

A Review of Theoretical Methodologies
for Locking in a
Concurrent Data Base Environment

by

Alice Ruth Galinat

B.S. Southeastern Massachusetts University, Dartmouth, 1976

A Masters Report

Submitted in partial fulfillment of the

requirements for the degree

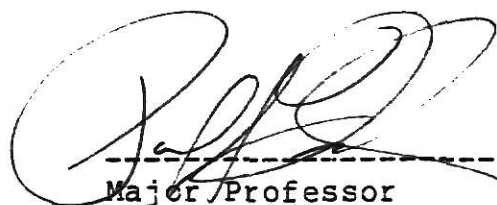
MASTER OF SCIENCE

Department of Computer Science

Kansas State University
Manhattan, Kansas

1980

Approved by:



Major Professor

Spec. Coll.
LD
2668
.R4
1980
G34
c.2

page ii

Table of Contents

	Page
List of Tables.....	iii
List of Figures.....	iv
Chapter	
1. Introduction.....	1
2. The Complications of Using a Data Base as a System Resource.....	3
3. Data Base Access, Serial Schedules, and Integrity.....	9
3.1 Data Base Access and Serialization.....	9
3.2 The Added Integrity Problems Caused by Predicate Locks.....	22
3.3 Types of Integrity and Consistency.....	26
4. Locking Protocols.....	30
4.1 Physical Locking Protocols Without Deadlock Prevention.....	31
4.2 Physical Locking Protocols With Deadlock Prevention.....	37
4.3 Predicate Locking Protocols.....	45
4.4 Concurrent Access to a File of Records.....	53
5. Conclusion.....	58
References.....	66

List of Tables

Table	Page
1. The Compatibility of the Lock Modes in [GRAY75]'s Protocol.....	33
2. The Compatibility of Lock Modes For [BAYE76]'S Protocol.....	45
3. An Extension of Table 1 for [GRAY75]'s Protocol.....	47
4. The Extension of A, IA, and SIA Modes to Table 1.....	47
5. Comparison of Lock Protocols.....	60

List of Figures

Figure	Page
1. Two Simple Transactions.....	15
2. The Dependency Graph of a Simple Non-Serial Schedule.....	15
3. Three Arbitrary Transactions.....	17
4. An Arbitrary Schedule for the Three Transactions in Figure 3.....	18
5. The Dependency Graph for the Arbitrary Schedule in Figure 4.....	19
6. The Schedule and Lock Results of an Arbitrary Schedule Using Predicate Locks.....	23
7. The Weighted Dependency Graph of the Arbitrary Schedule in Figure 6.....	24
8. The Lock Hierarchy Given for the Protocol in [GRAY75].....	31
9. The Ordering of Privileges Between Lock Modes for [GRAY75]'s Protocol.....	34
10. The Partitioning of a File From a Lock Protocol Designed to Handle Predicate Locks [GRAY75].....	35
11. The Schedule of Two Transactions that does not Maintain Integrity in a Weak Sense.....	40
12. The Dependency Graph of the Schedule Given in Figure 11.....	41
13. A Hierarchy of Records for Locking.....	42

14.	An Acceptable Locking Sequence for a Transaction Accessing Records with Figure 13's Hierarchy.....	43
15.	An Example of a B-tree With $N = 4$	53
16.	Key for Table 5.....	61

Chapter 1

Introduction

A data base is a system resource like other system resources that has to be managed by a data base management system (DBMS) or an operating system. Obviously there are differences between a data base and conventional system resources that have to be considered to achieve acceptable levels of resource utilization, throughput, and response time. These differences are discussed in the second chapter. The traditional solutions used for resource allocation are not adequate for a data base, and their problems are discussed. Some of the problems to avoid when creating new solutions are also discussed.

Unlike conventional system resources it is possible to separate a data base into distinct pieces that are treated as separate resources to allow concurrency. This leads to many more problems. The integrity of the data base is threatened by concurrent access. This is the subject of chapter 3. A dependency graph is defined to help study the integrity problems created by concurrent access, and two kinds of integrity are defined from the dependency graph.

The added integrity problems created by using predicate locks instead of physical locks is also discussed. The dependency graph is modified to help study these added problems.

Different levels of integrity are defined at the end of this chapter based on what a transaction (process or user) may do to specified data. The usefulness of the different levels for different kinds of applications is suggested.

Algorithms developed to handle these problems at different integrity levels are presented in chapter 4. The amount of concurrency possible and the overhead involved in each algorithm are discussed. These algorithms are designed for a central data base holding one copy of the data. The supplemental algorithms needed by some locking protocols to detect deadlock and the subject of restart and rollback are not considered.

In the last chapter, chapter 5, some simulation studies concerning the efficiency and concurrency of locking protocols are reviewed in terms of the overhead involved for the various algorithms. The protocols are also compared.

Chapter 2

The Complications of Using a Data Base as a System Resource

The problems of allocating data from a data base as a system resource are not present in a normal operating system environment and are explained below as suggested by Chamberlin [CHAM74]. Assume a record is a resource.

- Non-unique resource names: A record can usually be identified by more than one data item's value. This causes confusion as to whether a given record is already locked. For example, suppose a user wanted to obtain exclusive access to records of students who are in the computer science department. Another user then wanted exclusive access to all students taking calculus I. It is obvious that there is a possibility of the users' requests overlapping. This may not be easily determined by the system managing the data base as several system resources.

- Non-static resource categories: A process with exclusive access to a record may change the value of one or more of the record's fields and cause this record to move to a group of records currently locked by a different process.

For example, suppose a user is adding calculus I to particular students' schedules and at this same time another user is accessing all students with calculus I on their schedules. Should this user see all the students with calculus I being added to their schedule or just the ones added so far or none of the new calculus I students?

Records are also added to and deleted from a data base. This, of course, changes those records which other processes may want to access. Schlageter [SCHL78] calls this the variable object set.

- Interdependent locks: A process may wish to have exclusive access to some records and, based on information contained in these records, may or may not decide to obtain exclusive access to other records. For example, suppose a user looks at records containing calculus I class rosters. He may then want exclusive access to the individual student records to compute class statistics. The student records may be temporarily inaccessible because another process is using them. The possibility of deadlock is created when interdependent locks are involved and all the data with the potential of being accessed is not locked before execution.

- Increased complexity: The smaller the granularity of the lockable units in the data base, the greater the concurrency potential. This means that the records or even the data fields of the records could be treated as resources or lockable units. This can mean that literally millions of

resources potentially could be locked. The problem of resource allocation from within a data base is more difficult than allocation of other system resources.

The above problems and the nature of data give severe limitations to the traditional solutions used by operating systems for resource allocation. The limitations for some of these solutions in a data base ([BAYR75], [BAYR76], [EVER74]) are outlined below.

- Presequence processes: Processes that are competing for resources are presequenced so that conflict does not develop between them. This can be done by a DBMS when it is known in advance what resources will be needed. Often there is a lot of data that has the potential of being used by a given process. If it is all locked for the duration of the job, concurrency is severely limited. The potential of a large amount of data being accessed by a given process is partly caused by the problem of interdependent locks.

- Preempt Processes: Processes are not prevented from causing deadlock. A mechanism is used to detect deadlock, and to determine which process or processes should be backed out. This procedure is considerably more complicated in a data base because of the large number of resources. Also the integrity of the data base must be maintained when backing a process out which may be very difficult or even impossible without terminating all processing and backing out all potentially interacting processes.

- Preorder all system resources: Resources are claimed in a predefined order. Often data does not have a natural order. Processes may find it difficult to claim resources in a predefined order especially when interdependent locks and non-static resource categories are involved.

- Preclaim needed resources: A process must claim all needed resources before execution. If this is done in a DBMS then it may reduce concurrency and modifications are needed to increase concurrency. The inherent problems of interdependent locks and increased complexity reduces the potential concurrency.

- Deadlock prevention algorithms: These algorithms sometimes can be modified for a DBMS. Preclaiming resources can fall into this category as well as certain algorithms which require the user to follow a certain preorder imposed on the data base. Some of these algorithms are discussed later.

The new techniques used to overcome the obstacles created by concurrent access to a data base must deal with the problems outlined below. [GARD77]

- Lost updates: Updates made to the data base can be lost when two processes are updating the same data item. This can happen when a process reads a data item into its own buffer and another process reads this same data item into another buffer before the first process has updated the original copy. The first process then updates the original

value and the second process updates the original value too. This allows the second process to overwrite the update of the first process and consequently the first update is permanently lost.

- Non-reproducible reads: This can happen when a process reads a data item and later reads it again after the item has been changed by another process.

- Identity confusion: The address or key of a data item (record) is still in memory a short time after the corresponding data item (record) has been deleted or moved.

