

# Chapter 22

---

## **Transaction Management**

# Chapter 22 - Objectives

- **Function and importance of transactions.**
- **Properties of transactions.**
- **Concurrency Control**
  - **Meaning of serializability.**
  - **How locking can ensure serializability.**
  - **Deadlock and how it can be resolved.**
  - **How timestamping can ensure serializability.**
  - **Optimistic concurrency control.**
  - **Granularity of locking.**

# Chapter 22 - Objectives

- **Recovery Control**
  - Some causes of database failure.
  - Purpose of transaction log file.
  - Purpose of checkpointing.
  - How to recover following database failure.
- **Alternative models for long duration transactions.**

# 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.**

# Example Transaction

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

(a)

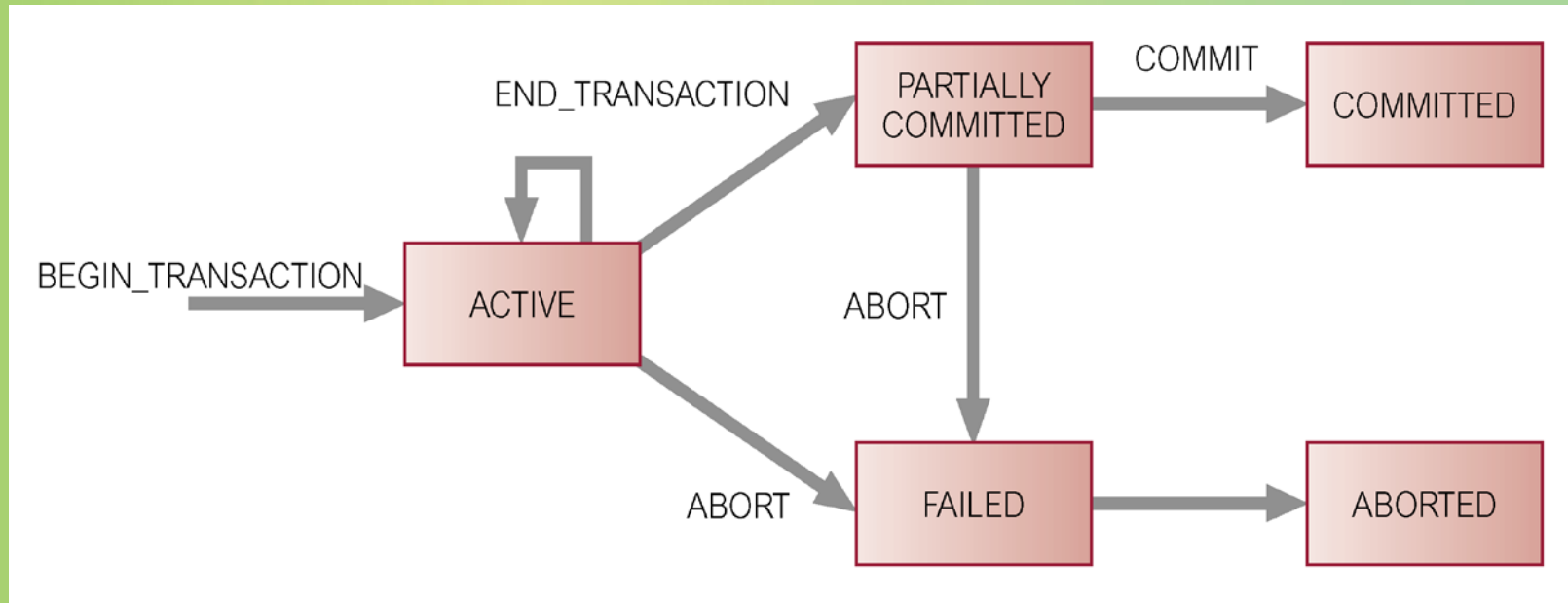
```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
        begin
            staffNo = newStaffNo
            write(propertyNo = pno, staffNo)
        end
    end
end
```

(b)

# 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.

# State Transition Diagram for Transaction



# Properties of Transactions

• Four basic (*ACID*) properties that define a transaction are:

Atomicity 'All or nothing' property.

