CS 5/7330

Transactions : Serializability

Table of contents

Concurrency control

- Why concurrency?
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.

Concurrency control

- Why concurrency control?
  - **Shared resources.** Many transaction may want to access the same object/tuple.
  - **Isolation.** One of the key requirement of transactions

Concurrency control -- schedule

- *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- Assumption: at any time, only one operation from one transaction can be executed
- However, DBMS may interleave operations from multiple transactions

Concurrency control -- schedule

- *Serial schedule:* schedules that does not allow interleaving between transactions (i.e. one transaction finishes before the other begin)
- Equivalent schedules: two schedules are equivalent if they "produce the same results"
  - Still need to define what it means by "produce the same results"

Concurrency control – schedule (example)

- Let *T*1 transfer \$50 from *A* to *B*, and *T*2 transfer 10% of the balance from *A* to *B*. The following is 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 1

Concurrency control – schedule (example)

- 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)

Schedule 3

Concurrency control – schedule (example)

- The following concurrent schedule does not preserve the value of the sum *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)

Schedule 4

Concurrency control – big question

- Why is schedule 3 equivalent to schedule 1, but schedule 4 is not?
  - Conflicts?
  - Order of conflicts?
  - Any general rules we can apply?

Serializable schedule – example (2)

- Consider the fund transfer operation (last lecture)

1. Find tuple for x’s account (database query)
2. Read x’s account info into main memory
3. Check if x have at least $k
4. Subtract $k from x’s account
5. Write x’s new balance back to the database (database update)
6. Find tuple for y’s account (database query)
7. Read y’s account info into main memory
8. Add $k to y’s account
9. Write y’s new balance to the database (database update)

Rewrite Using class notation →

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)
4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

Serializable schedule – example (2)

- Consider the dividend operation (last lecture)

1. Find tuple for x’s account (database query)
2. Read x’s account info into main memory
3. Add 1% to x’s account
4. Write x’s new balance back to the database (database update)
5. Find tuple for y’s account (database query)
6. Read y’s account info into main memory
7. Add 1% to y’s account
8. Write y’s new balance back to the database (database update)

Rewrite Using class notation

1. A1 <- Read(X)
2. A1 <- A1* 1.01
3. Write(X, A1)
4. A2 <- Read(Y)
5. A2 <- A2 * 1.01
6. Write(Y, A2)

Serializable schedule – example (2)

- Suppose x has $100, y has $200
- Consider two operations
  - x transfer $50 to y
  - Dividend
- For serial schedules
  - If transfer comes before dividend
    - X : 100 -> 50 -> 50.5
    - Y : 200 -> 250 -> 252.5
  - If dividend comes before transfer
    - X : 100 -> 101 -> 51
    - Y : 200 -> 202 -> 252
  - In both case, X + Y = 303

Serializable schedule – example (2)

- But with the following schedule

1. A1 <- Read(X)	
2. A1 <- A1 – k	
3. Write(X, A1)	
	1. A1 <- Read(X)
	2. A1 <- A1* 1.01
	3. Write(X, A1)
	4. A2 <- Read(Y)
	5. A2 <- A2 * 1.01
	6. Write(Y, A2)
4. A2 <- Read(Y)	
5. A2 <- A2 + k	
6. Write(Y, A2)	

X : 100 -> 50 -> 50.5; Y : 200 -> 202 -> 252; X+Y = 302.5

Serializable schedule – example (2)

- What cause the problem?
  - Contention of resources?
  - Interleaving?
- Is interleaving always bad?

Serializable schedule – example (2)

- But with the following schedule

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

1. A1 <- Read(X)
2. A1 <- A1* 1.01
3. Write(X, A1)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

4. A2 <- Read(Y)
5. A2 <- A2 * 1.01
6. Write(Y, A2)

X : 100 -> 50 -> 50.5; Y : 200 -> 250 -> 252.5; X+Y = 303

In this case, interleaving is ok!

