Transaction Processing Concepts

Prof.Dr. Adnan YAZICI
Department of Computer Engineering
Middle East Technical University (METU)
Ankara - Turkey
(Fall 2011)

What is a transaction?

- ***A* Transaction:** any one execution of a user program in a DBMS that includes one or more access operations (read - retrieval, write - insert or update, delete).
- *A transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.
- ***A* transaction (set of operations)** may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.
- **Transaction boundaries**: Begin and End transaction.
- An **application program** may contain several transactions separated by the Begin and End transaction boundaries.

What is a transaction?

- A transaction:
  - should either be done entirely or not at all.
  - if it succeeds, the effects of write operations persist (commit); if it fails, no effects of write operations persist (abort)
  - these guarantees are made despite concurrent activity in the system, and despite failures that may occur.

How things can go wrong?

- Suppose we have a table of bank accounts which contains the balance of the account. A deposit of $50 to a particular account # 1234 would be written as:

```
update Accounts
set balance = balance + $50
where account#= '1234';
```

- We will see that this entails a read and a write of the account's balance. What happens if two owners of the account make deposits simultaneously?

Concurrent deposits

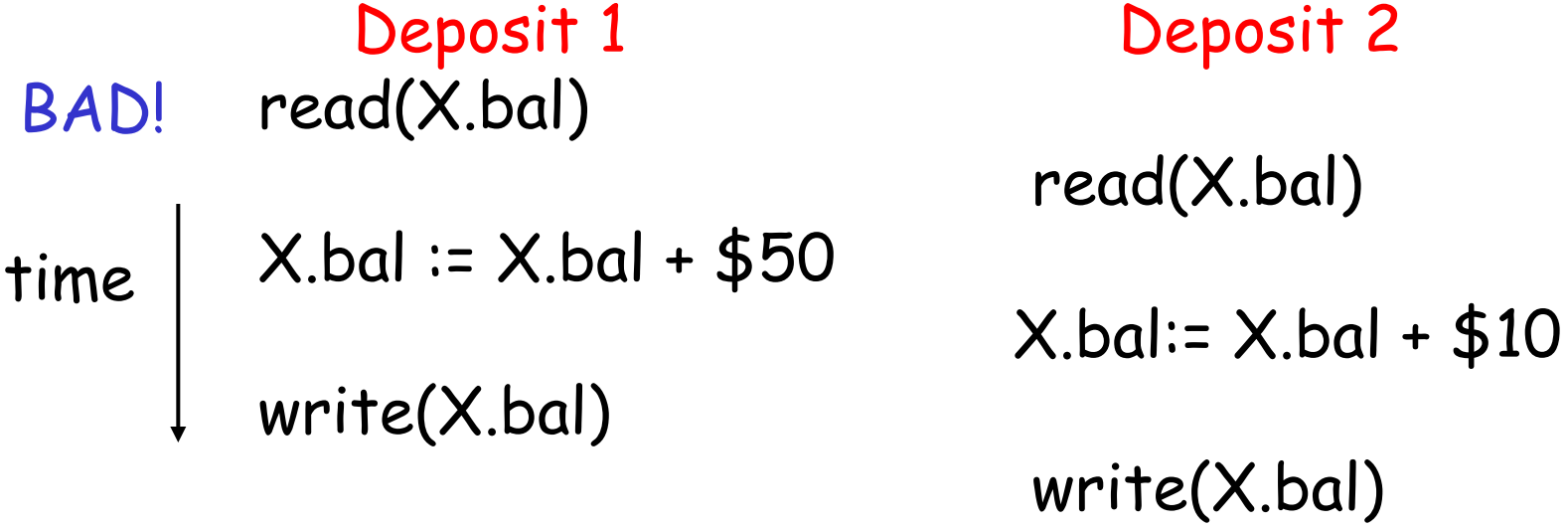
- This SQL update code is represented as a sequence of read and write operations on "data items":

Deposit 1	Deposit 2
read(X.bal)	read(X.bal)
X.bal := X.bal + $50	X.bal:= X.bal + $10
write(X.bal)	write(X.bal)

- Here, X is the data item representing the account with account# 1234.

