Chapter One

Transaction Management and Concurrency Control

Chapter 1 - 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 1 – Objectives cont'd...

- **Recovery Control**
  - **Some causes of database failure.**
  - **Purpose of transaction log file.**
  - **Purpose of check-pointing.**
  - **How to recover following database failure.**

Transaction Support

Transaction

Action, or series of actions, carried out by user or an 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. What are they?**
  - **Computation, Sorting ,Filtering, looping, branching**
- **Transaction transforms a 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)

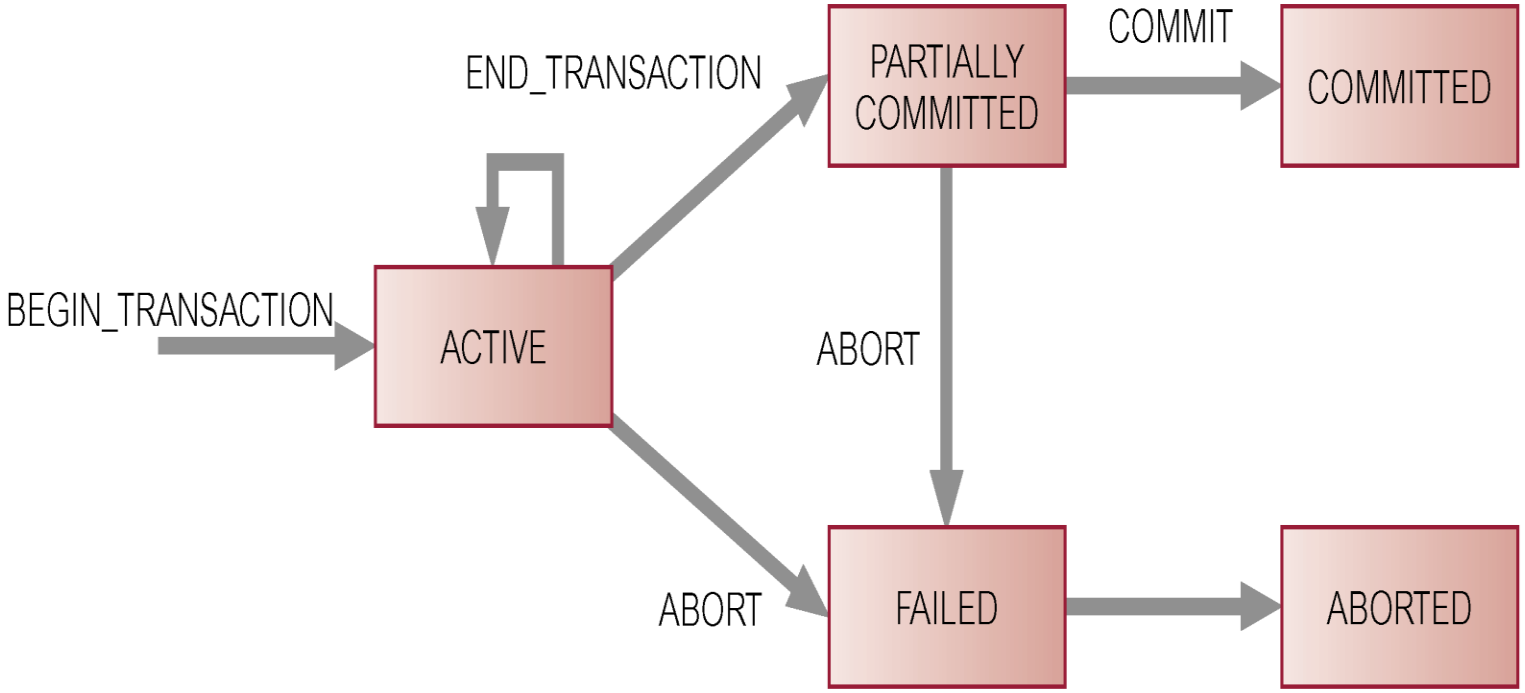
```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(b)

Transaction Support Cont'd...

