Principles of Database Systems (CS307)

Lecture 15: Transaction

Zhong-Qiu Wang

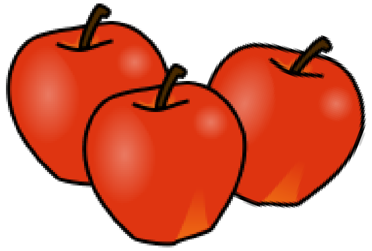
Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

Transaction in Real Life

- "An exchange of goods for money"
  - A series of steps
  - All or nothing

Flickr:BracketingLife (Clarence)



Transaction in Computer

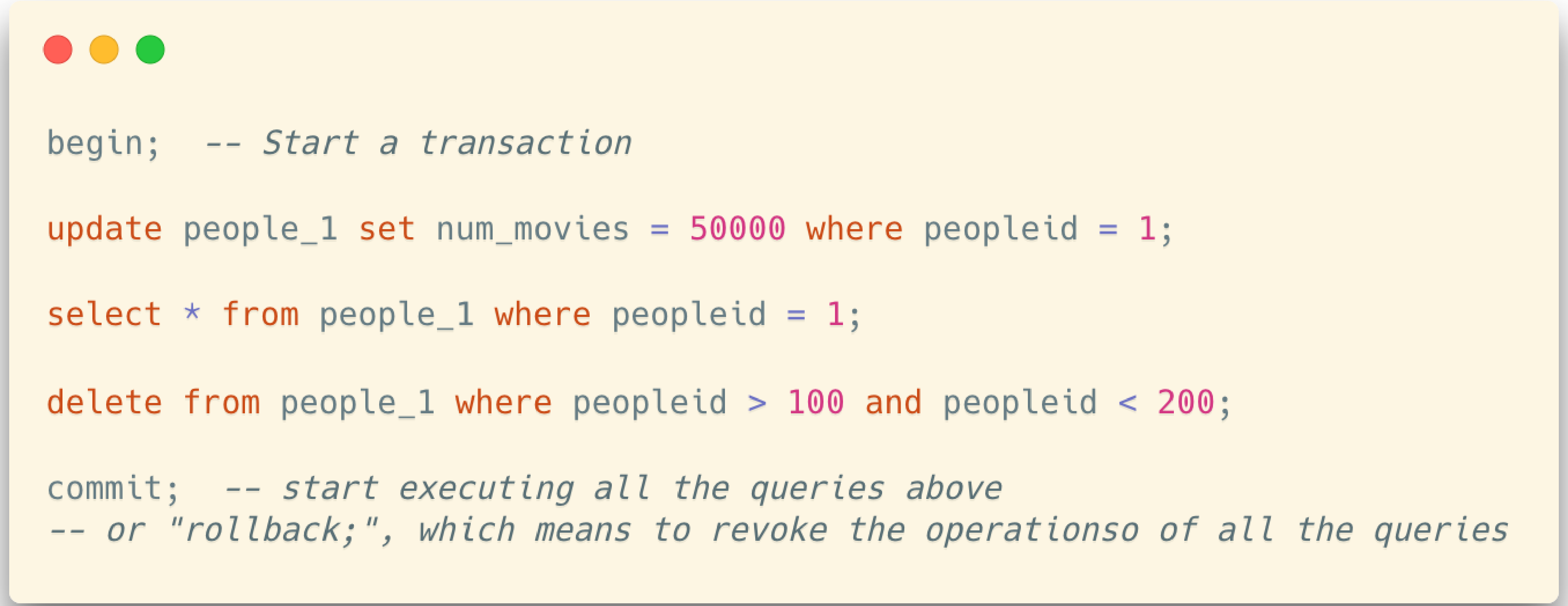
- A transaction is a unit of program execution that accesses and possibly updates various data items
  - A classical example in database: money transfer

E.g., transaction to transfer CNY ¥50 from account A to account B:

```
read(A) // read from disk to main memory
A := A – 50
write(A) // write from main memory to disk
read(B) // read from disk to main memory
B := B + 50
write(B) // write to disk
```

An Example of Transactions in PostgreSQL

- BEGIN, COMMIT, ROLLBACK



```
begin;  -- Start a transaction

update people_1 set num_movies = 50000 where peopleid = 1;

select * from people_1 where peopleid = 1;

delete from people_1 where peopleid > 100 and peopleid < 200;

commit;  -- start executing all the queries above
-- or "rollback;", which means to revoke the operationso of all the queries
```

