INTEGRITY, CONCURRENCY, AND RECOVERY IN DATABASES

Rudolf Bayer*
Institut für Informatik
Technische Universität München
8 München 2, Germany

1. INTEGRITY

In this paper we will discuss some aspects of obtaining correctness of databases. Several conditions must be met to achieve correctness:

1) proper operation of the hardware,

2) proper construction of the software,

3) proper use of the system.

1) will be of interest only to the extent that the recovery scheme described in section 10 can also be used to recover from hardware failures, see also [Wil 72]. This paper deals mainly with 2). 3) is concerned with techniques to prevent mischievous or accidental misuse of a computer system, i.e., with its security. This will not be discussed here.

Correctness is of particular concern in database systems for several reasons:

1) Longevity: Even rare errors will in the long run lead to contamination and degradation of a database. Purging erroneous data and their consequences from a database is difficult.

2) Limited repeatability: Even if errors are discovered, it may be impossible or useless to rectify them 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 checking the results repeatedly until correctness is obtained.

4) Multiaccess: Databases are manipulated by many users with different quality standards. It is infeasible to completely entrust the quality control to these users and difficult to track the source and the proliferation of errors.

*During the Academic Year 1975/76 visiting at IBM Research Laboratory, San Jose, CA 95193, USA

It seems hopeless to give a precise and complete description of what we mean intuitively by a "correct database." Still from our knowledge about the meaning of the data, a set of necessary (but not sufficient) conditions can be derived which must be satisfied by a correct database. We call much conditions integrity-constraints and we say that a database is in a state of integrity, if all integrity-countraints are satisfied.

Integrity might be enforced by allowing on certain data only a limited set of precisely specified meaningful, and therefore integrity preserving operations [Bay 74], [LiZ 74] by adopting a set of strict programming and interaction conventions, by dynamically checking the results of updates, or by proving for each program manipulating the database, that the integrity-constraints will be satisfied. Much work still needs to be done on how to describe, to enforce, and to implement such integrity-constraints.

2. TRANSACTIONS AND CONSISTENCY

In multiaccess database systems certain activities of users must, for several reasons, be considered as atomic operaTions, called transactions [CBT 74]. Thus, performing a set of transations $\{t_1,t_2,\ldots,t_n\}$ on a database must have the same effect as performing some sequence (precisely which is irrelevant) $(t_{i_1},t_{i_2},\ldots,t_{i_n})$ of transactions where $(i_1,i_2,\ldots,i_n)$ is some permutation of $(1,2,\ldots,n)$.

The execution of a transaction requires in reality a sequence of actions. We write $t_j = (a_{j_1},a_{j_2},\ldots,a_{j_k})$ for a transaction t_j composed of the actions $a_{j_1},a_{j_2},\ldots,a_{j_k}$. Performing transactions serially may be too costly (because of too low a utilization of hardware resources) or too slow (since other transactions must wait for people to react). Therefore transactions should run concurrently, subject to the constraint, that concurrency produces the same net effect on the database as some serial execution.

We call this last constraint consistency [EGLT 74]. It can be considered as a special integrity constraint establishing the conditions under which transactions can run concurrently. To achieve consistency, one could run transactions serially after all, or one might prove that running two transactions, t_1,t_2, concurrently is consistent.

More formally, we write $\{t_1,t_2\}$ for parallel execution, (t_1,t_2) for serial execution of t_1 followed by t_2, and $\equiv$ for equivalence of two executions in the sense of yielding the same result for the database. Showing that $\{t_1,t_2\}$ is consistent amounts to establishing the truth of

$$\{t_1,t_2\} \equiv (t_1,t_2) \vee \{t_1,t_2\} \equiv (t_2,t_1). \qquad (1)$$

The generalization of (1) to an arbitrary set of transactions is obvious.

A third technique is qenerally followed in database systems to guarantee consistency. Before a transaction is allowed to execute, it must acquire all the resources it might possibly need. Here <u>resources</u> are the data objects in the database. Resources are acquired by locking them in the proper mode. We will see later that more complicated lock-modes than the read-locks and write-locks (or shared and exclusive locks) commonly found in operating systems are needed for database locking.

Most work to date concerned with integrity has been limited to consistency, i.e., 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.

Locking techniques and the associated problems have been investigated extensively for operating systems. These techniques are of limited applicability in database locking and require considerable modifications to become useful here.