- **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(Undone). Oh! What if it was a mistake?**
- **Aborted transaction that is rolled back can be restarted (redone) later.**

State Transition Diagram for Transaction



Properties of Transactions

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

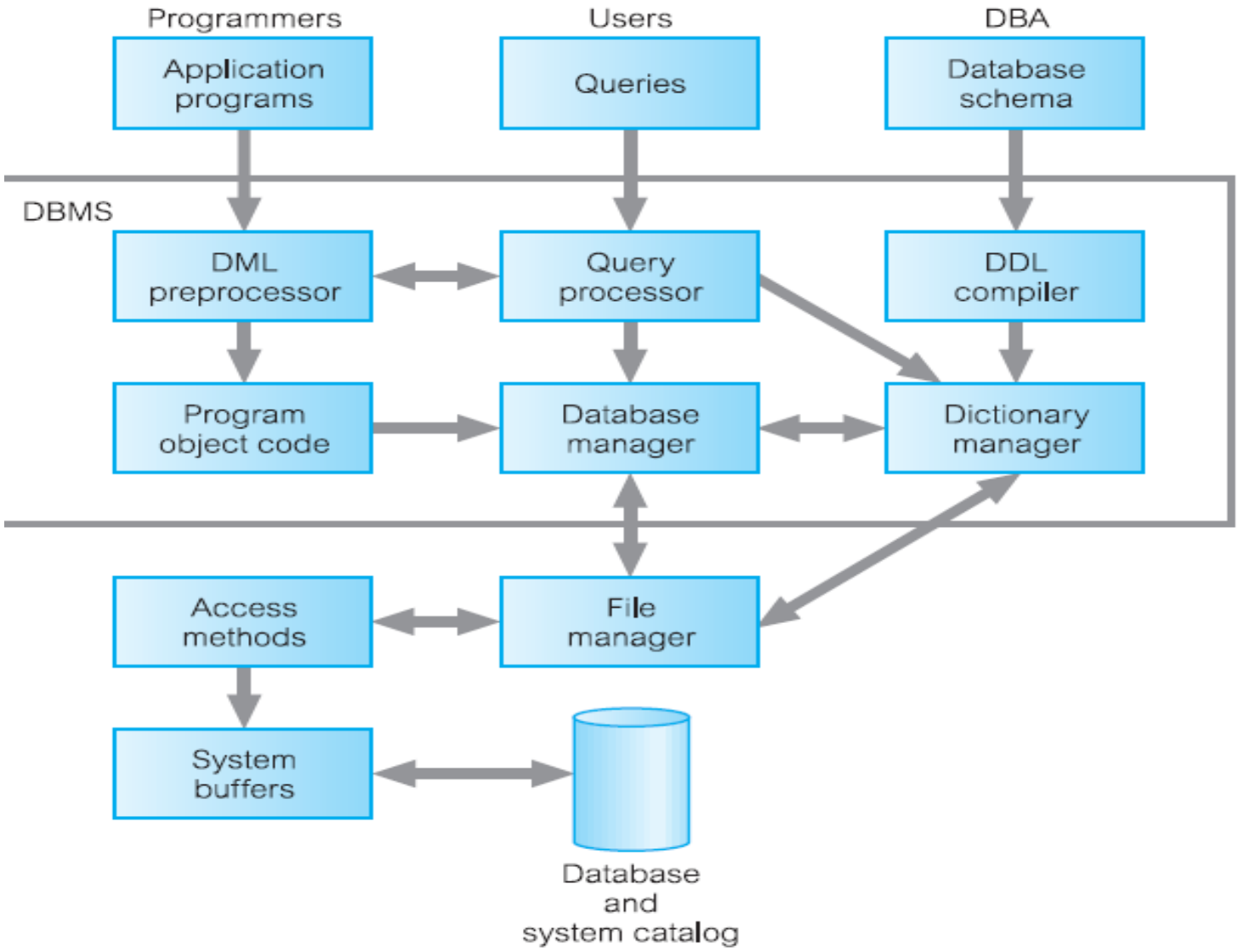
Atomicity 'All or nothing' property.

Consistency Must transform database from one consistent state to another.

