CS352 Lecture - The Transaction Concept

Last Revised 11/7/06

Objectives:

1. To introduce the notion of a transaction and the ACID properties of a transaction
2. To introduce the notion of the state of a transaction

Materials:

1. Projectable of alternate schedules for concurrent deposit and withdrawal from same account
2. Projectables of schedule for transfer and withdrawal and serial equivalent

I. Introduction

A. At the start of the course, we noted that one key responsibility is to preserve the integrity of the database by dealing with issues that could otherwise lead to its corruption. In the next few lectures, we want to deal with two of these in particular:

1. CONCURRENCY CONTROL deals with ensuring that the integrity of the database is preserved when it is being simultaneously accessed by more than one user (a critical capability for many systems using DBMS's)

2. CRASH CONTROL deals with ensuring that the integrity of the database is preserved even if a system crash (perhaps arising from factors outside the control of the DBMS) should occur while changes are being made to the database.

B. Though these seem like two rather diverse issues, it turns out that a key concept is at the heart of measures for dealing with both of them - the concept of a TRANSACTION.

C. A preliminary definition: We say that the database is in a CONSISTENT state if there are no contradictions between items stored in the database.

1. However, during the course of routine processing, it is sometimes necessary for the database to momentarily enter an inconsistent state

   Example:

   In a banking system, a customer requests a transfer of funds from a checking account to his savings account. Note that this operation results in a change to the balances in both accounts; however, the SUM of the two balances is not changed. Thus, the database is

consistent if the sum of the two balances is correct. However, in processing the transfer, it will necessarily be the case that one of the two new balances is written to disk before the other (they can't both be written at exactly the same time if they are stored in two different ocations on the disk.) Thus, during the brief interval between the wo write operations the database on disk is actually in an inconsistent state.

2. The DBMS must take measures to ensure that this momentary inconsistency does not become permanent.

   a) A failure or a crash at this instant could "freeze" that inconsistency.

   b) If another user were to access the data at this point, that user would see inconsistent data; and if user were performing an operation that updated the database, the inconsistent data might be incorporated into that update.

3. In the case of concurrent processing, overlapping of two operations on the same data could also result in inconsistency - e.g. if the the funds transfer were being executed with another computation to post interest, the following could occur:

Transfer Transaction	Interest Posting
Reads savings balance	
	Reads savings balance
Adds transfer amount	
	Computes and adds interest
Writes updated balance	
	Writes updated balance

   What happens in this case?

   ASK

4. Actually, it is also possible for a pair concurrent transactions to be executed in two different ways that are both consistent, yet produce different results.

   Example: Suppose we do the transfer transaction completely, then the interest posting. In this case, the balance on which the interest is computed includes the transferred amount. On the other hand, if the order is interest posting first, then transfer, the interest balance does not include the transferred amount. Thought these two results differ, we would regard both as consistent, since the difference depends on the relative order of external events.

II. The Transaction Concept

A. At the heart of strategies for preventing problems like these is that we conceive of the DBMS's work as basically involving the processing of a series of TRANSACTIONS.

1. Each transaction begins with the database in a consistent state, and ends with the database in a consistent state - but may momentarily place the database into an inconsistent state due to the necessity of performing updates one after another.

2. We can define a transaction formally as follows: a transaction is an atomic operation involving a series of processing steps, including:

a) The reading of zero or more items from the database, with each tem being read exactly once.

b) The writing of one or more items to the database, with each item being written exactly once.

(Actually, there is such a thing as a read only transaction that reads data from the database, but writes nothing; such transactions, however, do not cause problems of the sort we have discussed, though they do need to be considered in the context of concurrency.)

Further, if the database is in a consistent state when a transaction is begun, then it will still be in a consistent state when the processing of the entire transaction is complete.

B. To preserve system consistency, we must guarantee that each transaction satisfies four requirements. These are called the ACID properties, after the first letters of their names.

