# Chapter 22

## Transaction Management

# Transaction Support

**Transaction**

**Action, or series of actions, carried out by user or application, which reads or updates contents of database.**

- **Logical unit of work on the database.**
- **Application program is series of transactions with non-database processing in between.**
- **Transforms database from one consistent state to another, although consistency may be violated during transaction.**

Pearson Education © 2009

# Example Transaction

```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, new_salary)
```

(a)

```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(b)



Pearson Education © 2009

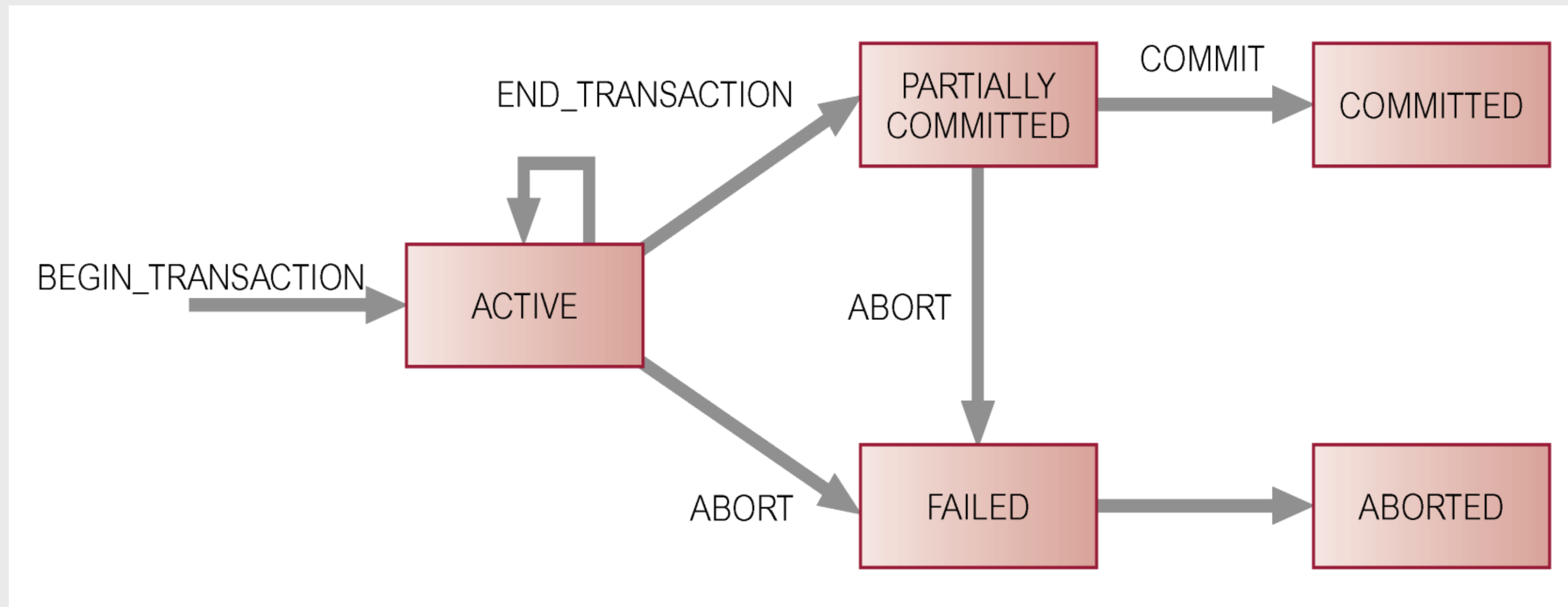
# Transaction Support

- **Can have one of two outcomes:**
  - **Success - transaction *commits* and database reaches a new consistent state.**
  - **Failure - transaction *aborts*, and database must be restored to consistent state before it started.**
  - **Such a transaction is *rolled back* or *undone*.**
- **Committed transaction cannot be aborted.**
- **Aborted transaction that is rolled back can be restarted later.**



Pearson Education © 2009

# State Transition Diagram for Transaction



# Properties of Transactions

- Four basic *(ACID)* properties of a transaction are:

**Atomicity** 'All or nothing' property.

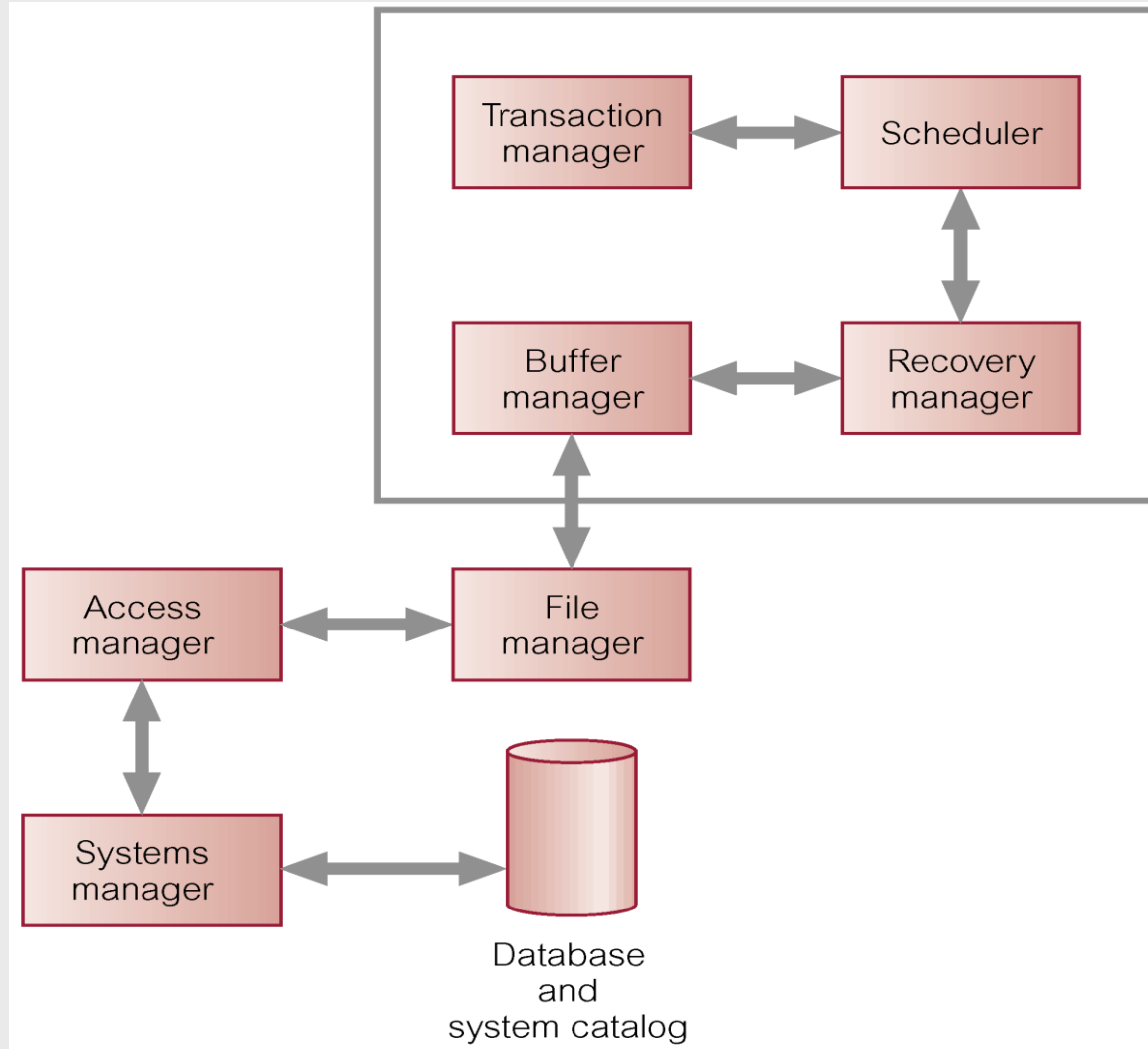
**Consistency** Must transform database from one consistent state to another.

**Isolation** Partial effects of incomplete transactions should not be visible to other transactions.