3. LOCKING IN OPERATING SYSTEMS

We will briefly survey locking techniques developed for operating systems and indicate, why they are not satisfactory for database applications. As usual in this field we use "process" as the analogon for "transaction." The list of techniques is adopted from [Eve 74] and [CES 71]. Our main interest is in the treatment of the deadlock problem.

Presequence Processes: Processes potentially competing for resources must be presequenced and must execute one after the other. For database 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. Therefore, no parallelism is possible. Still presequencing transactions, e.g., through time-stamping, may be useful for other purposes, like preventing indefinite delay 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 database locking, too, but its application is difficult due to the large number of transactions and resources involved. Deadlock discovery, preemption of resources, and transaction backup to recover a state of integrity of the database become complicated and expensive. Suitable algorithms to solve these problems are challenging.

Preorder all System Resources: The processes are required to claim their resources according to a total order defined on the set of all resources. In databases the resources are data objects, which often do not have such a 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 must 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.

In databases 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 database. These searches themselves 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.

4. A PROPOSAL FOR DATABASE LOCKING

In [CBT 74] a technique is proposed to provide consistency for database locking. The technique can be considered as a modification and combination of several methods described in section 3. Since transactions are to be considered atomic, integrity of the database 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 database for exclusive or shared use. The scheme proposed in [CBT 74] therefore requires each transaction to lock all its resources (parts of a database, e.g., individual records or fields of records) during a so-called "seize phase" before starting the "execution phase." During the seize phase the database must not be modified by the seizing transaction. For such a transaction preemption of locked resources and backup to wait for the preempted resource are easy.

Once a transaction has started its execution phase, it is not allowed to claim more resources, thus no backup will be necessary due to deadlock. 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 and 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 indefinite delay. It is then shown in [CBT 74] that in the scheme described each transaction will eventually be processed. This requires, of course, suitable algorithms for discovery of deadlocks between transactions in their seize phases, for preemption of resources, and for backing up transactions 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.

5. ON-LINE TRANSITIVE CLOSURE ALGORITHMS FOR DEADLOCK DISCOVERY

The deadlock discovery algorithm [KiC 73] mentioned as useful in [CBT 74] is not really applicable, since it requires that a transaction t may wait for at most one other transaction. In general, however, t may be waiting for several transactions.

To clarify this, let us assume that a lock request for a resource r is a pair (t,μ) where t is a transaction and μ is a lock mode, meaning that t requests a lock of mode μ. Associated with each resource r there is a FIFO waiting queue Q(r) of lock requests and a set G(r) of granted lock requests, also called the <u>granted group</u> [GLP 75]. The next request in Q(r) can be granted if the mode of the request is compatible with the modes of all requests in G(r).

To grant the request (t,μ) perform:

$G(r) := G(r) \cup \setminus \{(t,\mu)\};$

$Q(r) := Q(r)$ remove $\{(t,\mu)\}$

We also say that t now has a μ-lock on r.

To release the granted request (t,μ) perform:

$G(r) := G(r) \setminus \{(t,\mu)\}$

To issue a request (t,μ) for r perform:

$Q(r) := Q(r)$ append $\{(t,\mu)\}$

When t_i issues a request (t_i,μ) for r, t_i will enter a wait state until the request is granted. Thus, t_i will be waiting in at most one queue for a resource r. Before the request (t_i,μ) can be granted all transactions with lock-request-modes incompatible with μ which are in G(r) or ahead of t_i in Q(r) must finish their execution phase and release their locks. We denote this set by $B_i = \{t_{i_1}, t_{i_2}, \ldots, t_{i_{k_i}}\}$ and say that t_i is (directly) waiting for t_j if $t_j \in B_i$. Therefore, any transaction t_k, for which $B_k \neq \phi$ is in a wait state. We may then formally define the wait relation $w \subseteq T \times T$ where T is the set of transactions, such that $(t_i, t_j) \in w$ iff $t_j \in B_i$.

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 $t \varepsilon T : (t,t) \varepsilon w^+$.

Maintaining w is trivial. 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] [Pur 70] steps, where n is the number of nodes in G_w and m the number of arcs, i.e., $n = |T|$, $m = |w|$. 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:

$$\begin{array}{lll} \text{Given } w, w^+, & & \text{calculate} \\ & w', w'^+ & \text{where} \\ w' = & w \cup \{(t_i, t_j)\} & \text{or} \\ w' = & w \ \{(t_i, t_j)\}. & \end{array}$$

Although it is 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 to 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^+.