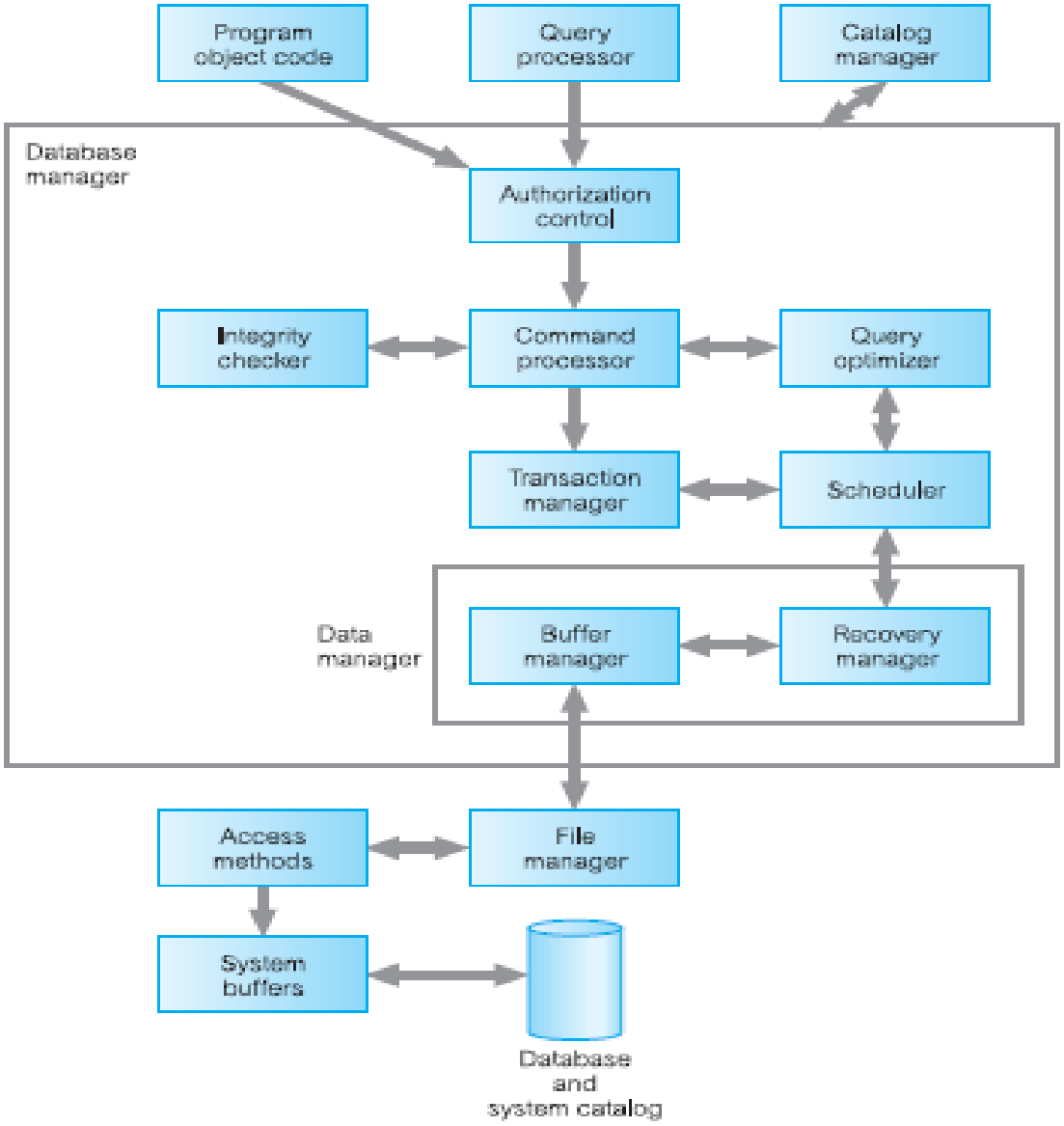
Isolation Partial effects of incomplete transactions should not be visible to other transactions.