The opposite condition can happen when a process deletes a pointer immediately after another process has read it. This other process may read invalid data because it accesses free space under the assumption it is valid data [BERN69].

This can also happen when there are two processes accessing different records on the same page. One of the processes may invoke a data compaction routine without any regard for the other process [ASRT76]. This can cause the other process to unknowingly access the wrong data.

- Inconsistent observation: A process computes and outputs inconsistent data because, during the time interval it is retrieving the data, another process changes some of the data it has not yet retrieved. This process sees several different states or a transitional state of the data base.

- Loss of consistency: A process modifies a transitionary state of the data base so it is not possible to back a process out to a consistent state without a complete backup and restore.

Chapter 3

Data Base Access, Serial Schedules, and Integrity

In this section serialization will be examined as a means of preserving integrity during concurrent access. A dependency graph will be introduced to help study serialized schedules. General approaches to lock protocols used to insure integrity will be introduced without very much regard to the granularity or the size of the lockable units. Terms will be defined which can be used on a wide range of granularity sizes. Several levels of consistency for a data base will be defined based on the amount of concurrent access allowed between transactions operating on overlapping sets of data.

3.1 Data Base Access and Serialization

When a user accesses a data base the user assumes that all the data read or updated is not in a transitionary state and does not contain contradicting information. Certain

assertions or consistency constraints about the data are assumed by the user even though they are not all at a conscious level or defined by the DBMS. If the data base starts out consistent then it is not necessary for the consistency constraints to be defined provided every user sees a snapshot view of the data base and is constrained in interacting with it in a manner that will not create inconsistencies. These requirements must be implemented by an operating system or DBMS that can oversee the users.

This overseer function assumes each user maintains these assertions when accessing the data base alone and therefore does not threaten the integrity of the data base. A user's job may be broken up into separate sections that maintain consistency when run alone. These groups of actions that maintain consistency are called transactions ([ESWA74],[ESWA76]). It is not necessary and not always possible for a transaction to keep a consistent state during execution. As an example, suppose some money is transferred from one bank account to another. The total amount of money in both accounts before and after the transfer is the same. This is the consistency constraint. However there is a time when one account has been reduced and the other not incremented. At this time the total amount of money in both accounts is not equal to the initial or final total. This is a necessary temporary inconsistency [ESWSA76]. During this period of inconsistency, other users must be prevented

from accessing these accounts. This is usually done by a locking mechanism in the DBMS.

When transactions are run one after the other or in a serial manner the data base obviously has no inconsistencies due to concurrency, so a natural way of maintaining consistency is to force the transactions to access the data base in a manner that is equivalent to serial access.

The definitions that follow are defined for individual entities but also apply to larger lockable units in this context.

Formally ([ESWA74],[ESWA76]), a transaction T is a sequence

$$T = ((t, a_i, e_i)) \text{ for } i = 1 \text{ to } N$$

where

t = transaction name

a_i = the action performed by step i

e_i = the entity (data item or record) that action a_i operates on.

A schedule S ([ESWA74],[ESWA76]) taken from a set of transactions $T = (T_1, \dots, T_n)$ is any sequence

$$S = ((t_i, a_i, e_i)) \text{ for } i = 1 \text{ to } m$$

for each $j = 1, \dots, n$

$$T_j = ((t_i, a_i, e_i) \in S \mid t_i = t_j) \text{ for } i = 1 \text{ to } m.$$

Note that the order of the actions within each transaction is preserved. Also note that the length of M is the sum of all the n lengths from the transactions.

If a transaction does not access any data items accessed by another transaction there is no need to be concerned about the transactions being run concurrently. When two transactions have data in common, each transaction should see either all the changes made by the other transaction or none. This is the same as running them serially. A dependency relation between transactions [ESWA76] shows the order the transactions, with overlapping sets of entities, are performed. Formally this dependency relation, caused by schedule S , is a ternary relation on $T \times E \times T$, and is defined by $(T_1, e, T_2) \in \text{Dep}(S)$ iff for some $i < j$

$$S = (\dots, (T_1, a_i, e), \dots, (T_2, a_j, e))$$

and there is no k such that $i < k < j$ and $e_k = e$

where

T is the set of all transactions in S

E is the set of all entities

e denotes a member of set E .

If a schedule has entity e accessed by more than one transaction the order of these actions on entity e is given by the dependency relation. If there are no dependency relations then the transactions' data sets are disjoint and any schedule obviously gives results equal to the results of a serial schedule. Two schedules are said to be equivalent if their dependency relations are the same [ESWA76]. If the dependency relation of a schedule is the same as the dependency relation of a serial schedule then the schedule

is said to be equivalent to a serial schedule or serialized [ESWA76].

Different serialized schedules give different final results and consequently have different dependency relations reflecting this. As an example, a transaction T1 may give a 5% cost of living raise to all employees. Another transaction T2 may give certain employees merit raises. The question of whether the merit raise should be included in the cost of living raise is for management to decide. But the DBMS or the operating system must produce a schedule with reproducible results and maintain some specified level of integrity when these transactions are run concurrently. Hawley et al [HAWL75] believes these processes should be executed with an arbitrary serial schedule. If the merit raise is given before the cost of living raise the payroll is different then if the cost of living raise is given before the merit raise. Obviously different serialized schedules will produce different results.

Schlageter [SCHL78] defines a schedule as equivalent to a serial schedule differently than Eswaran [ESWA76]. He requires an equivalent schedule's final output to the data base to be equal to the final output of a serial schedule and the output of each transaction to be equal to the output of the corresponding transaction in the serial schedule. Note that in this definition it is possible for the order of actions within a transaction not to be preserved and for

transactions to be intermixed in a nonserial way when there are commutative actions. However he only considers schedules that are equivalent to serial schedules without commutativity for simplicity. It would be very difficult to implement a system that checks for commutative operations and still insures serialized schedules.

A reduced dependency graph [SCHL78] is quite helpful in determining if a schedule is serialized and for studying serialization. Actions involving the same entity are ordered such that one is performed before the other. A conflict or $<$ relation is defined by a schedule from dependency relations. $T_i < T_j$ is read T_i conflict T_j if the dependency relation $(T_i, e, T_j) \in \text{Dep}(S)$ holds for $i < j$. A dependency graph consists of a set of nodes each representing the set of entities used by a transaction. There is an arc from node T_1 to node T_2 if there is a $<$ relation between them.

If all of the transactions act on disjoint sets of lockable units then there will be no arcs between nodes and any schedule is serialized. An arc represents overlapping sets of lockable units between transactions. If there is an arc from T_1 to T_2 and then another arc from T_2 to T_1 then the schedule is not serialized ([SCHL78],[SILB80]). For an example, transaction T_1 performed the action of adding 10 to entity 'a'. Then T_2 performed the action of multiplying entity 'a' by 1.5. Up to this point the schedule can be

viewed as being serialized. But when T1 acts again on entity 'a' it does not generally have the same value after T1's first action. This is obviously not a serialized schedule. The schedule and the dependency graph for the two simple transactions in Figure 1 are given in Figure 2 below.

T1	T2
T11 $a = a + 10$	T21 $a = a * 1.5$
T12 $a = a + 2$	

Figure 1.

Two Simple Transactions

Non-serialized
Schedule

Dependency Graph

T11
T21
T12

T21 < T12

<T1>

T11 < T21

<T2>

Figure 2.

The Dependency Graph of a Simple Non-serial Schedule

A transaction is well-formed if any entity that is locked is not released until a lock is never needed again on that entity by this transaction. Formally a transaction is well-formed [ESWA74,ESWA76]

```

    if for any step  $i = 1, \dots, n$ 
         $a_i = \text{lock entity } e$ 
    then for  $j = 1$  to  $i-1$ 
         $a_j \text{ not} = \text{lock } e$ 
    and if  $a_i \text{ not} = \text{lock}$  then
        for some  $j \quad 1 < j < i$ 
             $a_i = \text{lock } e$ 
    and for the last step  $n$  of schedule  $S$ 
         $a_n = \text{unlock } e(n)$ 

```

The above discussion did not take into account different kinds of actions performed on entities. If a transaction only reads certain entities there is no reason why another transaction can not read them too. Two actions operating on the same entity or data set are said to be in conflict if at least one of the actions is modifying the entity [SCHL78]. The relation $<$ can now be weakened to only those dependencies involving a conflict by this new definition. A less restrictive dependency graph can be drawn using this new $<$ relation.

Integrity in a weak sense (IWS) is defined [SCHL78] such that for any pair of transactions, T_1 and T_2 , where $T_1 < T_2$ then T_2 is not $< T_1$. This is the same as if there are

no cycles between any two nodes. IWS does not guarantee that a schedule is serialized [SCHL78]. To see this consider the three transactions in Figure 3 executing with the arbitrary schedule given in Figure 4.

