Unit VIII: Transactions

Dr. Anushree Tripathi
Department of Computer Science
National Institute of Technology Patna
Patna

Overview

- Transaction concept
- Transaction state
- System log
- Commit point
- Desirable Properties of a Transaction
- Concurrent executions
- Serializability
- Recoverability
- Implementation of isolation
- Transaction definition in SQL
- Testing for serializability

Transaction concept

Basic

- Unit of program execution that accesses and updates data items
- All operations between begin transaction and end transaction

Example

- Transaction to transfer $50 from account A to account B:

read(A)

$A := A - 50$

write(A)

read(B)

$B := B + 50$

write(B)

Transaction concept

Properties

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

Atomicity

- Either all operations of transaction reflected or none
- If transaction fails after step 3 or step 6 leads to inconsistent database

Consistency

- Sum of A and B unchanged by the execution of the transaction
- Requirements
- Explicitly specified integrity constraints such as primary keys and foreign keys
- Implicit integrity constraints

Isolation

- Each transaction is unaware of other transactions executing concurrently in the system

Durability

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

Transaction concept

Properties

Isolation

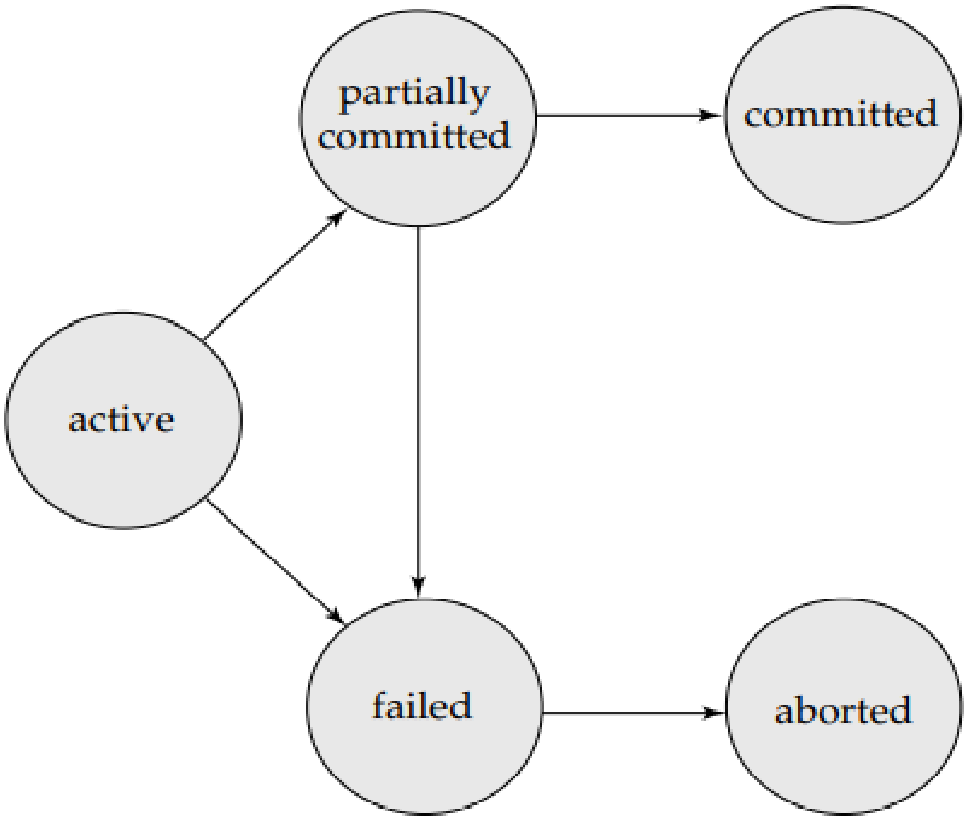
T1	T2
1. **read**(*A*)	
2. A: = A-50	
3. write(A)	
	read(A),read(B), print(A+B)
4.read(B)	
5. B: = B+50	
6. Write(B)	

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).

Transaction state

States of transaction	Active: Initial state, transaction stays while executing
	Partially committed: State after final statement executed
	Failed: State after finding normal execution no longer proceed
	Rollback or Aborted: Indicates transaction has ended unsuccessfully, changes must be undone
	Committed: State after successful completion

Transaction state (contd.)