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

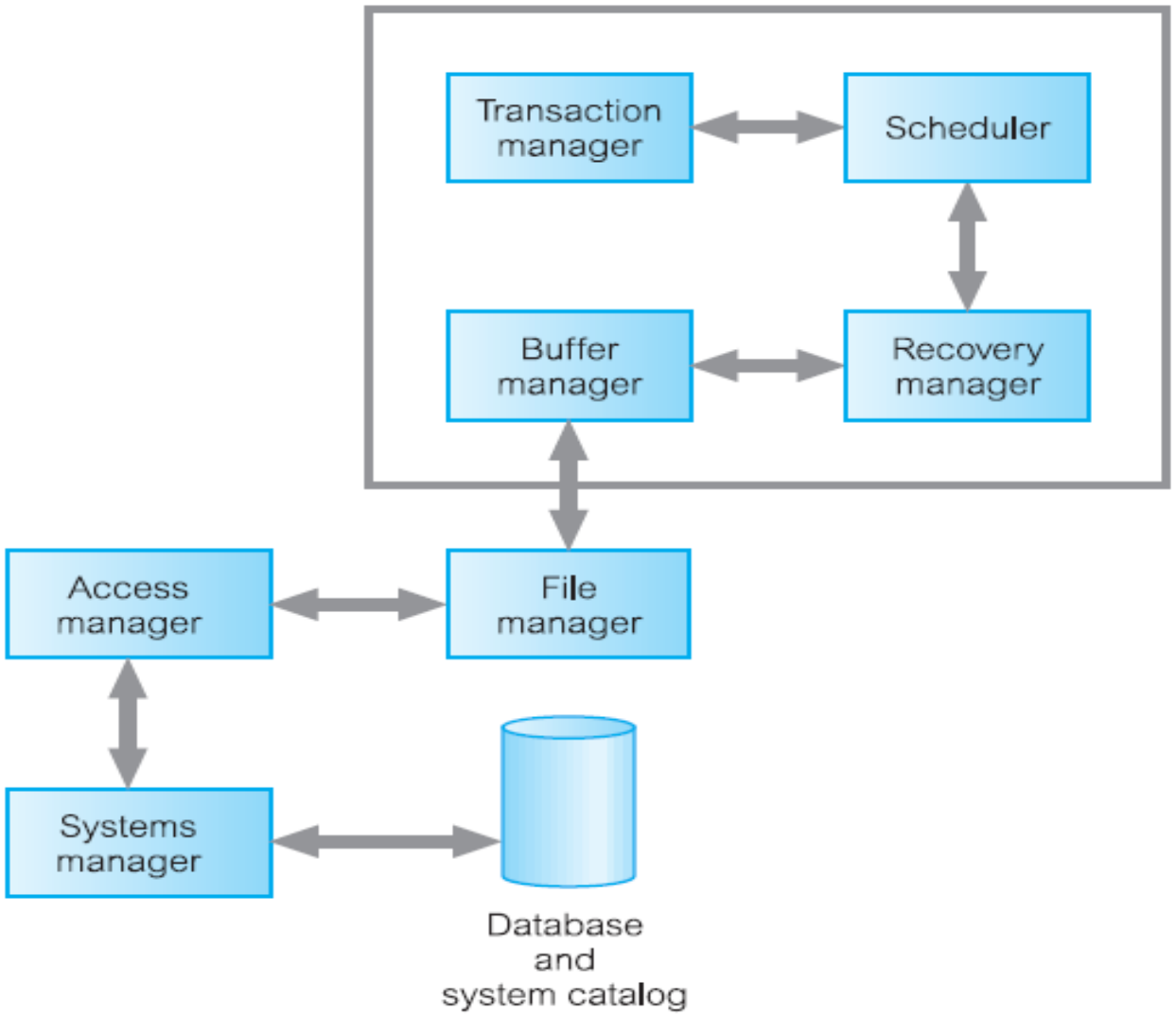
DBMS Architecture



Database Manager



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

- **Need to increase *efficiency and throughput by interleaving operations* from different transactions.**

- **Three examples of potential problems that can be 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

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

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

Uncommitted Dependency Problem (or dirty read)

- 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.

Uncommitted Dependency Problem

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

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

Inconsistent Analysis Problem

- **Occurs when transaction reads several values but second transaction updates some of them during execution of the first.**
- **Sometimes referred to as *fuzzy read* or *unrepeatable read*. (if reading value of a data item being Modified)**
- ***Phantom read (*Additional Tuples are read) is the name used when the "updating" transaction inserts new records**
- **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

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

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

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. (Most programs block for I/O and most systems have DMA-separate module for I/O). So, could run other programs in the meantime.