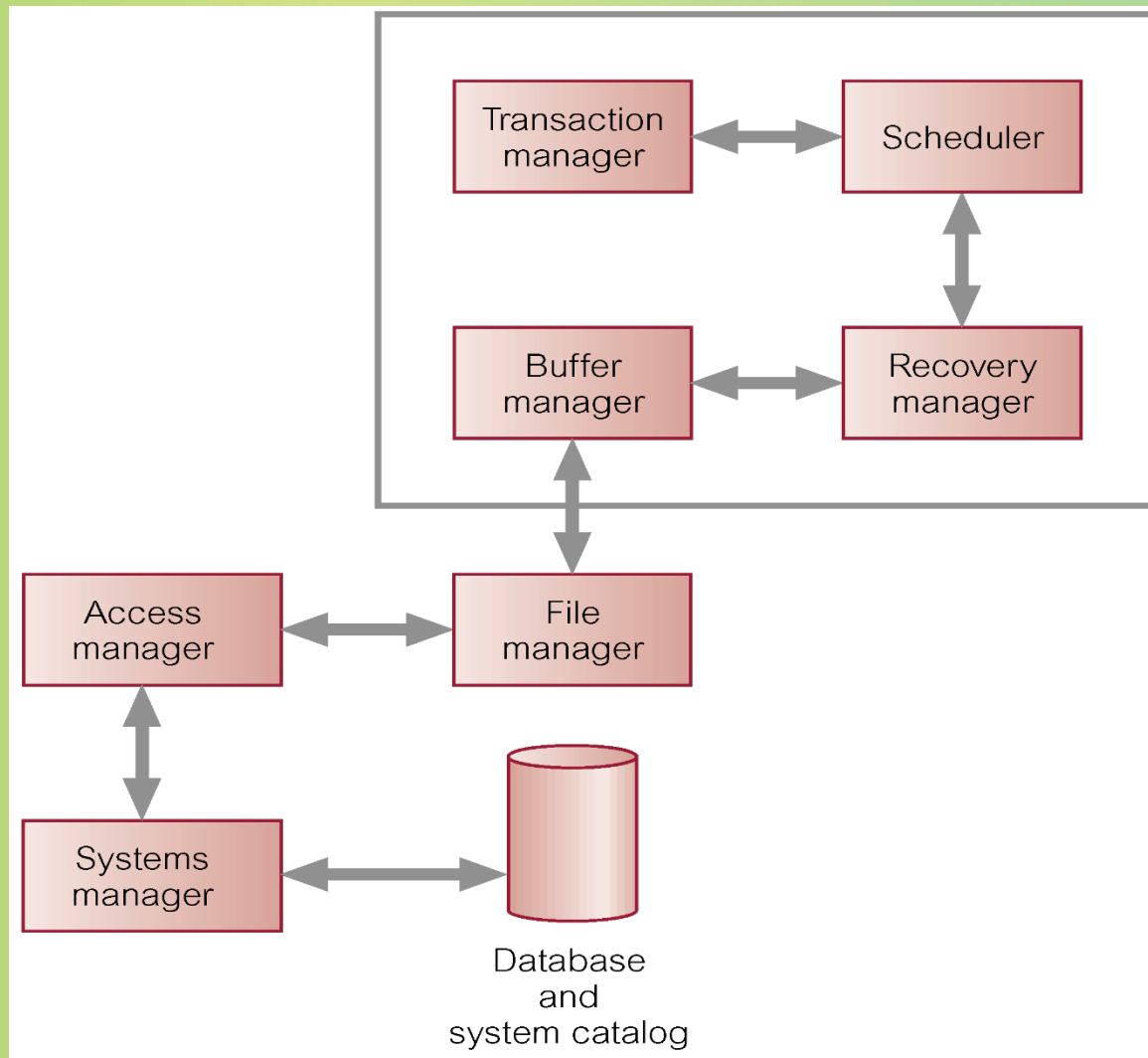
Consistency Must transform database from one consistent state to another.

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.



# DBMS Transaction Subsystem



# 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.**

# Need for Concurrency Control

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

# 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.

# 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.

# Uncommitted Dependency Problem

- Occurs when one transaction can see intermediate results of another transaction before it has committed.
- Referred to as a dirty read
- $T_4$  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.

# Uncommitted Dependency Problem

Time	T <sub>3</sub>	T <sub>4</sub>	bal <sub>x</sub>
t <sub>1</sub>		begin_transaction	100
t <sub>2</sub>		read(bal <sub>x</sub> )	100
t <sub>3</sub>		bal <sub>x</sub> = bal <sub>x</sub> + 100	100
t <sub>4</sub>	begin_transaction	write(bal <sub>x</sub> )	200
t <sub>5</sub>	read(bal <sub>x</sub> )	:	200
t <sub>6</sub>	bal <sub>x</sub> = bal <sub>x</sub> - 10	rollback	100
t <sub>7</sub>	write(bal <sub>x</sub> )		190
t <sub>8</sub>	commit		190

- Problem avoided by preventing T<sub>3</sub> from reading bal<sub>x</sub> until after T<sub>4</sub> commits or aborts.

# Inconsistent Analysis Problem

- Occurs when transaction reads several values but second transaction updates some of them during execution of first.
- $T_6$  is totaling balances of account x (£100), account y (£50), and account z (£25).
- Meantime,  $T_5$  has transferred £10 from  $bal_x$  to  $bal_z$ , so  $T_6$  now has wrong result (£10 too high).



# Inconsistent Analysis Problem

Time	T <sub>5</sub>	T <sub>6</sub>	bal <sub>x</sub>	bal <sub>y</sub>	bal <sub>z</sub>	sum
t <sub>1</sub>		begin_transaction	100	50	25	
t <sub>2</sub>	begin_transaction	sum = 0	100	50	25	0
t <sub>3</sub>	read(bal <sub>x</sub> )	read(bal <sub>x</sub> )	100	50	25	0
t <sub>4</sub>	bal <sub>x</sub> = bal <sub>x</sub> - 10	sum = sum + bal <sub>x</sub>	100	50	25	100
t <sub>5</sub>	write(bal <sub>x</sub> )	read(bal <sub>y</sub> )	90	50	25	100
t <sub>6</sub>	read(bal <sub>z</sub> )	sum = sum + bal <sub>y</sub>	90	50	25	150
t <sub>7</sub>	bal <sub>z</sub> = bal <sub>z</sub> + 10		90	50	25	150
t <sub>8</sub>	write(bal <sub>z</sub> )		90	50	35	150
t <sub>9</sub>	commit	read(bal <sub>z</sub> )	90	50	35	150
t <sub>10</sub>		sum = sum + bal <sub>z</sub>	90	50	35	185
t <sub>11</sub>		commit	90	50	35	185

- Problem avoided by preventing T<sub>6</sub> from reading bal<sub>x</sub> and bal<sub>z</sub> until after T<sub>5</sub> completed updates.