Transaction in Computer

- A transaction is a unit of program execution that accesses and possibly updates various data items
  - A classical example in database: money transfer

E.g., transaction to transfer CNY ¥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)
```

  - Two main issues to deal with:
    - Failures of various kinds, such as hardware failures and system crashes
    - Concurrent execution of multiple transactions

Requirements in Transactions

- Atomicity Requirement
  - If the transaction ***fails*** after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that <u>updates of a partially executed transaction are not reflected in the database</u>

E.g., transaction to transfer CNY ¥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)
```

Requirements in Transactions

- 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** even if there are software or hardware failures.

Requirements in Transactions

- Consistency Requirement
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - Application-dependent consistency constraints that are too complex to state using the SQL constructs for data integrity
    - e.g., sum of balances of all accounts
      - In the example: The sum of A and B is unchanged by the execution of the transaction

E.g., transaction to transfer CNY ¥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)
```

Requirements in Transactions

- Isolation Requirement
  - If between steps 3 and 6, another transaction **T2** 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

```
T1                          T2
1.  read(A)
2.  A := A – 50
3.  write(A)
                               read(A), read(B), print(A+B)


4.  read(B)
5.  B := B + 50
6.  write(B)
```

  - Isolation can be ensured trivially by running transactions serially, that is, one after the other
    - However, executing multiple transactions concurrently has significant benefits

ACID Properties

- A transaction is a unit of program execution that accesses and possibly updates various data items
  - To preserve the integrity of data the database system must ensure:

Atomicity: Either all operations of the transaction are properly reflected in the database, or none are

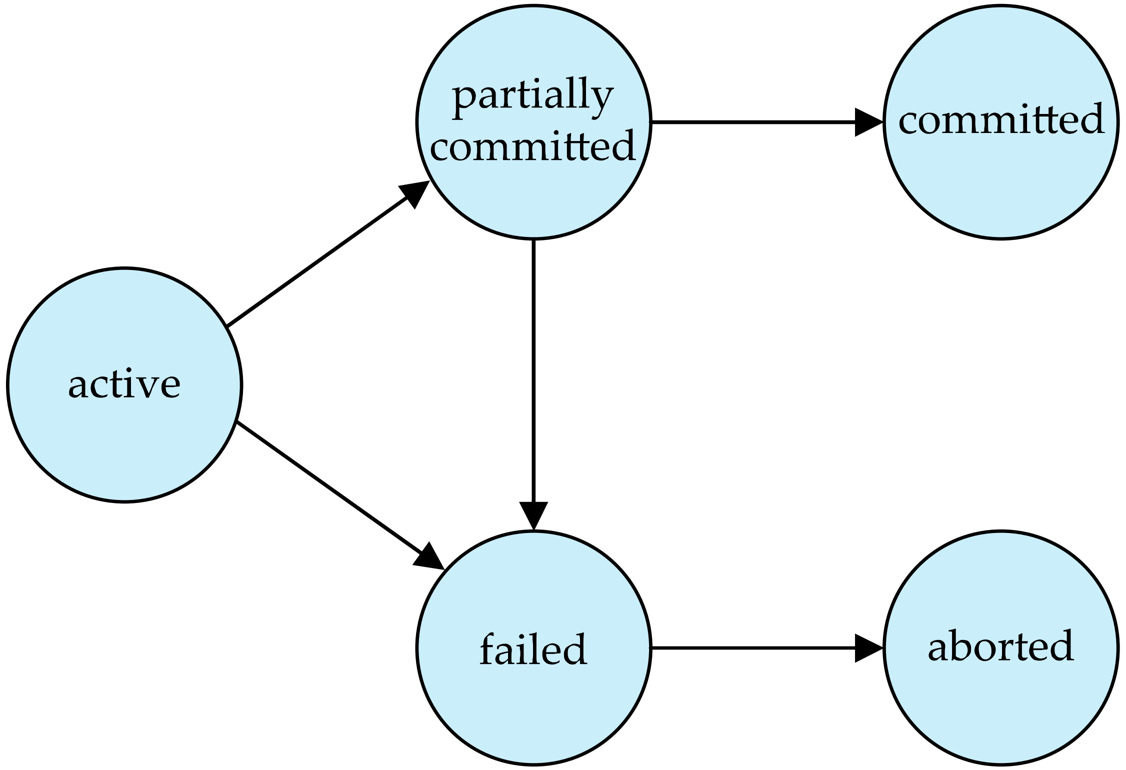
Consistency: Execution of a transaction in isolation preserves the consistency of the database.

Isolation: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

- That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.

Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State



- Active
  - The initial state; the transaction stays in this state while it is executing
- Partially committed
  - After the final statement has been executed
- Failed
  - After the discovery that normal execution can no longer proceed
- Aborted – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - Can be done only if no internal logical error
  - Kill the transaction
- Committed
  - After successful completion

Concurrent Executions

- Isolation can be ensured by running transactions serially (one after another)
- Multiple transactions are allowed to run concurrently in the system
  Advantages are:
  - Increased processor and disk utilization, leading to better transaction throughput
    - E.g., 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 do not need to wait behind long ones