1. ATOMICITY: We must guarantee that each transaction is processed ATOMICALLY - i.e. either none of it is done, or all of it is done. It must NOT be possible for only part of a transaction to be carried out.

a) This means that if a transaction is aborted for any reason (due to a logical error in the data or a request by the user, then all effects of the transaction must be removed from the database and the database must be restored to the state it was in before the transaction was begun.

b) This also means that if a system crash occurs in the middle of processing a transaction, then either:

(1) Upon system restart, the system must be restored to its state before the transaction was started (in which case the transaction can be restarted from scratch.)

or

(2) Upon system restart, the work that was not done because of the crash is completed before any new work is begun.

2. CONSISTENCY: If a transaction is executed in isolation (with no other transactions executing concurrently), and the database is in a consistent state when the transaction starts, then it will still be in a consistent state when it is finished.

3. ISOLATION: Even if transactions are executing concurrently, the overall result is the same as if they executed serially - i.e. as if each transaction executed in isolation, with one transaction completing before the next begins.

   a) This was the problem with our earlier example about transferring money from a checking account to a savings account at the same time interest was being posted to savings accounts. Each transaction was consistent in isolation, but they interacted in such a way as to produce inconsistency.

   b) Note that we consider the isolation property to be satisfied if the result is equivalent to ANY serial ordering of the transactions being processed - e.g. as we already noted execution of both a transfer and interest posting would give two different final balances in the savings account if the transfer was done before interest was posted or after it was posted - but either result is acceptable (as long as it applies consistently to all the accounts involved.)

4. DURABILITY: Once a transaction is completed, its effects on the database must persist, even if there is a subsequent system crash. This may mean restoring some data that was destroyed by a crash upon restart.)

C. In SQL, transactions are can either be defined explicitly by `BEGIN TRANSACTION` and `END TRANSACTION` (within embedded code), or - more typically - implicitly, by the `COMMIT [ WORK ]` and `ROLLBACK [ WORK ]` statements.

1. At the start of program execution, a transaction is implicitly started.

2. When a successful `COMMIT` or `ROLLBACK` is executed, the current transaction is ended, and a new transaction is started.

3. If any failure occurs in a transaction (e.g. the violation of a constraint), any subsequent operation in that transaction wil also fail. Thus, if a transaction fails, it must be explicitly rolled back - otherwise its failure will pollute all subsequent operations.

   NOTE: Some DBMS's allow the testing of certain constraints to be deferred until the transaction is committed. In this case, of course, it is the commit operation that actually fails An unsuccessful `COMMIT` does not end the transaction. The failed transaction must be explicitly rolled back.

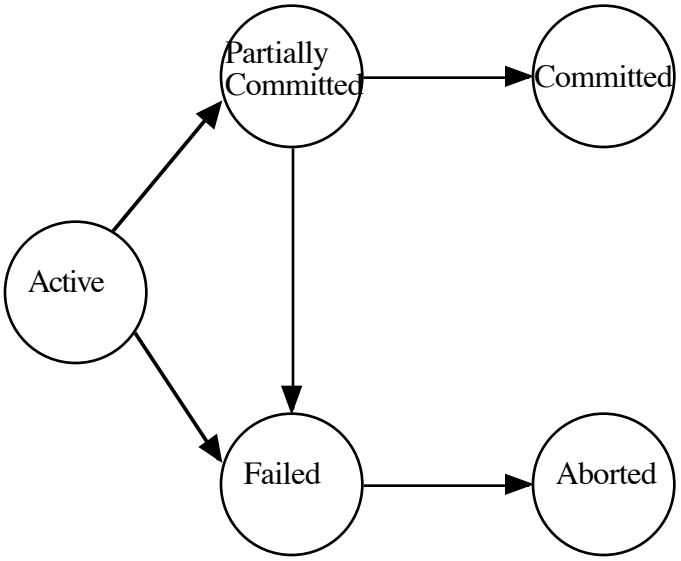
4. Some DBMS's support an "autocommit" mode, in which each SQL statement is automatically committed after being executed.

   a) The db2 command interpreter operates in this mode by default.

   b) An embedded SQL program can turn autocommit off or on. (For your project, autocommit is turned off, so you need to commit explicitly, which is appropriate since some of your transactions will require multiple SQL statements)

   c) If program not using autocommit terminates with an incomplete transaction pending, it is handled in some default way - normally by being rolled back automatically.