Serializable schedule – example (2)

- Let’s change slightly:

```
1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)
                              1. A1 <- Read(X)
                              2. A1 <- A1* 1.01
                              3. Write(X, A1)
                              4. A2 <- Read(Y)
4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)
                              5. A2 <- A2 * 1.01
                              6. Write(Y, A2)
```

X : 100 -> 50 -> 50.5; Y : 200 -> 250 -> 202; X+Y = 252.5

In this case, interleaving is very bad!

Serializable schedule – example (2)

- Let’s change slightly (again):

1. A1 <- Read(X)	
2. A1 <- A1 – k	
3. Write(X, A1)	
	1. A1 <- Read(X)
	2. A1 <- A1* 1.01
4. A2 <- Read(Y)	
5. A2 <- A2 + k	
6. Write(Y, A2)	
	3. Write(X, A1)
	4. A2 <- Read(Y)
	5. A2 <- A2 * 1.01
	6. Write(Y, A2)

X : 100 -> 50 -> 50.5; Y : 200 -> 250 -> 252.5; X+Y = 303

In this case, interleaving is good again!

Serializable schedule – example (2)

- What’s going on here?
  - Interleaving can be very bad.
  - However, some interleaving does not cause problems.
  - How can we determine what kind of interleaving is “nice”?

Serializable schedule – example (2)

- Notice in example
  - The value of X (and Y) is changed twice
  - Let’s call the values
    - Old value (before any change)
    - Intermediate value (after one change)
    - Final value (after all changes)

Serializable schedule – example (2)

- But with the following schedule

1. A1 <- Read(X)	
2. A1 <- A1 – k	
3. Write(X, A1)	
	1. A1 <- Read(X)
	2. A1 <- A1* 1.01
	3. Write(X, A1)
	4. A2 <- Read(Y)
	5. A2 <- A2 * 1.01
	6. Write(Y, A2)
4. A2 <- Read(Y)	
5. A2 <- A2 + k	
6. Write(Y, A2)	

X : 100 -> 50 -> 50.5; Y : 200 -> 202 -> 252; X+Y = 302.5

Serializable schedule – example (2)

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

1. A1 <- Read(X)
2. A1 <- A1* 1.01
3. Write(X, A1)
4. A2 <- Read(Y)
5. A2 <- A2 * 1.01
6. Write(Y, A2)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

Notice that the two transaction use inconsistent values as input.

But: how to formalize this notion?

Serializability

- How to formalize the notion?
  - One can look at final results
  - If the schedule produce the same result as a serial schedule, then it’s fine.
  - However, this may be due to luck and/or “commutative” operators

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

1. A1 <- Read(X)
2. A1 <- A1 + m
3. Write(X, A1)
4. A2 <- Read(Y)
5. A2 <- A2 - m
6. Write(Y, A2)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

A better notion is needed

Conflict serializability

- Suppose two transactions (T_1, T_2) want to operate on the same data object (X)
- Four possible scenarios
  - T_1 Read(X), T_2 Read(X)
  - T_1 Read(X), T_2 Write(X)
  - T_1 Write(X), T_2 Read(X)
  - T_1 Write(X), T_2 Write(X)
- How does the order of these operations affect the results of the transactions?

Conflict serializability

- T_1 Read(X), T_2 Read(X)
  - No effect.
- T_1 Read(X), T_2 Write(X)
  - Order will determine what value of X T_1 reads
  - Affect the results of T_1
- T_1 Write(X), T_2 Read(X)
  - Order will determine what value of X T_2 reads
  - Affect the results of T_2
- T_1 Write(X), T_2 Write(X)
  - No effect on T_1 and T_2
  - But affect on the next transaction that read X
- Thus, in 2nd to 4th case order matters
- We denote that the pair(s) of operations are in conflict

Conflict serializability

- What's conflict have to do with it?
- Consider a schedule of two transactions (T_1, T_2)
  - Suppose T_1 execute operation I_1, and than T_2 execute operation I_2
  - If I_1 and I_2 has conflict
    - Then swapping them implies potential problem
  - Else, (we claim) the results are not affected

Conflict serializability -- example

- Remember this schedule?

