

# Lecture 12

# Transaction Processing (事务处理)

Shuigeng Zhou

June 4, 2014  
School of Computer Science  
Fudan University

# Outline

- ❑ Transaction Concept
- ❑ Concurrent Executions
- ❑ Serializability
- ❑ Recoverability
- ❑ Implementation of Isolation
- ❑ Transaction Definition in SQL
- ❑ Testing for Serializability

# Transaction Concept

- A ***transaction*** is a unit of program execution that accesses and possibly updates various data items
  - During transaction execution the database may be inconsistent
  - When the transaction is committed, the database must be consistent
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

# ACID Properties

## □ Atomicity (原子性)

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

## □ Consistency (一致性)

- Execution of a transaction in isolation preserves the consistency of the database.

## □ Isolation (隔离性)

- Although multiple transactions may execute concurrently, each transaction must be unaware of other transactions.

## □ Durability (持久性)

- 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( $B$ )`
- ❑ **Consistency requirement** – the sum of  $A$  and  $B$  is unchanged by the execution of the transaction.
- ❑ **Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

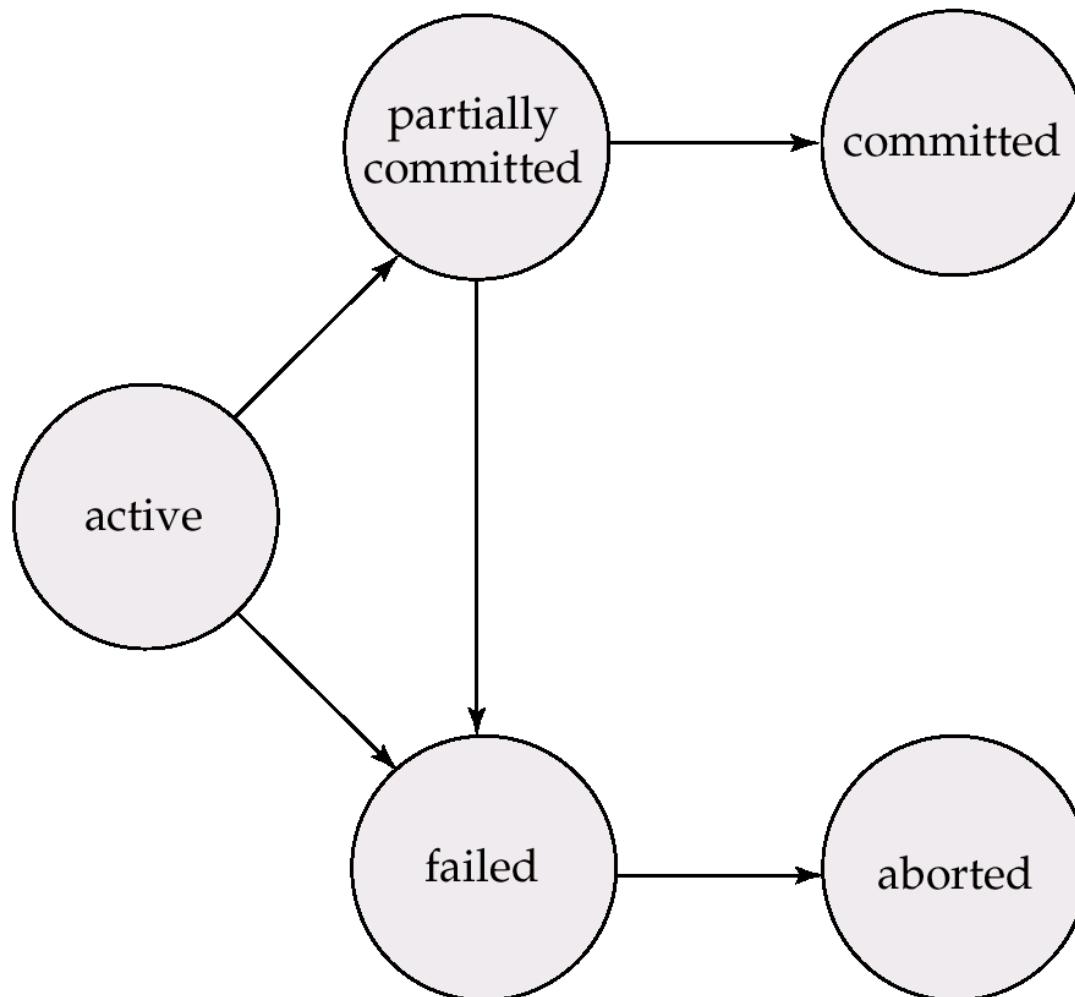
# Example of Fund Transfer (Cont.)