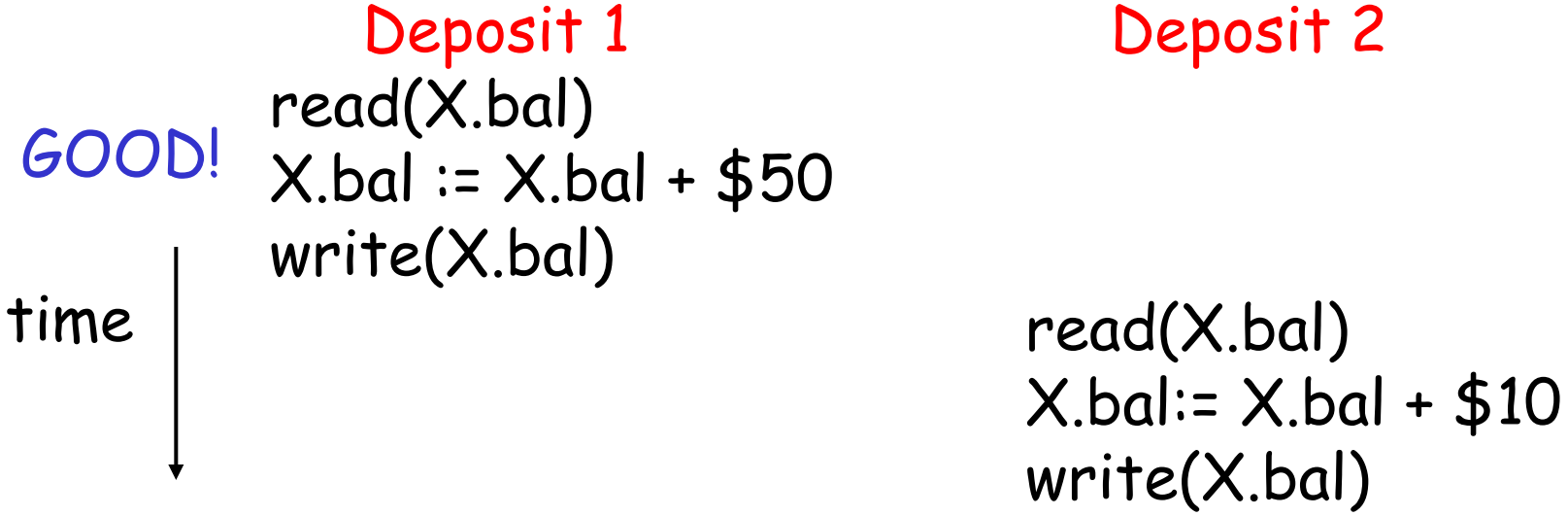
A "bad" concurrent execution

- But only one "action" (e.g. a read or a write) can happen at a time, and there are a variety of ways in which the two deposits could be simultaneously executed:



A "good" execution

- The previous execution would have been fine if the two accounts were different (i.e. one were X and one were Y).
- The following execution is a serial execution, and executes one transaction after the other:



Good executions

- An execution is "good" if is it serial (i.e. the transactions are executed one after the other) or serializable (i.e. equivalent to some serial execution)

Deposit 1	Deposit 3
read(X.bal)	
	read(Y.bal)
X.bal := X.bal + $50	
	Y.bal:= Y.bal + $10
write(X.bal)	
	write(Y.bal)

- This execution is equivalent to executing Deposit 1 then Deposit 3, or vice versa.

Motivation for Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.
- Ensuring transaction isolation while permitting concurrent execution is difficult but necessary for performance reasons.
  - **increased processor and disk utilization**, leading to better transaction *throughput* (the ave. no. of transactions completed in a given time): one transaction can be using the CPU while another is reading from or writing to the disk.
  - **reduced average response time** for transactions (average time taken to complete a transaction): short transactions need not wait behind long ones.

SIMPLE MODEL OF A DATABASE

- A database - collection of named data items
- Granularity of data locking–
  - a field,
  - a record,
  - a whole disk block, or
  - a table

- Transaction Concept is independent of granularity
- Basic operations are read and write.

SIMPLE MODEL OF A DATABASE
(for purposes of discussing transactions)

read_item(X) command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the buffer to the program variable named X.

write_item(X) command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

What causes a Transaction to fail
(why Transaction Recovery may be necessary)

1. **A computer failure (system crash):** A hardware or software error occurs in the computer system during transaction execution.
2. **A transaction or system error:** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.
3. **Local errors or exception conditions detected by the transaction:**

   -certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

   -a programmed abort in the transaction causes it to fail.

What causes a Transaction to fail
(why Transaction/Data Recovery may be necessary)

4. **Concurrency control enforcement:** The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.
5. **Disk failure:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
6. **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Additional Operations In Transaction Management

For recovery purposes, the recovery manager keeps track of the following operations:

begin_transaction: marks the beginning of transaction execution.

read or write: specify read or write operations on the database items executed as a part of a transaction.

end_transaction: specifies that read and write transaction operations have ended and marks the end limit of transaction execution.

commit_transaction: signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.

rollback (or abort): signals that the transaction has *ended* unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

Recovery Techniques Use The Following Operations:

undo: Similar to rollback except that it applies to a single operation rather than to a whole transaction.

redo: specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

Aborting a Transaction

- If a transaction *Ti* is aborted, all of its updates must be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well! ➔ *cascading aborts.*

- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits.

- In order to *undo* the updates of an aborted transaction, the DBMS maintains a *log* in which every write is recorded.