T1	T2	T3
$B = B + 1$	$C = C * 2$	$E = E * 2$
$C = C + 2$	$D = D - 1$	$F = F + 1$
$A = A * 2$	$E = E + 1$	$A = A + 1$

Figure 3.

Three Arbitrary Transactions

Schedule 1

```
T11    B = B + 1
T12    C = C + 2
T21    C = C * 2
T22    D = D - 1
T23    E = E + 1
T31    E = E * 2
T32    F = F + 1
T33    A = A + 1
T13    A = A * 2
```

Figure 4.

An Arbitrary Schedule

For The Three Transaction in Figure 3

To see the problem when executing according to schedule 1, suppose transaction T1 starts to execute with $A = 1$, $B = 2$, and $C = 3$. The problem appears when the schedule gets down to executing the last action $A = A * 2$ because the value of 'A' is '2' due to step T33. This is not the original value of 'A' at the beginning of transaction 1. This schedule would cause T1 to output information based on a transitional state. T1 would not see a consistent data base.

The dependency relations for schedule 1 are then:

$T12 < T21$ c is passed to T21 from T12 without any

other actions performed on it.

T23 < T31 e is passed to T31 from T23 without any other actions performed on it.

T33 < T13 a is passed to T13 from T33 without any other actions performed on it.

This last action causes T1 to see a view of the data base that never existed all at one time. This inconsistency causes the dependency graph to contain a circular chain of dependencies or a cycle. The dependency graph is shown in Figure 5 below.

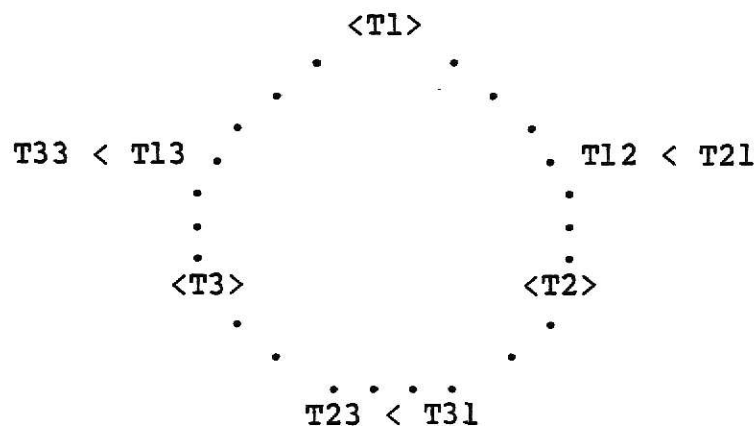


Figure 5.

The Dependency Graph

For the Arbitrary Schedule in Figure 4.

Integrity in the pure sense is insured when the dependency graph contains no cycles at all. In this case

the schedule is equivalent to a serial schedule.

One approach to locking mechanisms which can be used to prevent transactions from causing inconsistencies is the two-phased approach that was first used in a protocol by Chamberlin [CHAM74] and analyzed by Eswaran [ESWA74]. The first, or growing phase, requests all the needed locks on the data base. No data may be unlocked by the locking transaction during this phase. If all the locks are requested before the transaction begins execution, any denied locks will cause this transaction to be delayed and possibly to release all of the locks temporarily before starting all over. If this phase does not request all of the required locks before execution, there is a possibility that the transaction will be denied a lock and will have to start all over due to deadlock. In this case, any data already changed by the transaction will have to be returned to its original state to insure integrity.

The second, or shrinking phase, wherein no locks can be requested, releases the locks. If locks have been released on data already locked by another transaction, as is done in some methods of detecting non-serialized schedules during execution, then transactions that have locked data already unlocked by this transaction have to be backed up so that inconsistent observations and updates do not occur. It is better to release all the locks at the end of the transaction so that other transactions can not lock them in

the meantime. This is analogous to locking all needed data before a transaction begins execution. If the releasing transaction has to be backed up for some reason (external to the transaction generally) the transactions setting locks on data already released have to be backed up also.

A two-phased and well-formed lock protocol, that locks all entities accessed, insures that transactions completing execution are integrity preserving [ESWA74, SCHL78]. Some thought given to a two-phased protocol on a dependency graph will convince you. Everytime a new transaction tries to lock data already locked by an executing transaction, it is prevented from setting a lock until the transaction holding the lock is completely finished with the data. The transaction that is holding the lock will never request this lock or any other lock once one lock has been released. Therefore no cycle can ever be created involving the transaction holding all its locks. Each transaction will hold all its locks before it finishes so that no transaction can cause a cycle. The transactions are forced to execute with a schedule equivalent to a serialized schedule.

It has been noticed ([CHAM74],[GARD76]) that when all locks are claimed before execution, deadlock may occur only before execution and that no integrity problems are created. Both ([CHAM74],[GARD76]) give deadlock detection algorithms designed for locking protocols claiming all required locks ahead of time.

3.2 The Added Integrity Problems Caused by Predicate Locks

All the previous discussion has assumed that all the data being locked was addressed by name. This is called physical locking. The value of a data item, or the values of a combination of data items, have not been taken into consideration. Logical locking addresses data by a value, or a combination of values in a data record or set. A predicate such as "all employee records with a salary greater than \$25,000" is used to identify data to be locked. The problems of non-unique resource names, non-static resource categories, and interdependent locks surface in this environment. Preserving the integrity of a data base is considerably more difficult with logical locks.

With logical locks it is possible for a transaction to lock or not lock the necessary data because other transactions have changed the data during the time interval that locks were being set. For a simple example [SCHL78], transaction T1 tests data item "a" negatively and data item "b" positively and consequently "b" is locked. Transaction T2 now tests and locks "a", "c", and "d". "d" is then changed to "d'" and "a" to "a'". T2 releases its locks on "a'" and "d'". T1 now tests "d'" and locks it. T1 now has "b" and "d'" and sees a transitional state of the data base

that never existed. This schedule is given in Figure 6 below.

Schedule	Result
T11 test a	negative
T12 test b	positive - b is locked
T21 test a	positive - a is locked
T22 test c	positive - c is locked
T23 test d	positive - d is locked
T24	d is changed to d'
T25	a is changed to a'
T26	d' is unlocked
T14 test d'	positive - d' is locked

Figure 6.

The Schedule and Lock Results of an Arbitrary Schedule Using Predicate Locks

The dependency graph can be adjusted to include the dependencies caused by logical locking [SCHL78]. A test with a positive result (ptr) and a test with a negative result (nt) is now added to the set of actions included in a schedule. When a dependency relation holds between two transactions and the first action is a ptr action from T1, a write and read arc (wr-arc) is drawn from T1 to T2.

Otherwise a test arc (t-arc) is drawn from T1 to T2.

From the above example an adjusted or weighted dependency graph is drawn below in figure 7 with a t-arc from T1 to T2 because T1 tests 'a' negatively and T2 tests 'a' positively. A wr-arc is drawn from T2 to T1 because T1 tests and locks 'd' after T2 has released 'd'. A cycle of length 2 is created showing this threat to integrity.

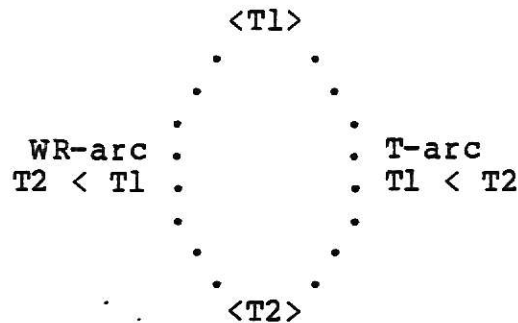


Figure 7.

The Weighted Dependency Graph
of the Arbitrary Schedule in Figure 6

If there are no cycles of length two with at least one wr-arc (dependency relation) in the dependency graph, integrity in the weak sense is preserved [SCHL78]. Pure integrity is not preserved. Two t-arcs in a cycle can, but do not necessarily, imply that the schedule is not serialized [SCHL78]. For example, two transactions, T1 and T2, may both change data such that T2 would have locked

different data if T2 executed following T1, and T2 may change data such that T1 would have locked different data if T2 executed prior to T1.

This problem can be solved easily if transactions lock all records they test and then hold them until the transactions are completed. This prevents possible concurrency among transactions. The approach of synchronization by adaption [SCHL78] has been suggested as a solution. Transactions that test overlapping sets of records are monitored as to the changes they make to the records so that the system managing the locks can prevent this problem. This is a difficult problem to solve, and adaption is difficult to implement.