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

i. A1 <- Read(X)
ii. A1 <- A1* 1.01
iii. Write(X, A1)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

iv. A2 <- Read(Y)
v. A2 <- A2 * 1.01
vi. Write(Y, A2)

X : 100 -> 50 -> 50.5; Y : 200 -> 250 -> 252.5; X+Y = 303

In this case, interleaving is ok!

Conflict serializability -- example

- Remember this schedule?

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

i. A1 <- Read(X)
ii. A1 <- A1* 1.01

4. A2 <- Read(Y)

iii. Write(X, A1)

5. A2 <- A2 + k
6. Write(Y, A2)

iv. A2 <- Read(Y)
v. A2 <- A2 * 1.01
vi. Write(Y, A2)

(4) And (iii) are not in conflict, so can swap

Conflict serializability -- example

- Remember this schedule?

1. A1 <- Read(X)	
2. A1 <- A1 – k	
3. Write(X, A1)	
	i. A1 <- Read(X)
4. A2 <- Read(Y)	
	ii. A1 <- A1* 1.01
	iii. Write(X, A1)
5. A2 <- A2 + k	
6. Write(Y, A2)	
	iv. A2 <- Read(Y)
	v. A2 <- A2 * 1.01
	vi. Write(Y, A2)

(4) And (ii) are not in conflict, so can swap

Conflict serializability -- example

- Remember this schedule?

1. A1 <- Read(X)	
2. A1 <- A1 – k	
3. Write(X, A1)	
4. A2 <- Read(Y)	
	i. A1 <- Read(X)
	ii. A1 <- A1* 1.01
	iii. Write(X, A1)
5. A2 <- A2 + k	
6. Write(Y, A2)	
	iv. A2 <- Read(Y)
	v. A2 <- A2 * 1.01
	vi. Write(Y, A2)

(4) And (i) are not in conflict, so can swap

Conflict serializability -- example

- Remember this schedule?

T1	T2
1. A1 <- Read(X)	
2. A1 <- A1 – k	
3. Write(X, A1)	
4. A2 <- Read(Y)	
5. A2 <- A2 + k	
	i. A1 <- Read(X)
	ii. A1 <- A1* 1.01
	iii. Write(X, A1)
6. Write(Y, A2)	
	iv. A2 <- Read(Y)
	v. A2 <- A2 * 1.01
	vi. Write(Y, A2)

Similarily (5) And (i) – (iii) are not in conflict, so can swap

Conflict serializability -- example

- Remember this schedule?

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)
4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

i. A1 <- Read(X)
ii. A1 <- A1* 1.01
iii. Write(X, A1)
iv. A2 <- Read(Y)
v. A2 <- A2 * 1.01
vi. Write(Y, A2)

Similarily (6) And (i) – (iii) are not in conflict, so can swap

We obtain a serial schedule by this transformation (swapping process)

Conflict serializability -- example

- Now, remember this schedule?

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

    i. A1 <- Read(X)
    ii. A1 <- A1* 1.01
    iii. Write(X, A1)
    iv. A2 <- Read(Y)
    v. A2 <- A2 * 1.01
    vi. Write(Y, A2)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

X : 100 -> 50 -> 50.5; Y : 200 -> 202 -> 252; X+Y = 302.5

Conflict serializability -- example

- Now, remember this schedule?

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

i. A1 <- Read(X)
ii. A1 <- A1* 1.01
iii. Write(X, A1)
iv. A2 <- Read(Y)
v. A2 <- A2 * 1.01
vi. Write(Y, A2)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

(3) And (i) has conflict, so can’t swap

(4) And (vi) has conflict, so can’t swap

Conflict serializability -- example

- Can you work through this case?