The System Log

- The log keeps track of all transaction operations that affect the values of database items. This information may be needed to permit recovery from transaction failures.
- Log records are chained together by Xact id, so it's easy to undo a specific Xact.
- Log is kept in disk and often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

The System Log

Types of Log Records:

1. [start_transaction,T]: Records that transaction T has started execution.
2. [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
3. [read_item,T,X]: Records that transaction T has read the value of database item X.
4. [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. [abort,T]: Records that transaction T has been aborted.

Why concurrence control is needed

1. The temporary update (or dirty read) problem
2. Unrepeatable read problem
3. Lost update problem
4. The incorrect summary problem

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2:

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B)
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B)	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of $(A + B)$.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

1. The temporary update (or dirty read) problem (WR conflicts)

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A), C	

One source of anomalies is that T2 could read a DB object A that has been modified by another transaction T1, which has not yet committed.

The value of object A read by T2 is called dirty data, because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the dirty read problem.

2. Unrepeatable read problem (RW Conflicts)

T1:	R(A),		R(A), W(A), C
T2:		R(A), W(A), C	

This problem may occur, when a transaction T1 reads an item twice and the item is changed by another transaction T2 between the two reads.

Hence, T1 receives different values for its two reads of the same item.

2. Unrepeatable read problem (RW Conflicts)

Unrepeatable read problem may occur, for example, if during an airline reservation transaction, a customer is inquiring about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation.

3. Lost update problem (WW Conflicts)

T1:	W(A),		W(B), C
T2:		W(A), W(B), C	

A source of anomalous behaviour is that a transaction T2 could overwrite the value of an object A, which has already been modified by a transaction T1, while T1 is still in progress.

3. Lost update problem (WW Conflicts)

- Ex: Suppose that Harry and Larry are two employees, and their salaries must be kept equal. T1 sets their salaries to 2000 TL and T2 sets their salaries to 1000 TL. If we execute these in the serial order T1 followed by T2, both receives the salary 1000 TL; the serial order T2 followed by T1 gives each the salary 2000 TL. Either of these is acceptable from a consistency standpoint.

3. Lost update problem (WW Conflicts)

- Now, consider the following interleaving of the actions of T1 and T2: T2 sets Harry's salary to 1000 TL, T1 sets Larry's salary to 2000 TL, T2 sets Larry's salary to 1000 TL and commits, and T1 sets Harry's salary to 2000 TL and commits. The result is not identical to the result of either of two possible serial executions, and interleaved schedule is therefore not serializable.
- It violates the desired consistency criterion that the two salaries must be equal.
- Note that neither transaction reads a salary value before writing it- such a write is called blind write.
- The problem is that we have a lost update. The first transaction to commit, T2, overwrote Larry's salary as set by T1. In the serial order T2 followed by T1, Larry's salary should reflect T1's update rather than T2's, but T1's update is 'lost'.