Fortunately, we have a very special case yielding a simple on-line transitive closure algorithm. Since we remove deadlocks immediately the wait graph G_w is acyclic. Therefore, we can represent w^+ as an integer matrix M with the interpretation: M[t,u] := number of different paths in G_w from t to u. Then tw^+u iff M[t,u] > 0.

To insert or delete an arc from t to u in G_w update M as follows:

$$\begin{aligned} M[s,v] &:= M[s,v] \pm M[s,t].\ M[u,v] &&; s \neq t;\ u \neq v \\ M[s,u] &:= M[s,u] \pm M[s,t] &&; s \neq t \\ M[t,v] &:= M[t,v] \pm M[u,v] &&; u \neq v \\ M[t,u] &:= M[t,u] \pm 1 \end{aligned}$$

It is clear that the worst case complexity of this on-line update of w^+ for inserting or deleting a single arc in G_w is $O(n^2)$. This algorithm can easily be modified to have an average complexity of $O((\frac{|w^+|}{n})^2)$.

To discover deadlock, tentatively update M when inserting an arc. Then check the diagonal of M for the existence of a t such that M[t,t] > 0. If such a t exists, the insertion of the arc caused a deadlock. The action to be taken to break the deadlock is described in section 6. The tentative update of M is cancelled. If a deadlock does not exist make the tentative update of M definite.

Another highly special case arises when w^+ and therefore M must be updated at the end of a transaction t when t releases all its locks. Since t was executing, it was not waiting for other resources, and therefore it is a sink of G_w. t and all arcs leading into t can be removed from G_w. w^+ is now simply updated by removing a row and a column from M if necessary, a very simple operation. A simililarly simple update of M is performed when granting a lock request to t. For further details on on-line transitive closure algorithms see also [Bay 75].

6. BREAKING A DEADLOCK

Assume that adding t to the end of Q(r) and updating w and w^+ accordingly would cause a deadlock. Then one of the following actions can be taken to break the deadlock.

6.1: Move t forward in Q(r): Since t issued a lock-request before causing the deadlock, t was not waiting. Therefore moving t forward in Q(r) reduces the number of transactions t is waiting for, hereby the deadlock may be broken. The deadlock will definitely be broken, if t can be added to the granted group G(r). Note that this strategy overrides the FIFO rule for servicing lock-requests for the purpose of breaking deadlocks only. Note also that the available w^+ can be used to find out very easily, how far t should be moved forward in Q(r) in order to break the deadlock.

6.2: Backup t: Note that all cycles in G_w after tentatively appending t to Q(r) and updating w^+ pass through t. Thus backing up t towards the beginning of its seize phase, taking away resources previously granted to t in the opposite order, in which they were granted, will eventually break all cycles in G_w.

6.3: Minimal-cost node cut set: Find a cheapest node cut set breaking all cycles in G_w and back up all transactions in this cut set to the beginning of their seize phases, thus taking away all their resources. This technique may be attractive, if the cost of backing up and rerunning the seize phase of a transaction is available and is very high.

7. PREVENTING INDEFINITE DELAY

The methods just described can cope with the deadlock problem, but they do not guarantee that a transaction t will eventually reach its execution phase and complete. t might be prevented from execution by being backed up again and again.

The relation w^+ can be used advantageously to cope with indefinite delay. Let us assume that all transactions in the system are time-sttamped. We can then use w^+ to partition the set T into priority classes $P_1, P_2, \ldots, P_k$ for some integer k. Let $\bar{t}_1$ be the oldest transaction in T, the set of transactions in the system. Define the highest priority class P_1 as follows: $P_1 := \{\bar{t}_1\} \cup \{t : \bar{t}_1 \; w^+ \; t\}$. In general, define the classes, $P_1, P_2, \ldots, P_k$ by:

$$\begin{aligned}
&U_o := \phi \\
&\bar{t}_i := \text{oldest transaction in } T \setminus U_{i-1} \\
&P_i := \{\bar{t}_i\} \cup \{t : \bar{t}_i \ w^+ \ t\} \setminus U_{i-1} \\
&U_i := U_{i-1} \cup P_i \\
&\text{for } i = 1,2,\ldots,k \text{ where } U_k = T
\end{aligned}$$

We now propose four increasingly effective, but also increasingly radical strategies for preventing indefinite delay. It seems quite feasible 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: Use the priority classes for scheduling, starting with the highest priority class P_1. This strategy will tend to move older transactions through the system faster.

Strategy 2: Stop all transactions in seize phases, except those in P_1.