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

    1. A1 <- Read(X)
    2. A1 <- A1* 1.01
    3. Write(X, A1)
    4. A2 <- Read(Y)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

    5. A2 <- A2 * 1.01
    6. Write(Y, A2)

X : 100 -> 50 -> 50.5; Y : 200 -> 250 -> 202; X+Y = 250.5

In this case, interleaving is very bad!

Conflict serializability -- example

- From previous slides, it seems
  - A schedule can be transformed to a serial schedule $\Rightarrow$ good! (achieve isolation)
  - A schedule cannot be transformed to a serial schedule $\Rightarrow$ bad! (do not achieve isolation)
- Can we generalize?
  - Yes.

Conflict serializability -- definitions

- Schedule: – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
- Complete schedule: Schedule that contain commit/abort decision for each transaction in the schedule
- Serial schedule: A schedule where there is no interleaving of operations by multiple transactions.
  - Denoted by < T1, T2, … , Tn>

Conflict serializability -- definitions

- Given a schedule *S*, let *p* and *q* be two operations in *S*, we write $p <_S q$ if *p* occurs before *q* in *S*.
  - We write $<$ instead of $<_S$ if the schedule is understood from context.
- We use a subscript to indicate the transaction that issues an operation, e.g., p_i is an operation issued by transaction T_i.

Conflict serializability -- definitions

- Conflict equivalent transformation: swapping *adjacent* operation on a schedule which does not conflict
- Conflict equivalent: Two schedules S and S’ are conflict equivalent if S can be transformed to S’ by successive conflict equivalent transformations

Conflict serializability -- definitions

- Given a schedule *S*, the *committed projection* of *S*, denoted by *C(S),* is the schedule obtained from *S* by deleting all operations that do not belong to transactions committed in *S*.

Conflict serializability -- definitions

- Conflict serializability: a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule
- Example of a schedule that is not conflict serializable:

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

We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

Test for serializability

- Consider the following schedules:

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

i. A1 <- Read(X)
ii. A1 <- A1* 1.01
iii. Write(X, A1)
iv. A2 <- Read(Y)
v. A2 <- A2 * 1.01
vi. Write(Y, A2)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

Not serializable

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

i. A1 <- Read(X)
ii. A1 <- A1* 1.01
iii. Write(X, A1)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

iv. A2 <- Read(Y)
v. A2 <- A2 * 1.01
vi. Write(Y, A2)

Serializable

Test for serializability

- Label all the conflicting operations

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

i. A1 <- Read(X)
ii. A1 <- A1* 1.01
iii. Write(X, A1)
iv. A2 <- Read(Y)
v. A2 <- A2 * 1.01
vi. Write(Y, A2)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

Not serializable

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

i. A1 <- Read(X)
ii. A1 <- A1* 1.01
iii. Write(X, A1)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

iv. A2 <- Read(Y)
v. A2 <- A2 * 1.01
vi. Write(Y, A2)

Serializable

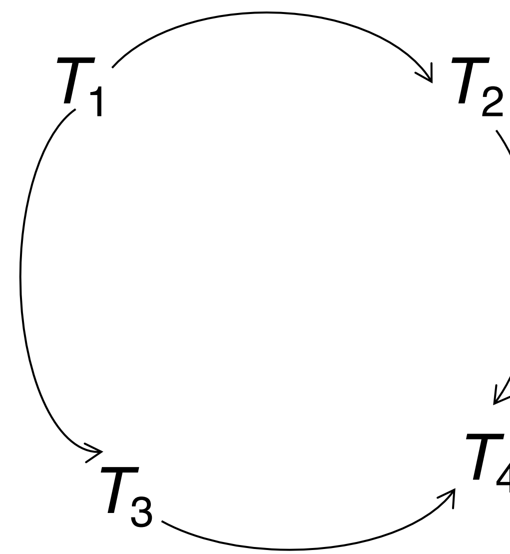
What is the difference?