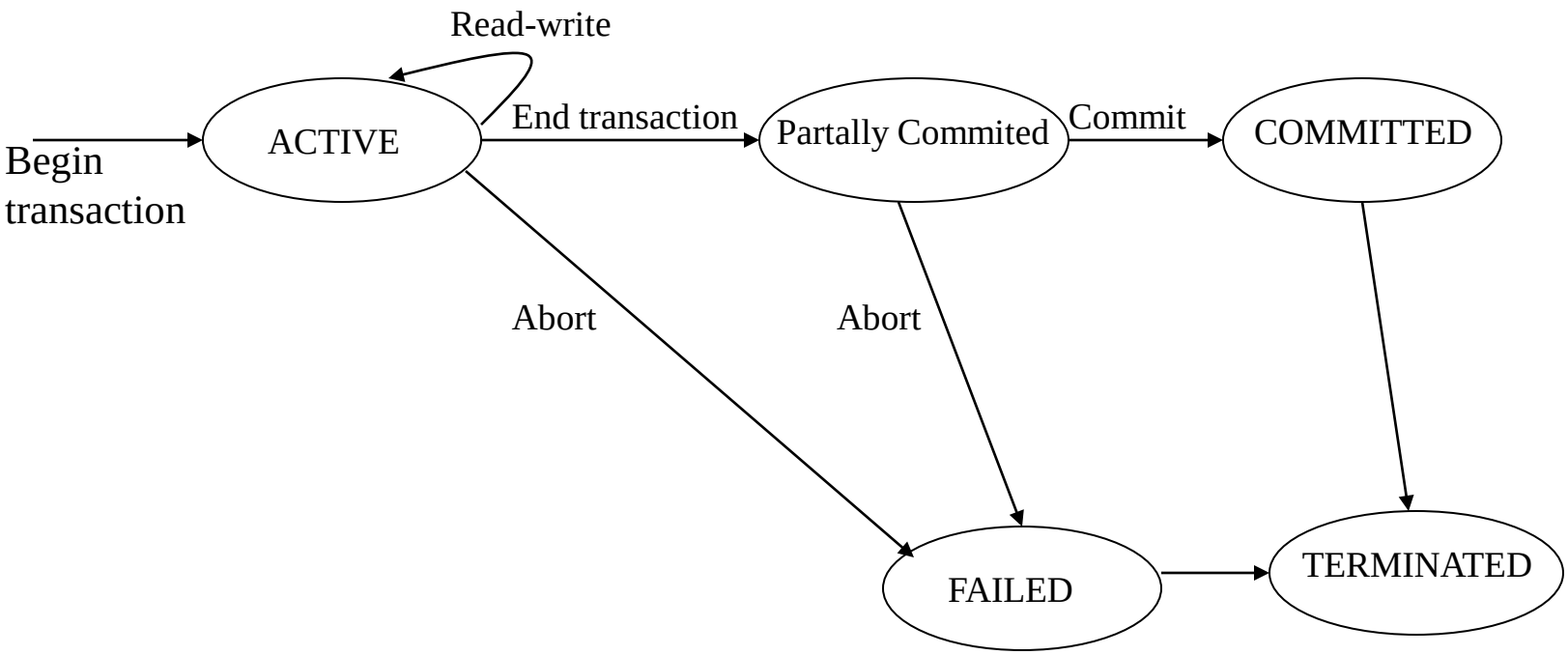
4. The incorrect summary problem

T1	T2
	sum =0;
	R(X)
	sum = sum+M;

R(X);	
X = X-N;	
W(X);	
	R(X);
	sum= sum+X;
	R(Y);
	W(Y);
R(Y);	
Y = Y + N;	
W(Y)	

T2 reads X after N is substracted and reads Y before N is added; a wrong summary is the result (off by N)

State transition diagram for transaction execution



State transition diagram illustrating the states for transaction execution

Commit Point of a Transaction

Definition: *A* transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log.

Roll Back of transactions: Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

Redoing transactions: Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise, they would not be committed, so their effect on the database can be *redone* from the log entries.

Force writing a log: Before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called *force-writing* the log file before committing a transaction.

Desirable Properties of Transactions

- Popularly known as the **ACID properties**, they should be enforced by the concurrency control and recovery methods of the dbms.

1. **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
2. **Consistency preservation**: A correct execution of the transaction must take the database from one consistent state to another.
3. **Isolation**: A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the **temporary update problem (lost update)** and makes **cascading rollbacks** of transactions unnecessary.
4. **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Example of Fund Transfer

- Transaction to transfer $50 from account *A* to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- **Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- **Consistency requirement** – the sum of *A* and B is unchanged by the execution of the transaction.

Example of Fund Transfer (Cont.)

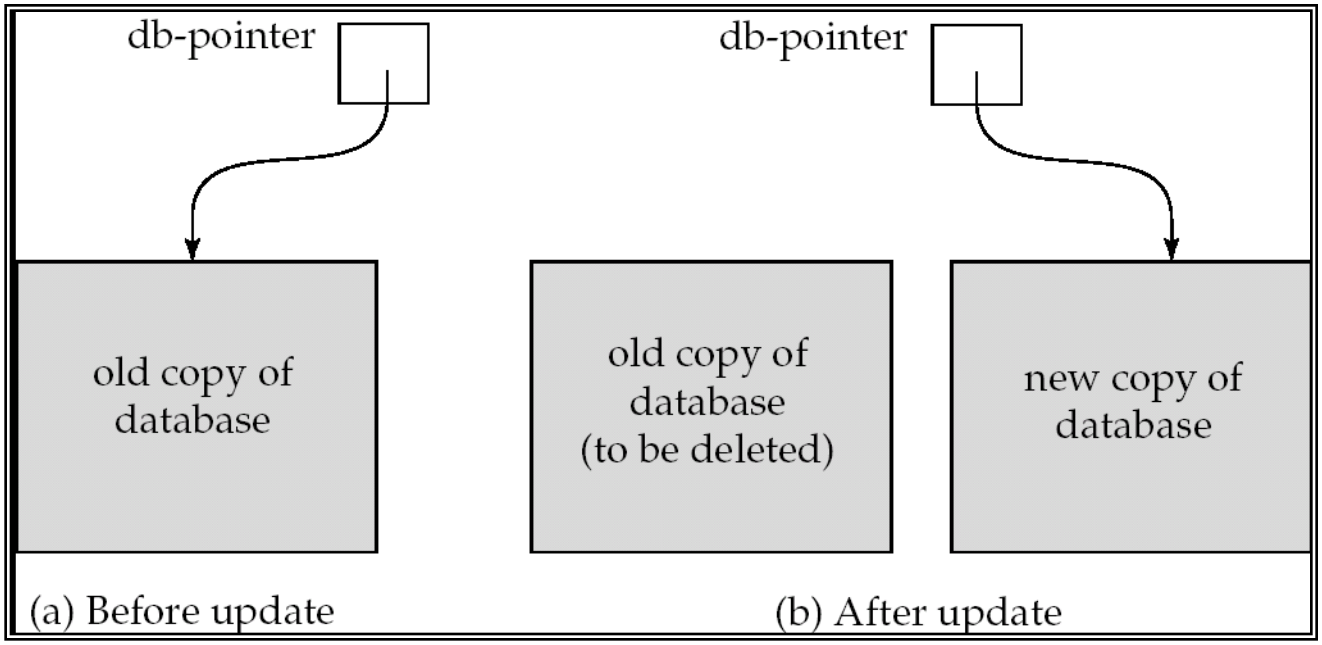
- **Isolation requirement** — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).
  - Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
  - However, executing multiple transactions concurrently has significant benefits, as we mentioned before (throughput, response time).
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.