# 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.

# 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.**

# 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*.

# 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 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.

# Example of Conflict Serializability

Schedule 1

Schedule 2

Schedule 3

Time	T <sub>7</sub>	T <sub>8</sub>	T <sub>7</sub>	T <sub>8</sub>	T <sub>7</sub>	T <sub>8</sub>
t <sub>1</sub>	begin_transaction		begin_transaction		begin_transaction	
t <sub>2</sub>	read( <b>bal<sub>x</sub></b> )		read( <b>bal<sub>x</sub></b> )		read( <b>bal<sub>x</sub></b> )	
t <sub>3</sub>	write( <b>bal<sub>x</sub></b> )		write( <b>bal<sub>x</sub></b> )		write( <b>bal<sub>x</sub></b> )	
t <sub>4</sub>		begin_transaction		begin_transaction	read( <b>bal<sub>y</sub></b> )	
t <sub>5</sub>		read( <b>bal<sub>x</sub></b> )		read( <b>bal<sub>x</sub></b> )	write( <b>bal<sub>y</sub></b> )	
t <sub>6</sub>		write( <b>bal<sub>x</sub></b> )	read( <b>bal<sub>y</sub></b> )		commit	
t <sub>7</sub>	read( <b>bal<sub>y</sub></b> )			write( <b>bal<sub>x</sub></b> )		begin_transaction
t <sub>8</sub>	write( <b>bal<sub>y</sub></b> )		write( <b>bal<sub>y</sub></b> )			read( <b>bal<sub>x</sub></b> )
t <sub>9</sub>	commit		commit			write( <b>bal<sub>x</sub></b> )
t <sub>10</sub>		read( <b>bal<sub>y</sub></b> )		read( <b>bal<sub>y</sub></b> )		read( <b>bal<sub>y</sub></b> )
t <sub>11</sub>		write( <b>bal<sub>y</sub></b> )		write( <b>bal<sub>y</sub></b> )		write( <b>bal<sub>y</sub></b> )
t <sub>12</sub>		commit		commit		commit
	(a)		(b)		(c)	

# Serializability

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

# 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  $T_i$ .
- **If precedence graph contains cycle schedule is not conflict serializable.**

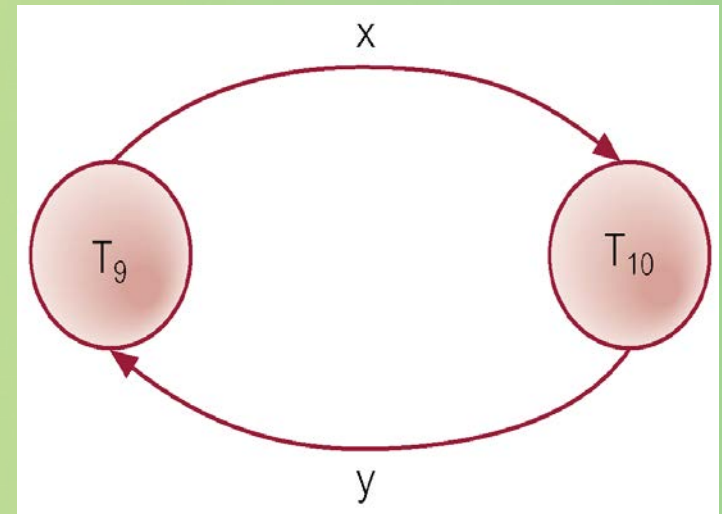


# Example - Non-conflict serializable schedule

- $T_9$  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.

# 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	



# 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).

# 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  $T_j$ .**

# 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.**

# 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.

# 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.

# Locking - Basic Rules

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



# Example - Incorrect Locking Schedule

- For two transactions above, a valid schedule using these rules is:

$S = \{ \text{write\_lock}(T_9, \text{bal}_x), \text{read}(T_9, \text{bal}_x), \text{write}(T_9, \text{bal}_x), \text{unlock}(T_9, \text{bal}_x), \text{write\_lock}(T_{10}, \text{bal}_x), \text{read}(T_{10}, \text{bal}_x), \text{write}(T_{10}, \text{bal}_x), \text{unlock}(T_{10}, \text{bal}_x), \text{write\_lock}(T_{10}, \text{bal}_y), \text{read}(T_{10}, \text{bal}_y), \text{write}(T_{10}, \text{bal}_y), \text{unlock}(T_{10}, \text{bal}_y), \text{commit}(T_{10}), \text{write\_lock}(T_9, \text{bal}_y), \text{read}(T_9, \text{bal}_y), \text{write}(T_9, \text{bal}_y), \text{unlock}(T_9, \text{bal}_y), \text{commit}(T_9) \}$

# 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  $bal_x = 220$  &  $bal_y = 340$ .
- S is not a serializable schedule.

