT1 requires $O(n^2)$ operations. If, however, there are m arcs in w^+, then each node on the average will have m/n arcs into it and m/n arcs out of it. Accordingly the average number of operations will be

$$O\left(n \cdot (m/n)\right) = O(m).$$

VI. FOUR STRATEGIES FOR PREVENTING INDEFINITE DELAY

With the locking and preemption schemes proposed it is still conceivable, that a transaction is delayed indefinitely from its execuiton phase. To deal with this problem, we propose four increasingly effective, but also increasingly costly strategies. It seems quite reasonable to employ several strategies within one system successively in order to force transactions which have passed a certain age threshold into their execution phase and out of the system.

Strategy 1: Let t_e be the eldest transaction. Schedule all transactions t, such that $t_e w^+ t$, with highest priority. This clearly has a tendency to speed up the processing of t_e. It is easy to find those t from the t_e-row of the Boolean matrix describing w^+.

Strategy 2: Stop all transactions in seize phases from further seizing except those t for which $t_e w^+ t$.

Strategy 3: For all r such that t_e is waiting in q(r) let t_r be the transaction that has locked r. If t_r is seizing or waiting, preempt r from t_r and give r to t_e. If t_r is executing, insert t_e in q(r) directly

behind t_r. No new deadlocks can arise if we assume that all these preemptions are performed together in one step. Then recalculate the new w'^+.

Strategy 4: Stop all transactions, which are not executing from seizing further. Then apply strategy 3 for t_e until t_e has reached its execution phase. Then let the other transactions proceed.

Some Oberservations on Strategies 1, 2, 3, 4: It is clear that all strategies will tend to bring t_e closer to its execution phase.

Strategy 1 can be generalized to establish a partition of the transactions into a linearly ordered set of priority classes, which can serve as the basis for a general scheduling strategy. Strategies 1 and 3 might still allow indefinite delay. It is easy to construct a plausibility argument, that strategies 2 and 4 will prevent indefinite delay of transactions.

VII. AN ALTERNATIVE APPROACH: PREEMPTION AND PARTIAL BACKUP

Although it seems feasible to maintain the basic locking and preempting mechanism proposed in [CBT 74] using the special algorithms described in the preceding sections, there is another argument supporting a more radical preemption than that proposed in the CBT-scheme:

Let us assume that r_1 is preempted from t_1 by t_2, which probably updates r_1. Depending on the value of r_1, t_1 might have locked other resources r_1', r_1'',... already. But since the value of r_1 changes, the decision of t_1 to lock r_1', r_1'', ... should be reconsidered. In other words, t_1 should be backed up within its seize phase to precisely the state it was in just before seizing r_1, it should then be waiting for t_2 on r_1, and the resources r_1', r_1'', ... locked by t_1 should be freed again.

In such a preemption scheme a transaction t_1 will generally be waiting for at most one other transaction t_2 on precisely one resource r_1. The wait relation $t_1 w t_2$ shall now mean that t_1 waits for the holder t_2 of r_1 and not for the predecessor in $q(r_1)$, since we do not need to maintain such queues. The resulting G_w is obviously a forest of oriented trees, the arcs pointing towards the roots. Only roots are processing in the execution or seize phases. All other transactions are waiting.

Since a transaction t_1 is waiting for t_2 on precisely one resource r_1, we may label that arc with r_1.

The following simple algorithms then describe the necessary operations.

Seize unsuccessfully:

Case 1, no deadlock arises: t_1 trying to lock r already locked by t_2 means that the tree with root t_1, i.e. $T(t_1)$, is appended as a subtree to t_2, the new arc being labelled with r.

Case 2, deadlock arises: Deadlock discovery is quite simple: Each seizing or executing transaction is the root node of one tree. Deadlock arises precisely when t_1 is also the root node of the tree in which t_2 is. To find this out, just follow the arcs from t_2 to the root. In this case a cycle would be generated by inserting an arc (t_1,t_2). The deadlock is resolved by preempting the resource r from t_2.

Preemption works as follows: t_2 must free r and all resources it locked after r. In the process - see the Free operation - corresponding subtrees of t_2 will be detached - allowing their roots to continue seizing - and the arc (t_2,t_3) from t_2 to its father t_3 in $T(t_1)$ will be deleted. The tree $T'(t_2)$ remaining after pruning $T(t_2)$ will be attached as a subtree of t_1 by introducing the new arc (t_2,t_1) with label r.