**Durability** Effects of a committed transaction are permanent and must not be lost because of later failure.

Pearson Education © 2009

# DBMS Transaction Subsystem






Pearson Education © 2009

# Concurrency Control

**Process of managing simultaneous operations on the database without having them interfere with one another.**

- **Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.**
- **Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.**

Pearson Education © 2009

# Need for Concurrency Control

- **Three examples of potential problems caused by concurrency:**
  - **Lost update problem.**
  - **Uncommitted dependency problem.**
  - **Inconsistent analysis problem.**


Pearson Education © 2009

# Lost Update Problem

- **Successfully completed update is overridden by another user.**
- **$T_1$ withdrawing £10 from an account with $bal_x$, initially £100.**
- **$T_2$ depositing £100 into same account.**
- **Serially, final balance would be £190.**

Pearson Education © 2009

# Lost Update Problem

| Time | $T_1$ | $T_2$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | begin_transaction | read($bal_x$) | 100 |
| $t_3$ | read($bal_x$) | $bal_x = bal_x + 100$ | 100 |
| $t_4$ | $bal_x = bal_x - 10$ | write($bal_x$) | 200 |
| $t_5$ | write($bal_x$) | commit | 90 |
| $t_6$ | commit | | 90 |

- **Loss of $T_2$'s update avoided by preventing $T_1$ from reading $bal_x$ until after update.**

Pearson Education © 2009

# Uncommitted Dependency Problem

- **Occurs when one transaction can see intermediate results of another transaction before it has committed.**
- **$T_4$ updates $bal_x$ to £200 but it aborts, so $bal_x$ should be back at original value of £100.**
- **$T_3$ has read new value of $bal_x$ (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.**


Pearson Education © 2009

# Uncommitted Dependency Problem

| Time | $T_3$ | $T_4$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | | read($bal_x$) | 100 |
| $t_3$ | | $bal_x = bal_x + 100$ | 100 |
| $t_4$ | begin_transaction | write($bal_x$) | 200 |
| $t_5$ | read($bal_x$) | ⋮ | 200 |
| $t_6$ | $bal_x = bal_x - 10$ | rollback | 100 |
| $t_7$ | write($bal_x$) | | 190 |
| $t_8$ | commit | | 190 |

- **Problem avoided by preventing $T_3$ from reading $bal_x$ until after $T_4$ commits or aborts.**



Pearson Education © 2009

## Inconsistent Analysis Problem

- **Occurs when transaction reads several values but second transaction updates some of them during execution of first.**
- **Sometimes referred to as *dirty read* or *unrepeatable read*.**
- **$T_6$ is totaling balances of accounts:**

  **x (£100), y (£50), and z (£25).**
- **Meantime, $T_5$ has transferred £10 from $bal_x$ to $bal_z$, so $T_6$ now has wrong result (£10 too high).**


Pearson Education © 2009

# Inconsistent Analysis Problem

| Time | $T_5$ | $T_6$ | $bal_x$ | $bal_y$ | $bal_z$ | sum |
|---|---|---|---|---|---|---|
| $t_1$ | | begin_transaction | 100 | 50 | 25 | |
| $t_2$ | begin_transaction | sum = 0 | 100 | 50 | 25 | 0 |
| $t_3$ | read($bal_x$) | read($bal_x$) | 100 | 50 | 25 | 0 |
| $t_4$ | $bal_x = bal_x - 10$ | $sum = sum + bal_x$ | 100 | 50 | 25 | 100 |
| $t_5$ | write($bal_x$) | read($bal_y$) | 90 | 50 | 25 | 100 |
| $t_6$ | read($bal_z$) | $sum = sum + bal_y$ | 90 | 50 | 25 | 150 |
| $t_7$ | $bal_z = bal_z + 10$ | | 90 | 50 | 25 | 150 |
| $t_8$ | write($bal_z$) | | 90 | 50 | 35 | 150 |
| $t_9$ | commit | read($bal_z$) | 90 | 50 | 35 | 150 |
| $t_{10}$ | | $sum = sum + bal_z$ | 90 | 50 | 35 | 185 |
| $t_{11}$ | | commit | 90 | 50 | 35 | 185 |

- **Problem avoided by preventing $T_6$ from reading $bal_x$ and $bal_z$ until after $T_5$ completed updates.**

Pearson Education © 2009

## Serializability

- **Objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference.**
- **Could run transactions serially, but this limits degree of concurrency or parallelism in system.**
- **Serializability identifies those executions of transactions guaranteed to ensure consistency.**


Pearson Education © 2009

# Serializability

**Schedule**

Sequence of reads/writes by set of concurrent transactions.

**Serial Schedule**

Schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions.

- No guarantee that results of all serial executions of a given set of transactions will be identical.


Pearson Education © 2009

## Nonserial Schedule

- **Schedule where operations from set of concurrent transactions are interleaved.**
- **Objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another.**
- **In other words, want to find nonserial schedules that are equivalent to *some* serial schedule. Such a schedule is called *serializable*.**

Pearson Education © 2009

# Serializability

- **In serializability, ordering of read/writes is important:**

**(a) If two transactions only read a data item, they do not conflict and order is not important.**

**(b) If two transactions either read or write completely separate data items, they do not conflict and order is not important.**

**(c) If one transaction writes a data item and another reads or writes same data item, order of execution is important.**

Pearson Education © 2009

# Example of Conflict Serializability

| Time | $T_7$ | $T_8$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | read($bal_x$) | |
| $t_3$ | write($bal_x$) | |
| $t_4$ | | begin_transaction |
| $t_5$ | | read($bal_x$) |
| $t_6$ | | write($bal_x$) |
| $t_7$ | read($bal_y$) | |
| $t_8$ | write($bal_y$) | |
| $t_9$ | commit | |
| $t_{10}$ | | read($bal_y$) |
| $t_{11}$ | | write($bal_y$) |
| $t_{12}$ | | commit |


Pearson Education © 2009

# Example of Conflict Serializability

| $T_7$ | $T_8$ |
| --- | --- |
| begin_transaction | |
| read(**$bal_x$**) | |
| write(**$bal_x$**) | |
| | begin_transaction |
| | read(**$bal_x$**) |
| read(**$bal_y$**) | |
| | write(**$bal_x$**) |
| write(**$bal_y$**) | |
| commit | |
| | read(**$bal_y$**) |
| | write(**$bal_y$**) |
| | commit |

| $T_7$ | $T_8$ |
| --- | --- |
| begin_transaction | |
| read(**$bal_x$**) | |
| write(**$bal_x$**) | |
| read(**$bal_y$**) | |
| write(**$bal_y$**) | |
| commit | |
| | begin_transaction |
| | read(**$bal_x$**) |
| | write(**$bal_x$**) |
| | read(**$bal_y$**) |
| | write(**$bal_y$**) |
| | commit |

# Serializability

- **Conflict serializable schedule orders any conflicting operations in same way as some serial execution.**
- **Under *constrained write rule* where:**
  - **A transaction updates data item based on its old value, which is first read.**
- **Use *precedence graph* to test for serializability.**


Pearson Education © 2009

## Precedence Graph

- **Create:**
  - **node for each transaction;**
  - **a directed edge $T_i \rightarrow T_j$, if $T_j$ reads the value of an item written by $T_i$;**
  - **a directed edge $T_i \rightarrow T_j$, if $T_j$ writes a value into an item after it has been read by $T_i$.**
  - **a directed edge Ti → Tj, if Tj writes a value into an item after it has been written by Ti.**
- **If precedence graph contains a cycle, schedule is not conflict serializable.**



Pearson Education © 2009

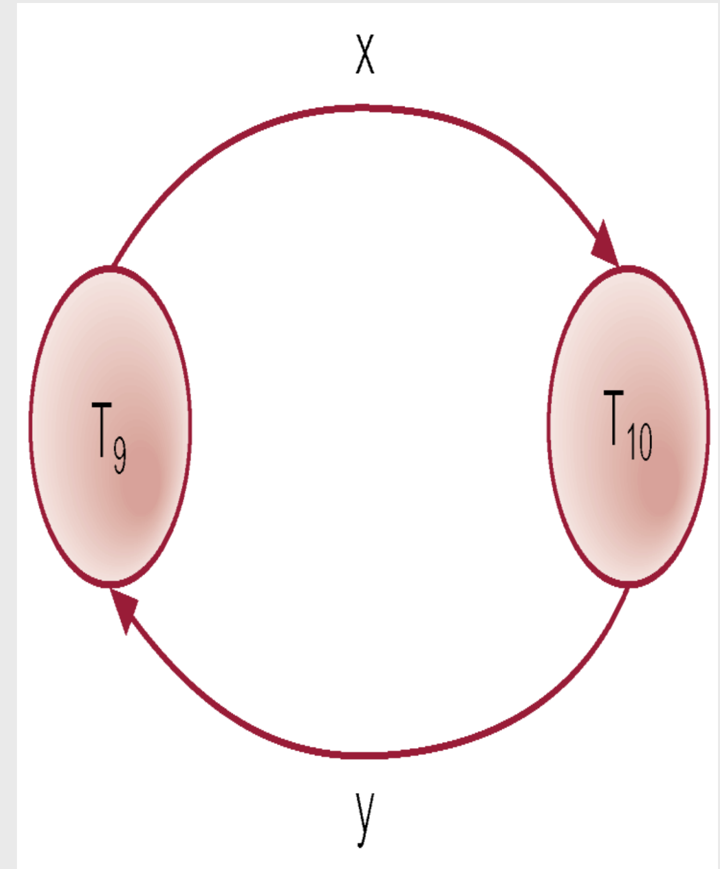
# Example - Non-conflict serializable schedule

- $T_9$ is transferring £100 from one account with balance $bal_x$ to another account with balance $bal_y$.
