COMP 3311
DATABASE MANAGEMENT SYSTEMS

LECTURE 20
TRANSACTIONS

TRANSACTIONS: OUTLINE

Overview

Schedules

Serializability

OVERVIEW

A *transaction* is a *unit* of program execution that accesses, and possibly updates, the database.

- A transaction must preserve database consistency.
  - During transaction execution the database may be inconsistent.
  - However, when the transaction commits (ends), the database must be consistent.
- Two main issues to deal with:
  - Concurrent execution of multiple transactions ⟹ concurrency control.
  - Failures of various kinds, such as hardware failures and system crashes ⟹ database recovery.

ACID PROPERTIES

Atomicity

Either all operations of a transaction are properly reflected in the database or none are.

Consistency

Execution of a transaction *in isolation* (i.e., it is the only transaction executing) preserves the consistency of the database.

Isolation

Concurrently executing transactions must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executing transactions.

- 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 started execution after T_i finished (i.e., the execution of T_i and T_j is serial).

Durability

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

ACID PROPERTIES: FUNDS TRANSFER EXAMPLE

Consider a transaction to transfer $50 from account A to account B.

Atomicity Requirement

Handled by database recovery.

If the transaction fails *after* step 3 and *before* step 6, the system should ensure that the transaction's updates are not reflected in the database, else an inconsistency will result.

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

transaction failure

Consistency Requirement

Handled by concurrency control.

The sum of A and B is unchanged by the execution of the transaction.

ACID PROPERTIES: FUNDS TRANSFER EXAMPLE (CONT'D)

Isolation Requirement

Handled by concurrency control.

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.

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

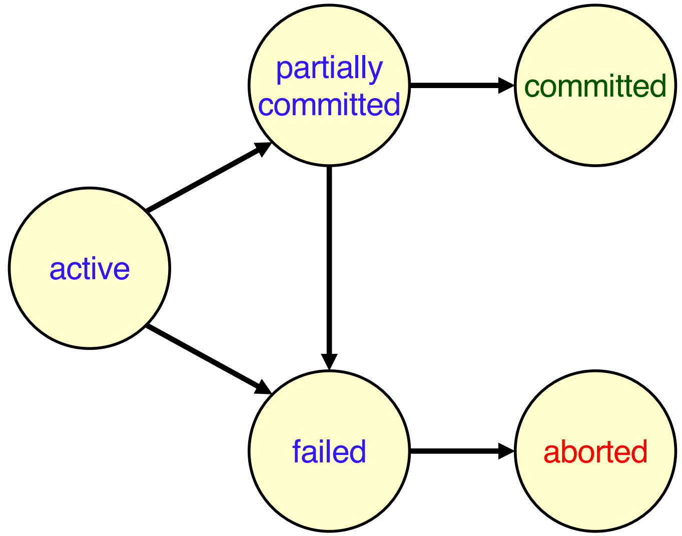
transaction failure

Durability Requirement

Handled by database recovery.

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.

TRANSACTION STATE



Active

The initial state; the transaction stays in this state while executing.

Partially committed

After the final instruction in the transaction 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. There are two options for how to handle a transaction after it has been aborted:

- restart the transaction – only if there is no internal logical error.
- kill the transaction.

Committed

After *successful completion*.

CONCURRENT EXECUTIONS AND ISOLATION

- Allowing concurrent execution of transactions has the following advantages:
  - increased processor and disk utilization, leading to better transaction *throughput* since one transaction can be using the CPU while another is reading from or writing to the disk.
  - reduced average response time for transactions. Short duration transactions need not wait behind long duration ones.

☞ **Concurrent execution can lead to violation of the isolation property.**

- **Concurrency control schemes** are mechanisms used to achieve isolation in the presence of concurrent executions.

☞ They control the interaction among the concurrent transactions to prevent them from destroying the database consistency due to problems that can arise when concurrent transactions access the same data.

SCHEDULES

- A schedule is a sequence that indicates 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.
  - A schedule must preserve the order in which the instructions appear in each individual transaction.

- To preserve the consistency of the database, a schedule for executing transactions must be one of the following.

  serial — the transactions are executed one after the other
  ⟹ all the instructions of a transaction are executed *before* the next transaction starts executing.

  serializable — the transactions are executed concurrently (i.e., at the same time) in an *interleaved* manner so that their execution is equivalent to a serial schedule.

SERIAL SCHEDULE

T_1 transfers \$50 from A to B; T_2 transfers 10% of the balance from A to B.