Free r': If t_2 frees a resource r' either due to being backed up in its seize phase or due to finishing an execution phase and there is an arc (t_4,t_2) labelled r', then this arc can be deleted, thereby t_4 becomes a root and can proceed in its seize phase. To free such arcs one must represent these trees by data structures in which it is possible to follow arcs in both directions.

VIII. PREVENTING INDEFINITE DELAY

It is possible that for individual transactions t a situation similar to a deadlock might again arise due to t being preempted and backed up in its seize phase again and again. Strategies 1 and 2 of section VI are easily adapted to work for the preemption and backup technique of section VII.

The analogon to strategy 3 of section VI is much easier to implement now: Let t_e be the eldest transaction in the system again. Assume that t_e is waiting for t on r and t is not executing. (If t is executing, nothing can be done except scheduling t with highest priority until t has finished executing.) Then t_e will preempt r from t and t will be backed up in its seize phase to a state just before seizing r. t_e becomes a root and continues seizing. A new arc (t,t_e) labelled r is introduced. The preemption process works precisely as described in section VII. The main difficulty of strategy 3 of section VI has disappeared, since we do not explicitly store w^+. Instead, cycles are discovered by just following the path from an arbitrary node of a tree to its root, a simple and fast operation. To prevent pathological cases of data bases changing faster than t_e being able to catch up in its own seize phase, we can apply an analogon to strategy 4 of section VI again. Instead, however, it suffices to prevent that transactions will enter from their seize phases into their execution phases. Let this be strategy 5. Since only finitely many transactions are in the system at any one time, and since each executing transaction will run only a finite time, t_e will eventually be able to finish both its seize and execution phase, and indefinite delay of t_e cannot occur.

IX. PHANTOMS AND PREDICATE LOCKS

In [EGLT 74] a technique is described to use so called predicate locks ("predicate locking") for locking logical, i.e. existing as well as potential subsets of a data base instead of locking individual data objects ("individual object locking"). This technique also solves the "phantom problem". To explain briefly, what phantoms are, let us assume that there is a universe $\mathcal{D}$ of data objects (called "entities" in [EGLT 74] and "records" in [CBT 74]) which are the potential data objects in the data base B. Thus $B \subseteq \mathcal{D}$. Two transactions t_1, t_2 may have successfully locked all their needed resources, and they may be executing. t_1 may add a new object $r_1 \in \mathcal{D}$ to B and t_2 may add a new object $r_2 \in \mathcal{D}$ to B, such that t_1 would have locked r_2 and t_2 would have locked r_1, if t_1 or t_2 would have seen r_2 or r_1 resp. during their seize phases. r_1 and r_2 are called "phantoms", since they might, but not necessarily will appear in B (materialize) while t_1 or t_2 are in their execution phases.

The appearance of just a single phantom, say r_1, does not cause any difficulty, since this has the same effect as running the transactions t_1, t_2

serially, namely in the order t_2 followed by t_1. In this case also t_2 would not see the object r_1 created by t_1 and therefore t_2 could not be able to lock r_1. It is the goal of predicate locking to schedule transactions in parallel as far as possible under the restriction, that the parallel schedule is equivalent to - i.e. has exactly the same total effect on the data base as - a serial schedule. One also says that such a schedule is a "consistent schedule", or that each transaction sees a "consistent view" of the data base.

To enforce consistent schedules each transaction t is required to lock (for read or write access) all data objects $E(t) \subseteq \mathcal{D}$ - irrespective of whether they are in B or are just phantoms - which might in any way influence or be influenced by the effect of t on B. E(t) shall be locked by specifying a predicate P defined on $\mathcal{D}$ (or on a part of $\mathcal{D}$, e.g. on a relation [Cod 70]) such that $E(t) \subseteq S(P)$ where S(P) is the subset of elements of $\mathcal{D}$ satisfying P.

Two transactions t_1, t_2 are then said to be in conflict, if for their predicates P_1, P_2 it is true that $\exists r \in S(P_1) \cap S(P_2)$ and t_1 or t_2 performs a write action on r. Thus conflict can arise even if r is a phantom. In this case t_1, t_2 cannot run in parallel, but must be run serially. The order in which they are run is irrelevant for consistency. This order might be important for other reasons which are not of interest here.

The main difficulties in using such a locking and scheduling method seem to be the following:

1) Find a suitable predicate P_t for t. Ideally $E(t) = S(P_t)$ should hold, but then P_t might be too complicated. If P_t is chosen in a very simple way, then $S(P_t)$ might be intolerably large, increasing the danger of phantoms, which are really artificial phantoms.