# 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.

# 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.**

# Preventing Lost Update Problem using 2PL

Time	T <sub>1</sub>	T <sub>2</sub>	bal <sub>x</sub>
t <sub>1</sub>		begin_transaction	100
t <sub>2</sub>	begin_transaction	write_lock(bal <sub>x</sub> )	100
t <sub>3</sub>	write_lock(bal <sub>x</sub> )	read(bal <sub>x</sub> )	100
t <sub>4</sub>	WAIT	bal <sub>x</sub> = bal <sub>x</sub> + 100	100
t <sub>5</sub>	WAIT	write(bal <sub>x</sub> )	200
t <sub>6</sub>	WAIT	commit/unlock(bal <sub>x</sub> )	200
t <sub>7</sub>	read(bal <sub>x</sub> )		200
t <sub>8</sub>	bal <sub>x</sub> = bal <sub>x</sub> - 10		200
t <sub>9</sub>	write(bal <sub>x</sub> )		190
t <sub>10</sub>	commit/unlock(bal <sub>x</sub> )		190

# 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

# Preventing Inconsistent Analysis Problem using 2PL

Time	T <sub>5</sub>	T <sub>6</sub>	bal <sub>x</sub>	bal <sub>y</sub>	bal <sub>z</sub>	sum
t <sub>1</sub>		begin_transaction	100	50	25	
t <sub>2</sub>	begin_transaction	sum = 0	100	50	25	0
t <sub>3</sub>	write_lock( <b>bal<sub>x</sub></b> )		100	50	25	0
t <sub>4</sub>	read( <b>bal<sub>x</sub></b> )	read_lock( <b>bal<sub>x</sub></b> )	100	50	25	0
t <sub>5</sub>	<b>bal<sub>x</sub> = bal<sub>x</sub> - 10</b>	WAIT	100	50	25	0
t <sub>6</sub>	write( <b>bal<sub>x</sub></b> )	WAIT	90	50	25	0
t <sub>7</sub>	write_lock( <b>bal<sub>z</sub></b> )	WAIT	90	50	25	0
t <sub>8</sub>	read( <b>bal<sub>z</sub></b> )	WAIT	90	50	25	0
t <sub>9</sub>	<b>bal<sub>z</sub> = bal<sub>z</sub> + 10</b>	WAIT	90	50	25	0
t <sub>10</sub>	write( <b>bal<sub>z</sub></b> )	WAIT	90	50	35	0
t <sub>11</sub>	commit/unlock( <b>bal<sub>x</sub>, bal<sub>z</sub></b> )	WAIT	90	50	35	0
t <sub>12</sub>		read( <b>bal<sub>x</sub></b> )	90	50	35	0
t <sub>13</sub>		sum = sum + <b>bal<sub>x</sub></b>	90	50	35	90
t <sub>14</sub>		read_lock( <b>bal<sub>y</sub></b> )	90	50	35	90
t <sub>15</sub>		read( <b>bal<sub>y</sub></b> )	90	50	35	90
t <sub>16</sub>		sum = sum + <b>bal<sub>y</sub></b>	90	50	35	140
t <sub>17</sub>		read_lock( <b>bal<sub>z</sub></b> )	90	50	35	140
t <sub>18</sub>		read( <b>bal<sub>z</sub></b> )	90	50	35	140
t <sub>19</sub>		sum = sum + <b>bal<sub>z</sub></b>	90	50	35	175
t <sub>20</sub>		commit/unlock( <b>bal<sub>x</sub>, bal<sub>y</sub>, bal<sub>z</sub></b> )	90	50	35	175

# 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.



# Cascading Rollback

Time	T <sub>14</sub>	T <sub>15</sub>	T <sub>16</sub>
t <sub>1</sub>	begin_transaction		
t <sub>2</sub>	write_lock( <b>bal<sub>x</sub></b> )		
t <sub>3</sub>	read( <b>bal<sub>x</sub></b> )		
t <sub>4</sub>	read_lock( <b>bal<sub>y</sub></b> )		
t <sub>5</sub>	read( <b>bal<sub>y</sub></b> )		
t <sub>6</sub>	<b>bal<sub>x</sub> = bal<sub>y</sub> + bal<sub>x</sub></b>		
t <sub>7</sub>	write( <b>bal<sub>x</sub></b> )		
t <sub>8</sub>	unlock( <b>bal<sub>x</sub></b> )	begin_transaction	
t <sub>9</sub>	⋮	write_lock( <b>bal<sub>x</sub></b> )	
t <sub>10</sub>	⋮	read( <b>bal<sub>x</sub></b> )	
t <sub>11</sub>	⋮	<b>bal<sub>x</sub> = bal<sub>x</sub> + 100</b>	
t <sub>12</sub>	⋮	write( <b>bal<sub>x</sub></b> )	
t <sub>13</sub>	⋮	unlock( <b>bal<sub>x</sub></b> )	
t <sub>14</sub>	⋮	⋮	
t <sub>15</sub>	rollback	⋮	
t <sub>16</sub>		⋮	begin_transaction
t <sub>17</sub>		⋮	read_lock( <b>bal<sub>x</sub></b> )
t <sub>18</sub>		rollback	⋮
t <sub>19</sub>			rollback

# 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.

# 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 <sub>17</sub>	T <sub>18</sub>
t <sub>1</sub>	begin_transaction	
t <sub>2</sub>	write_lock( <b>bal<sub>x</sub></b> )	begin_transaction
t <sub>3</sub>	read( <b>bal<sub>x</sub></b> )	write_lock( <b>bal<sub>y</sub></b> )
t <sub>4</sub>	<b>bal<sub>x</sub></b> = <b>bal<sub>x</sub></b> - 10	read( <b>bal<sub>y</sub></b> )
t <sub>5</sub>	write( <b>bal<sub>x</sub></b> )	<b>bal<sub>y</sub></b> = <b>bal<sub>y</sub></b> + 100
t <sub>6</sub>	write_lock( <b>bal<sub>y</sub></b> )	write( <b>bal<sub>y</sub></b> )
t <sub>7</sub>	WAIT	write_lock( <b>bal<sub>x</sub></b> )
t <sub>8</sub>	WAIT	WAIT
t <sub>9</sub>	WAIT	WAIT
t <sub>10</sub>	:	WAIT
t <sub>11</sub>	:	:

# 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).
- However, in practice DBMS cannot restart aborted transaction since it is unaware of transaction logic even if it was aware of the transaction history (unless there is no user input in the transaction or the input is not a function of the database state).

# Deadlock

- **Three general techniques for handling deadlock:**
  - **Timeouts.**
  - **Deadlock prevention.**
  - **Deadlock detection and recovery.**

# Timeouts

- Transaction that requests lock will only wait for a system-defined period of time.
- If lock has not been granted within this period, lock request times out.
- In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.

# Deadlock Prevention

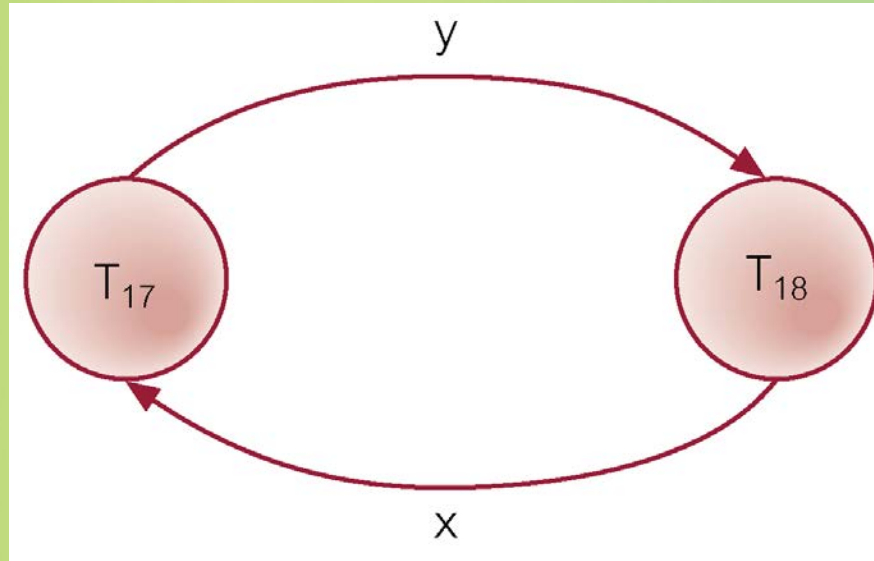
- DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.
- Could order transactions using transaction timestamps:
  - Wait-Die - only an older transaction can wait for younger one, otherwise transaction is aborted (*dies*) and restarted with same timestamp.
  - Wound-Wait - only a younger transaction can wait for an older one. If older transaction requests lock held by younger one, younger one is aborted (*wounded*).

# Deadlock Detection and Recovery

- DBMS allows deadlock to occur but recognizes it and breaks it.
- Usually handled by construction of wait-for graph (WFG) showing transaction dependencies:
  - Create a node for each transaction.
  - Create edge  $T_i \rightarrow T_j$ , if  $T_i$  waiting to lock item locked by  $T_j$ .
- Deadlock exists if and only if WFG contains cycle.
- WFG is created at regular intervals.



# Example - Wait-For-Graph (WFG)



# Recovery from Deadlock Detection

- **Several issues:**
  - **choice of deadlock victim;**
  - **how far to roll a transaction back;**
  - **avoiding starvation.**

# Timestamping

- Transactions ordered globally so that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.
- Conflict is resolved by rolling back and restarting transaction.
- No locks so no deadlock.

# Timestamping

## Timestamp

A unique identifier created by DBMS that indicates relative starting time of a transaction.

- Can be generated by using system clock at time transaction started, or by incrementing a logical counter every time a new transaction starts.

# Timestamping

- Read/write proceeds only if *last update on that data item* was carried out by an older transaction.
- Otherwise, transaction requesting read/write is restarted and given a new timestamp.
- Also timestamps for data items:
  - read-timestamp - timestamp of last transaction to read item;
  - write-timestamp - timestamp of last transaction to write item.

# Timestamping - Read(x)

$ts(T) < write\_timestamp(x)$

- x already written by younger transaction.
- Abort and Restart the transaction.
- Otherwise, operation is accepted and executed. Update the  $read\_timestamp = \max((ts(T), read\_timestamp(x)))$

# Timestamping – Transaction issues Write(x)

- Consider a transaction T with timestamp  $ts(T)$ :

$ts(T) < write\_timestamp(x)$

- x already updated by younger (later) transaction.
- Transaction must be aborted and restarted with a new timestamp.