Implementation of Atomicity and Durability

- The **recovery-management** component of a database system implements the support for atomicity and durability.
- The ***shadow-database*** scheme:
  - assume that only one transaction is active at a time.
  - a pointer called **db_pointer** always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

Implementation of Atomicity and Durability (Cont.)

The shadow-database scheme:



Transaction Support in SQL2

A single SQL statement is always considered to be atomic.

- With SQL, there is no explicit Begin Transaction_statement. Transaction initiation is done implicitly when particular SQL statements are encountered.
- Every transaction must have an explicit end statement, which is either a commit or rollback.

Three characteristics specified by a set transaction statement in SQL2:

1. diagnostic size n, specifies an integer value n, indicating the number of error conditions that can be held simultaneously in the diagnostic area. (supply user feedback information)
2. access mode: read only or read write. The default is read write unless the isolation level of read uncomitted is specified, in which case read only is assumed.
3. isolation level <isolation>, where <isolation> can be read uncommitted, read committed, repeatable read or serializable.

Read-only transactions

- When a transaction only reads information, we have more freedom to let the transaction execute in parallel with other transactions.
- We signal this to the system by stating

```
set transaction read only;
select * from Accounts
where account#='1234';
...
```

Read-write transactions

- If we state read-only, then the transaction cannot perform any updates; that is ,insert, delete, update, and create commands cannot be executed.

```
                    set transaction read only;
ILLEGAL!            update Accounts
                    set balance = balance - $100
                    where account#= '1234'; ...
```

- Instead, we must specify that the transaction read-write, updates (the default):

```
set transaction read write;
update Accounts
set balance = balance - $100
where account#= '1234'; ...
```

Isolation Levels

- The default isolation level is SERIALIZABLE
- To signal to the system that a dirty read is acceptable,

```
SET TRANSACTION READ ONLY
ISOLATION LEVEL READ UNCOMMITTED;
```

- In addition, there are

```
SET TRANSACTION
ISOLATION LEVEL READ COMMITTED;

SET TRANSACTION
ISOLATION LEVEL REPEATABLE READ;
```

Implementing Isolation Levels

- The approach for implementing these isolation levels is to use locking:
  - each data item is either locked (in some mode, e.g. shared or exclusive) or is available (no lock)
  - an action on a data item can be executed if the transaction holds an appropriate lock
- Appropriate locks:
  - before a read operation can be executed, a shared lock must be acquired
  - before a write operation can be executed, an exclusive lock must be acquired.
  - Locks are acquired at some level: record, page, table, etc.

Locking

- Locks on a data item are granted based on a lock compatibility matrix:

Request mode	Current Mode of Data Item: None	Shared	Exclusive
Shared	Y	Y	N
Exclusive	Y	N	N

- When a transaction requests a lock, it must wait (block) until the lock is granted.

"Bad" execution revisited

- If the system used locking, the first "bad" execution could have been avoided :

Deposit 1	Deposit 2
xlock(X)	
read(X.bal)	
	{xlock(X) is not granted}
X.bal := X.bal + $50	
write(X.bal)	
release(X)	
	xlock(X)
	read(X.bal)
	X.bal:= X.bal + $10
	write(X.bal)
	release(X)

Isolation levels and locking

- READ UNCOMMITTED allows queries in the transaction to read data without acquiring any lock. For updates, exclusive locks must be obtained and held to end of transaction.
- READ COMMITTED requires a read-lock to be obtained for all tuples touched by queries, but releases the locks immediately after the read. Exclusive locks must be obtained for updates and held to end of transaction.

Isolation levels and locking, cont.

- REPEATABLE READ places shared locks on tuples retrieved by queries, holds them until the end of the transaction. Exclusive locks must be obtained for updates and held to end of transaction.
- SERIALIZABLE places shared locks on tuples retrieved by queries as well as the index, holds them until the end of the transaction. Exclusive locks must be obtained for updates and held to end of transaction.
- Holding locks to the end of a transaction is called "strict".