2) The problem "$S(P_1) \cap S(P_2) \neq \phi$" may be very hard. In general this problem is even undecidable. Thus for practical applications and a given it is necessary to find a suitable class of locking predicates, for which the problem "$S(P_1) \cap S(P_2) \neq \phi$" is not only decidable, but for which a very efficient decision procedure is known. For more details and a candidate class for suitable locking predicates see [EGLT 74].

3) Phantoms might turn out to be a very serious but mostly artificial obstacle to parallel processing in the following sense: phantoms in

$S(P_1) \cap S(P_2)$ prohibit t_1 and t_2 from being run in parallel. But if these phantoms do not materialize, and if furthermore $S(P_1) \cap S(P_2) \cap B = \phi$, then, of course, t_1 and t_2 could have been run in parallel. How much of an artificial obstacle phantoms are to parallel processing seems to be unknown and can probably be answered only for concrete instances of data bases.

X. A UNIFICATION OF INDIVIDUAL OBJECT LOCKING AND PREDICATE LOCKING

Let us start with the crucial observation for this section:

"Transactions,which are pure readers, do not need to lock phantoms".

A transaction is a "pure reader", if it is composed of read actions only. Obviously for many data base applications the pure readers are a very important class of transactions.

To understand our observation, consider two pure readers t_1, t_2 first. Since there are no write actions at all, there is no possibility for phantoms to materialize, thus they need not be locked. Phantom locking is only necessary to control the interaction with a transaction, say t_3, which also performs write operations. We call t_3 a "writer". Consider the interaction between t_1 and t_3. Let us assume that there is a phantom $r \in S(P_1) \cap S(P_3)$ such that t_3 might perform a write on r. Then t_1 and t_3 could not run concurrently, if t_1 would use predicate locking. If however, t_1 uses individual object locking and successfully terminates its seize phase, then t_1 can run in parallel with t_3 provided that

$$\widehat{S(P_1)} \cap S(P_3) = \phi$$

where $\widehat{S(P_1)} = S(P_1) \cap B$, i.e. the set of real data objects (without phantoms) in B which t_1 needs to lock in order to see a consistent view of B. But now $\widehat{S(P_1)}$ can be locked by t_1 using conventional "individual object locking" as e.g. described in [CBT 74] instead of predicate locking. If t_3 should materialize phantoms, then running t_1 and t_3 in parallel still is consistent and has the same effect as the serial schedule t_1 followed by t_3.

The following observation should also be clear now: To control the interaction between the writer t_3 and the pure reader t_1 if suffices, that t_3 use individual object locking according to [CBT 74]. t_3 need not lock its phantoms, since t_1 is not interested in phantoms anyway. We can con-

clude that the problem of phantoms - and therefore predicate locking - arises only between writers.

The preceding observations suggest several alternative approaches for handling the phantom problem:

Strategy 1 - Serialize Writers:
Since, as we just observed, phantoms cause difficulties only between writers, the simplest solution is, not to schedule any writers to run concurrently. Concurrency is possible between arbitrarily many pure readers and at most one writer. Consistency is guaranteed by individual object locking and by handling deadlocks and preemptions as described in the earlier part of this paper. The problem of phantoms does not arise.

As mentioned before, in many applications most transactions are pure readers. Serializing writers in those cases should notcause a significant loss of concurrency and has the advantage that predicate locking with its associated difficulties is not needed.

Strategy 2 - Predicate Locks between Writers:
Use predicate locks as described in [EGLT 74] only to determine whether two writers t_3, t_4 can run in parallel. After a writer is allowed to proceed on account of his predicate locks, he then starts individual object locking to compete for further processing with other transactions, which are pure readers, exactly as in strategy 1. For more details on the individual object locking phase, in particular the types of locks, see strategy 3.

Using predicate locking and individual object locking at this point allows a more general notion of conflict than that used in [EGLT 74]. Let U_1 or R_1 be the set of objects including phantoms which are updated or only read respectively by a transaction t_1. Define U_2 and R_2 for t_2 analogously. Obviously $U_1 \cap R_1 = \phi$ and $U_2 \cap R_2 = \phi$.

Then define
$$\begin{aligned} B_1 &:= U_1 \cap U_2 \\ B_2 &:= U_1 \cap R_2 \\ B_3 &:= R_1 \cap U_2 \\ B_4 &:= R_1 \cap R_2 \end{aligned} \qquad (X.1)$$

Diagrammatically this can be shown as in Fig. 4.