- $T_{10}$ is increasing balance of these two accounts by 10%.
- Precedence graph has a cycle and so is not serializable.

Pearson Education © 2009

# Example - Non-conflict serializable schedule

| Time | $T_9$ | $T_{10}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | read($bal_x$) | |
| $t_3$ | $bal_x = bal_x + 100$ | |
| $t_4$ | write($bal_x$) | begin_transaction |
| $t_5$ | | read($bal_x$) |
| $t_6$ | | $bal_x = bal_x * 1.1$ |
| $t_7$ | | write($bal_x$) |
| $t_8$ | | read($bal_y$) |
| $t_9$ | | $bal_y = bal_y * 1.1$ |
| $t_{10}$ | | write($bal_y$) |
| $t_{11}$ | read($bal_y$) | commit |
| $t_{12}$ | $bal_y = bal_y - 100$ | |
| $t_{13}$ | write($bal_y$) | |
| $t_{14}$ | commit | |




Pearson Education © 2009

# Recoverability

- Serializability identifies schedules that maintain database consistency, assuming no transaction fails.
- Could also examine recoverability of transactions within schedule.
- If transaction fails, atomicity requires effects of transaction to be undone.
- Durability states that once transaction commits, its changes cannot be undone (without running another, compensating, transaction).



Pearson Education © 2009

# Recoverable Schedule

**A schedule where, for each pair of transactions $T_i$ and $T_j$, if $T_j$ reads a data item previously written by $T_i$, then the commit operation of $T_i$ precedes the commit operation of $T_j$.**


Pearson Education © 2009

## Concurrency Control Techniques

- **Two basic concurrency control techniques:**
  - **Locking,**
  - **Timestamping.**
- **Both are conservative approaches: delay transactions in case they conflict with other transactions.**
- **Optimistic methods assume conflict is rare and only check for conflicts at commit.**


Pearson Education © 2009

## Locking

**Transaction uses locks to deny access to other transactions and so prevent incorrect updates.**

- **Most widely used approach to ensure serializability.**
- **Generally, a transaction must claim a *shared* (*read*) or *exclusive* (*write*) lock on a data item before read or write.**
- **Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.**

Pearson Education © 2009

# Locking - Basic Rules

- If transaction has shared lock on item, can read but not update item.
- If transaction has exclusive lock on item, can both read and update item.
- Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.
- Exclusive lock gives transaction exclusive access to that item.