Summary of Isolation Levels

Isolation Level	Type of Violation		
	Dirty Read	Unrepeatable Read	Phantoms
READ UN-COMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return the same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others (phantom problem).
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

- Note: Lower degrees of consistency useful for gathering approximate information about the database

Levels of Consistency in SQL-92

- phantom problem:
- New rows being read using the same read with a condition
- A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause. Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1. If T1 is repeated, then T1 will see a row that previously did not exist, even though it does not modify any of these tuples itself. This is called a phantom.
- In short, a transaction retrieves a collection of tuples twice and sees different results.

Levels of Consistency in SQL-92

- Ex: T1: "find the youngest sailor whose rating = 8"
  T2: "insert a new sailor aged 16"
- Suppose that the DBMS sets shared locks on every existing Sailors row with rating = 8 for T1. This does not prevent Xact T2 from creating a brand new row with rating=8 and setting an exclusive lock on this row. If this new row has a smaller age value than existing rows, T1 returns an answer that depends on when it is executed relative to T2. However, the locking scheme imposes no relative order on these two transactions.
- This phenomenon is called the phantom problem: a transaction retrieves a collection of tuples twice and sees different results, even though it does not modify any of these tuples itself.
- To prevent phantom, DBMS must conceptually lock all possible rows with rating = 8 on behalf of T1. One way to do this is to lock the entire table, at the cost of low concurency. It is possible to take advantage of indexes to do better, but in general preventing phantoms can have a significant impact on concurrency.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g. a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g. database statistics computed for query optimization can be approximate
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

The theory of serializability

- A schedule of a set of transactions is a linear ordering of their actions. E.g. for the simultaneous deposits example:

  $R_1(X.bal)\ R_2(X.bal)\ W_1(X.bal)\ W_2(X.bal)$

- A serial schedule is one in which all the steps of each transaction occur consecutively. That is, serial schedule does not interleave the actions of different transactions.
  - The example above is not serial.

Scheduling Transactions

- Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- A serializable schedule is one which is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Conflicting actions

- Consider a schedule S in which there are two consecutive actions Ii and Ij of transactions Ti and Tj respectively.
  - (1). They belong to different transactions,
  - (2). They access the same data item X,
  - (3). At least one of the actions is a write_item(X).
  - If Ii and Ij refer to different data items then swapping Ii and Ij does not matter.
  - If both Ii and Ij are R (Q) then swapping Ii and Ij does not matter.
  - If Ii and Ij refer to the same data item Q, then swapping Ii and Ij matters if and only if one of the actions is a write operation.
- $R_i(Q)W_j(Q)$ produces a different final value for Ti than $W_j(Q)R_i(Q)$

Testing for serializability

- Given a schedule S, we can construct a di-graph G=(V,E) called a precedence graph
  - V : all transactions in S
  - E : Ti $\rightarrow$ Tj whenever an action of Ti precedes and conflicts with an action of Tj in S
- Theorem: A schedule S is conflict serializable if and only if its precedence graph contains no cycles.
- Note that testing for a cycle in a digraph can be done in time $O(|V|^2)$

Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way

- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

The theory of serializability

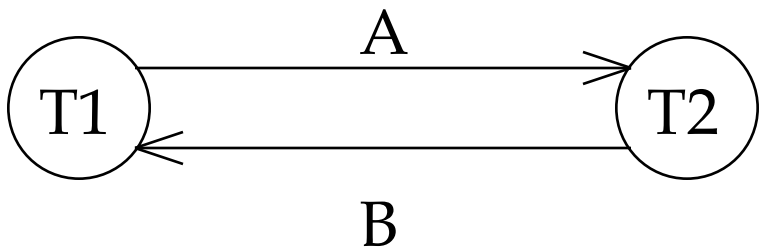
Alg.: Testing conflict serializability of a schedule S

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph (or serialization graph)
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule S is serializable iff the precedence graph has no cycles.

Example

- A schedule that is not conflict serializable:

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

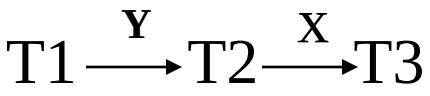


Dependency graph

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Another example

T1	T2	T3
	R(X)	
	W(X)	
		R(X)
		W(X)
R(Y)		
W(Y)		
	R(Y)	
	W(Y)	

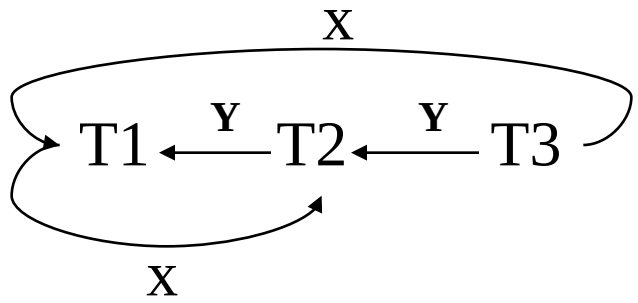


Acyclic: serializable

A schedule: T1,T2,T3

An example

T1	T2	T3
		R(X,Y,Z)
R(X)		
W(X)		
	R(Y)	
	W(Y)	
R(Y)		
	R(X)	
		W(Z)

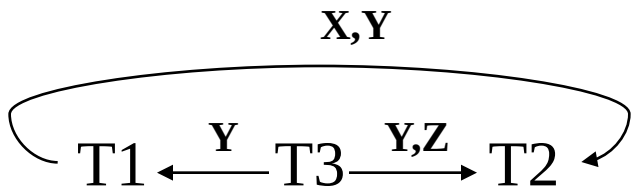


Cyclic: Not serialiable.