D. As a transaction is being processed, it passes through a series of STATES.



1. Active: from the time it starts, until it either fails or reaches its last statement.

2. Partially committed: its last statement has executed, but its changes to the database have not yet been made permanent.

   Note: The SQL `COMMIT` statement places the transaction into the partially committed state.

3. Committed: its changes to the database have been made permanent. A soon as a transaction has partially committed, the DBMS attempts to move it to the committed state - though there is no guarantee it will be able to successfully do so. Once a transaction has reached the committed state, the DBMS is obligated preserve its results, even if there is a crash

4. Failed: a logical error or user abort has precluded completion, so any changes it has made to the database must be undone.

   Note: The SQL `ROLLBACK` statement places the transaction into the failed state.

   A SQL `COMMIT` that fails due to a constraint violation also places the transaction into a failed state.

   Once a transaction has failed, the DBMS must move the transaction to the aborted state.

5. Aborted: all effects of the transaction have been removed from the database.

6. Some further points to note

   a) There is a one-way connection from partially committed to failed - a partially committed transaction can still fail; but a failed transaction must end up aborted

   b) Externally visible effects of the transaction (those seen by a user) should be deferred until after the transaction is fully committed. These include:

      (1) The writing of messages to the user terminal such as "Transaction complete."

      (2) Changes to data seen by other users concurrently accessing the database.

III. Implementation of the ACID Properties

A. It is the responsibility of the author of the code for carrying out a transaction to ensure the consistency property - i.e. the DBMS "believes" that any transaction that is submitted to it, if executed in isolation, would be consistent. (Though some inconsistencies due to user error might be detected as a result of constraint violations.)

B. Strategies for achieving atomicity and durability are the subject of a ater chapter in the book and the lecture on Crash Control.

1. The current chapter of the book gives a simplistic and impractical solution that at least illustrates how something like this could be done.

   ASK (Use of shadow copy of the database plus a pointer)

2. Actual systems make use of one of two key ideas:

   a) The use of a LOG, in which information about changes is recorded before the database itself is changed. If a crash occurs, data in the log can be used to either complete the changes or undo the ones that were made by a transaction that did not complete.

   b) A variation on the idea of a shadow copy of the database, called shadow paging, which uses this idea with individual sections of the database, rather than the whole database.

C. Strategies for achieving isolation depend on the notions of SCHEDULES and SERIALIZABILITY.

1. A transaction consists of a mixture of various kinds of operations, including reading data from the database, doing computation, and writing data back to the database. From the standpoint of ensuring isolation, it is the reads and writes that are critical, since it is through these that transactions can end up interacting with each other (i.e. by one transaction reading what another wrote or by one transaction over-writing what another wrote.)

2. Thus, for purposes of managing concurrency, a transaction is regarded as a series of read and write operations. When two or more transactions are executing concurrently, the relative sequence of the read and write operations in the two transactions constitutes a SCHEDULE.