- **Durability requirement** — once the user has been notified that the transaction has completed, the updates to the database by the transaction must persist despite failures.
- **Isolation requirement** — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database
  - Can be ensured trivially by running transactions *serially*, that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.

# Transaction State

- **Active** (活跃) , the initial state; the transaction stays in this state while it is executing
- **Partially committed** (部分提交) , after the final statement has been executed.
- **Failed** (失败) , after the discovery that normal execution can no longer proceed.
- **Aborted** (夭折) , after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
  - restart the transaction – only if no internal logical error happens in the transaction
  - kill the transaction – problems arising with the transaction, input data, no desirable data found in the database
- **Committed** (提交), after successful completion.

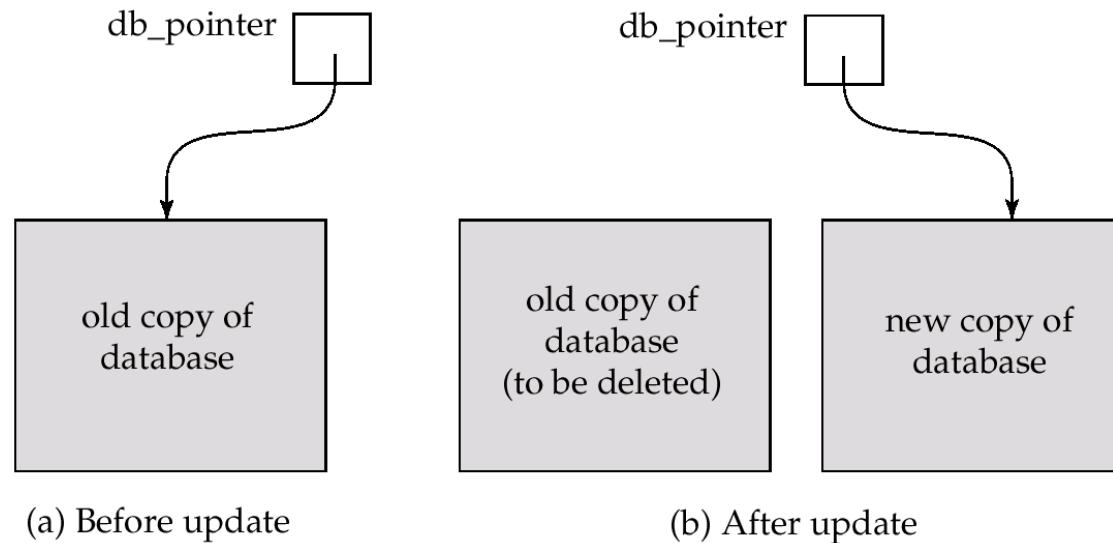
# Transaction State (Cont.)



# Implementation of Atomicity and Durability

- ❑ The recovery-management component of a database system implements the support for **atomicity** and **durability**.
- ❑ The **shadow-database** scheme:
  - assume that only one transaction is active at a time.
  - a pointer called **db\_pointer** always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database, and **db\_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - in case transaction fails, old consistent copy pointed to by **db\_pointer** can be used, and the shadow copy can be deleted.

# Implementation of Atomicity and Durability (Cont.)



- ❑ Assumes disks to not fail
- ❑ Useful for text editors, but extremely inefficient for large databases: executing a single transaction requires copying the entire database.

# Schedules

# Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - increased processor and disk utilization
  - reduced average response time
- *Concurrency control schemes* – mechanisms to achieve isolation.
  - i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the *consistency* of the database

# Schedules (调度计划)

- **Schedules** – sequences that indicate 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.

# Example Schedules

- Let  $T_1$  transfer \$50 from  $A$  to  $B$ , and  $T_2$  transfer 10% of the balance from  $A$  to  $B$ . The following is 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( $B$ )

# Example Schedule (Cont.)

- 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.

$$\begin{aligned} & A' = (A - 50) * 0.9 \\ & B' = B + 50 + (A - 50) * 0.1 \\ & A' + B' = A + B \end{aligned}$$

$T_1$	$T_2$
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( $B$ )

In both Schedule, the sum  $A + B$  is preserved.