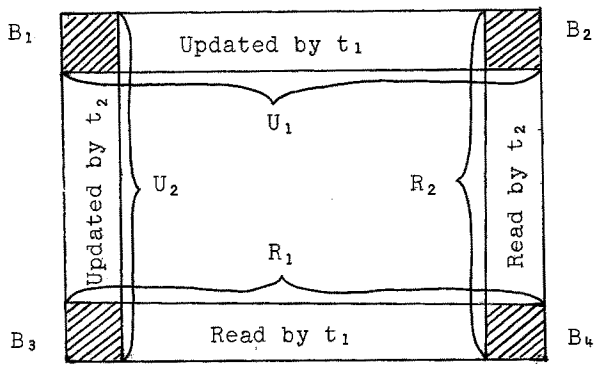

Fig. 4: Possible intersections of update and read-only sets.

For t_1 and t_2 to proceed in parallel with individual object locking the following conditions must hold:

$$B_1 = \phi \qquad (X.2)$$
$$B_2 = \phi \vee B_3 = \phi$$

Without individual object locking the stronger condition $B_2=\phi \wedge B_3=\phi$ is required in [EGLT 74]. To see that our weakend condition suffices let us assume without loss of generality that $B_2=\phi$ and $B_3 \neq \phi$.

B_3 is read only by t_1, but is updated by t_2. Also B_3 may contain phantoms which are materialized by t_2. Let us assume that both t_1 and t_2 are successful in their seize phases, i.e. while locking individual objects excluding phantoms, and then continue to run in parallel. We claim that this is equivalent to the serial schedule t_1 followed by t_2.

Since B_1 and B_2 are both empty, the effect of t_1 on B cannot in any way influence t_2, thus t_2 has the same effect on B if it is run after t_1 or parallel to t_1.

B_3 is not empty, but t_1 successfully locked all the resources it needed to see a consistent view of the data base. t_1 may have missed phantoms materialized by t_2 , thus the effect of t_1 will be the same as in the serial schedule t_1 t_2. Consequently running t_1 and t_2 in parallel is equivalent to the serial schedule t_1t_2, and is therefore consistent.

The conditions (X.2) for t_1 and t_2 to proceed in parallel can be gener-

alized for $t_1, t_2, \ldots, t_n$ to proceed concurrently. This is left to the reader.

Strategy 3: This strategy sacrifices some concurrency, but is much simpler to implement than strategy 2. There a writer t_i was required to perform individual object locking both in the sets U_i and R_i. It turns out that with the conflict condition of [EGLT 74] between writers, writers need perform object locking only within U_i, but they need not set any read locks.

Assume that a writer t_i first locks the sets U_i and R_i by specifying the predicates P_U^i and P_R^i. The condition for two writers t_i and t_j to run concurrently then is:

$$\begin{aligned} S(P_U^i) \cap S(P_U^j) &= \phi \\ S(P_U^i) \cap S(P_R^j) &= \phi \\ S(P_R^i) \cap S(P_U^j) &= \phi \end{aligned} \qquad (X.3)$$

After successfully locking U_i and R_i the writer then proceeds to perform individual object locking within U_i by setting "u-locks" for exclusive use of data objects to be updated. These u-locks are necessary for preventing pure readers from reading those objects while they are being updated. Since the sets $S(P_U^i)$ are pairwise disjoint, there is never any possibility for conflict between u-locks of different writers.

We observe that writers need not set any individual read-locks, called "r-locks", since $S(P_R^i) \cap S(P_U^j) = \phi$, and conflict of u-locks of one writer and r-locks of another would not be possible anyway. Furthermore, several r-locks of readers and writers on the same data object would be allowed, since data objects are shareable as long as they are only read.

The only potential conflict still remaining is between read-access of pure readers and update-access of a writer to the same data object s, which is not a phantom. To control this we require pure readers to set r-locks on individual data objects s to be read. This must happen during a seize phase. Several r-locks can be on s, but not both r-locks of pure readers and a u-lock of a writer. Thus if a reader (writer) sets an r-lock (u-lock) first then a writer (reader) trying to set a u-lock (r-lock) on the same data object must wait for the reader (writer) to release s. This leads to the usual wait situations with the possibility for deadlock and the need for preemption and backup as described in the

first part of this paper.