Test for serializability

- The *serialization graph* (*SG*) (or the precedence graph used in the textbook) for a schedule *S*, denoted *SG(S),* is a directed graph (*V*, *E*) such that:
  - *V* (nodes) = the set of transactions that are *committed* in *S*.
  - *E* (edges) = $T_i \rightarrow T_j$ ($i \neq j$) if one of T_i's operation precedes and conflicts with one of T_j's operations in *S*.
- Theorem: A schedule is conflict serializable iff its serialization graph is acyclic

Test of serializability -- example

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)				



Test of serializability

- Proof: (if)
  - *SG(S)* is acyclic $\Rightarrow$ there exists a topological order of the committed transactions in *S*.
  - W.l.o.g., let $S' = T_1, T_2, \ldots, T_m$ be such an order. We can show that $C(S) \equiv S'$.
  - Let p_i and q_j ($i \neq j$) be two conflicting operations in *S* such that T_i and T_j are committed in *S*.
  - Clearly, p_i & $q_j \in C(S)$.
  - If $p_i <_S q_j$, we have an edge $T_i \rightarrow T_j$ in *SG(S)*.
  - Therefore, T_i must precede T_j in *S'*. That is, all operations of T_i appear before all operations of T_j in *S'*.
  - Hence, $p_i <_{S'} q_j$.