A serial schedule is shown to the right, in which T_1 is followed by T_2.

Note: A schedule in which T_2 is followed by T_1 is also a serial schedule (*but may have different final results*).

☞ For *n* transactions there exist *n*! different valid serial schedules.

Start with A=100, B=200; A+B=300

	T_1	T_2	
A=100	read(A)		
	A := A - 50		
A=50	write(A)		
B=200	read(B)		
	B := B + 50		
B=250	write(B)		
	commit		
		read(A)	A=50
		temp := A * 0.1	*temp*=5
		A := A - *temp*	
		write(A)	A=45
		read(B)	B=250
		B := B + *temp*	
		write(B)	B=255
		commit	

End with A=45, B=255; A+B=300

NOT SERIAL BUT CORRECT SCHEDULE

T_1 transfers \$50 from A to B; T_2 transfers 10% of the balance from A to B.

This schedule is equivalent to the previous serial schedule.

☞ **In both schedules, the sum A + B is preserved.**

Start with A=100, B=200; A+B=300

	T_1	T_2	
A=100	read(A)		
	A := A - 50		
A=50	write(A)		
		read(A)	A=50
		temp := A * 0.1	*temp*=5
		A := A - *temp*	
		write(A)	A=45
B=200	read(B)		
	B := B + 50		
B=250	write(B)		
	commit		
		read(B)	B=250
		B := B + *temp*	
		write(B)	B=255
		commit	

Again end with A=45, B=255; A+B=300

NOT SERIAL AND INCORRECT SCHEDULE

T_1 transfers \$50 from A to B; T_2 transfers 10% of the balance from A to B.

This schedule is incorrect and should not be allowed.

☞ **The schedule does not preserve the value of sum A + B.**

Start with A=100, B=200; A+B=300

	T_1	T_2	
A=100	read(A)		
	A := A - 50		
		read(A)	A=100
		temp := A * 0.1	*temp*=10
		A := A - *temp*	
		write(A)	A=90
		read(B)	B=200
A=50	write(A)		
B=200	read(B)		
	B := B + 50		
B=250	write(B)		
	commit		
		B := B + *temp*	
		write(B)	B=210
		commit	

Now end with A=50, B=210; A+B=260

SERIALIZABILITY

Basic Assumption
Each transaction preserves database consistency.

☞ **A serial execution of a set of transactions preserves database consistency.**

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.

- We ignore operations other than read and write instructions, and 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.**

CONFLICT SERIALIZABILITY

Instructions I_i and I_j of transactions T_i and T_j respectively, conflict *if and only if* there exists some data item Q accessed by both I_i and I_j, and at least one of these instructions writes Q.

1. I_i = read(Q), I_j = read(Q). I_i and I_j do not conflict.
2. I_i = read(Q), I_j = write(Q). I_i and I_j conflict.
3. I_i = write(Q), I_j = read(Q). I_i and I_j conflict.
4. I_i = write(Q), I_j = write(Q). I_i and I_j conflict.

- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
- If I_i and I_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 (CONT'D)

Conflict Equivalent

Schedules S and S' are conflict equivalent if S can be transformed into S' by a series of swaps of non-conflicting instructions.

Conflict Serializable

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

Example: A schedule that is not conflict serializable.

T_1	T_2
read(Q)	
	write(Q)
write(Q)	

☞ Instructions in the above schedule cannot be swapped to obtain either the serial schedule <T_1, T_2>, or the serial schedule <T_2, T_1>.

CONFLICT SERIALIZABLE SCHEDULE

The following schedule is equivalent to a serial schedule, where T_2 follows T_1, by a series of swaps of non-conflicting instructions.

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

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

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

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

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

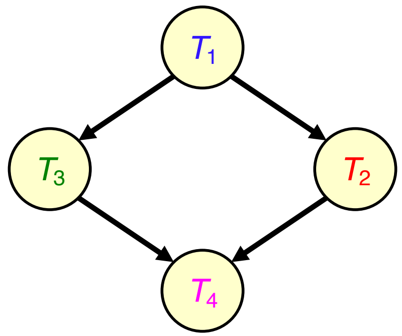
serial schedule

TESTING FOR SERIALIZABILITY

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n.
- A precedence graph is a directed graph where the vertices are the transactions (i.e., their names).
- We draw an arc from T_i to T_j if the two transactions conflict and T_i accessed the data item on which the conflict arose earlier than T_j.
- We can label the arc by the data item that was accessed, if desired.