Two more problems caused by logical locking are the creation of records (the phantom problem), and the deletion of records. These problems create non-static resource categories and raise the question of what records a transaction should see. For example, suppose two transactions, T1 and T2, execute concurrently. T1 may create several records T2 is interested in accessing. If T2 is permitted to access the records T1 created it may only see some of the newly created records. This may not be acceptable. If the two transactions run serially T2 would see all the newly created records or none. T1 may also delete records that T2 is accessing. This may confuse T2 and cause T2 to update an inconsistent state. This

execution is also not serial. A solution to this problem is to check predicate locks from other transactions before creating and deleting records. A discussion of this is given by Eswaran [ESWA76] and is briefly discussed by Chamberlin [CHAM75].

Bayer [BAYE75] shows that the problem of phantoms arises only between writers. A reader who locks all the records it accesses is not concerned about having a serialized schedule when phantoms are created. The result of a reader setting locks on what it accesses and a writer setting locks on what it updates can at worst give results equal to the serial execution of the reader and then the writer.

3.3 Types of Integrity and Consistency

The question of how much integrity should be preserved is debatable. With predicate locks, two transactions with potentially overlapping data sets, (non-static resource categories) that were initially non-overlapping, may give perfectly acceptable results. Integrity in the weak sense is preserved in this situation. The output of each transaction at worst will be the same as if each transaction ran first. If the result wanted from these two transactions

is based on the data's current state then their concurrent execution gives the results wanted. In other situations pure integrity may be desired.

Gray defines [GRAY76] various levels of consistency (or integrity) based on when and who gets to see changes before they are permanently written to the data base. Once a change has been written to the data base permanently it is said to be committed. Changes to the data base that have not been committed are called dirty data. The various consistency levels, for a given transaction, are as follows:

Degree 0: The transaction must not overwrite dirty data from other transactions.

Degree 1: Degree 0 and the transaction must not commit any updates until it is all done executing.

Degree 2: Degree 1 and the transaction must not read dirty data from other transactions.

Degree 3: Degree 2 and other transactions may not dirty any data the transaction reads before its execution is terminated.

Notice that level 3 with a two-phased and well-formed lock protocol preserves integrity in the weak sense for predicate locks and pure integrity for physical locks.

Degrees 0 and 1 insure that the data base is consistent as long as dirty data from other transactions is not used to determine what data to update and that phantoms and deletions do not occur. Degree 1 has the advantage of

permitting an easier methodology for backing up a transaction. When degree 0 is backed up, committed updates may need to be undone in order to preserve integrity. Updates may be committed before the end of the transaction and other transactions may have locked these data.

Degree 2 can only allow inconsistencies to develop in the data base caused by phantoms, deletions, and changed records because the data base does not all have to exist at the same time. A two-phased lock protocol is not a requirement for degree 2 consistency. This can still be resolved without going to degree 3 consistency if predicate locks are used with synchronization by adaption. Degrees 0, 1, and 2 allow readers to read dirty data and allow concurrency between a reader and an updater which may give perfectly acceptable results for statistical use in a large data base [ASTR76] where exact results are not required.

Degree 3 insures that all reads are repeatable and that all the data read by the transaction is consistent and existed all at the same moment. The previous degrees do not insure this.

Date defines [DATE79] five levels of integrity consistency for groups of records instead of for an entire transaction. They are as follows:

level 1: The transaction must not update dirty data of this record type that was dirtied by other transactions.

- level 2: Level 1 and the transaction must not read dirty records of this record type from another transaction.
- level 3: level 2 and the transaction must not allow other transactions to update dirty records of this record type.
- level 4: level 3 and the transaction must not let other transactions access any records of this type accessed by it.
- level 5: The transaction must not have any knowledge of other transactions.

The low levels of consistency are introduced to decrease overhead and to increase concurrency. Fewer computations and less storage are required for low degrees of consistency [GRAY78]. Gray also believes the low levels of consistency are not a good idea. The effect of allowing users to update a data base at low levels of consistency can create an inconsistent data base over a long period of time since users will make mistakes occasionally.

Chapter 4

Locking Protocols

In the following section, lock protocols used to preserve various levels of integrity (mostly very high levels of integrity) are described in theoretical terms. The granularity and the placement of locks in the algorithms are described. The possible concurrency between transactions is also explained.

To produce schedules that maintain a certain level of integrity, schedules can be forced to maintain the desired level of integrity by some synchronization mechanism before execution or by detecting when this degree of integrity is lost during execution. This second method of detecting the loss of integrity by a schedule during execution requires that a transaction be backed out when completion would cause inconsistencies in the data base. The algorithms that are used to detect schedules that do not preserve the desired integrity and to detect schedules that have caused deadlock are not discussed. The locking protocols presented are designed for a central data base.

4.1 Physical Locking Protocols Without Deadlock Prevention

Gray ([GRAY75],[GRAY76],[GRAY78]) proposes a lock protocol that allows some concurrency and insures varying levels of data base integrity. It partitions the data base into a lock hierarchy with finer and finer granularity as given in Figure 8 [GRAY75]:



Figure 8.

The Lock Hierarchy

Given for the Protocol by Gray [GRAY75]

Any node can be locked provided the immediate ancestor is locked in an acceptable mode and the new lock is compatible with the already existing locks on the node. The intention modes do not give specific access to a node but

allow a transaction to set stronger locks on lower level nodes. They also alert other transactions that this transaction intends to read (or write) lower level nodes. There are six access modes as follows:

- Null (NL): no access has been requested on the node.

- Intention Share (IS): Access is granted to the requested node to permit descendents to be locked in IS or S modes. This level is for transactions that are reading and that want to lock lower level nodes in S mode to prevent other transactions exclusive access at lower levels.

- Intention Exclusive (IX): Access is granted to the requested node to permit descendents to be locked in X, S, SIX, IX or IS mode. This mode is for transactions that are updating and that want to use X mode on lower level nodes to read and write.

- Shared (S): Shared access is granted to this node and to all descendents without lower level locking required. It is used by readers and must be set on any part of the data base a reader does not want changed.

- Shared Intention exclusive (SIX): It permits the requestor to lock descendents in X, SIX, or IX mode. It is used by an updater searching for an item(s) to update while still permitting readers access.

- Exclusive (X): exclusive access is granted to this node and all descendents without lower level locking being

required. It is used by updaters and must be set on any part of the data base being changed.

The compatibility table shown as Table 1 below gives the allowable simultaneous locks on a node.

	NL	IS	IX	S	SIX	X
NL	yes	yes	yes	yes	yes	yes
IS	yes	yes	yes	yes	yes	no
IX	yes	yes	yes	no	no	no
S	yes	yes	no	yes	no	no
SIX	yes	yes	no	no	no	no
X	yes	no	no	no	no	no

Table 1.

The Compatibility of the Lock Modes
in Gray's [GRAY75] Protocol

The rules for transactions requesting locks are:

1) Before granting an S or IS lock, all ancestors must be held in IS or IX mode by the requestor.

2) Before granting an X, SIX, or IX lock, all ancestors must be held in X, SIX, or IX mode by the requestor.

3) Locks should be released at the end of the transaction and in leaf to root order.

This protocol has the following changes for directed acyclic graphs:

1) Before granting an S or IS lock, at least one ancestor must be held in IS or IX mode.

2) Before granting IX, SIX or X lock all ancestors must be in at least IX mode.

The ordering of privileges between modes is given in Figure 9 below.

```

      X
    SIX
  S   IX
    IS
    NL
  
```

Figure 9.

The Ordering of Privileges Between Lock Modes
For Gray's [GRAY75] Lock Protocol

Predicate locks in a file are implemented by further partitioning the files into un-indexed fields and indices from the records instead of using the entire records as shown in Figure 10 below.

be categorized by the following added restrictions:

Degree	Restriction
3	T is well formed. T is two-phased.
2	T is well formed. T is two-phased with respect to writers.
1	T is well formed with respect to writers. T is two-phased with respect to writers.
0	T is well formed with respect to writers.

This protocol, as proposed, has the drawback of not preventing deadlock between transactions for any degree of integrity. See Gray [GRAY76] for detail on degrees of integrity preserved by backup. It also prevents possible concurrency between transactions with all degrees of integrity because of the extensive locking of nodes. Later protocols will allow more potential for concurrency with high levels of integrity. Astrahan [ASTR76] briefly discusses an implementation of this protocol for System R.

There is considerable overhead involved in this protocol. All granted locks have to be recorded and checked against requests. Queues have to be maintained for requests granted to insure integrity and to detect deadlock.

King [KING73] presents a protocol that locks records individually. A lock list is kept with all records holding exclusive locks. A reader that does not want writers to change records while it is reading must place these records

in the lock list too.

A directed graph is kept from which deadlock is determined and some level of integrity maintained. Dirty data is not accessible to other transactions. However, King [KING73] does not state that transactions must be two-phased making it possible for readers and writers to see transitionary states. See King [KING73] for details of deadlock detection and recovery.