State diagram of a transaction

Reference: Silberschatz−Korth−Sudarshan, Database System Concepts, Sixth Edition

System log

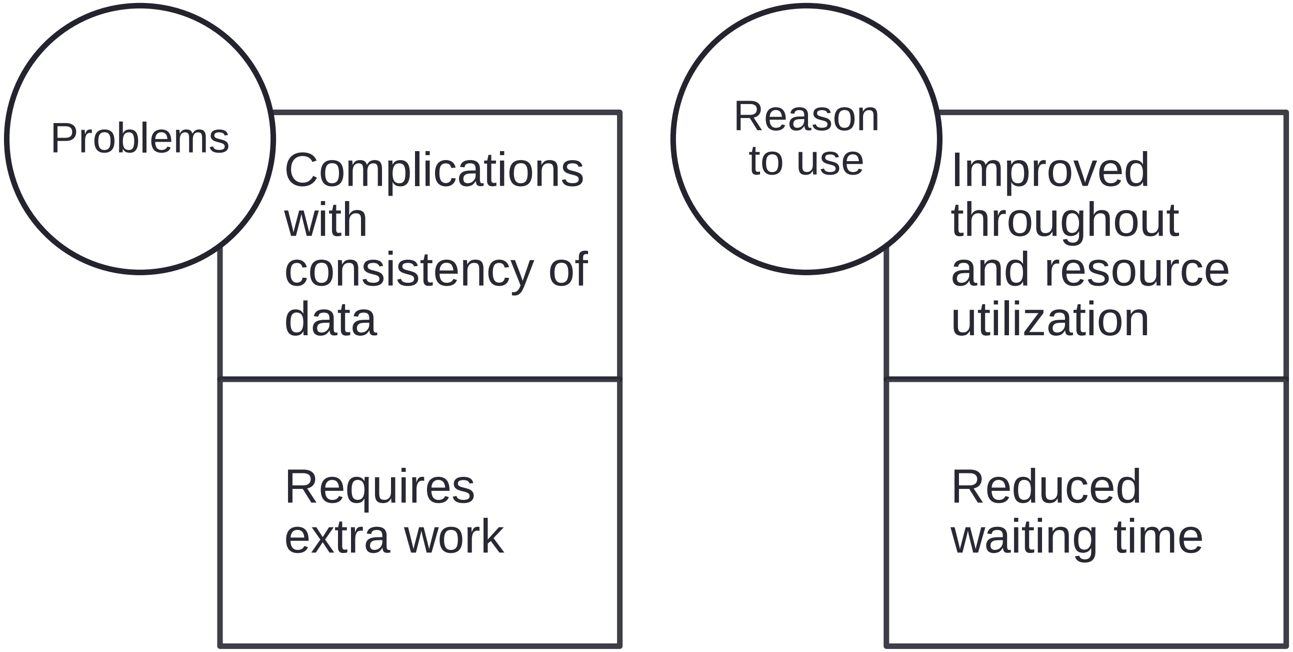
- To recover from transaction failure by restoring most recent consistent state
- To keep information about changes to data items
- Strategies for recovery
  - Reconstructs a more current state by reapplying the operations of committed transactions from backed up log in case of extensive damage
  - Identify any changes that may cause an inconsistency in database. System log entries determine appropriate actions for recovery
- Contains list of active transactions (checkpoints). Actions of checkpoints:
  - Suspend execution of transaction
  - Force-write all main memory buffers, modified to disk
  - Write a record to the log
  - Resume execution transaction

Commit point

- 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 have been recorded in the log
- Its effect must be permanently recorded

Concurrent executions

- Transaction-processing systems allow multiple transactions to run concurrently



Concurrent executions

Concurrency control schemes

- To achieve isolation
- To control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Concurrent executions

Schedules

Sequence of instructions to specify chronological order of execution

Must preserve order of instructions

Concurrent executions

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)

Concurrent executions

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)	

Concurrent executions

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)