Strategy 3: Let $\bar{t}_1$ preempt r from $t \in P_1$ if $\bar{t}_1$ is directly waiting for t unless t is executing, i.e., move $\bar{t}_1$ ahead of t in Q(r) or replace t in G(r) by $\bar{t}_1$.

Strategy 4: Stop all t which are not in their execution phases, apply strategy 3.

Strategy 5: When breaking a deadlock do not allow t to pass $\bar{t}_1$ in 6.1, do not backup $\bar{t}_1$ in 6.2 or 6.3, but backup some cutset of transactions (not necessarily the cheapest) not containing $\bar{t}_1$.

It is clear that all five strategies will tend to bring $\bar{t}_1$ closer to its execution phase. Strategies 1 and 3 might still allow indefinite delay in very special circumstances, but it is easy to construct plausibility arguments that strategies 2, 4, and 5 do prevent indefinite delay.

8. PHANTOMS AND PREDICATE LOCKING

In [EGLT 74] a technique is described to use 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 $\mathscr{D}$ 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 $B \subseteq \mathscr{D}$. 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 $r_1 \varepsilon \mathscr{D}$ to B and t_2 may add a new object $r_2 \varepsilon \mathscr{D}$ 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. It is clear that individual object locking as described so far does not solve the phantom problem.

The appearance of a single phantom, say r_1, does not cause any difficulty. The parallel schedule $\{t_1,t_2\}$ has the same effect as running the transactions t_1,t_2 serially, namely in the order t_2 followed by t_1, therefore, $\{t_1,t_2\}$ is consistent. It is the goal of predicate locking to schedule transactions in parallel as far as possible under the restriction, that consistency is preserved.

To enforce consistent schedules each transaction t is required to lock (for read or write access) all data objects $E(t) \subseteq \mathscr{D}$ - 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 $\mathscr{D}$ (or on a part of $\mathscr{D}$, e.g., on a relation [Cod 70]) such that $E(t) \subseteq S(P)$ where S(P) is the subset of elements of $\mathscr{D}$ 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 $\exists r \varepsilon S(P_1) \cap S(P_2)$ 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 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 are 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 "$S(P_1) \cap S(P_2) = \phi$" may be very hard. For first order predicates this problem is undecidable. Thus for practical applications and a given $\mathscr{D}$ 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].

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 databases.

9. LOCK MODES AND PROTOCOLS FOR A PARTITIONED DATABASE

Let us start with an important observation which will lead to the simple strategy 1 for handling phantoms: "Transactions, which are readers, do not need to lock phantoms." A transaction is a reader, if it does not contain any write actions, it is a writer otherwise. Obviously for many database applications the readers are a very important class of transactions.

To understand our observation, consider two readers t_1, t_2 first. Since there are no write actions, there is no possibility for phantoms to materialize, and they need not be locked. Now let t_3 be a writer. Consider the interaction between t_1

and t_3. Assume there is a phantom $r \varepsilon S(P_1) \cap S(P_3)$ 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 obviously $\{t_1,t_3\} \equiv (t_1,t_3)$.

The following observation should also be clear now: To control the interaction between the writer t_3 and the reader t_1 if suffices, that t_3 use individual object locking according to [CBT 74]. t_3 need not lock its phantoms except as they are materializing since t_1 is not interested in phantoms anyway. We can conclude that the problem of phantoms - and therefore of predicate locking - arises only between writers. The preceding observations suggest several alternative approaches for handling the phantom problem:

<u>Strategy 1</u>: <u>Serialize Writers</u>:

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 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 readers. Serializing writers in those cases should not cause a significant loss of concurrency and has the advantage that predicate locking with its associated difficulties is not needed. Several more involved strategies are described in [Bay 75].

Strategy 2: Partition Database:

The following approach can be thought of as a highly specialized and simplified predicate locking technique, although predicates are no longer used explicitely. The technique allows a high degree of concurrency between transactions, and avoids the phantom problem, if all transactions use the proper locking protocols. A similar technique is described in [GLP 75], [GLPT 75] and has been implemented in System R, an experimental relational database system [ABC 76]. For explanatory purposes we first describe a simplified case, a sketch of the general case should then be easy to understand.

Assume that our database B is partitioned into a finite number of blocks $B_1, B_2, \ldots, B_k$. Each object of B belongs to a unique block. When inserting, deleting or modifying an object in B, we assume that the block or blocks affected by such an update are known. Partitioning of databases, e.g., into files or relations, is a widely used technique. In terms of predicate locking, the sets $B_1, B_2, \ldots, B_k$ can be thought of as:

$$\begin{aligned} B_1 &= B \cap S(P_1) \\ B_2 &= B \cap S(P_2) \\ &\vdots \\ B_k &= B \cap S(P_k) \end{aligned}$$

for a fixed set of predicates $P_1, P_2, \ldots, P_k$ having the special property that for all possible database contents B and blocks B_i, B_j, $i \neq j$ it is true that

$$B_i \cap B_j = (B \cap S(P_i)) \cap (B \cap S(P_j)) = B \cap S(P_i) \cap S(P_j) = \phi.$$

Now assume that we can lock the blocks $B_1,\ldots,B_k$ and also individual objects of B in various lock modes as we shall see suitable shortly.

Partition Locking: Obviously a transaction t could then guard itself against phantoms which might materialize in B_i by simply locking B_i in exclusive mode X. It suffices, however, that t lock B_i in a mode preventing other updaters from modifying B_i in any way while still allowing objects of B_i to be read by t and by other transactions. A shared lockmode S allowing read access to all objects in a partition block serves that purpose as long as locks used for update purposes on B_i are not compatible with this lock.

In addition to allowing X- and S-locks on blocks, we also introduce an analysis-lock, called an A-lock. This A-lock can be used by a transaction t while analyzing a block with the intent to update it. Also a new version of the block can already be prepared while still maintaining the old version available for read access to other transactions.

During this analysis phase no phantoms may be created by t or other transactions. Thus read accesses, but no update accesses by other transactions are allowed. Furthermore, no analysis accesses by other transactions can be allowed since the intended update of B_i by t would invalidate such analyses.

To finally perform the update, or to finalize (commit) an already prepared update for B_i, first convert the A-lock into an X-lock. Thus the update cannot be committed, until all S-locks which might coexist with the A-lock on a block have been released. From the intended use of the locks it is now clear that the compatibility matrix for the three lock-modes S,A,X should be defined as follows:

$$\begin{array}{c|ccc} & S & A & X \\ \hline S & + & + & - \\ A & + & - & - \\ X & - & - & - \end{array}$$

+ means compatible

- means incompatible

Fig. 1: Compatibility Matrix for Lock Modes S,A,X.

Data Object Locking: Locking at such a coarse level as the partition blocks may cause an inacceptable decrease of concurrency since much larger parts of the database than really necessary might have to be locked. Although it requires more overhead in setting locks it is, therefore, desirable to be able to lock at a finer level, namely individual data objects, in order to increase potential concurrency of transactions. This is expecially true for transactions, e.g., readers as we saw before, which are not concerned about phantoms and which can run consistently, as long as they are able to lock all objects in the database, in which they are interested, in the proper mode.

In addition to readers there are many writers which have additional knowledge about the semantics of the database and need not be concerned with guarding themselves against phantoms in order to obtain consistency. Therefore, they do not need to lock out other transactions at the block level.

In order to make partition locking still work it must be known what sort of object locking is going on within a block. This can be achieved by leaving certain traces at the block level [Ram 74] or equivalently by introducing additional lock modes at the block level [GLP 75], [GLPT 75]. These lock-modes are:

IS: (intention share) to indicate that a transaction intends to set S-locks on objects within the block locked with IS.

IX: (intention exclusive) to indicate the intention for setting S-locks or X-locks.

SIX: (shared, intention exclusive) to guard against phantoms within the block and thereby getting (without further object locking) read access to all objects in the block, and to indicate the intention to set X-locks on objects in the block locked with SIX.

Note: It is not necessary to set S-locks on objects in a block already SIX-locked. Also there is no need for an SIS-lock (with the obvious meaning) since SIX would imply read access to all elements within the block already without setting further object locks. Thus, using S instead of SIS serves the same purpose.

These considerations then lead to the following system of locks which is similar to the one arrived at in [GLP 75] with a somewhat different motivation than the one given here. This lock system with appropriate protocols serves to solve the phantom problem, allows locking at the proper granularity - to allow a trade-off between overhead work for setting locks and the degree of concurrency achievable - and still allows highly concurrent access to a partitioned database using a relatively simple locking protocol.

Locks for Objects: The objects in a block can be locked in one of the following modes:

a) S-mode for concurrent read access

b) X-mode for exclusive access for read or write purposes.

c) although the A-mode could be allowed, it seems hardly worthwhile and is omitted here.

Locks for Partition Blocks: The partition blocks can be locked in any of the modes S,A,X,IS,IX,SIX. Locking at the block level really serves two rather different purposes:

1) to guard against phantoms

2) to avoid the overhead of setting many locks on objects, thereby potentially accepting a decrease in concurrency.

To perform the actual update of a block after an analysis, t can follow two courses:

1) Convert A to X as described before, then update.

2) Convert A to SIX and individually X-lock those objects in the block which must be updated.

We can now expand the compatibility matrix of Fig. 1 by adding the lock modes IS, IX, and SIX. From the meaning of those modes described before it should be clear that IS must be incompatible with X only, but can be compatible with S, A, IS. An IX-lock is used to indicate that X-locks are set at the object level to perform updates and possibly to create phantoms. Therefore IX must be incompatible with S, A, X, it can be compatible with IS and IX only. An SIX lock is used to guard against phantoms and to set X-locks on objects for updates. Thus SIX must be incompatible with S, A, X, IX, and SIX, it is only compatible with IS. Since compatibility is symmetric, we obtain the complete compatibility matrix of Fig. 2.

	S	X	A	IS	IX	SIX
S	+	-	+	+	-	-
X	-	-	-	-	-	-
A	+	-	-	+	-	-
IS	+	-	+	+	+	+
IX	-	-	-	+	+	-
SIX	-	-	-	+	-	-

Fig. 2: The Compatibility Matrix for the Modes S, X, A, IS, IX, SIX.

The following combinations of locks are then obviously meaningful and constitute allowed lock protocols:

lock held on partition block:	meaningful locks for objects:
S	-
X	-
A	-
IS	S
IX	S,X
SIX	X

Generalization to Hierarchy: In a next step one can now iterate the scheme and use a sequence of partitions $\Pi_1,\Pi_2,\ldots,\Pi_\ell$ such that Π_{i+1} is a refinement of Π_i, i.e., Π_{i+1} splits the blocks of Π_i into smaller blocks. This leads to a database, which for locking purposes is hierarchically organized. The hierarchy describes which blocks are subsets of other blocks. To prevent phantoms and gain read access within a partition block one of the locks S, A, X, and SIX can be used. SIX, IX, and IS are used to indicate what locks may be set at the next level of refinement. It is now prudent to also add the additional lock modes IA and SIA with the following extensions of the compatiblity matrix:

	S	X	A	IS	IX	SIX	IA	SIA
IA	+	-	-	+	+	-	+	-
SIA	+	-	-	+	-	-	-	-

Fig. 3: The Compatibility of the Modes IA and SIA

Before t converts an A-lock on a block B_i to an X-lock it must convert all IA-locks and SIA-locks it holds on ancestors of B_i to IX- or SIX-locks, respectively. To avoid potential deadlocks due to conversions this conversion should be performed top-down in the hierarchy, obviously the conversions of A to X, of IA to IX, and of SIA to SIX cannot be granted until any coexisting S-locks have been released.

Note: The two lock-modes A and SIA have the same compatibilities, but they differ in the way they should be converted. Thus SIA could be omitted allowing the conversion of A to both SIX or X after finding out which conversion is preferable.

The following table describes the generalized lock protocols, assuming locks are set top-down in the hierarchy:

locks for partition blocks	allowable on next level	allowable on object level
S	-	
X	-	
A	-	
IS	S, IS	S
IX	all modes	S,X,A
SIX	X,A,IX,IA	X,A
IA	S,A,IS,IA,SIA	S,A
SIA	A,IA	A

Note: There would be no difficulty to allow S, IS, SIX, SIA locks on sons of a block with an SIX-lock, or to allow S, IS, SIA locks on the sons of a block with an SIA-lock.

The locking techniques described in this section offer a means to handle the phantom problem. The issue has been investigated in more detail, but without using the A, IA, and SIA modes in [GLP 75] and [GLPT 75]. In a partitioned database transactions can choose to lock at a coarse or a fine level depending on the desired tradeoff between low locking overhead and high concurrency.

The issue of potential deadlock still prevails, unless additional restrictions on locking protocols are imposed [Ram 74]. The techniques described in sections 5, 6, 7 can be used to handle the deadlock issue.

10. RECOVERY

Backup of a database to an earlier state of integrity should be planned for and may become necessary for many reasons: hardware failure, system failure due to program errors, deadlock and deadlock resolution, prevention of indefinite delay, violation of integrity constraints discovered during or at the end of a transaction, attempt to perform unauthorized operations.

We will present the basic design principles, ignoring many technical details, for a revocery scheme. It is assumed that a safe pseudo-random-access store, e.g., a disk containing the database is available and that recovery of a previous state of integrity must be possible from the contents of the backup store alone, if one of the above failures including failure of the central processor or main memory occurs.

We assume that such a failure has precisely the same effect as halting the operation of the whole system at an arbitrary moment resulting in an undefined state of the central processor and main memory. Any read or write operations in progress on the backup store shall also halt, without having any destructive side effects on the state of the backup-store. Only the contents of the backup-store shall be useable to determine, how far such operations might have progressed at the moment of the system halt.

The recovery problem to be solved then can be summarized as follows: The effect of incomplete transactions, i.e., transactions which had started but not yet completed at the moment of failure, on the contents of the database must be undone.

To describe the update of a single partition block, let us assume that the whole block will be rewritten. In a technical implementation the unit of information for read and write operations on the backup-store will often be a page. The basic recovery principle can be adopted to this case.

The key of the recovery scheme is the notion of a shadow block [Lor 76]. When a transaction updates a block, the old block - called the shadow - will be kept for the contingency that a failure occurs before the transaction completes and that the shadow is needed for recovery. The shadow will be released only after the new updated block has been properly constructed on the backup store. The main difficulty of such a scheme is the design of the update-commitment operation which performs the switchover from the shadow to the new block. This operation itself may fail and recovery must still be possible.

After a new block has been constructed by an updating transaction two versions of a block reside on the backup store, each representing part of a state of integrity of the database. To perform the update-commitment, the transaction must first assure, that the shadow is no longer used by other transactions. This can be done by either performing updates under an X-lock on the block or by performing updates with an A-lock and converting the A-lock to an X-lock as part of the commitment. After obtaining an X-lock on the block, the new block can then be validated and the shadow can be freed.

Since at any time we may have two versions of a block, we need two address maps defining for each block the physical locations of the shadow and the new block on the backup store. These maps must themselves reside on the backup-store. Let us assume that each map entry is time-stamped, e.g., with the system-time of the moment of creation of the entry. Also associated with each map entry shall be a separately writeable Boolean validation variable. A value true shall mean that the corresponding physical block represents part of a state of integrity of the database. The timestamp of the map entry can be used to identify that state.

With these preliminaries we can now introduce the data objects of the commitment algorithm to be performed by a transaction t_j to update a block B_i:

FL: A list of free storage areas to place new blocks. The details of FL are irrelevant here. FL can be in main or backup store, FL is not needed for recovery purposes.

VO,VI: These are two arrays located on the backup store to represent the two address maps and the timestamps for the physical blocks. VO[i] and VI[i] contain the addresses and timestamps of the shadow and the new block of B_i.

GO,GI: Two Boolean arrays located on the backup store to represent the two validation variables for each block.

Vc: An address map located in main or backup store. Vc[i] contains the address of the currently valid version of block B_i which is made available for read access to other transactions while B_i may be updated by t_j. Vc is not needed for recovery.

To read the block B_i a transaction requests an S-lock on B_i and holds this lock for as long as repeatable reads and the prevention of phantoms in B_i are desired. To update B_i a transaction can request an X-lock on B_i before starting to construct the update. Alternatively, since the shadow of B_i is available until the update is committed, the shadow can be left available for read access while the update is being constructed by requesting an A-lock on B_i first. To commit the update the A-lock is then converted to X. We present the second more complicated update protocol in detail:

1) Place A-lock on B_i;
2) Get free slot for new B_i via FL;
3) Prepare and write new block B_i;
4) Update and timestamp VO[i] or VI[i];
5) Validate GO[i] or GI[i];
6) Invalidate GI[i] or GO[i];
7) Convert A-lock on B_i to X-lock;
8) Update Vc[i] with address of new B_i;
9) Release X-lock on B_i;
10) Free shadow and update FL

Fig. 4: Locking Protocol for an Updater

The main desirable properties of this update protocol are:

a) For most of the duration of the update block B_i can still be read by other transactions, or equivalently, an update can already be started while the old version of B_i is still locked for read access.

b) An exclusive lock must be held only in steps 7), 8), 9). These steps are very fast, since no operations on backup store are involved.

c) Except for steps 2) and 10), parts of which may have to be programmed as critical regions, updates of different blocks can be performed concurrently, unless prevented by other locks in the hierarchy.

d) The technique of using the shadow blocks for recovery causes only a very small disturbance of the overall operation. No quiescing of the system for taking checkpoints is needed and no extra operations are needed on the backup store beyond the maintenance of the structures VO, VI, GO, and GI.

e) Backup of a single transaction, e.g., because of an integrity violation or a deadlock, although not discussed here in detail, is easily possible without stopping or affecting other transactions.