$ts(T) < read\_timestamp(x)$

- x already read by younger transaction.
- Roll back transaction and restart it using a later timestamp.
- Otherwise proceed and set  $write\_stamp = ts(T)$

# Example – Basic Timestamp Ordering

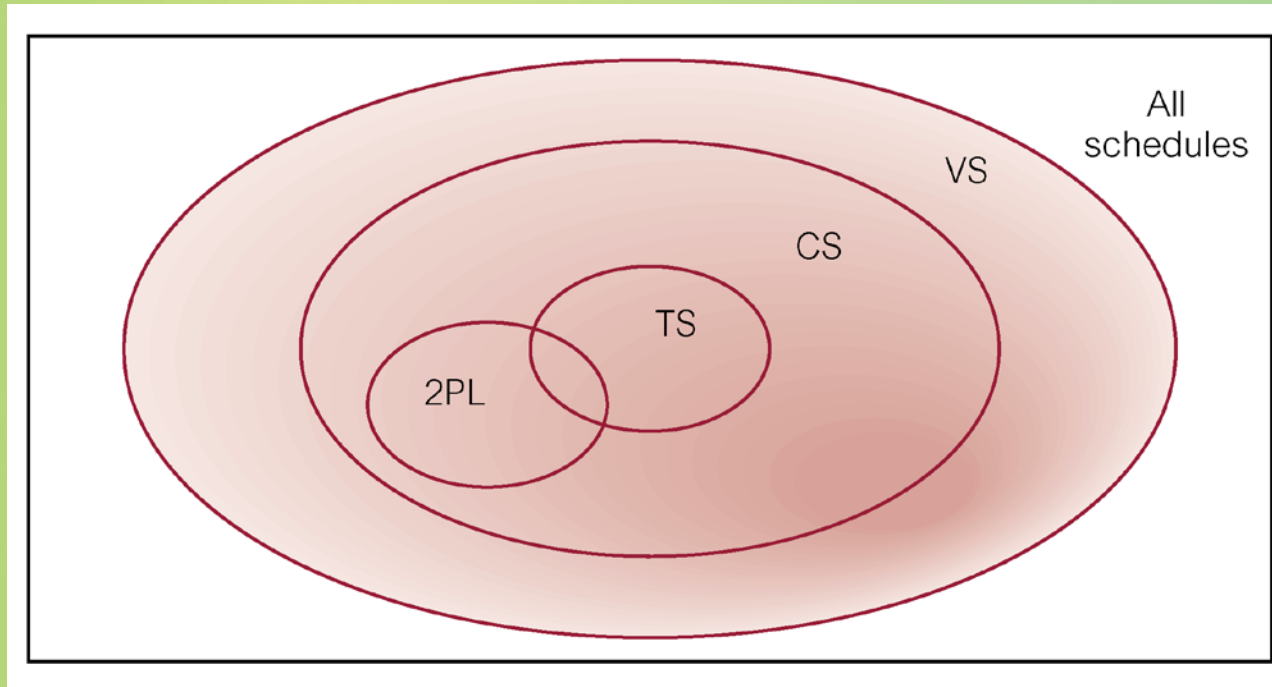
Time	Op	T <sub>19</sub>	T <sub>20</sub>	T <sub>21</sub>
t <sub>1</sub>		begin_transaction		
t <sub>2</sub>	read( <b>bal<sub>x</sub></b> )	read( <b>bal<sub>x</sub></b> )		
t <sub>3</sub>	<b>bal<sub>x</sub></b> = <b>bal<sub>x</sub></b> + 10	<b>bal<sub>x</sub></b> = <b>bal<sub>x</sub></b> + 10		
t <sub>4</sub>	write( <b>bal<sub>x</sub></b> )	write( <b>bal<sub>x</sub></b> )	begin_transaction	
t <sub>5</sub>	read( <b>bal<sub>y</sub></b> )		read( <b>bal<sub>y</sub></b> )	
t <sub>6</sub>	<b>bal<sub>y</sub></b> = <b>bal<sub>y</sub></b> + 20		<b>bal<sub>y</sub></b> = <b>bal<sub>y</sub></b> + 20	begin_transaction
t <sub>7</sub>	read( <b>bal<sub>y</sub></b> )			read( <b>bal<sub>y</sub></b> )
t <sub>8</sub>	write( <b>bal<sub>y</sub></b> )		write( <b>bal<sub>y</sub></b> ) <sup>+</sup>	
t <sub>9</sub>	<b>bal<sub>y</sub></b> = <b>bal<sub>y</sub></b> + 30			<b>bal<sub>y</sub></b> = <b>bal<sub>y</sub></b> + 30
t <sub>10</sub>	write( <b>bal<sub>y</sub></b> )			write( <b>bal<sub>y</sub></b> )
t <sub>11</sub>	<b>bal<sub>z</sub></b> = 100			<b>bal<sub>z</sub></b> = 100
t <sub>12</sub>	write( <b>bal<sub>z</sub></b> )			write( <b>bal<sub>z</sub></b> )
t <sub>13</sub>	<b>bal<sub>z</sub></b> = 50	<b>bal<sub>z</sub></b> = 50		commit
t <sub>14</sub>	write( <b>bal<sub>z</sub></b> )	write( <b>bal<sub>z</sub></b> ) <sup>‡</sup>	begin_transaction	
t <sub>15</sub>	read( <b>bal<sub>y</sub></b> )	commit	read( <b>bal<sub>y</sub></b> )	
t <sub>16</sub>	<b>bal<sub>y</sub></b> = <b>bal<sub>y</sub></b> + 20		<b>bal<sub>y</sub></b> = <b>bal<sub>y</sub></b> + 20	
t <sub>17</sub>	write( <b>bal<sub>y</sub></b> )		write( <b>bal<sub>y</sub></b> )	
t <sub>18</sub>			commit	

<sup>+</sup> At time t<sub>8</sub>, the write by transaction T<sub>20</sub> violates the first timestamping write rule described above and therefore is aborted and restarted at time t<sub>14</sub>.

<sup>‡</sup> At time t<sub>14</sub>, the write by transaction T<sub>19</sub> can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T<sub>21</sub> at time t<sub>12</sub>.