4.2 Physical Locking Protocols With Deadlock Prevention

In [SEKI76] another protocol is presented with deadlock prevention using a preordering of resources (pointers). The data base is partitioned into lockable units (records or groups of records) whose pointers are all linearly ordered. Records are locked by writers and readers who are requesting exclusive access. When a transaction updates a lockable unit the following actions take place.

- 1) The pointer is locked with an exclusive lock.
- 2) The record is read.
- 3) The record is modified.
- 4) The exclusive locks are released.

To add a new structure

- 1) The pointer is locked with an exclusive lock.

- 2) The new structure is built with exclusive locks.
- 3) The pointers are written and modified.
- 4) The pointers are unlocked.

Generally transactions place locks in the linear order the pointers have been given. The pointers at the left or at the beginning of the linear order are set first. The pointers at the right or the end of the linear order are generally set last. When requesting a lock on an already locked pointer the transaction must wait until it is unlocked. This linear ordering prevents deadlock.

The actions involved in updating and unlocking a lockable unit is called flushing. A transaction may request a lock to the left of its other locks provided that, if the request is not allowed, it flushes all records to the right before going into a wait state. This reduces flushes and prevents deadlock.

The amount of concurrency possible depends on the linear arrangement of the pointers and the access patterns transactions use to access the data base. Access to the left of the current locks, as explained below, can reduce concurrency by causing flushing and delaying the transactions.

The integrity maintained by this protocol is not very high. It was designed to prevent processes from canceling out each others updates and to prevent deadlock. It does both of these things. A transaction is not constrained to

release locks in the same order in which it set them nor is it prevented from setting locks to the right of a transaction holding locks already to the right of this transaction. This allows a transaction to see and update a transitional state of the data base and to commit updates based on a transitional state of the data base.

For example, consider the alphabet to be the set of pointers all linearly arranged. A transaction T1 is holding a lock on pointer m and previous to this has held locks on pointers a and b. Transaction T2 started after T1 and locked pointers a and b. T2 can now hold a lock on pointer p to the right of pointer m and T1 may later set a lock on p too. If T1 later locks p, T1 and T2 will not have a serialized schedule. T1 started by accessing data at pointers a and b. T2 then accessed data at pointers a and b. On the dependency graph an arc would be drawn from T1 to T2 since $T1 < T2$. Later T2 accessed data at pointer p. Then if T1 accesses data at pointer p an arc from T2 to T1 would be drawn. This creates a cycle and proves this protocol is not serialized. This schedule is given in Figure 11 and the dependency graph in Figure 12 below.

Schedule

T11 lock a

T12 lock b

perform actions on a and b

T13 unlock a

T14 unlock b

T15 lock m

T21 lock a

T22 lock b

perform actions on a and b

T23 unlock a

T24 unlock b

T25 lock p

perform actions on p

T26 unlock p

T16 lock p

perform action on p

Figure 11.

The Schedule of two Transactions
that does not Maintain Integrity in a Weak Sense

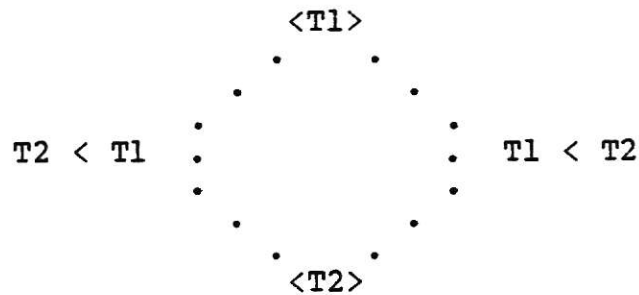


Figure 12.

The Dependency Graph
of the Schedule Given in Figure 11

If transactions release locks in the order they are set, and if transactions can not set locks to the right of a transaction on their right, then flushing to the right of transaction $T1$ still allows $T1$ to read and update a transitional state because $T1$ may access data to the left. $T2$ may have changed the data $T1$ can access after the flushing of all lockable units to the right. This can cause the data base to contain inconsistencies as well as produce output to a user which will be inconsistent. Sekino [SEKI76] notes that the need to lock a record to the left of the transaction's current locks is rare.

There is no overhead incurred to detect deadlock. The locked records have to be recorded and compared against requests. Queues of denied requests do not get as long as in some protocols because of the linear ordering. Transactions stop executing when denied a lock so they are not held simultaneously in several queues.

In [SILB80] a protocol based on a hierarchy of records is presented. Once a record is locked all descendents are accessible to the transaction for locking. All locking goes down the hierarchy from parent to child. A transaction can release any lock once it is completely done accessing it. $E(t)$ denotes the first record on which a transaction sets a lock. All locks are exclusive and there is no mention of readers. The formal protocol is the following:

- 1) T may lock a record not equal to $E(t)$ iff T is holding a lock on the record's father.
- 2) After unlocking a record, T may not lock it again.
- 3) T can only access records it has a lock on.

An example of this kind of data base hierarchy is in Figure 13 below [SILB80].

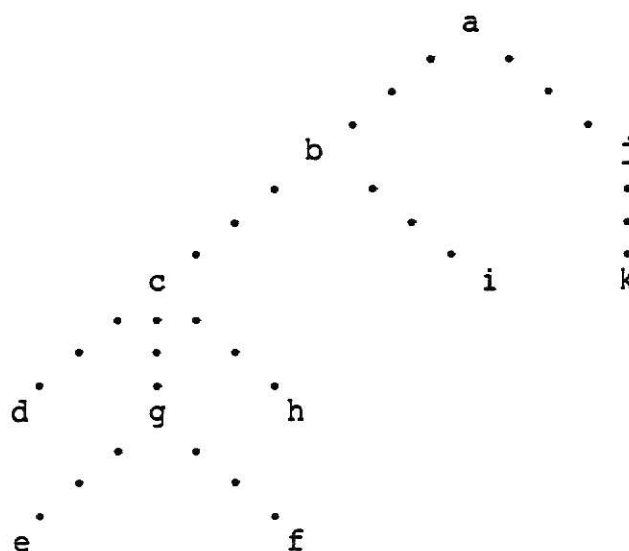


Figure 13

A Hierarchy of Records for Locking

An acceptable sequence of locks for Figure 13's hierarchy is in Figure 14 below [SILB80].

```
lock b
lock c
lock d
unlock d
lock g
unlock b
unlock c
lock e
lock f
unlock g
unlock f
unlock e
```

Figure 14.

An Acceptable Locking Sequence for a Transaction
Accessing Records With Figure 13's Hierarchy

This algorithm prevents deadlock. To see this, notice that a transaction's first lock must be an ancestor of all subsequently accessed nodes. Once a lock is released only lower level records with a direct ancestor still locked are eligible to be locked. A transaction, T2, which requests a

lock on a record locked by T1 will be delayed until the record is released. T1 will not request a lock on a record locked by T2 because it cannot lock ancestors. T2 cannot lock any record if T1 has a lock on one of its older ancestors. This prevents deadlock. A formal proof is given Silberschatz [SILB80].

This protocol insures that a set of transactions is executed by a schedule equivalent to a serial schedule if there are no readers or if all readers set exclusive locks. It also allows potential concurrency between writers that Gray's protocol [GRAY76] does not because ancestor locks can be released when they are no longer needed. Gray's protocol requires some kind of a lock on all ancestors of an accessed record because a record is a subset of all its ancestors.

A transaction may not be placed in a queue at a level lower than an ancestor node with a lock request that has been denied already since the transaction will not be able to get that far. This keeps queues of waiting transactions (processes) from getting very long. Overhead can be high if a transaction does not release high level nodes at an early stage during its execution.

4.3 Predicate Locking Protocols

The first protocol suggested by Bayer [BAYE76] is that the data base be broken up into disjoint blocks each of which satisfies a different predicate from a set P of predicates. A shared lock and exclusive lock are defined as usual. A new lock, called analysis (A), is used by a transaction to analyze a block that the transaction may update. Each block is defined by a predicate. No transaction may update a block with an A lock. To perform an update the A lock must be changed to an X lock after all transactions with S locks on smaller parts of the block have released their locks.

The compatibility table is below in Table 2.

	S	A	X
S	yes	yes	no
A	yes	no	no
X	no	no	no

Table 2.

The Compatibility of Lock Modes
For Bayer's [BAYE76] Protocol

This protocol can prevent possible concurrency because the size of the blocks may be unnecessarily large. It does take care of the phantom problem and other predicate lock problems by locking a whole predicate type of existing and potentially existing records. It maintains pure integrity. Deadlock is possible only during the seize phase because a two-phased lock protocol is used. Deadlock is possible in the same way as Gray's [GRAY76].

Sometimes finer locking is desirable for more concurrency at the cost of more overhead. IS, IX, and SIX are introduced for blocks with the same meaning as used by Gray [GRAY76] but with the intention of solving the phantom problem while providing more concurrency than the previous protocol. This does increase overhead. A transaction can now do either of the following two actions when updating a block

- 1) Convert the A lock to an X lock
- 2) Convert the A lock to an SIX lock and place X locks on individual objects.