# Example Schedules (Cont.)

- The following concurrent schedule does not preserve the value of the the sum  $A + B$ .

- $A' = A - 50$

- $B' = B + A * 0.1$

- $A' + B' = A + B - 50 + A * 0.1 \neq A + B$

$T_1$	$T_2$
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( $B$ )

# **Serializable Schedule(可串行化调度)**

# Serializability (可串行性)

- Basic Assumption – Each transaction preserves database consistency, thus serial execution of a set of transactions preserves database consistency.
- A schedule is serializable if it is equivalent to a serial schedule.
  1. conflict serializability (冲突可串行性)
  2. view serializability (视图可串行性)
- We ignore operations other than **read** and **write** instructions. Our simplified schedules consist of only **read** and **write** instructions.

# Conflict Serializability

- Given instructions  $I_i$  and  $I_j$  of transactions  $T_i$  and  $T_j$  respectively, **conflict** iff there exists some item  $Q$  accessed by both  $I_i$  and  $I_j$ , and at least one of these instructions wrote  $Q$ 
  1.  $I_i = \text{read}(Q)$ ,  $I_j = \text{read}(Q)$ .  $I_i$  and  $I_j$  don't conflict.
  2.  $I_i = \text{read}(Q)$ ,  $I_j = \text{write}(Q)$ . They conflict.
  3.  $I_i = \text{write}(Q)$ ,  $I_j = \text{read}(Q)$ . They conflict
  4.  $I_i = \text{write}(Q)$ ,  $I_j = \text{write}(Q)$ . They 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.

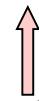
# Conflict Serializability (Cont.)

- 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
- Example of a schedule that is not conflict serializable:

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

# Conflict Serializability (Cont.)

- Schedule 1 can be transformed into Schedule 2, a serial schedule where  $T_2$  follows  $T_1$ , by series of swaps of non-conflicting instructions. Therefore Schedule 1 is conflict serializable.

$T_1$	$T_2$	$T_1$	$T_2$
read(A) write(A)  read(B) write(B)	read(A) write(A)  read(B) write(B)	read(A) write(A) <b>Read(B) Write(B)</b>	<b>Read(A) Write(A)</b> read(B) write(B)

*Schedule 1*      *Schedule 2*

# View Serializability

- $S$  and  $S'$  are **view equivalent** if the following three conditions are met:
  1. For each data item  $Q$ , if transaction  $T_i$  reads the initial value of  $Q$  in schedule  $S$ , then transaction  $T_i$  must, in schedule  $S'$ , also read the initial value of  $Q$ .
  2. For each data item  $Q$  if transaction  $T_i$  executes **read( $Q$ )** in schedule  $S$ , and that value was produced by transaction  $T_j$  (if any), then transaction  $T_i$  must in schedule  $S'$  also read the value of  $Q$  that was produced by transaction  $T_j$ .
  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  $S'$ .
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

# View Serializability (Cont.)

- A schedule  $S$  is **view serializable**, it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- A schedule which is view-serializable but **not** conflict serializable. Equivalent to  $T_3, T_4, T_6$

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

- Every view serializable schedule that is not conflict serializable has **blind writes** --- write without read

# Other Notions of Serializability

- Schedule given below produces same outcome as the serial schedule  $\langle T_1, T_5 \rangle$ , yet is not conflict equivalent or view equivalent to it.

$T_1$	$T_5$
read( $A$ ) $A := A - 50$ write( $A$ )	
	read( $B$ ) $B := B - 10$ write( $B$ )
read( $B$ ) $B := B + 50$ write( $B$ )	
	read( $A$ ) $A := A + 10$ write( $A$ )

- Determining such equivalence requires analysis of operations other than read and write.