Pearson Education © 2009

# Locking - Basic Rules

- **Some systems allow transaction to upgrade read lock to an exclusive lock, or downgrade exclusive lock to a shared lock.**



Pearson Education © 2009

# Example - Incorrect Locking Schedule

| Time | $T_9$ | $T_{10}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | read($bal_x$) | |
| $t_3$ | $bal_x = bal_x + 100$ | |
| $t_4$ | write($bal_x$) | begin_transaction |
| $t_5$ | | read($bal_x$) |
| $t_6$ | | $bal_x = bal_x * 1.1$ |
| $t_7$ | | write($bal_x$) |
| $t_8$ | | read($bal_y$) |
| $t_9$ | | $bal_y = bal_y * 1.1$ |
| $t_{10}$ | | write($bal_y$) |
| $t_{11}$ | read($bal_y$) | commit |
| $t_{12}$ | $bal_y = bal_y - 100$ | |
| $t_{13}$ | write($bal_y$) | |
| $t_{14}$ | commit | |

- **For these two transactions, a valid schedule using these rules is: S = {**

**T09- write_lock($bal_x$), read($bal_x$), write($bal_x$), unlock($bal_x$).**

**T10- write_lock($bal_x$), read($bal_x$), write($bal_x$), unlock($bal_x$).**

**T10- write_lock($bal_y$), read($bal_y$), write( $bal_y$), unlock($bal_y$), commit($T_{10}$)**

**T09- write_lock($bal_y$), read($bal_y$), write($bal_y$), unlock($bal_y$), commit($T_9$)}**

Pearson Education © 2009

# Example - Incorrect Locking Schedule

- **If at start, $bal_x = 100$, $bal_y = 400$, result should be:**
  - **$bal_x = 220$, $bal_y = 330$, if $T_9$ executes before $T_{10}$, or**
  - **$bal_x = 210$, $bal_y = 340$, if $T_{10}$ executes before $T_9$.**
- **However, result gives $bal_x = 220$ and $bal_y = 340$.**
- **S is not a serializable schedule.**



Pearson Education © 2009

# Example - Incorrect Locking Schedule

- Problem is that transactions release locks too soon, resulting in loss of total isolation and atomicity.
- To guarantee serializability, need an additional protocol concerning the positioning of lock and unlock operations in every transaction.



Pearson Education © 2009

## Two-Phase Locking (2PL)

Transaction follows 2PL protocol if all locking operations precede first unlock operation in the transaction.

- Two phases for transaction:
  - Growing phase - acquires all locks but cannot release any locks.
  - Shrinking phase - releases locks but cannot acquire any new locks.

Pearson Education © 2009

# Preventing Lost Update Problem using 2PL

| Time | $T_1$ | $T_2$ | $bal_x$ |
| --- | --- | --- | --- |
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | begin_transaction | write_lock($bal_x$) | 100 |
| $t_3$ | write_lock($bal_x$) | read($bal_x$) | 100 |
| $t_4$ | WAIT | $bal_x = bal_x + 100$ | 100 |
| $t_5$ | WAIT | write($bal_x$) | 200 |
| $t_6$ | WAIT | commit/unlock($bal_x$) | 200 |
| $t_7$ | read($bal_x$) | | 200 |
| $t_8$ | $bal_x = bal_x - 10$ | | 200 |
| $t_9$ | write($bal_x$) | | 190 |
| $t_{10}$ | commit/unlock($bal_x$) | | 190 |

Pearson Education © 2009

# Preventing Uncommitted Dependency Problem using 2PL

| Time | $T_3$ | $T_4$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | | write_lock($bal_x$) | 100 |
| $t_3$ | | read($bal_x$) | 100 |
| $t_4$ | begin_transaction | $bal_x = bal_x + 100$ | 100 |
| $t_5$ | write_lock($bal_x$) | write($bal_x$) | 200 |
| $t_6$ | WAIT | rollback/unlock($bal_x$) | 100 |
| $t_7$ | read($bal_x$) | | 100 |
| $t_8$ | $bal_x = bal_x - 10$ | | 100 |
| $t_9$ | write($bal_x$) | | 90 |
| $t_{10}$ | commit/unlock($bal_x$) | | 90 |

Pearson Education © 2009

# Preventing Inconsistent Analysis Problem using 2PL

| Time | $T_5$ | $T_6$ | $bal_x$ | $bal_y$ | $bal_z$ | sum |