# Comparison of Methods



# Granularity of Data Items

- Size of data items chosen as unit of protection by concurrency control protocol.
- Ranging from coarse to fine:
  - The entire database.
  - A file.
  - A page (or area or database spaced).
  - A record.
  - A field value of a record.

# Granularity of Data Items

- **Tradeoff:**
  - coarser, the lower the degree of concurrency;
  - finer, more locking information that is needed to be stored.
- **Best item size depends on the types of transactions.**

# Hierarchy of Granularity

- Could represent granularity of locks in a hierarchical structure.
- Root node represents entire database, level 1s represent files, etc.
- When node is locked, all its descendants are also locked.
- DBMS should check hierarchical path before granting lock.

# Hierarchy of Granularity

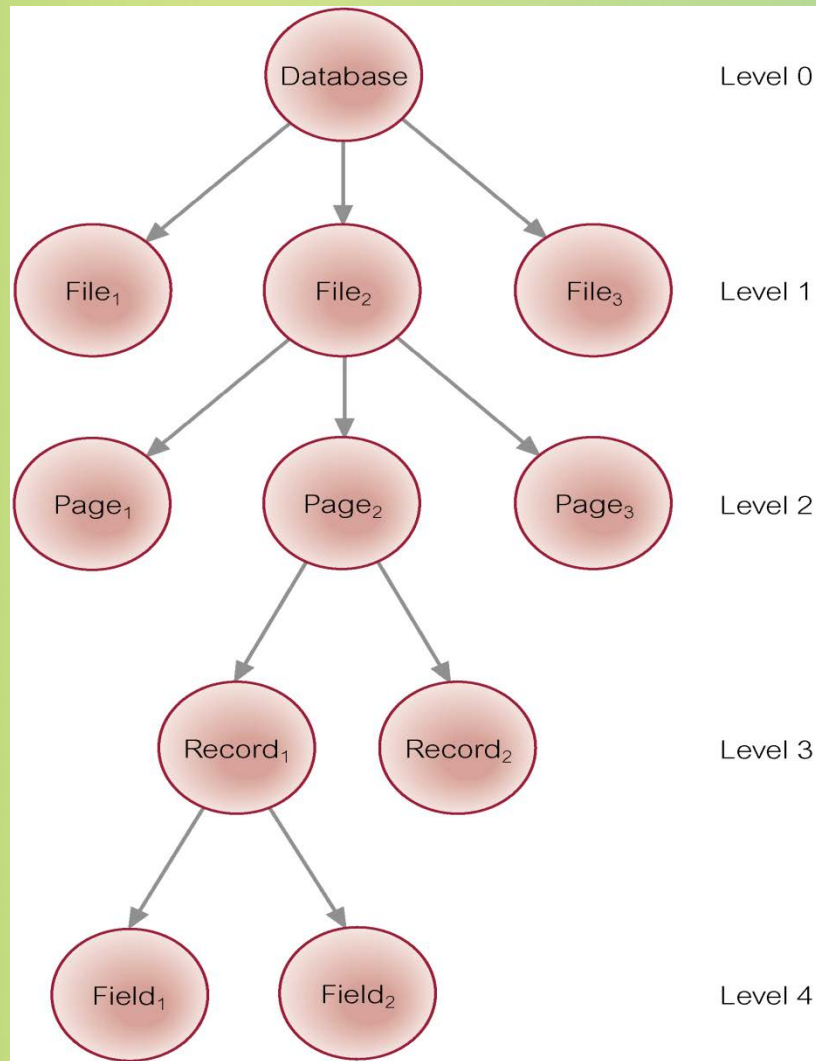
- *Intention lock* could be used to lock all ancestors of a locked node.
- Intention locks can be read or write. Applied top-down, released bottom-up.

**Table 20.1** Lock compatibility table for multiple-granularity locking.

	IS	IX	S	SIX	X
<i>IS</i>	✓	✓	✓	✓	✗
<i>IX</i>	✓	✓	✗	✗	✗
<i>S</i>	✓	✗	✓	✗	✗
<i>SIX</i>	✓	✗	✗	✗	✗
<i>X</i>	✗	✗	✗	✗	✗

✓ = compatible, ✗ = incompatible




# Levels of Locking



# Database Recovery

**Process of restoring database to a correct state in the event of a failure.**

## **Need for Recovery Control**

-  **Two types of storage: volatile (main memory) and nonvolatile.**
-  **Volatile storage does not survive system crashes.**
-  **Stable storage represents information that has been replicated in several nonvolatile storage media with independent failure modes.**

# Types of Failures

- **System crashes, resulting in loss of main memory.**
- **Media failures, resulting in loss of parts of secondary storage.**
- **Application software errors.**
- **Natural physical disasters.**
- **Carelessness or unintentional destruction of data or facilities.**
- **Sabotage.**



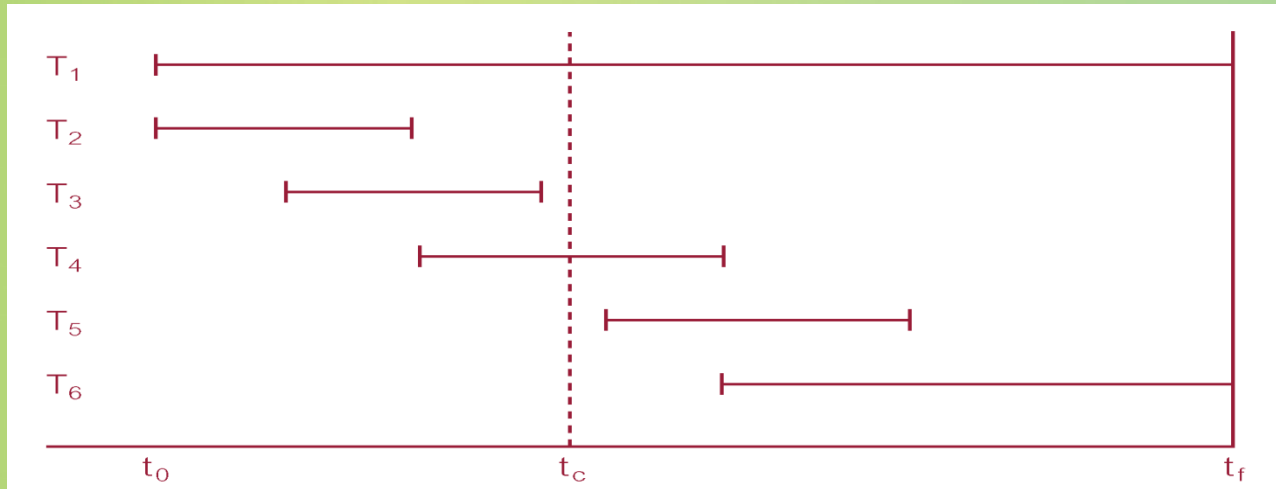
# Transactions and Recovery

- Transactions represent basic unit of recovery.
- Recovery manager responsible for atomicity and durability.
- If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, recovery manager has to *redo* (*rollforward*) transaction's updates.

# Transactions and Recovery

- If transaction had not committed at failure time, recovery manager has to *undo (rollback)* any effects of that transaction for atomicity.
- Partial undo - only one transaction has to be undone.
- Global undo - all transactions have to be undone.

# Example



- DBMS starts at time  $t_0$ , but fails at time  $t_f$ . Assume data for transactions  $T_2$  and  $T_3$  have been written to secondary storage.
- $T_1$  and  $T_6$  have to be undone. In absence of any other information, recovery manager has to redo  $T_2$ ,  $T_3$ ,  $T_4$ , and  $T_5$ .

# Recovery Facilities

- **DBMS should provide following facilities to assist with recovery:**
  - Backup mechanism, which makes periodic backup copies of database.
  - Logging facilities, which keep track of current state of transactions and database changes.
  - Checkpoint facility, which enables updates to database in progress to be made permanent.
  - Recovery manager, which allows DBMS to restore database to consistent state following a failure.

# Log File

- **Contains information about all updates to database:**
  - Transaction records.
  - Checkpoint records.
- **Often used for other purposes (for example, auditing).**

# Log File

- **Transaction records contain:**
  - Transaction identifier.
  - Type of log record, (transaction start, insert, update, delete, abort, commit).
  - Identifier of data item affected by database action (insert, delete, and update operations).
  - Before-image of data item.
  - After-image of data item.
  - Log management information.

# Sample Log File

Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
T1	10:12	START				0	2
T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new value)	3	5
T2	10:17	DELETE	STAFF SA9	(old value)		4	6
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
T3	10:18	START				0	11
T1	10:18	COMMIT				2	0
	10:19	CHECKPOINT	T2, T3				
T2	10:19	COMMIT				6	0
T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
T3	10:21	COMMIT				11	0

# Log File

- **Log file may be duplexed or triplexed.**
- **Log file sometimes split into two separate random-access files.**
- **Potential bottleneck; critical in determining overall performance.**



# Checkpointing

## Checkpoint

**Point of synchronization between database and log file. All buffers are force-written to secondary storage.**

- **Checkpoint record is created containing identifiers of all active transactions.**
- **When failure occurs, redo all transactions that committed since the checkpoint and undo all transactions active at time of crash.**

# Checkpointing

- In previous example, with checkpoint at time  $t_c$ , changes made by  $T_2$  and  $T_3$  have been written to secondary storage.
- Thus:
  - only redo  $T_4$  and  $T_5$ ,
  - undo transactions  $T_1$  and  $T_6$ .

# Recovery Techniques

- **If database has been damaged:**
  - Need to restore last backup copy of database and reapply updates of committed transactions using log file.
- **If database is only inconsistent:**
  - Need to undo changes that caused inconsistency. May also need to redo some transactions to ensure updates reach secondary storage.
  - Do not need backup, but can restore database using before- and after-images in the log file.

# Main Recovery Techniques

- **Three main recovery techniques:**
  - **Deferred Update**
  - **Immediate Update**
  - **Shadow Paging**

# Deferred Update

- Updates are not written to the database until after a transaction has reached its commit point.
- If transaction fails before commit, it will not have modified database and so no undoing of changes required.
- May be necessary to redo updates of committed transactions as their effect may not have reached database.

# Immediate Update

- Updates are applied to database as they occur.
- Need to redo updates of committed transactions following a failure.
- May need to undo effects of transactions that had not committed at time of failure.
- Essential that log records are written before write to database. *Write-ahead log protocol.*

# Immediate Update

- If no “*transaction commit*” record in log, then that transaction was active at failure and must be undone.
- Undo operations are performed *in reverse order in which they were written to log*.

# Shadow Paging

- Maintain two page tables during life of a transaction: *current* page and *shadow* page table.
- When transaction starts, two pages are the same.
- Shadow page table is never changed thereafter and is used to restore database in event of failure.
- During transaction, current page table records all updates to database.
- When transaction completes, current page table becomes shadow page table.