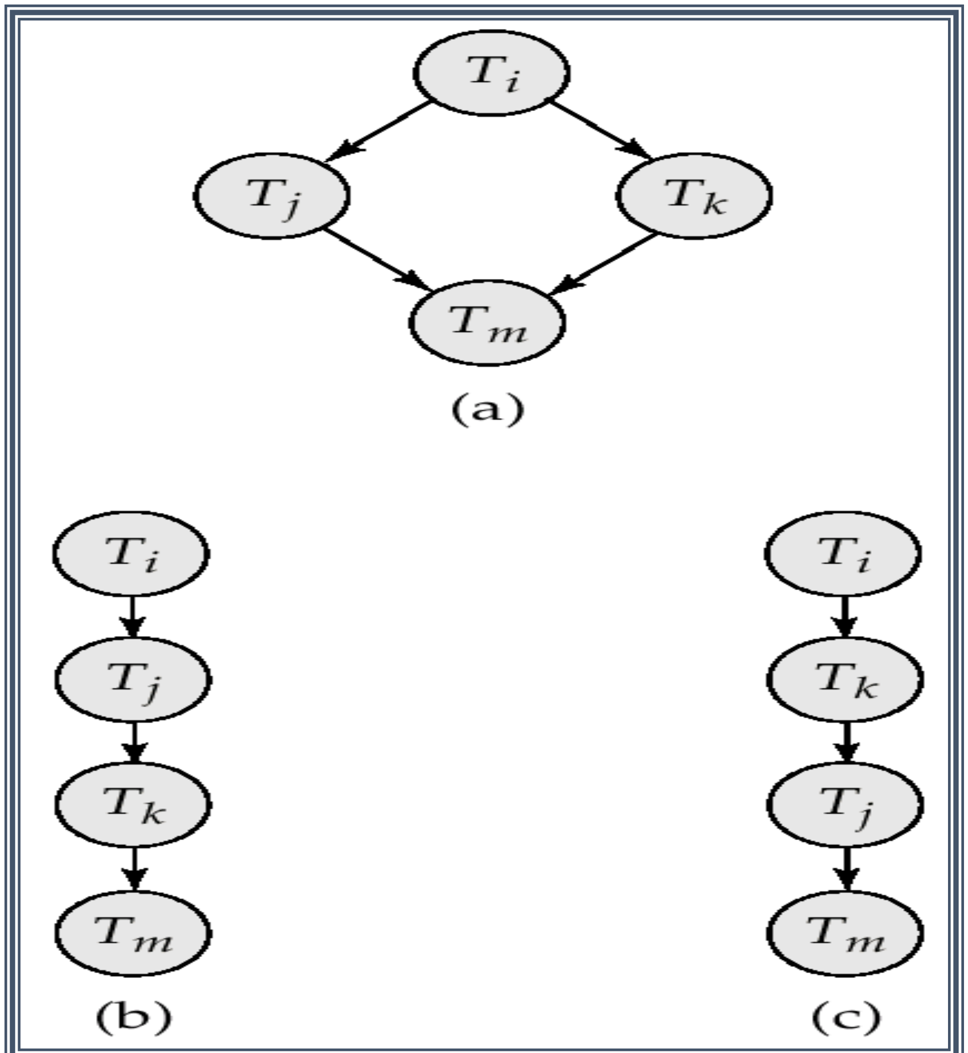
Test of serializability

- Proof: (only if)
  - S is serializable $\Rightarrow$ $C(S) \equiv$ to some serial schedule S'.
  - If an edge $T_i \rightarrow T_j$ exists in $SG(S)$, then, T_i must appear before T_j in S'.
  - Now, if $SG(S)$ has a cycle, w.l.o.g., let it be $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$.
  - We have T_1's operations must appear before T_2's operations, which must appear before T_3's operations, …, T_k's operations must appear before T_1's operations.
  - A contradiction.

Test of serializability

- 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.

Example of topological sort



Recoverability

- Abort transactions caused another problem:
- Suppose T_9 commits.
- What happen when T_8 has to abort after read(B)?
- T_9 will in fact be reading an inconsistant value
- Can be problematic (e.g. if T_9 print the value of A)

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

Recoverability

- We say that T_i reads x from T_j in a schedule S if
  - $W_j[x] <_S R_i[x]$, *(Ti read an x value written by Tj)*
  - A_j not $<_S R_i[x]$, *(Tj has not aborted when Ti read x)*
  - If $\exists\ W_k[x]$ s.t. $W_j[x] <_S W_k[x] <_S R_i[x]$, then $A_k <_S R_i[x]$. *(Any transaction that update x between Tj write x and Ti read x is aborted before the read – that means the value of x read by Ti is actually written by Tj)*
- We say that T_i reads from T_j in S if T_i reads some data item from T_j in S.

Recoverability

- A schedule S is called *recoverable* if, whenever T_i reads from T_j ($i \neq j$) in S and $C_i \in S$ then $C_j < C_i$
  - I.e., each transaction commits after the commitment of all transactions (other than itself) from which it read from
- This implies all aborted transaction will not make values that are read by committed transaction obsolete/inconsistent.

Recoverability

- Even for recoverable schedule, trouble may still bestows.
- Suppose none of the transactions committed yet.
- Suppose T10 aborts after read(A)
- T11 needs to be aborted (as it read values from T10)
- T12 needs to be aborted too
- Cascade aborts

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

Recoverability

- We say that a schedule S *avoids cascading abort* if, when T_i reads x from T_j ($i \neq j$), $C_j < R_i[x]$.
  - I.e., a transaction may read only those values that are written by committed transaction or by itself.
- A schedule S is *strict* if whenever $W_j[x] < O_i[x]$ ($i \neq j$), either $A_j < O_i[x]$ or $C_j < O_i[x]$ where $O_i[x]$ is $R_i[x]$ or $W_i[x]$.
  - I.e., no data item may be read or written until the transaction that previously wrote into it terminates, either by aborting or by committing.

Serializability – In practice

- Scheduling is usually not up to the DBMS
  - Operating systems do it, so why spend time reinventing the wheel?
- Detection of serializability is limited in usage
  - Cycle detection (while not NP complete) is not very cheap
  - Testing for serializability after execution is not really helpful
- Goal – to develop concurrency control protocols that will assure serializability.
  - not examine the precedence graph as it is being created;
  - instead a protocol will impose a discipline that avoids nonseralizable schedules.
  - However, test for serializability will help one to understand and proof the correctness of such protocals

Serializability – In practice

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- Levels of consistency specified by SQL-92:
  - **Serializable** — default
  - **Repeatable read**
  - **Read committed**
  - **Read uncommitted**

Serializability – In practice

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **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.

(We will revisit these terms later)