Example: consider two transactions, one of which deposits $50.00 to a checking account and another of which withdraws $100.00 from the same account. (Perhaps, unknown to each other, a husband and wife simultaneously access their joint account rom two different ATM's.)

a) In SQL, the transactions would look like this (assuming the account number is in the program variable ACCT):

```
UPDATE CHECKING_ACCOUNTS
     SET BALANCE = BALANCE + 50.00
     WHERE ACCOUNT_NO = :ACCT

UPDATE CHECKING_ACCOUNTS
     SET BALANCE = BALANCE - 100.00
     WHERE ACCOUNT_NO = :ACCT
```

b) It appears that each transaction involves a single step, but actually, from the vantage point of the database, there are two separate steps in each:

(1) read the current value of BALANCE

(2) write a new value to BALANCE

with some computation in between.

c) From the standpoint of the database, the crucial operations are the reads and writes. These two transactions could actually be executed n any one of six different sequences (called SCHEDULES). Several different results are possible, depending on the schedule chosen. (Assume in each case the starting balance is $1000. The numbers after each read or write show the value that would be obtained or written; note that the value a transaction writes is computed solely on the basis of what it read. Note that the correct final balance is 950):

PROJECT

	Deposit (T1)	Withdrawl (T2)	Final balance
(S1)	read (1000)		
	write (1050)		
		read (1050)	
		write (950)	
			950
(S2)	read (1000)		
		read (1000)	
	write (1050)		
		write (900)	
			900
(S3)	read (1000)		
		read (1000)	
		write (900)	
	write (1050)		
			1050
(S4)		read (1000)	
		write (900)	
	read (900)		
	write (950)		
			950
(S5)		read (1000)	
	read (1000)		
		write (900)	
	write (1050)		
			1050
(S6)		read (1000)	
	read (1000)		
	write (1050)		
		write (900)	
			900

d) The six possible schedules lead to three different possible results. Only two schedules lead to the correct result.

It happens that the correct orders involve doing one transaction in its entirety first - then the other. These are called SERIAL SCHEDULES. A serial schedule will always lead to a consistent result, since each individual transaction executes independently and each individual transaction produces a consistent result.

e) In more complex situations, it is possible to get a consistent result from certain non-serial schedules (but not all).

Example: Suppose the deposit transaction were actually a transfer of $50.00 from the person's savings account to checking. Then the steps in this transaction might be:

read savings balance
compute new savings balance
write savings balance
read checking balance
compute new checking balance
write checking balance

In this case, the following non-serial schedule would still produce the correct result (assume the initial savings balance is $2000):

PROJECT

Transfer (T1)	Withdrawl (T2)	Final balances
read savings (2000)		
	read checking (1000)	
write savings (1950)		
		1950 (savings)
	write checking (900)	
read checking (900)		
write checking (950)		
		950 (checking)

3. In order to preserve the integrity of the database when doing concurrent processing, we will take measures to ensure that the actual schedule of concurrent operations by two or more transactions is SERIALIZABLE - that is, that it is EQUIVALENT TO SOME SERIAL SCHEDULE. (Recall that a serial schedule will always be consistent if the individual transactions comprising it are.)

a) Such a schedule must be consistent if each of the individual transactions is.

b) For a particular set of transactions, it may be possible to produce a non-serializable schedule that also produces consistent results; but this depends on detailed analysis of the computations done by the transactions which is not algorithmically feasible. So we insist on serializability to be sure the results are correct.

4. Unfortunately, a complicating factor that arises at this point is that there are two different definitions of equivalence - one more stringent than the other.

   a) The more stringent definition is called CONFLICT EQUIVALENCE. A schedule is said to be CONFLICT SERIALIZABLE if it is conflict equivalent to some serial schedule.

   b) The less stringent definition is called VIEW EQUIVALENCE. A schedule is said to be VIEW SERIALIZABLE if it is view equivalent to some serial schedule.

   c) Every schedule that is conflict serializable is also view serializable; however, the reverse is not necessarily true.

   d) However, it turns out that testing a schedule to see if it is conflict serializable is algorithmically feasible, whereas testing it to see if it is view serializable may, in some cases, require exponential time.

IV. Equivalence of Schedules

A. In general, two schedules are equivalent if we can interchange the operations in one of the schedules in such a way as to turn it into the other schedule, without altering the computation that would be produced. (Making no assumptions about what kind of computation takes place between operations).

Example: The schedule

T1	T2
read A	
	read B
write A	
	write B

Can be turned into the serial schedule:

T1	T2
read A	
write A	
	read B
	write B

by exchanging the write A and read B operations. Since switching these two operations cannot possibly have any impact on the final state of the database (given that A and B are two different data items), these two schedules are equivalent.

B. Of course, we cannot interchange two operations occurring in the SAME transaction, since that could easily result in changing the computation it performs. Rather, we consider only changng the relative order of operations occurring in different transactions.

C. Conflict Equivalence.

1. We say that two operations occurring in two different transactions CONFLICT if:

a) they access the same data item.

(Note: by a data item we mean a particular field of a particular record. No conflict occurs if two transactions access the same field from DIFFERENT records, or DIFFERENT fields from the same record.)

b) At least one of them is a write

c) In general, changing the relative order to two conflicting operations can result in different final outcome.

Examples

(1)

T1	T2
write A	
	read A

Clearly, if these two operations are switched, T2 will read a different value for A - the value that was in the database before T1 and T2 started.

(2)

T1	T2
read A	
	write A

Here, T1 will read the value for A that was in the database before T1 and T2 started. If the operations are switched, T1 will read the value written by T2 instead.

(3)

T1	T2
write A	
	write A

Here, if the operations are switched, there will be a different final value in the database when both transactions complete.

(4) But note: if T1 and T2 both read the same item, there is no conflict.

T1	T2
read A	
	read A

will always produce the same results as

T1	T2
	read A
read A	

2. We say that two schedules S1 and S2 (consisting of the same set of transactions) are CONFLICT EQUIVALENT if one can be transformed into the other by a series of interchanges of non-conflicting operations of different transactions.

   Example: The following schedules are conflict-equivalent

T1	T2		T1	T2
read A			read A	
	read B		write A	
write A				read B
	write B			write B

   (They differ in the relative orders of read B and write A, which access different data items.

   Example: The following schedules are NOT conflict equivalent

T1	T2		T1	T2
read A			read A	
	read A		write A	
write A				read A
	write B			write B

3. We say that a schedule is CONFLICT SERIALIZABLE if there exists a serial schedule to which it is confict equivalent.

   Example: The first schedule in the first pair above is not aq serial schedule, but it is conflict equivalent to the second schedule which is serial, so it is conflict serializable.

   Example: The first schedule in the second pair above is not conflict equivalent to any serial schedule; therefore it is not conflict serializable.

D. View Equivalence

1. It turns out that, if our interest is in whether two schedules have the same final impact on the database, conflict equivalence is sometimes too strict of a test.

   Example: Consider the following pair of schedules

T1	T2	T3		T1	T2	T3
read A				read A		
	write A					write A
		write A			write A	
write A				write A		

These schedules are not conflict equivalent, because transforming one to another involves exchanging T2 and T3's write A operation, which conflict. However, given that no read A operations intervene between T2 and T3's write A operations and T1's, the two schedules do have the same impact on the database, since the only value of A any subsequent transaction will see is the one written by T1

(The write A operations in T2 and T3 are called USELESS WRITES because no other transaction ever reads the values they write).

2. Though not equivalent by the standard of conflict equivalence, these two schedules are equivalent by the standard of VIEW EQUIVALENCE. Two schedules S1 and S2 (consisting of the same set of transactions) are view equivalent if:

   a) If in S1, some transaction T reads the initial value of some item Q, then in S2, T also reads the initial value of Q.

   b) For each pair of transactions T_i and T_j such that in S1 T_i reads a certain data item Q that was written by T_j, then the same holds in S2, and vice-versa.

   c) For each transaction T_i that does the LAST write to a certain data item Q in S1, T_i also does the last write to Q in S2.

      Example: The two schedules in the last example above, though not confict equivalent, are view equivalent.