An example

T1	T2	T3
		R(Y);
		R(Z);
R(X);		
W(X);		W(Y);
		W(Z);
	R(Z);	
R(Y);		
W(Y);		
	R(X);	
	W(Y);	
	R(X);	
	W(X);	

Schedule A

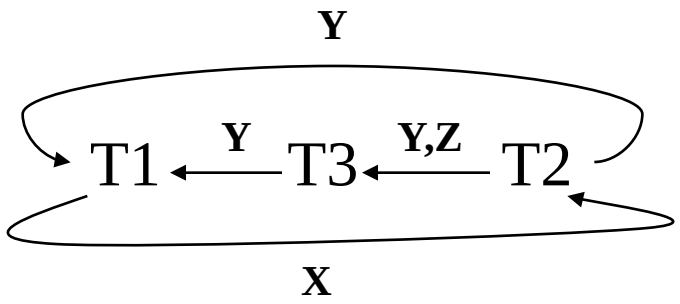


No Cycle: serialiable.

Equivalent serial schedule:
T3;T1;T2;

An example

T1	T2	T3
	R(Z);	
	R(Y);	
	W(Y);	
		R(Y);
		R(Z);
R(X);		
W(X);		W(Y);
		W(Z);
	R(X);	
R(Y);		
W(Y);		
	W(X);	



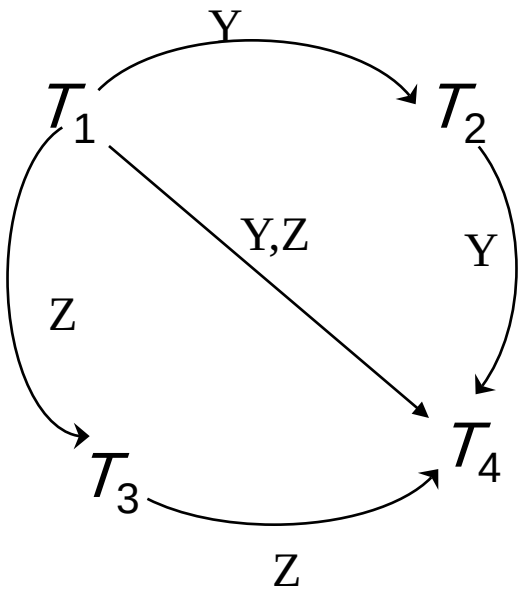
Cyclic: nonserializable.

Equivalent serial schedules: None

Schedule B

Example Schedule Precedence Graph

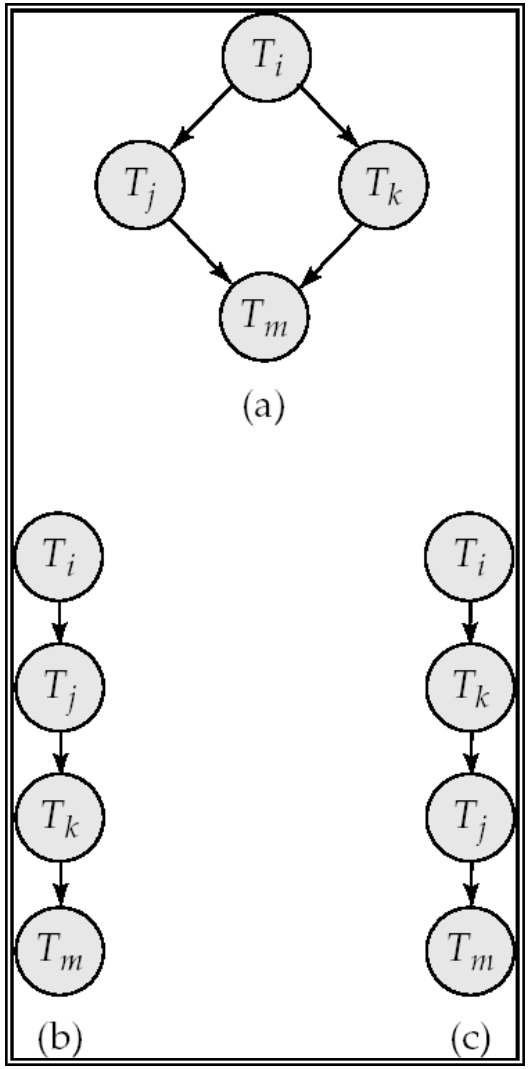
T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