Concurrent executions

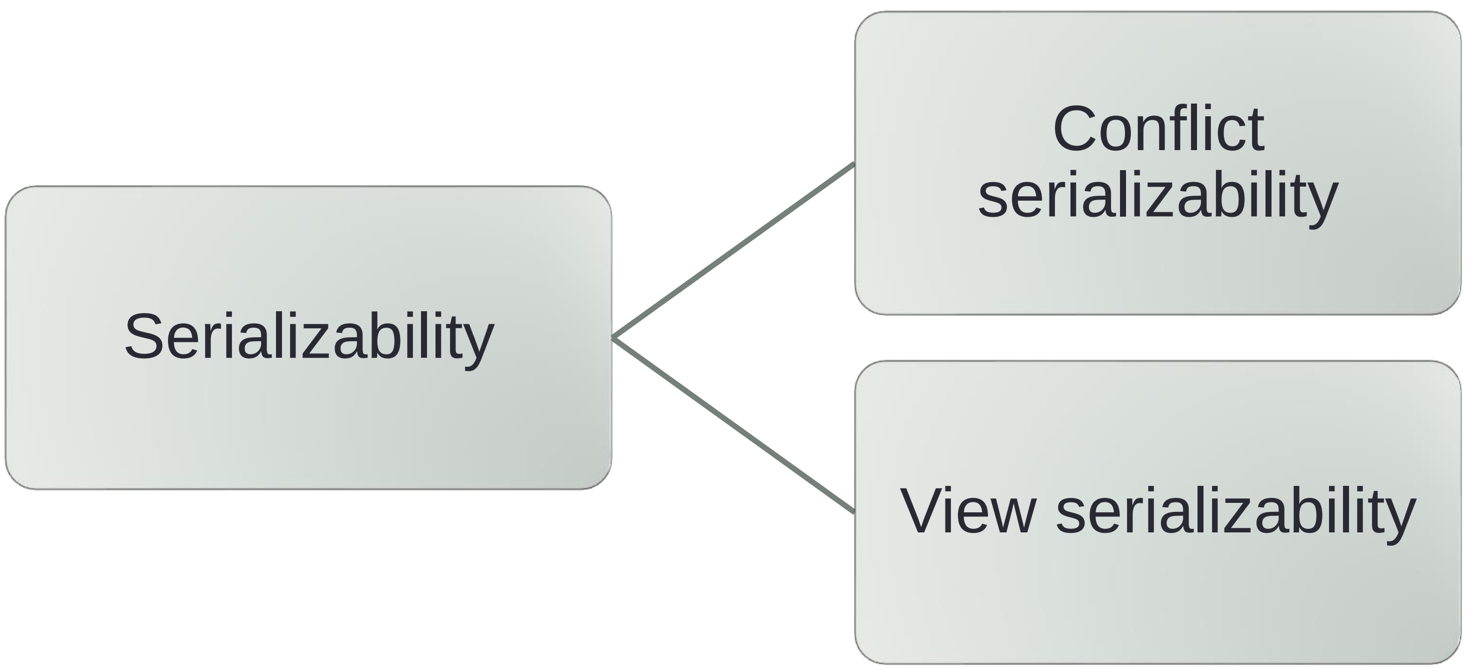
Schedule 4

- 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)

Serializability

- Need
- ➢To preserves database consistency
- ➢Schedule is serializable if it is equivalent to a serial schedule
-



Conflict serializability

Four cases	l_i = **read**(Q), l_j = **read**(Q) l_i and l_j No conflict
	l_i = **read**(Q), l_j = **write**(Q) Conflict
	l_i = **write**(Q), l_j = **read**(Q) Conflict
	l_i = **write**(Q), l_j = **write**(Q) Conflict

Conflict serializability (contd.)

S and *S´* are conflict equivalent if *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions

S is conflict serializable if it is conflict equivalent to a serial schedule

Conflict serializability (contd.)

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(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

Conflict serializability (contd.)

- No conflict serializability
- No swapping of instructions either the serial schedule <T_3, T_4>, or the serial schedule <T_4, T_3>

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

Testing for serializability

How to determine conflict serializability of a schedule

Precedence graph

Directed graph where the vertices are the transactions

G= (V,E)

Set of edges consists of all edges $Ti \rightarrow Tj$

Cycle detection algorithm

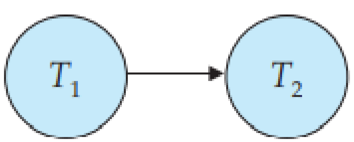
Based on depth-first search, require on order of n^2

Schedule is conflict serializable if and only if its precedence graph is acyclic

Testing for serializability (contd.)

Schedule 1

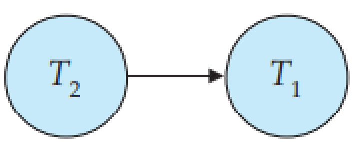
T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit



Precedence graph for schedule 1

Schedule 2

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	



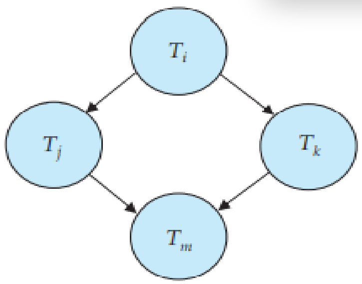
Precedence graph for schedule 2

Reference: Silberschatz−Korth−Sudarshan, Database System Concepts, Sixth Edition

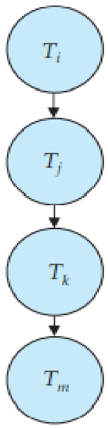
Testing for serializability (contd.)

Topological sorting

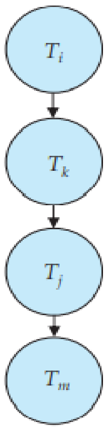
- To find serializability order, if precedence graph is acyclic



(a)



(b)



(c)

Reference: Silberschatz−Korth−Sudarshan, Database System Concepts, Sixth Edition

Recoverability

- 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
- Schedule 11 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

Basics

Single transaction failure leads to a series of transaction rollbacks

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

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

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.
- Every cascadeless schedule is also recoverable

Concurrency Control

mechanism that will ensure that all possible schedules are

either conflict or view serializable, and

are recoverable and preferably cascadeless

to develop concurrency control protocols that will assure serializability

Implementation of isolation

Concurrency control mechanisms

Locking

Two-phase locking to ensure serializability

First phase: Transaction acquires locks

Second phase: Transaction releases locks

Shared locks

Transaction reads

Exclusive locks

Transaction writes

Timestamps

Technique assigns each transaction a timestamp, when it begins

For each data item, system keeps two timestamps for reading and writing data

Transaction definition in SQL

DML must include a construct for specifying the set of actions that comprise a transaction

In SQL, 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.

In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully