Transaction Management

Chapter 14

What we want to cover

- Transaction model
- Transaction schedules
- Serializability
- Atomicity

Chapter 14

TRANSACTION MODEL

Transaction Requirements

- Eg. 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, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - the system should ensure that updates of a partially executed transaction are not reflected in the database
- **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.

Transaction Requirements (Cont)

- 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*)
- **Consistency requirement** in above example:
  - the sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction must start with a consistent database and leave one when it completes

Transaction Requirements (Cont)

- **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

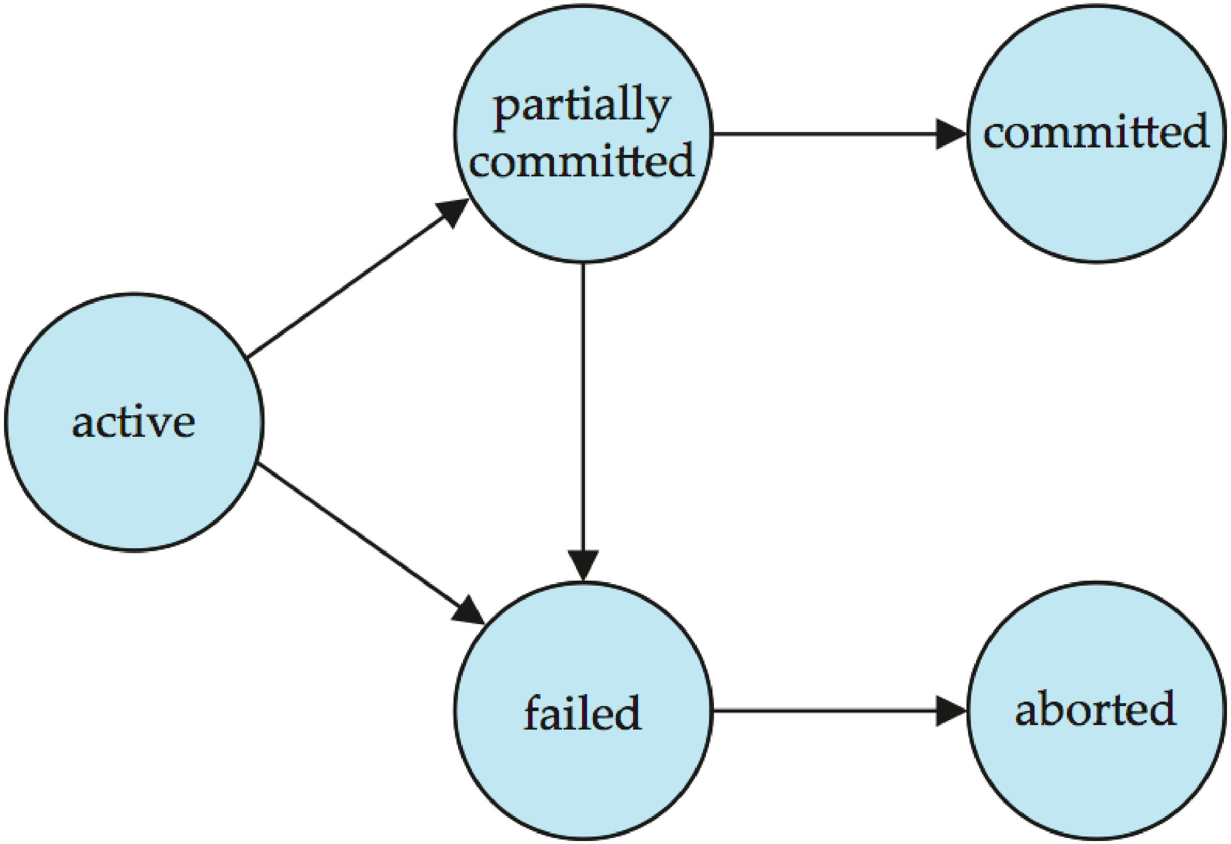
T1	**T2**
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), read(B), print(A+B)
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*	

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.

ACID Properties

- **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



Chapter 14

TRANSACTION SCHEDULES

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.
- Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- How do we model and analyze concurrent behaviour?

Schedules

- **Schedule** – a sequence of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - must consist of all instructions of those transactions
  - must preserve the order of individual transactions.
- A transaction that successfully completes its execution will have a commit as the last statement
- A transaction that fails to successfully complete its execution will have an abort as the last statement

Example 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*)	
commit	
	read (*A*)
	$temp := A * 0.1$
	$A := A - temp$
	write (*A*)
	read (*B*)
	$B := B + temp$
	write (*B*)
	commit

Example 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	

Example Schedule 3

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

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

Example Schedule 4

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

Chapter 14

SERIALIZABILITY

Serializability

- **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.

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). don't conflict.
  2. l_i = **read**(Q), l_j = **write**(Q). conflict.
  3. l_i = **write**(Q), l_j = **read**(Q). conflict
  4. l_i = **write**(Q), l_j = **write**(Q). conflict
- 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 (Cont.)

- Schedule S1 can be transformed into Schedule S1', a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule S1 is conflict serializable.

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

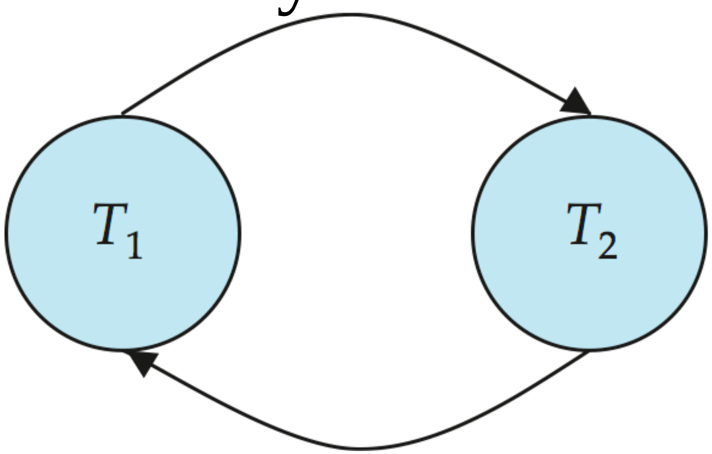
S1

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

S1'

Testing for Serializability

- Consider some schedule of a set of transactions $T_1, T_2, \ldots, T_n$
- **Precedence graph** — a directed graph where the vertices are the transactions (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 before T_j.
- We may label the arc by the item that was accessed.
- **Example 1**

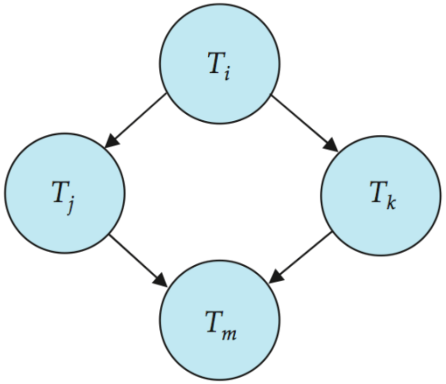


Test for Conflict Serializability

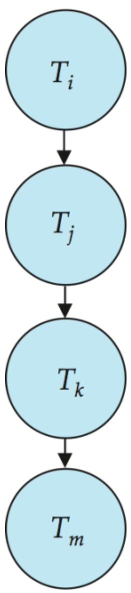
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, 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 orders for Schedule (a) are

    $T_i \rightarrow T_j \rightarrow T_k \rightarrow T_m$

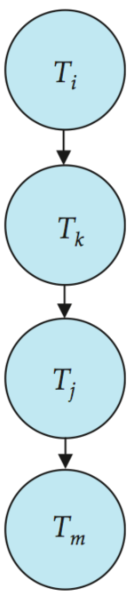
    $T_i \rightarrow T_k \rightarrow T_j \rightarrow T_m$



(a)



(b)



(c)

Serializable?

T1	**T2**	**T3**
read(X)		
read(Z)		
write(X)		
	read(X)	
write(Z)		
commit		
	write(X)	
	read(Z)	
	read(Y)	
		read(Y)
		write(Y)
		commit
	write(Z)	
	commit	

Chapter 14

ATOMICITY & ISOLATION LEVELS

Recoverable Schedules

- **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)
	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.

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)
abort		

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

WHAT'S COMING UP IN THE COURSE

- **Week 3 – Sept 26 - 30**
  - Lectures – RDBMS implementation issues, RDBMS architectures
- **Week 4 – Oct 3 – 7**
  - Assignment 1 due Oct 4
  - Bluemix tutorial – Oct 4
  - Lectures – RDBMS architectures
  - ***Big Data 175 Lecture*** – Oct 4 6:30 pm, Goodes Hall Commons
- **Week 5 – Oct 10 – 14**
  - 832 paper proposal due Oct 14
  - Lectures – RDBMS architectures