A sechedule:
T1,T2,T3,T4

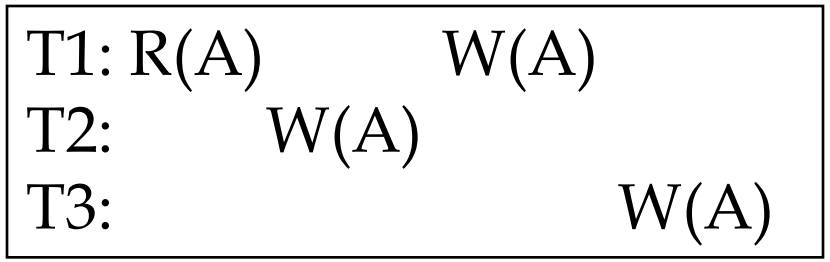
Test for Conflict Serializability

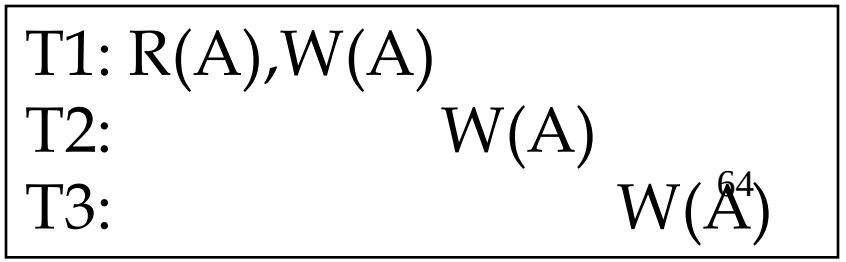
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for the previous schedule would be $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$
    - Are there others?



View Serializability

- Conflict serializability is sufficient but necessary fro serializability. View serializability is a more geberal sufficient condition for serializability.
- View serializability: definition of view serializability based on view equivalence. A schedule is view serializable if it is view equivalent to a serial schedule.
- Schedules S1 and S2 are view equivalent if:
  - The same set of transactions participates in S1 and S2, and S1 and S2 include the same operations of those transactions.
  - If Ti reads the initial value of object A in S1, then Ti also reads the initial value of A in S2.
  - If Ti reads a value of A written by Tj in S1, then Ti also reads the value of A written by Tj in S2.
  - For any data object A, If Ti writes final value of A in S1, then Ti also writes final value of A in S2





View Equivalence and View Serializability

- The idea behind view equivalence is that, as long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same result. The read operation hence said to **see** the same view in both schedules.

- Constraint write assumption states that any write operation $w_i(X)$ in T_i is preceded by a $r_i(X)$ in T_i and that the value written by $w_i(X)$ in T_i depends only on the value of X read by $r_i(X)$.

View Equivalence and View Serializability

- The definition of view serializability is less restrictive than that of conflict serializability under the unconstrained write assumption, where the value written by an operation $w_i(X)$ in T_i can be independent of its old value from the database. This is called blind write.
- Example: S_g: $r_1(X);w_2(X);w_1(X);w_3(X);c_1;c_2;c_3;$

  In this schedule $w_2(X)$ and $w_3(X)$ are blind writes, since T_2 and T_3 do not read the value of X. The schedule S_g is view serializable, since it is view equivalent to the serial schedule S_a: T_1, T_2, T_3.

  (S_a: $r_1(X)$; $w_1(X)$; $w_2(X)$; $w_3(X);c_1;c_2;c_3;$)

  However, this is not conflict serializable, since it is not conflict equivalent to any serial schedule.

View Serializability

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- Every view serializable schedule that is not conflict serializable has **blind writes**.

View Serializability

- Consider the schedule:
  $R_1(X),R_2(Y),W_3(X),R_2(X),R_1(Y)$
  - Is it view serializable?
  - Yes, it is view serializable because it is view equivalent to
    $T_1,T_3,T_2 = R_1(X),R_1(Y),W_3(X),R_2(Y),R_2(X)$

- Consider the schedule:
  $W_1(X),R_2(Y),R_2(X),W_2(Y),R_1(Y)$
  - Is it view serializable?
  - No. As T_2 read from T_1, T_1 read from T_2

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability. Enforcing or testing view serializability is very expensive.
  - Extension to test for view serializability has the cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus existence of an efficient algorithm is extremely unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule is not recoverable if T_9 commits immediately after the read

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work, cascading rollbacks.

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j (..Wi(X), ci, Rj(X)...)
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Summary

- Transactions are all-or-nothing units of work guaranteed despite concurrency or failures in the system.
- Theoretically, the "correct" execution of transactions is serializable (i.e. equivalent to some serial execution).
- Tests for serializability help us understand why a concurrency control protocol is correct.
- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- Practically, this may adversely affect throughput $\Rightarrow$ isolation levels.
- With isolation levels, users can specify the level of "incorrectness" they are willing to tolerate.