Recovery: Now let us consider the recovery problem assuming that the system can fail, i.e., halt its operation with an undefined state of the processor and the main memory, at any moment of the update protocol.

Case 1: A failure occurs before step 5). Then exactly one of GO[i] or GI[i] will be true. The corresponding address map entry VO[i] or VI[i] will point to a valid version of block B_i representing a state of integrity.

Case 2: If step 5) itself fails we will not know whether the validation took place or not, and we have two cases:

Case 2a: The validation was not performed, i.e., only one of GO[i] or GI[i] is true and we continue the recovery as in Case 1.

Case 2b: The validation was performed, i.e., both GO[i] and GI[i] are true, we

have two valid versions of B_i and can use the timestamps of VO[i] and VI[i] to determine the newest version to be used for recovery.
Since step 5) consists of writing only a single bit we can assume that only the two cases 2a and 2b can arise.
Case 3: Step 6) fails: depending on how step 6) fails, there will be one or two validated versions of B_i and recovery is performed exactly as in Case 2.
Case 4: A failure occurs after step 6). There will be exactly one validated version of B_i on the backup store which is used for recovery.

To conclude we want to indicate some of the additional complications arising in a viable implementation of the techniques presented.

1) The lock protocol presented in Fig. 4 must be generalized to handle updating of several blocks within one transaction. This is easily possible.
2) The hardware of some machines allows to combine the write operations of steps 3) and 4) and the validation of step 5) in a single operation. The update protocol can be modified to take advantage of this possibility [Lor 76].
3) When updating several blocks a generalized conversion operation is desirable to convert a set of A-locks in a single operation rather than converting A-locks one at a time. Such an operation can increase concurrency considerably.
4) Instead of backing up a transaction completely for recovery it may be desirable to introduce additional intermediate safe points and to use partial backup for recovery. Additional bookkeeping is then necessary to inform transactions, how far they have been backed up and how much work must be repeated [Lor 76] [ABC 76].
5) An additional level of recovery is needed in reality to guard against failures involving the backup store. The techniques presented here can be generalized to serve that purpose [Lor 76].

REFERENCES

[ABC 76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade and V. Watson, "System R: A Relational Approach to Database Management" to appear in ACM Transactions on Database Systems, 1, 2 (1976).

[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, September 1974.

[Bay 75] Bayer, R., "On the Integrity of Databases and Resource Locking." In: Data Base Systems (ed. Hasselmeier, H. and Spruth, W. G.), Lecture Notes in Computer Science, 39, Springer, 1976.

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

[CBT 73] Chamberlin, D. D., Boyce, R. F., Traiger, I. L., "A Deadlock-free Scheme for Resource Locking in a Database 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." ACM 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 Database System." IBM Research Report RJ 1487, December 1974.

[ESC 75] Eswaran, K. P., Chamberlin, D. D., "Functional Specifications of a Subsystem for Database Integrity. "IBM Research Report RJ 1601, June 1975.

[Eve 74] Everest, G. C., "Concurrent Update Control and Database Integrity." In: Database 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 Database Management System." In: Database Management (ed. Klimbie, J. W., and Koffman, K. L.), North Holland 1974, 271-288.

[GLP 75] Gray, J. N., Lorie, R. A., Putzolu, G. R., "Granularity of Locks in a Large Shared Database." IBM Research Report RJ 1606, June 1975.

[GLPT75] Gray, J. N., Lorie, R. A., Putzolu, G. R., Traiger, I. L., "Granularity of Locks and Degrees of Consistency in a Shared Database." IBM Research Report RJ 1654, September 1975.

[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.

[LiZ 74] Liskov, B. H., Zilles, S. N., "Progrmaming with Abstract Data Types" ACM Sigplan Notices, 9, 4 (April 1974), 50-59.

[Lor 76] Lorie, R. A., "Physical Integrity in a Large Segmented Database." IBM Research Report RJ 1767, April 1976.

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

[Pur 70] Purdom, P., Jr., "A Transitive Closure Algorithm." Bit 10 (1970), 76-94.

[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 1976), 11-12.