3. Note that any two schedules that are conflict equivalent are also view equivalent - the operations permitted to transform one schedule to another in a conflict equivalent way do not result in violating any of the rules for view equivalence.

4. We say that a schedule is VIEW SERIALIZABLE if it is view equivalent to some serial schedule.

E. Note that equivalence - by either criterion - is a stronger condition than just saying that the two schedules produce the same final result

1. Two equivalent schedules (by either standard) will always produce the same final result.

2. But the converse is not necessarily true: schedules can produce the same final result without being equivalent.

Example: When we enumerated six possible schedules for the simple pair of deposit and withdrawl transactions, we saw that two of them (S1 and S4) gave the correct result. However, they are not equivalent by either definition.

3. Thus, equivalence of schedules is a stronger requirement than saying that two schedules produce the same final result. This is because in defining equivalence, we say nothing about the kinds of computation we will allow to occur between the reads and writes. In fact, it is certainly possible to conceive of a pair of transactions such that schedules similar to S1 and S4 WILL produce different results - for example, if T1 were Deposit $50.00 (as before), and T2 were Credit 10% interest to the balance.

F. In order to ensure correctness of concurrent operation, we must ensure that the schedule we follow is serializable.

1. In our initial example with deposit and withdrawl transactions, the two schedules that produced correct results (S1 and S4) were both serial schedules. None of the erroneous schedules (S2, S3, S5, or S6) are serializable. (We will prove this shortly.)

2. In our example of a transfer and a withdrawl, we exhibited a non-serial schedule that gives a correct answer.

PROJECT

Transfer (T1)	Withdrawl (T2)	Final balances
read savings (2000)		
	read checking (1000)	
write savings (1950)		
		1950 (savings)
	write checking (900)	
read checking (900)		
write checking (950)		
		950 (checking)

This schedule is conflict serializable; it is conflict equivalent to:

PROJECT

Transfer (T1)	Withdrawl (T2)	Final balances
	read checkng (1000)	
	write checking (900)	
read savings (2000)		
write savings (1950)		
		1950 (savings)
read checking (900)		
write checking (950)		
		950 (checking)

which is a serial schedule.

3. We observe, then, that any serializable schedule will give a consistent result, while a non-serializable schedule may not give a consistent result. (But note that two serializable schedules could give different results, just as two serial schedules can. However, we will regard either result as acceptable since it is consistent.)

V. Testing for Serializability

A. Since any schedule that is serializable produces consistent results, and a non-serializable schedule may not do so, it is clearly desirable to be able to test a given schedule to see if it is serializable. Unfortunately, this can be computationally expensive for the general definition of view serializability. If we use the more stringent standard of conflict serializability, then we can test for serializability more easily.

B. We can test for conflict serializability by constructing a PRECEDENCE GRAPH as follows:

1. Let each transaction be a node in the precedence graph.

2. Let there be a directed edge from a transaction T1 to a transaction T2 if one or more of the following occur:

   a) T1 executes a read on some item before T2 executes a write on it

   b) T1 executes a write on some item before T2 executes a read on it

   c) T1 executes a write on some item before T2 executes a write on it

   Each such edge represents the existence of a pair of conflicting operations which cannot be interchanged. Thus, in any conflict equivalent schedule, T1's operation must occur before T2's - which means in a conflict equivalent serial schedule ALL of T1 must occur before all of T2.

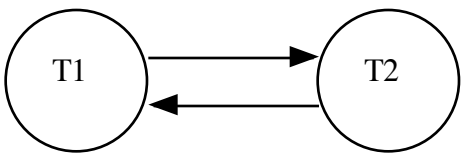
3. If the resulting graph contains a cycle, then the schedule is not conflict serializable.

4. It the graph is acyclic, then any topological sorting of the resultant precedence graph will give an equivalent serial schedule.