EXAMPLE SCHEDULE & PRECEDENCE GRAPH

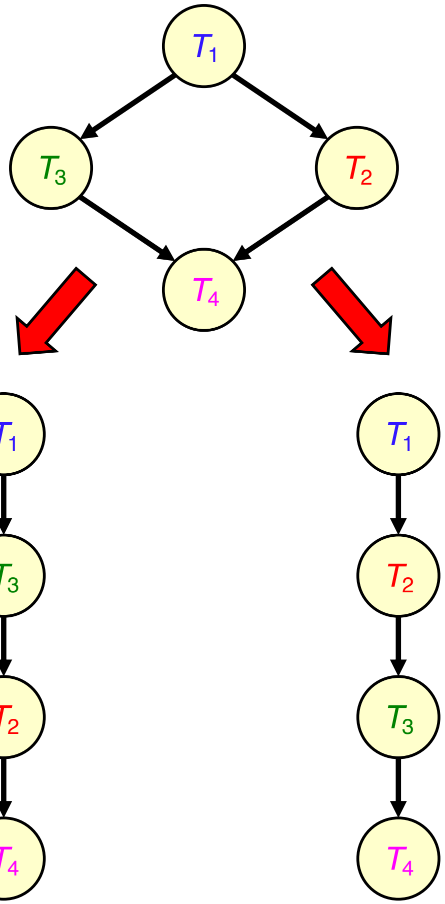
T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Precedence graph for the schedule.

CONFLICT SERIALIZABLE PRECEDENCE GRAPH

The precedence graph can be topologically sorted to find a linear order consistent with the partial order of the precedence graph.



☞ **If the precedence graph has a cycle, then the schedule is <u>not</u> conflict serializable.**

RECOVERABILITY

A schedule is recoverable if a transaction, T_j, that reads a data item *previously written* by a transaction, T_i, commits after T_i. (i.e., T_i commits before T_j).

☞ **A schedule that is not recoverable violates the durability property.**

Example

- T_2 commits immediately after the read.
- If T_1 aborts, then T_2 should also abort since it read a data item written by T_1.
- However, since T_2 has committed, it cannot abort and so can show to the user an inconsistent database state violating the durability property.

T_1	T_2
read(A)	
write(A)	
	$\boxed{\text{read(A)}}$
	commit
read(B)	

☞ **All schedules must be recoverable.**

CASCADING ROLLBACK

- Cascading rollback happens when a single transaction failure leads to a series of transaction rollbacks.
- Although this schedule is recoverable, because every transaction T_i commits after all transactions that wrote data items which T_i read, if T_1 fails, T_2 and T_3 must also be rolled back.
- This can lead to the undoing of a significant amount of work.

How would you (re)order the commit statements to make the schedule cascadeless?

T_1	T_2	T_3
read(A)		
write(A)		
	read(A)	
	write(A)	
		read(A)
commit		
	commit	
		commit

Cascadeless schedule

T_1	T_2	T_3
read(A)		
write(A)		
commit		
	read(A)	
	write(A)	
	commit	
		read(A)
		commit

CASCADELESS SCHEDULES

A schedule is cascadeless if, for each pair of transactions T_i and T_j where 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.

- Cascadeless schedules are schedules where cascading rollbacks cannot occur.

  ☞ **Every cascadeless schedule is also recoverable.**

- It is highly desirable to restrict the schedules to those that are cascadeless.

IMPLEMENTATION OF ISOLATION

- Schedules must be serializable, and recoverable, for the sake of database consistency, and preferably, cascadeless.
  - A policy whereby only one transaction can execute at a time generates serial schedules, but provides no concurrency.

☞ **Testing a schedule for serializability *after* it has executed is too late!**

Goal: Develop protocols that both allow concurrency and ensure serializability.

- A protocol will generally not examine the precedence graph as it is being created; instead, it will impose a discipline on the order in which data items are accessed that avoids nonseralizable schedules.

☞ **Protocols trade-off between the amount of concurrency allowed and the amount of overhead that is incurred to ensure that schedules are serializable and recoverable.**

TRANSACTIONS: SUMMARY

- A transaction is a unit of program execution that accesses and possibly updates the database.
- Transactions are required to have the ACID properties: Atomicity, Consistency, Isolation, and Durability.
- Concurrent execution of transactions improves throughput of transactions and system utilization and reduces waiting time.
- When transactions execute concurrently, the system may need to control the interaction among the concurrent transactions.
- A schedule captures the key actions of transactions that affect concurrent execution.
  - A schedule must be serializable, recoverable and preferably cascadeless.