The compatibility table is the same as Table 1 [GRAY76] with the A lock added as given below in Table 3.

	NL	IS	IX	S	SIX	X	A
A	yes	yes	no	yes	no	no	no

Table 3.

An Extension of Table 1 for
Gray's [GRAY75] Protocol

The objects in a block can only be locked in S or X mode and the block must have the appropriate lock.

Bayer also suggests [BAYE76] that the set of lock modes be extended to intention analysis (IA) and shared intention analysis (SIA) modes with the table extension given below in Table 4.

	NL	IS	IX	IA	SIA	S	X	A
IA	yes	yes	yes	yes	no	yes	no	no
SIA	yes	yes	no	no	no	yes	no	no

Table 4.

The extension of A, IA, and SIA Modes
to Table 1

This is used on a hierarchy having finer granularity. This protocol maintains pure integrity when the transaction follows a two-phased lock protocol and all the appropriate

locks are used. It is more cumbersome than other protocols discussed. Deadlock is possible in the same way as in Gray's protocol [GRAY76].

In another protocol by Bayer [BAYE75] predicate locks are used to cover a relatively large set of records and to determine if two writers potentially interfere with each other. After the writer is granted predicate locks on the sets of all potential update records, it then places physical locks on the records it is going to update. This allows readers to read other records that satisfy the predicate lock but which are not going to be updated.

The problems created by using predicate locks according to Bayer [BAYE75] are:

- 1) Finding a suitable predicate that is simple and easy to use. If it is simple then the set of records it denotes may be too large to allow very much concurrency among writers and it may create artificial phantoms. If it covers only a small part of the data base it may be too complicated for easy use.

- 2) Checking whether two predicates overlap may be quite difficult. For practicality a set of predicates which makes it easy and efficient to determine overlapping record sets should be found.

- 3) Preventing phantoms from occurring inhibits possible concurrency and may be artificial from time to time. It is quite possible for two writers to operate on two disjoint

data sets belonging to a common predicate and have both create records that would have no bearing on the other writer. Synchronization by adaption mentioned earlier provides a method of permitting this possible concurrency [SCHL78].

Bayer's first suggestion [BAYE75] to solve the phantom problem is to schedule writers serially since, as already mentioned, predicate lock problems occur only between writers.

His second protocol [BAYE75] uses predicate locks to determine whether two writers' predicates potentially overlap. After this determination, writers set locks on individual records (or small units of the data base) and compete with readers.

The third and final protocol [BAYE75] for writers and readers (which sacrifices some possible concurrency for the ease of implementation) is:

- 1) Lock the predicates on the read only data sets and lock the predicates on the update data sets of this transaction. If the conditions below are satisfied between this transaction T1 and all other transactions T2 then proceed to step 2, otherwise wait for these conditions.

- a) The update predicates of T1 have no potential for conflict with other T2 writers in existence.
- b) The update predicates of T1 have no potential for conflicts with other T2 readers' predicates.

c) The read only predicates of T1 have no potential for conflicts with T2 writers' predicates.

2) Start the seize phase for exclusive access to all data objects to be updated by T1 that already have a group predicate lock. If conflict arises with S locks wait or backup the seize phase.

3) Start the seize phase for shared access to all data objects to be read by T1 that already have a predicate group lock. If conflict develops with update locks the reader must wait or be backed up.

This algorithm provides pure integrity for both physical and logical locking. It does, however, have deadlock potential during its seize phase. The locking overhead is somewhat high but may well be worth it.

Schlageter presents [SCHL78] a slightly different protocol. The problem of phantoms is separated from the other problems of predicate locks involving two or more updaters. This later problem is solved by synchronization by adaption [SCHL78] previously mentioned. The phantom problem is solved similiarly to Bayer's [BAYE75] solution.

The transactions with the potential for creating phantoms are marked by the user. A predicate is set by the transaction creating phantoms similiar to Bayer's [BAYE75]. The actual protocol is:

1) The transaction is marked as a possible phantom creator or not.

2) If the transaction has the potential for creating phantoms, a lock predicate is determined for the phantom records. Other transactions creating phantoms must check their predicate against this one.

3) The predicate of this transaction is checked against other creating transactions' predicates. If there is no conflict the transaction then goes into a seize phase and then proceeds to execute. Synchronization by adaption is utilized to prevent further phantom problems.

This protocol preserves pure integrity and also prevents two creators from simultaneously creating the same record. Deadlock is possible during its seize phase but not during execution. The amount of concurrency and overhead depends on the predicates used.

Chamberlin suggests [CHAM74] the first two-phased lock protocol. All transactions that want to update records or prevent other transactions from updating records being read set exclusive locks on all appropriate records before execution begins. Chamberlin [CHAM74] gives a solution to the deadlock problem occurring during the seize phase. Deadlock cannot occur during execution because a two-phased lock protocol is used which, as mentioned before, sets all needed locks before execution. No transactions that have begun execution will need to be pre-empted because of this protocol.

This protocol uses predicate locks in its seize phase

and checks the predicates of records being released against the predicates of blocked transactions. It preserves pure integrity by using this kind of synchronization by adaption [SCHL78]. Two transactions may be able to set all needed locks simultaneously and both cause records to be moved into each others predicate group with this protocol, but the second transaction attempting to do this gets delayed. This forces the schedule to be serialized.

Schlageter suggests [SCHL76] two solutions to predicate lock problems. The first is that each predicate on a particular record type must come from the same transaction (process). This is a simple solution for phantoms and the deletion of records that allows no concurrency between transactions involving the same record type. It preserves pure integrity when a two-phased lock protocol is used.

The second solution for setting a predicate lock on a record type which already has a predicate lock is to prove by testing that the second predicate does not overlap the first predicate. Testing can create considerable overhead but allows much more concurrency. The testing [SCHL76] should test for all possible instances of records of this record type. It [SCHL76] does not really give a solution to this but his later [SCHL78] paper discusses synchronization by adaption as a solution.

4.4 Concurrent Access to a File of Records

The following algorithms are designed for concurrent access to a file of records and not for an entire data base. The file is structured as a B-tree. A B-tree is not a binary tree but is a search tree structure with each non-leaf node containing a set of keys $k_1, \dots, k_n$ and pointers $p_1, \dots, p_{n+1}$. Each key k_{i-1} is less than the key k_i and all keys in the subtree pointed to by p_i are less than or equal to the key k_i . The number n used above must be a number chosen for the whole tree. An example of a B-tree with $n = 4$ is below in Figure 15. The brackets represent the nodes and the numbers inside the brackets are the keys. The dotted lines are the pointers.

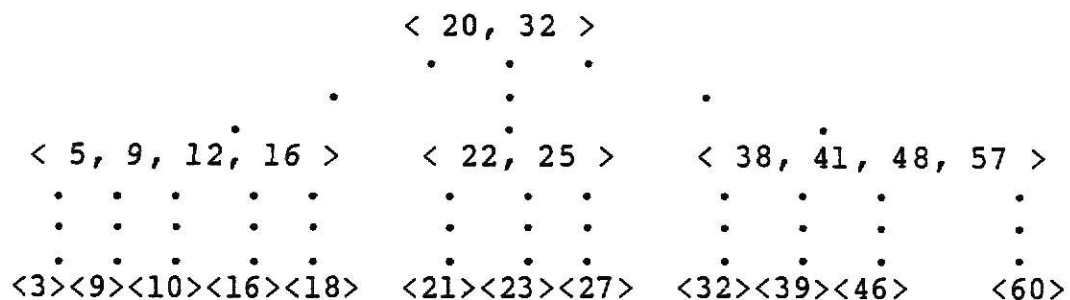


Figure 15.

An Example of a B-tree with $n = 4$

The tree must be structured so that every path from the root to any leaf is the same length. The leaves are linearly ordered and access to a leaf (record) is considerably faster than by a linear search. The time it takes to access a leaf is dependent on the size of the tree and the number n .

Bayer presents [BAYE77] several protocols for concurrent access. The first protocol has two lock modes. Shared mode (S) is for readers and exclusive mode (X) is for writers. An S lock is compatible with other S locks but not an X lock. An X lock is also not compatible with other X locks. A reader traverses the tree by starting at the root and repeatedly locking the appropriate son in S mode and then releasing the lock on the son's father until finally arriving at a leaf node.

A writer's protocol is somewhat different since the insertion of a new leaf may require some rearrangement of the nodes to force the nodes to have no more than n keys. A node is said to be safe if another key can be inserted without going over the maximum number n or a leaf can be inserted without going over $n+1$. A writer traverses the B-tree by starting at the root and repeatedly locking the appropriate son in X mode. The writer does not release the son's father unless the son is safe. This prevents the transaction from having any interference from other transactions when restructuring the tree. The transaction

may hold several levels of X locks. The locks are not released until a safe node has been reached.

This protocol prevents deadlock due to the hierarchical nature of the protocol and also insures pure integrity as long as the transaction is only accessing one record from the file or takes the appropriate actions to lock all required records simultaneously.

It does suffer a loss of some potential concurrency by not allowing readers to access nodes at high levels in the tree simultaneously with writers. The second protocol [BAYE77] increases this potential concurrency. Readers access the B-tree in the same way as in the previous solution. The writers' protocol is changed. A writer traverses the tree as a reader until it gets down to a leaf. If the node which the leaf belongs to is not safe, the first writer protocol is tried; otherwise just the leaf is locked in X mode.

If n is small the tree will be traversed a second time often. For example, if $n = 3$ there is a 50% chance of the node not being safe since a node has an equal chance of containing 3 or 4 leaves. As the number n increases the probability of traversing the tree a second time decreases rapidly. The third protocol [BAYE77] eliminates this problem for steep trees (small n) by adding a new mode, alpha, to allow a writer to prevent other writers access and still allow readers access. A writer is allowed to convert

an alpha lock to an X lock. This conversion takes place before any other lock requests waiting in the queue are granted. X and alpha locks can be held simultaneously on a node.

A reader follows the same protocol as before. A writer traverses the tree by setting alpha locks and releasing previous alpha locks every time the current node is safe. When the writer is down to a leaf node all the alpha locks still held by the transaction are converted to X locks from the top down. This protocol prevents deadlock as do the other solutions because of the top down access pattern. It does, however, prevent two or more writers from traversing the tree simultaneously when there are crossing paths, but with the added overhead of lock conversions.

Ellis presents [ELLI78] two protocols for a 2-3 tree ($n = 2$). The third algorithm [BAYE77] is modified to allow writers concurrent access to nodes that are involved in tree restructuring. Only one node is X locked at a time. In the first protocol [ELLI78], the 2-3 locking algorithm, the same lock modes as in the third algorithm in [BAYE77] are used. Readers do not miss any information but may read the same information twice. The basic idea is that readers read from the top down and from left to right. A writer process that is restructuring works from the bottom up and from right to left. Thus one node has the possibility of being crossed simultaneously by both transactions. A reader also has the

possibility of seeing four sons and not three when a leaf is being added to the structure.

X locking is only needed when deleting sons that have been transferred. This is done by X locking and releasing the father before locking the current node. After this time the sons can be deleted. The father does not have to stay locked because the son has an X lock preventing other transactions access. This protocol gives considerably more potential concurrency than Bayer's [BAYE77] and it prevents deadlock while still insuring pure integrity.

The second protocol [ELLI78] for 2-3 trees, the 2-3 pipeline algorithm, increases the potential concurrency by allowing writers to concurrently search and restructure the B-tree in a pipeline manner. The B-tree structure is modified so that the father of the leaves is a linked list of pointers to the leaves rather than a fixed set of two to four pointers. The linked lists may increase up to the number of writer transactions plus three during the actual insertion of leaves. An alpha' lock is introduced which is used by writers on the father of a leaf being inserted. It excludes other writers during the actual insertion. Transactions do not necessarily have the same path to restructure as when they searched down the tree so a father link is inserted into every non-leaf node.

The algorithm for a reader is the same as before except that now the linked lists are searched.

Chapter 5

Conclusion

The protocols presented in chapter 4 varied considerably in the amount of concurrency permitted, the level of integrity preserved, the method of deadlock prevention or detection, and the complexity of the protocol. The amount of concurrency varied because the granularity varied from very fine to very coarse by locking various sized parts of the data base. The amount of concurrency permitted is also effected by how the protocol sets and releases locks. The amount of integrity preserved by the protocols varied just as much as and, to some extent, inversely to the amount of concurrency allowed. Some protocols released locks as soon as the record was used and consequently allowed transitionary states of the data base to be accessed and sometimes updated. This allowed more concurrency than holding locks until execution is completed. Other lock protocols claimed all required locks before execution and did not release any locks until execution was completed. More strict protocols checked

predicates of records being released to the data base against predicates of active transactions. These protocols preserved pure integrity and permitted somewhat less concurrency than some other protocols.

Some hierarchical protocols and predicate lock protocols allowed some concurrency while still preserving a high level of integrity at the cost of complexity and overhead. Deadlock has been avoided by the nature of some protocols and other protocols have permitted it to occur only during the seize phase of a two-phased protocol requesting all required locks. Other protocols have ignored the possibility of deadlock and require supplemental algorithms for deadlock detection and backup.

A comparison of the lock protocols for an entire data base is given in Table 5 and the key for Table 5 is in Figure 16.

Protocol	Deadlock	Concurrency	Integrity	Complexity
[GRAY75]	2	2	1	3
[KING73]	3	1	2	1
[SEKI76]	1	2	3	2
[SILB79]	1	2	1	2
[BAYE76] all	2	3	1	3
[BAYE75] 1st	2	4	1	1
[BAYE75] 2nd	2	2	1	2
[BAYE75] 3rd	2	3	1	3
[SCHL78]	2	2	1	2
[CHAM74]	2	1	1	1
[SCHL76]	2	2	1	1

Table 5.

Comparison of Lock Protocols

Deadlock Key

- 1 - Deadlock is avoided by the design of the protocol.
- 2 - Deadlock is possible during the seize phase of this protocol when it is two-phased.
- 3 - Deadlock is not taken into account in the design of the protocol. Supplemental Algorithms are needed to handle deadlock.

Concurrency Key

- 1 - Records or record fields are locked individually to permit as much concurrency as possible.
- 2 - A lock hierarchy or a preorder is imposed on the data base that permits records and large segments of the data base to be locked separately.
- 3 - Median to small groups of the data base are locked individually. Usually these groups are files or blocks of records satisfying a predicate lock.
- 4 - Much larger segments or the entire data base is locked by a group lock. Almost no, or no concurrency is permitted.

Integrity Key

- 1 - Pure integrity is preserved. The final result of all output is equivalent to the output of a serial schedule.
- 2 - Integrity in the weak sense is preserved.
- 3 - Integrity in the weak sense is not preserved.

Complexity Key

- 1 - No more than two lock modes are used by the protocol and no hierarchy or preordering is imposed on the data.
- 2 - No more than two lock modes are used by the protocol and a hierarchy or preordering is imposed on the data.
- 3 - More than two lock modes are used by the protocol and a lock hierarchy or preordering is imposed on the data.

Figure 16.
Key for Table 5.

As mentioned in chapter 4 during the description of the algorithms, other factors in addition to those shown in the key affect the amount of concurrency permitted. Some of the algorithms can be tuned for more concurrency by the design of the lock hierarchy or the preorder and by having users access the data base in simple ways that do not inhibit concurrency. The different hierarchies affect concurrency in different ways and different amounts.

There are also more integrity levels than mentioned in the key. The question of whether it is desirable to use physical locks or predicate locks should also be taken into consideration when choosing a lock protocol. A carefully designed data base that uses physical locks can implement predicate locks by breaking up files into indices as is done by Gray [GRAY75] and can be used more easily than physical locks.

The amount of overhead also varies but this is hard to judge from the protocol without testing it. Complexity to some extent measures overhead. As mentioned below, some simulation studies have been done.

The best locking strategy seems to be application dependent. Some applications may require a lot of small independent accesses for verifying facts. This requires read only access to the data base. Changes to the data base may not be required instantaneously and it may be adequate to run them all at once when readers do not need access

(perhaps at night). Other applications may need to read and to change instantaneously a very small part of the data base. This situation is more complicated. If the granularity is extremely small (records or fields) and the probability of deadlock is very small, claiming locks as needed may be best. The probability of two people withdrawing money from the same bank account or the probability of two people trying to change the same plane reservation at the same time is essentially zero. The cost of only detecting deadlock in these situations may be worth the trouble.

Some simulation studies ([RIES77],[RIES78],[RIES79]) were made and it was concluded that coarse granularity is generally preferable when the transactions have mixed sized data base requirements. It was also shown that if the transactions were all accessing very small parts of the data base then small granularity is preferable.

Some lock protocols ([KING73],[SCKI76],[CHAM74]) with fine granularity may be the best solutions for applications that have many very small requests and no large requests. The integrity lost by Sekino's protocol [SEKI76] is not a problem if only one record is accessed at a time.

Some simulation studies [RIES79] indicate that the following factors help to support fine granularity:

- 1) If the transactions access less than 1% of the data base.

- 2) If the length of time that the locks are held is extremely long. For example, when someone is thinking about the data which was supplied as a result of a query.
- 3) If locking costs are cheap. The locks may be in core.
- 4) If there is a balance between I/O resources and CPU resources required by the transactions.