If a deadlock is discovered then a reader or a writer is backed up within its seize phase for setting r-locks or u-locks resp. as described before. For simplicity we can assume that locking with the predicates P_U and P_R is one indivisible operation, thus deadlock between writers is not possible during this phase of predicate locking.

To summarize, a writer t_i proceeds as follows:

1) Lock predicates P_U^i and P_R^i. If conditions (X.3) are satisfied for all other writers t_j which have successfully locked their predicates P_U^j and P_R^j then proceed with step 2), otherwise wait, until P_U^i and P_R^i can be successfully locked, then proceed with step 2).

2) Start a seize phase setting u-locks on individual data objects to be updated within $S(P_U^i)$. In case of conflict with r-locks wait or be backed up within this seize phase.

3) A pure reader performs a seize phase setting r-locks on data objects to be read. In case of conflict with u-locks the reader must wait or be backed up within this seize phase.

Summarizing the main advantages of strategy 3 we observe:

- Only writers use predicate locking to handle phantoms.
- Concurrency between writers is possible.
- Writers need an individual object locking phase for setting u-locks in their update areas only. In this phase phantoms are ignored.
- Pure readers do not use predicate locking, they set r-locks during an individual object locking phase only and ignore phantoms completely.

Note: Since predicate locking is now needed for writers only, it might be quite feasible to replace arbitrary predicates by a fixed partitioning of the data base or by a fixed family of subsets of $\mathcal{D}$, whose intersection properties are known once and for all and recorded in a Boolean matrix (intersection between two subsets is empty or not). Instead of locking predicates the above subsets are then locked by writers.

Acknowledgement: I wish to thank Mr. John Metzger, with whom I had many useful discussions during the writing of this paper.

Bibliography

[Bay 74] Bayer, R., "AGGREGATES: A Software Design Method and its Application to a Family of Transitive Closure Algorithms". TUM-Math. Report No. 7432, Technische Universität München, Sept. 1974

[Bjo 73] Bjork, L.A., "Recovery Semantics for a DB/DC System". Proceedings ACM Nat'l. Conference 1973, 142-146

[CBT 74] Chamberlin, D.D., Boyce, R.F., Traiger, I.L., "A Deadlock-free Scheme for Resource Locking in a Data Base Environment". Information Processing 1974, 340-343

[Cod 70] Codd, E.F., "A Relational Model for Large Shared Data Banks". Comm. ACM 13, 6 (June 1970), 377-387

[CES 71] Coffman, E.G. Jr., Elphick, M.J., Shoshani, A., "System Deadlocks". Computing Surveys 3, 2 (June 1971), 67-78

[Dav 73] Davies, C.T., "Recovery Semantics for a DB/DC System". Proceedings ACM Nat'l. Conference 1973, 136-141

[EGLT74] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger I.L., "On the Notions of Consistency and Predicate Locks in a Data Base System". IBM Research Report RJ 1487, Dec. 30, 1974

[Eve 74] Everest, G.C., "Concurrent Update Control and Data Base Integrity". In: Data Base Management (ed. Klimbie, J.W., and Koffeman, K.L.), North Holland 1974, 241-270

[Fos 74] Fossum, B.M., "Data Base Integrity as Provided for by a Particular Data Base Management System". In: Data Base Management (ed. Klimbie, J.W., and Koffeman, K.L.), North Holland 1974, 271-288

[Hab 69] Habermann, A.N., "Prevention of System Deadlocks". Comm. ACM 12, 7 (July 1969), 373-377, 385

[KiC 73] King, P.F., Collmeyer, A.J., "Database Sharing - an Efficient Mechanism for Supporting Concurrent Processes". AFIPS Nat'l. Comp. Conf. Proceedings 1973, 271-275

[Oll 74] Olle, T.W., "Current and Future Trends in Data Base Management Systems". Information Processing 1974, 998-1006

[Ram 74] Ramsperger, N., "Verringerung von Prozeßbehinderungen in Rechensystemen". Dissertation, Technische Universität München, 1974

[Sch 74] Schroff, R., "Vermeidung von totalen Verklemmungen in bewerteten Petrinetzen". Dissertation, Technische Universität München, 1974

[War 62] Warshall, S., "A Theorem on Boolean Matrices". Journal ACM 9, 1 (January 1962), 11-12

[Wil 72] Wilkes, M.V., "On Preserving the Integrity of Data Bases". The Computer Journal, 15, 3 (August 1972), 191-194