Module 17: Transactions

Database System Concepts, 7th Ed.

Outline

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Types of Serializability
  - Conflict Serializability
  - View Serializability
- Test for Serializability

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., 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*)
- Two main issues to deal with during transaction execution:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties of a Transaction

To preserve the integrity of data, the database system must maintain the following properties of transactions:

- **Atomicity (A):** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency (C):** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation (I):** 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 (D):** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

- 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 failure.
  - 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.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include the following:
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will observe an inconsistent database (the sum $A + B$ will be less than that it should be).

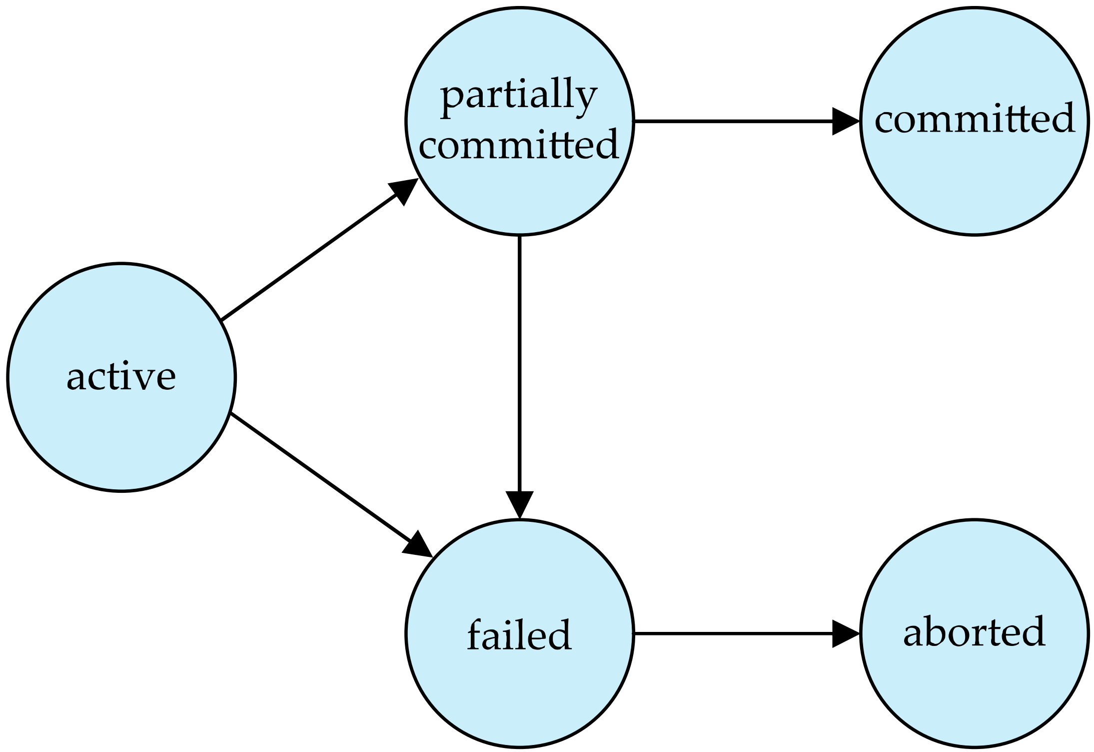
T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), write(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B)	

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

Transaction State

- **Active** – it is the initial state; the transaction stays in this state while it is executing
- **Partially committed** – This is the state in which a transaction reaches after the final statement has been executed.
- **Failed --** A transaction enters the failed state after the system determines that the transaction can no longer proceed with the normal execution.
- **Aborted** – Such a transaction can be rolled back and the database is restored to its state prior to the start of the transaction. At this point the database system has two options:
  - **Restart the transaction**
    - Can be done only if there is no internal logical error
  - **Kill the transaction**
- **Committed** – after successful completion.

Transaction State (Cont.)



Concurrent Executions

- If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a **context switch**, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on.
- With multiple transactions, the <u>CPU time is shared among all the transactions</u>.
- In concurrent execution, several execution sequences are possible, since the various instructions from both transactions may now be interleaved.
- In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction.
- Thus, the number of possible schedules for a set of *n* transactions is larger than *n*!.

Advantages of Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.
- The advantages of concurrent execution of transactions are:
  1. **Increased processor and disk utilization**, leading to better transaction *throughput.* For E.g., one transaction can be using the CPU while another is reading from or writing to the disk
  2. **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.
- The DBS must control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.
- This is done through a variety of mechanisms known as <u>concurrency control schemes</u>.

Schedules of a Transaction

- **Schedule** – a sequences of instructions that specify 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.
- A transaction that successfully completes its execution will have a **commit instruction** as the last statement
  - By default, a transaction is 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

Serial Schedule

- As the name says, all the transactions are executed serially one after the other.
- In serial Schedule, a transaction does not start execution until the currently running transaction finishes execution.

Transaction T1	Transaction T2
R(A)	
W(A)	
R(B)	
W(B)	
commit	
	R(A)
	W(B)
	commit

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

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	

Concurrent Schedule

- In a concurrent/non-serial schedule, multiple transactions execute concurrently, unlike the serial schedule, where one transaction must wait for another to complete all its operations.
- In this schedule, the other transaction proceeds without the completion of the previous transaction.
- All the transaction operations are interleaved or mixed with each other.

Transaction T1	Transaction T2
R1(A)	
W1(A)	
	R2(A)
	W2(A)
R1(B)	
W1(B)	
	R2(B)
	W2(B)

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

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

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

Serializability

- The database system must control concurrent execution of transactions, to ensure that the database state remains consistent.
- **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.
- Thus, if a non-serial schedule can be transformed into its corresponding serial schedule, it is said to be **serializable**.
- There are two different forms of serializability in DBMS:
  1. **Conflict Serializability**
  2. **View Serializability**

Conflicting Instructions

- For simplicity, we consider only **read** and **write** instructions in a transaction
- **Conflicting instructions** in a DBMS are operations that are performed on the same data or resource concurrently, and the order in which they are executed affects the final outcome
- Conflicting instructions in a transaction occur when two or more transactions try to access the same data item at the same time
- 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 is a write operation on the same data item.
    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
- Thus, l_i and l_j ***conflict*** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation

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** where, *S*: concurrent schedule, *S*': serial schedule.
- In our previous examples, Schedule1 is not conflict equivalent to Schedule2.
- Schedule1 is conflict equivalent to Schedule3, because the non-conflicting instructions of *T1* - read(*B) and write(B)* can be swapped with the *read(A)* and *write(A) instructions of T2* in *Schedule3* to generate its equivalent serial schedule *Schedule1.*
- The concept of conflict equivalence leads to the concept of conflict serializability.
- We say that a non-serial schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule.

Example of Conflict Serializability

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—a serial schedule in which T_1 is followed by T_2.

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—a concurrent schedule equivalent to schedule 1.

Schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule Schedule 1.

Example of Conflict Serializability (Cont.)

- 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.

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

Example of Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

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

- The schedule consists of only the significant operations (that is, the read and write) of transactions *T*3 and *T*4.
- This schedule is not conflict serializable, since it is not equivalent to either the serial schedule <*T*3,*T*4> or the serial schedule <*T*4,*T*3>.

View Serializability

- In this form of serializability, we consider a form of equivalence that is less stringent than conflict equivalence, but like conflict equivalence, it is also based on only the read and write operations of transactions.
- Let *S* and *S'* be two schedules with the same set of transactions. *S* and *S'* are **view equivalent** if the following three conditions are met, for each data item *Q:*
  1. For each data item *Q*, if transaction *Ti* reads the initial value of *Q* in schedule *S*, then transaction *Ti* must, in schedule *S'*, also read the initial value of *Q*.
  2. For each data item *Q*, if transaction *Ti* executes read(*Q*) in schedule *S*, and if that value was produced by a write(*Q*) operation executed by transaction *Tj* , then the read(*Q*) operation of transaction *Ti* must also read the value of *Q* that was produced by the same write(*Q*) operation of transaction *Tj* in schedule *S'*.
  3. For each data item *Q*, the transaction (if any) that performs the final write(*Q*) operation in schedule *S* must perform the final write(*Q*) operation in schedule *S'*.

View Serializability

- Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation.
- Condition 3, ensures that both schedules result in the same final system state.
- Schedule 1 is view equivalent to schedule 3, because the values of account *A and B read by transaction T2 were* produced by *T1 in both schedules.*
- **View Equivalence:** The concept of view equivalence leads to the concept of view serializability. A schedule *S is* ***view serializable if it is view equivalent to a serial schedule***
- Schedule 9 is view serializable. It is view equivalent to the serial schedule *<T3, T4, T6>, since the one read(Q) instruction reads* the initial value of *Q in both schedules, and T6 performs the final write of Q in both the* schedules.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Schedule 9—a view-serializable schedule.

View Serializability (Cont.)

- Every conflict-serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable.
- Schedule 9 is not conflict serializable, since every pair of consecutive instructions conflicts, and, thus, no swapping of instructions is possible.
- In Schedule 9, transactions *T4 and T6 perform write(Q) operations* without having performed a read(*Q) operation. Writes of this sort are called* ***blind*** **writes.**
- <u>Blind writes appear in any view-serializable schedule that is not conflict serializable</u>**.**

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Schedule 9—a view-serializable schedule.

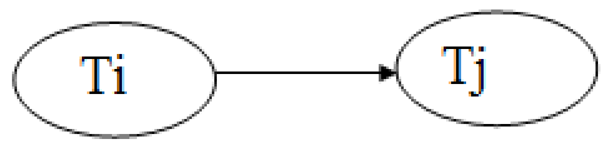
Testing for Serializability

- Given a particular schedule *S*, how to determine, whether the schedule is serializable or not.
- We now present a simple and efficient method for determining conflict serializability of a schedule.
- Consider a schedule *S*. We construct a directed graph, called a **precedence graph**, from *S*.
- This graph consists of a pair ***G* = (*V*, *E*)**, where ***V*** is a set of **vertices** and ***E*** is a set of **edges**.
- The set of vertices consists of all the transactions participating in the schedule.
- The set of edges consists of all edges *Ti* →*Tj* for which one of three conditions holds:
  1. *Ti* executes write(*Q*) before *Tj* executes read(*Q*).
  2. *Ti* executes read(*Q*) before *Tj* executes write(*Q*).
  3. *Ti* executes write(*Q*) before *Tj* executes write(*Q*).

Testing for Serializability

- If a precedence graph of a Schedule *S* contains a single edge Ti → Tj, then all the instructions of Ti are executed before the first instruction of Tj is executed.

Precedence graph for Schedule S



- If a **precedence graph for schedule S contains a cycle**, then **S is non-serializable**.
- If the **precedence graph** has **no cycle**, then **S is known as serializable**.