|---|---|---|---|---|---|---|
| $t_1$ | | begin_transaction | 100 | 50 | 25 | |
| $t_2$ | begin_transaction | sum = 0 | 100 | 50 | 25 | 0 |
| $t_3$ | write_lock($bal_x$) | | 100 | 50 | 25 | 0 |
| $t_4$ | read($bal_x$) | read_lock($bal_x$) | 100 | 50 | 25 | 0 |
| $t_5$ | $bal_x = bal_x - 10$ | WAIT | 100 | 50 | 25 | 0 |
| $t_6$ | write($bal_x$) | WAIT | 90 | 50 | 25 | 0 |
| $t_7$ | write_lock($bal_z$) | WAIT | 90 | 50 | 25 | 0 |
| $t_8$ | read($bal_z$) | WAIT | 90 | 50 | 25 | 0 |
| $t_9$ | $bal_z = bal_z + 10$ | WAIT | 90 | 50 | 25 | 0 |
| $t_{10}$ | write($bal_z$) | WAIT | 90 | 50 | 35 | 0 |
| $t_{11}$ | commit/unlock($bal_x$, $bal_z$) | WAIT | 90 | 50 | 35 | 0 |
| $t_{12}$ | | read($bal_x$) | 90 | 50 | 35 | 0 |
| $t_{13}$ | | sum = sum + $bal_x$ | 90 | 50 | 35 | 90 |
| $t_{14}$ | | read_lock($bal_y$) | 90 | 50 | 35 | 90 |
| $t_{15}$ | | read($bal_y$) | 90 | 50 | 35 | 90 |
| $t_{16}$ | | sum = sum + $bal_y$ | 90 | 50 | 35 | 140 |
| $t_{17}$ | | read_lock($bal_z$) | 90 | 50 | 35 | 140 |
| $t_{18}$ | | read($bal_z$) | 90 | 50 | 35 | 140 |
| $t_{19}$ | | sum = sum + $bal_z$ | 90 | 50 | 35 | 175 |
| $t_{20}$ | | commit/unlock($bal_x$, $bal_y$, $bal_z$) | 90 | 50 | 35 | 175 |



Pearson Education © 2009

# Cascading Rollback

- **If *every* transaction in a schedule follows 2PL, schedule is serializable.**
- **However, problems can occur with interpretation of when locks can be released.**


Pearson Education © 2009

# Cascading Rollback

| Time | $T_{14}$ | $T_{15}$ | $T_{16}$ |
|---|---|---|---|
| $t_1$ | begin_transaction | | |
| $t_2$ | write_lock($bal_x$) | | |
| $t_3$ | read($bal_x$) | | |
| $t_4$ | read_lock($bal_y$) | | |
| $t_5$ | read($bal_y$) | | |
| $t_6$ | $bal_x = bal_y + bal_x$ | | |
| $t_7$ | write($bal_x$) | | |
| $t_8$ | unlock($bal_x$) | begin_transaction | |
| $t_9$ | ⋮ | write_lock($bal_x$) | |
| $t_{10}$ | ⋮ | read($bal_x$) | |
| $t_{11}$ | ⋮ | $bal_x = bal_x + 100$ | |
| $t_{12}$ | ⋮ | write($bal_x$) | |
| $t_{13}$ | ⋮ | unlock($bal_x$) | |
| $t_{14}$ | ⋮ | ⋮ | |
| $t_{15}$ | rollback | ⋮ | |
| $t_{16}$ | | ⋮ | begin_transaction |
| $t_{17}$ | | ⋮ | read_lock($bal_x$) |
| $t_{18}$ | | rollback | ⋮ |
| $t_{19}$ | | | rollback |



Pearson Education © 2009

# Cascading Rollback

- **Transactions conform to 2PL.**
- **$T_{14}$ aborts.**
- **Since $T_{15}$ is dependent on $T_{14}$, $T_{15}$ must also be rolled back. Since $T_{16}$ is dependent on $T_{15}$, it too must be rolled back.**
- **This is called *cascading rollback*.**
- **To prevent this with 2PL, leave release of *all* locks until end of transaction.**



Pearson Education © 2009

# Deadlock

**An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.**

| Time | $T_{17}$ | $T_{18}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | write_lock($bal_x$) | begin_transaction |
| $t_3$ | read($bal_x$) | write_lock($bal_y$) |
| $t_4$ | $bal_x = bal_x - 10$ | read($bal_y$) |
| $t_5$ | write($bal_x$) | $bal_y = bal_y + 100$ |
| $t_6$ | write_lock($bal_y$) | write($bal_y$) |
| $t_7$ | WAIT | write_lock($bal_x$) |
| $t_8$ | WAIT | WAIT |
| $t_9$ | WAIT | WAIT |
| $t_{10}$ | ⋮ | WAIT |
| $t_{11}$ | ⋮ | ⋮ |



Pearson Education © 2009

# Deadlock

- Only one way to break deadlock: abort one or more of the transactions.
- Deadlock should be transparent to user, so DBMS should restart transaction(s).
- Three general techniques for handling deadlock:
  - Timeouts.
  - Deadlock prevention.
  - Deadlock detection and recovery.



Pearson Education © 2009