- Concurrency control schemes – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- **Schedule** – <u>a sequences of instructions</u> that specify the chronological order in which <u>instructions</u> 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

- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default, transaction assumed to execute commit instruction as its last step

- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer CNY ¥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)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

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)
	commit
read (A)	
$A := A – 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously
  - When DBMS executes several transactions concurrently, the corresponding schedules are no longer serial
  - The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1
    - In Schedules 1, 2 and 3, the sum A + B is preserved.

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)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 3 vs. 1

- Let T_1 and T_2 be the transactions defined previously
  - When DBMS executes several transactions concurrently, the corresponding schedules are no longer serial
  - The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1
    - In Schedules 1, 2 and 3, the sum A + B is preserved.

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)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 3

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

Schedule 1

Schedule 4

- Several execution sequences are possible, since the various instructions from both transactions may now be interleaved
  - Not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction
  - Not all concurrent executions result in a correct state
- The following concurrent schedule does not preserve the value of (A + B)
  - A is deducted by 50, but B is increased by A*0.1

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)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- To ensure database consistency
  - Ensure that any schedule executed has the same effect as a serial schedule

- Basic assumption:
  - Each transaction preserves database consistency
  - Thus, serial execution of a set of transactions preserves database consistency

- A (possibly concurrent) schedule is **serializable** if it is equivalent to a **serial schedule**
  - Different forms of schedule equivalence give rise to the notions of:
    - 1. Conflict serializability
    - 2. * View serializability

Simplified View of Transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions l_i and l_j , of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q
  - 1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict
  - 2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
  - 3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  - 4. l_i = **write**(Q), l_j = **write**(Q). They conflict
  - *If l_i and l_j refer to different items, no conflicts

- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
  - If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent**

- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1
  - … by series of swaps of non-conflicting instructions
  - Therefore, Schedule 3 is *conflict serializable*.