   Example: Consider schedule S2 from our original deposit/withdrawl transaction pair, which produced an incorrect final result.

   PROJECT

   Its precedence graph is:



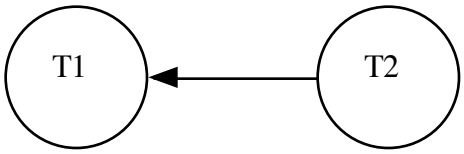
The edge from T1 to T2 arises because T1 must do its read before T2 does its write. The edge from T2 to T1 arises because T2 must do its read before T1 does its write.

Clearly, this graph contains a cycle, so schedule S2 is not serializable.

Example: Consider our non-serial, but serializable transfer/withdrawl schedule.

PROJECT

Its precedence graph is:



The edge from T2 to T1 arises for two reasons: T2 must do its read before T1 does its write, and T1 reads the value written by T2.

Since this graph is acylic, a topological sort is possible, with T2 done first, then T1. This yields the equivalent serial schedule we noted previously.

C. Testing for view serializability can also be done by using a precedence graph - but in some cases, the process is much more complex. We will not discuss the actual process here.

D. Of course, simply testing for serializability is not enough - we want to ensure serializability. This will be a topic in the next series of lectures - for now, we note that there are two general approaches that can be used.

1. We can make use of LOCKS, whereby a transaction is allowed to obtain exclusive access to some portion of the database for some period of time. Proper use of locking (a topic in the next lectures) can ensure that no unserializable schedule can occur.

2. We can make use of a rollback and restart strategy - whereby, when we detect that allowing a given transaction to complete would result in an unserializable schedule, we rollback an appropriate transaction and restart it from scratch. This, again, is a topic in the next lecture.

VI. Recoverabilty

A. One final issue we must deal with results from the fact that a transaction's results are not "official" until the transaction has committed. In particular, if some transaction writes a value that is then read by another transaction, and the first transaction fails for any reason before it commits, then any transaction that read what it had written before it committed must be rolled back and restarted.

B. However, what if the transaction that read the uncommitted value has itself committed before the first transaction is rolled back? We call such a schedule an UNRECOVERABLE schedule.

Example:

T1	T2
read A	
write A	
	read A
	write A
	fully commits
does some further computation, then fails	

This schedule could clearly lead to potential inconsistency - e.g. what if T1 and T2 were each adding 1 to the value of A? Correct execution of T1 and T2 should result in A being increased by 2; but execution of either alone should only increase A by 1. Here, even though T1 failed, A has been increased by 2. Further, if T1 were restarted, A would be increased by 3 even though each transaction has only “officially” executed once.

C. We therefore must ensure that any schedule that we produce as the result of concurrent execution is not only serializable, but also RECOVERABLE.

1. By this we mean that no transaction can commit until any ransaction that produced data it uses has itself committed.

2. If a transaction T2 uses data produced by T1, and T1 fails to commit, then T2 must also fail. (It can be restarted from scratch, but the current execution must not be allowed to commit.)

D. Of course, the possibility that the failure of one transaction might force the failure of another leads to the possibility of a chain of failures (e.g. T2 reads data produced by T1; then T3 reads data produced by T2; then T4 reads data produced by T3 ... then T1 fails - T2, T3, T4 etc. must also fail.)

1. This phenomenon is known as CASCADING ROLLBACK, and is obviously undesirable.

2. We may therefore chose to insist on producing only CASCADELESS SCHEDULES, in which cascading rollback cannot occur. In such a schedule, no transaction is allowed to read a value written by another transaction until the preceding transaction has fully committed. (A transaction that needs to read a data item that has just been written by another transaction must be delayed until the first transaction either fully commits or fails - in the latter case, the previous value of the item is read.)

3. Clearly, a cascadeless schedule is also recoverable.

4. Alternately, we may not require recoverability, but at the possible expense of cascading rollback (which, in a large database with many different items, is unlikely to involve many transactions