The simulation results also show that if there are one or more transactions requiring access to a large part of the data base concurrently with transactions requiring a small part of the data base, a lock hierarchy is best. This result was done with only two levels in the hierarchy and more levels may provide even better results with mixed sized transactions. These results give some insight into when to use protocols like [GRAY75], [GRAY76], [GRAY78], [SILB79], [BAYE75], [BAYE76].

Logical locking applied to a block of records by some ([BAYE75],[BAYE76]) protocols may give the best results in some circumstances where the set of records accessed is almost never the same and concurrent access is required. Logical locking requires more overhead than physical locks because the predicates must be stored and more CPU time is required for testing. With logical locks set on blocks of data, the difference in physical locking is not very great. A page or file can be locked by physical locks and if the

data base is designed well for logical locks the efficiency of the algorithm is about the same. Physical locking, in general, is easier to implement.

The lock protocols designed for the B-tree structures provide relatively fast access to records when records are accessed by a key. The concurrency is quite high and for accessing one record from a file, provide a good solution. Pure integrity is preserved when only one record is accessed by a transaction. The extra overhead involved in setting locks on data not accessed is at least partly offset by the efficiency of finding the right record quickly.

The different algorithms presented for B-trees vary in the amount of concurrency allowed between writers and between writers and readers. The best solution depends on the relative proportions of readers and writers and by the degree of randomness of the particular applications.

References

- ASTR76 Astrahan, M.M., et al., "System R: Relational Approach to database management", ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976, Pages 97-137.
- BAYE75 Bayer, R., "On the Integrity of Data Bases and Resource Locking", Proc. 5th Informatik Symposium: Lect. Notes in Comp. Sc., Springer-Verlag, Sept. 1975, Pages 339 - 361.
- BAYE76 Bayer, R., "Integrity, Concurrency, and Recovery in Database", Proc. ECI Conf. 1976: Lect. Notes in Comp. Sc., Vol. 44. Springer-Verlag, N.Y. August 1976, Pages 79-106.
- BAYE77 Bayer, R. and Schkolnick, M., "Concurrency of Operations on B-Trees", Acta Informatica, Vol. 9, Springer-Verlag, 1977, Pages 1-21.
- BERN69 Bernstein, A. and Shoshani, A., "Synchronization in a Parallel-Accessed Data Base", CACM, Vol. 12, No. 11, 1969.
- CHAM74 Chamberlin, D.D., Boyce, R.F., and Traiger, I.L., "A Deadlock-Free Scheme For Resource Locking In A Data-Base Environment", Information Processing 1974, Pages 340-343.
- CHAM75 Chamberlin, D.D., Gray, J.N., and Traiger, I.L., "Views, Authorization, and Locking in a Relational Data Base System", Proc. AFIPS National Computer Conf., Vol. 44, 1975, Pages 425-430.
- DATE79 Date, C.J., "Locking and Recovery in a Shared Database System: an Application Programming Tutorial", Proc. Int. Conf. on VLDB, 1979, Pages 1-15.

- ELLI78 Ellis, C. S., "Concurrent Search and Insertion in 2-3 Trees", Dept. of Computer Science, Univ. of Washington, Seattle, Washington 98195, Technical Report No. 78-05-01.
- ENGL76 Engles, R.W., "Currency and Concurrency in the Cobol Data Base Facility", Proc. IFIP TC-2 Working Conf. on Modelling in Data Base Management Systems, North Holland Publ. Co., 1976, Pages 339-363.
- ESWA74 Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L., "On The Notions of Consistency and Predicate Locks in a Data Base System", IBM Report RJ 1487 (No. 22786), IBM Research Laboratory, San Jose, Calif., 95193.
- ESWA76 Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L., "The Notions of Consistency and Predicate Locks in a Database System", CACM, Vol. 19, No. 11, November 1976, Pages 624-633.
- EVER74 Everest, G.C., "Concurrent Update Control and Database integrity", Proc. IFIP TC-2 Working Conf. on Management Systems, 1974, North Holland Publ. Co., Pages 241-260.
- GARD77 Gardarin, G. and Lebeux, P., "Scheduling Algorithms for Avoiding Inconsistency in Large Databases", Proc. Int. Conf. on VLDB, 1977, Pages 501-506.
- GARD76 Gardarin, G. and Spaccapietra, S., "Integrity of Data Bases: A General Lockout Algorithm With Deadlock Avoidance", Proc. IFIP TC-2 Working Conf. on Modelling in Data Base Management Systems, North Holland Publ. Co., 1976, Pages 395-411.
- GRAY75 Gray, J.N., Lorie, R.A., and Putzolu, G.R., "Granularity of Locks in a Shared Data Base", Proc. Int. Conf. on VLDB, 1975, Pages 428-451.

- GRAY76 Gray, J.N., Lorie, R.A., Putzolu, G.R. and Traiger, I.L. "Granularity of locks and Degrees of Consistency in a Shared Data Base" Proc. IFIP TC-2 Working Conf. on Modelling in Data Base Management Systems, North Holland Publ. Co., 1976, Pages 365-394.
- GRAY78 Gray, J., "Notes on Data Base Operating Systems", IBM Research Report RJ2188(30001)2/23/78, IBM Research Laboratory, San Jose, Calif., 95193.
- HAWL75 Hawley, D.A., Knowles, J.S. and Tozer, E.E., "Database Consistency and the CODASYL DBTG Proposals", Computer J., Vol. 18, No. 3, 1975, Pages 206-212.
- KING73 King, P.F. and Collmeyer, A.J., "Database Sharing - An Efficient Mechanism for Supporting Concurrent Processes", Proc. AFIPS NCC 1973.
- KUNG79 Kung, H.T. and Papadimitiou, C.H., "An Optimality Theory of Concurrency Control for Databases", Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Penn. 15213, 1979
- MUNZ77 Munz, R. and Krenz, G., "Concurrency in Database Systems - a Simulation Study", Proc. ACM SIGMOD Int. Conf. on Management of Data. Toronto, Canada, August 1977, Pages 111-120.
- PAPA79 Papadimitriou, C.H., "Serializability of Concurrent Updates", Journal of ACM, Vol. 26, No. 4, October 1979, Pages 631-653.
- RIES79 Ries, D.R., "The Effects of Concurrency Control on Database Management System Performance", Ph.D. Thesis, University of Calif., Berkeley, Calif., April 1979.
- RIES77 Ries, D.R. and Stonebraker, M., "Effects of Locking Granularity in a Database Management System", ACM Transactions on Database Systems, Vol. 2, No. 3, September 1977, Pages 233-246.

- RIES79 Ries, D.R. and Stonebraker, M.R., "Locking
Granularity Revisited" ACM Transactions on Database
Systems, Vol. 4, No. 2, June 1979, Pages 210-227.

- SAMA76 Samadi, B., "B-Trees in a System with Multiple
Users", Information Processing Letters, Vol. 5, No.
4, October 1976, Pages 107-113.

- SCHL76 Schlageter, G., "The Problem of Lock by value in
Large Data Bases", Computer J., Vol. 19, No. 1,
1976, Pages 17-20.

- SCHL78 Schlageter, G., "Process Synchronization in
Database Systems", ACM Transactions on Database
Systems, Vol. 3, No. 3, September 1978, Pages
248-271.

- SEKI76 Sekino, L.C., "Multiple Concurrent Updates", / Proc.
Int. Conf. on VLDB, 1975, Pages 505-507.

- SILB80 Silberschatz, A. and Kedem, Z., "Consistency in
Hierarchical Database Systems", Journal of ACM,
Vol. 27, No. 1, January 1980, Pages 72-80.

A Review of Theoretical Methodologies
for Locking in a
Concurrent Data Base Environment

by

Alice Ruth Galinat

B.S. Southeastern Massachusetts University, Dartmouth, 1976

An Abstract of a Master's Report

Submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

Kansas State University
Manhattan, Kansas

1980

Abstract

This report concentrates on the theoretical methods of locking in a data base. Locking is designed to insure integrity during concurrent access resulting from either multi-tasking or multi-processing such as might occur within a network environment. The report discusses the integrity problems caused by concurrent access and defines various types or levels of integrity. The level of integrity and the amount of concurrency maintained or allowed by each locking protocol is reviewed. The granularity or lockable unit size of the protocols is presented. The overhead required by the various protocols is evaluated against the amount of concurrency allowed from what is available in the literature.

The protocols which are discussed are applicable to a central data base holding one copy of the data. The supplemental algorithms needed by some protocols to detect deadlock and to backup transactions for restart are not discussed. The protocols presented are discussed in theoretical terms.

The protocols which are discussed are rated on the integrity preserved, the handling of deadlock, the amount of concurrency allowed, the complexity of the protocol, and the amount of overhead required. The appropriateness of the different protocols for different kinds of applications is suggested.