Operations on different data

- … and hence swappable in temporal order

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 4

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability

- 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 >.

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n

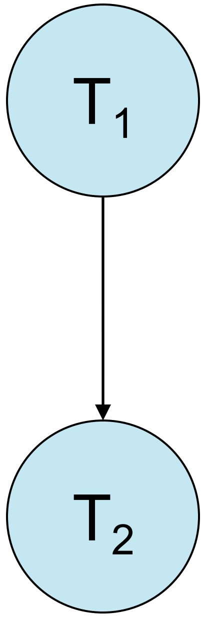
- Precedence graph
  - A directed graph where the vertices are the transactions (names of the transactions)
  - We draw an arc from T_i to T_j if the two transactions **conflict**
    - which means, in the schedule S, T_i **must** appear earlier than T_j
  - We may label the arc by the item that was accessed.

Conflict – At least one of the following situations exists for a data item Q:

- **T_i: write(Q) -> T_j: read(Q)**
- **T_i: read(Q) -> T_j: write(Q)**
- **T_i: write(Q) -> T_j: write(Q)**

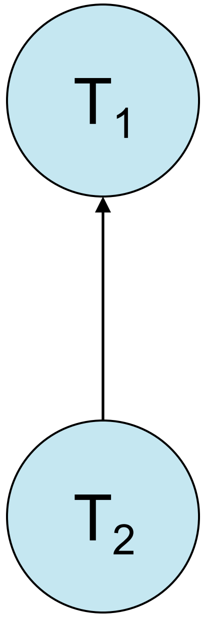
Testing for Serializability

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



Schedule 1

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

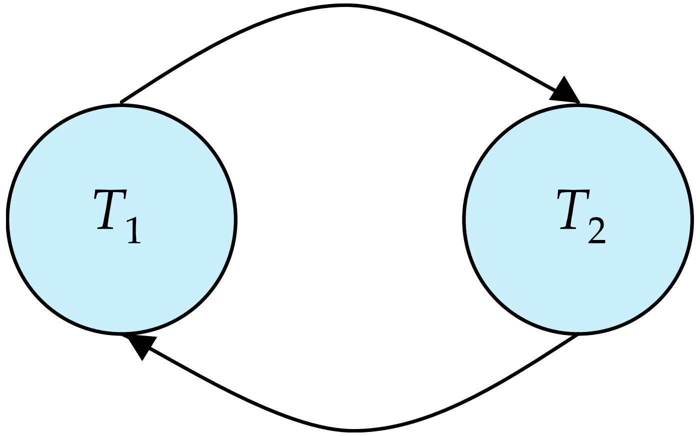


Schedule 2

Testing for Serializability

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)	
commit	
	$B := B + temp$
	write (B)
	commit

Schedule 4



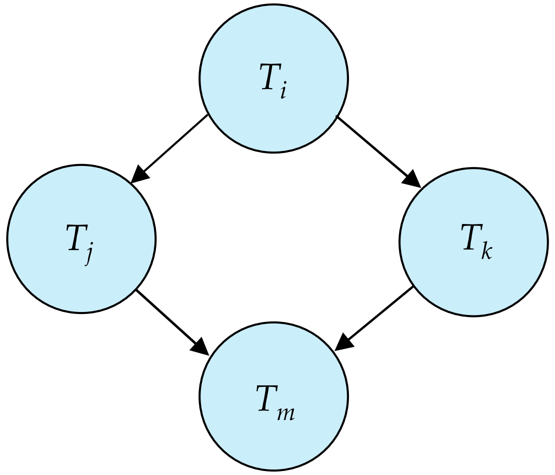
T1 -> T2, T1 reads A before T2 writes A
T2->T1, T2 reads B before T1 writes A

Testing for Serializability

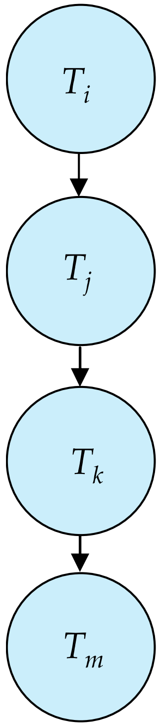
- A schedule is <u>conflict serializable</u> if and only if its precedence graph is acyclic

Cycle-detection: Cycle-detection algorithms exist which take n^2 time, where n is the number of vertices in the graph.

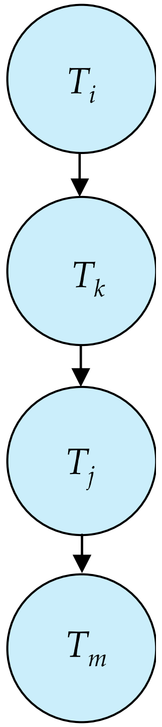
- If the precedence graph is acyclic, the <u>serializability order</u> can be obtained by a *topological sorting* of the graph
  - E.g., The topological order of (a) can be (b) and (c)



(a)



(b)



(c)

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

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
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.
  - To be recoverable, T_9 needs to commit after T_8

Weak Levels of Consistency

- If every transaction can maintain consistency if executed alone, then
  - Serializability can ensure that concurrent executions maintain consistency
  - but, too little concurrency can be achieved for certain applications

- 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
    - Such transactions do not need to be serializable with respect to other transactions
  - Purpose: Trade-off between accuracy and performance

Levels of Consistency (in SQL-92)

- Serializable (Strongest)
  - 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.
  - Successive reads of record may return different (but committed) values.
- Read uncommitted (Weakest) — even uncommitted records may be read.

Levels of Consistency

- Lower degrees of consistency can be useful for gathering approximate information about the database

- Warning: some database systems do not ensure serializable schedules by default
  - E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)

- Warning 2: All SQL-92 consistency levels infer that dirty writes are prohibited
  - Dirty write - when one transaction overwrites a value that has previously been written by another still in-flight transaction