# **Recoverable Schedule**

# Recoverability (可恢复性)

- **Recoverable schedule** (可恢复调度) — if a transaction  $T_j$  reads a data items previously written by a transaction  $T_i$ , 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( $B$ )	read( $A$ )

# Recoverability (Cont.)

- **Cascading rollback** (级联回滚) – 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( $A$ )	read( $A$ )

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

- Can lead to the undoing of a significant amount of work

# Recoverability (Cont.)

- **Cascadeless schedules** (无级联回滚调度) — 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

# Implementation of Isolation

- ❑ Schedules must be **conflict** or **view serializable**, and **recoverable**, for the sake of database consistency, and preferably **cascadeless**.
- ❑ A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency..
- ❑ Concurrency-control schemes **tradeoff** between the amount of concurrency they allow and the amount of overhead that they incur.
- ❑ Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

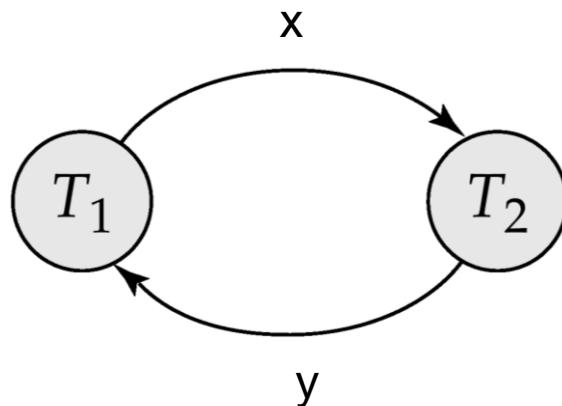
# Transaction Definition in SQL

- ❑ DML must include a construct for specifying the set of actions that comprise a transaction.
- ❑ In SQL, a 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.
- ❑ Levels of consistency specified by SQL-92:
  - **Serializable** — default
  - **Repeatable read**
  - **Read committed**
  - **Read uncommitted**

# Testing for Serializability

# Testing for Serializability

- Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$
- **Precedence graph** (优先图) — a direct graph where the vertices are the transactions (names).
- We draw an arc from  $T_i$  to  $T_j$  if the two transaction conflict, and  $T_i$  accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**



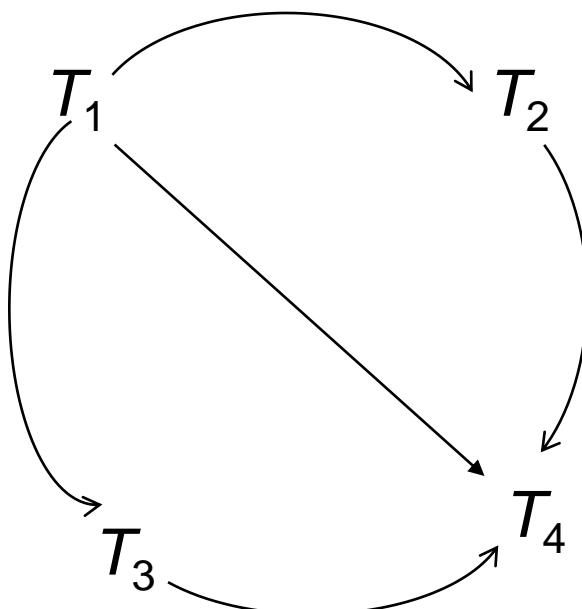
T1 write(x) before T2 read(x)  
T1 write(x) before T2 write(x)  
T1 read(x) before T2 write(x)

T2 write(y) before T1 read(y)  
T2 write(y) before T1 write(y)  
T2 read(y) before T1 write(y)

# Example Schedule (Schedule A)

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

# Precedence Graph for Schedule A



# Test for Conflict Serializability

- A schedule is conflict serializable iff its precedence graph is acyclic.
- 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 order for Schedule A would be
$$T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4 .$$

# Test for View Serializability

- The precedence graph test for conflict serializability must be modified to apply to a test for view serializability.
- The problem of checking if a schedule is view serializable falls in the **class of NP-complete problems**.
  - Thus existence of an efficient algorithm is unlikely.
  - However practical algorithms that just check some *sufficient conditions* for view serializability can still be used.

# Concurrency Control vs. Serializability Tests

- ❑ Testing a schedule for serializability after it has executed is a little too late!
- ❑ **Goal** – to develop concurrency control protocols that will assure serializability.
  - They will generally not examine the precedence graph as it is being created;
  - instead a protocol will impose a discipline that avoids nonserializable schedules.
- ❑ Tests for serializability help understand why a concurrency control protocol is correct.

# Assignments

- ❑ Practice Exercises: 15.6
- ❑ Exercises: 15.11