- 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(Operation Precedence).

Non-serial Schedule

- Schedule where operations from set of concurrent transactions are interleaved.
- Objective of serializability is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another.
- In other words, want to find non-serial 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 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.

Example of Conflict Serializability

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

Serializability premises

- **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 by the transaction),**
- **We can use *precedence(or serialization) graph* to test for serializability of a schedule.**

Precedence Graph

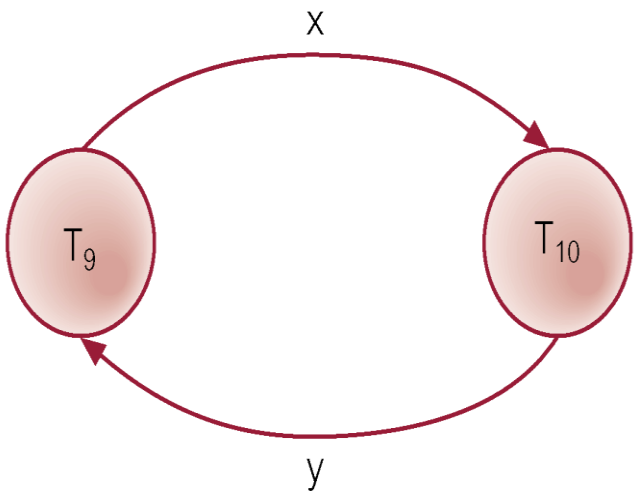
- **Create:**
  - **node for each transaction;**
  - **a directed edge $T_i \rightarrow T_j$, if T_j reads the value of an item written by T_i;**
  - **a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been read by T_i.**
  - **a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been written 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 partial effects of transaction to be undone.**
- **Durability states that once transaction commits, its changes cannot be undone (without running another, compensating, transaction). Effect should be permanent.**

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 pessimistic concurrency control techniques:
  - Locking,
  - Timestamping.
- Both are conservative(pessimistic) approaches: delay transactions in case they conflict with other transactions.
- Optimistic (a third approach) methods assume conflict is rare and only check for conflicts at commit time.

Locking

- A procedure used to control concurrent access to data.
- 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 Cont'd...

- **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(slide 28), a valid schedule using these rules(locking rules) is:**

S = {write_lock(T_9, bal_x), read(T_9, bal_x), write(T_9, bal_x), unlock(T_9, bal_x), write_lock(T_{10}, bal_x), read(T_{10}, bal_x), write(T_{10}, bal_x), unlock(T_{10}, bal_x), write_lock(T_{10}, bal_y), read(T_{10}, bal_y), write(T_{10}, bal_y), unlock(T_{10}, bal_y), commit(T_{10}), write_lock(T_9, bal_y), read(T_9, bal_y), write(T_9, bal_y), unlock(T_9, bal_y), 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 $bal_x = 220$ and $bal_y = 340$.
- 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.
- Although this may seem to allow greater concurrency, it also permits transactions to interfere with each other
- To guarantee serializability, need an additional protocol concerning the position of the *lock* and *unlock* operations in every transaction.

Two-Phase Locking (2PL)

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

- **Two phases for a transaction in handling locks:**
  - **Growing phase - acquires all locks but cannot release any locks.**
  - **Shrinking phase - releases locks but cannot acquire any new locks.**
  - **Which phase allows downgrading? and upgrading of locks?**

Variations of the 2PL

- **There are some variations of the 2PL protocol**
  - **Basic, Conservative, Strict, and Rigorous Two-Phase Locking**
- **The Basic 2PL variation makes sure that transactions follow growing and shrinking phases.**
- In practice, the ***most popular variation of 2PL is strict 2PL***, which guarantees strict schedules for transactions
  - In this variation, ***transaction will delay ONLY exclusive locks until after commit or rollback***.
- A more restrictive variation of 2PL is ***Rigorous 2PL***, which also guarantees strict schedules.
  - In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts
- **Conservative 2PL (or static 2PL)** requires a transaction to lock all the items it accesses before the transaction begins execution, by **predeclaring** its read-set and write-set.

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

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_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

Cascading Rollback

- If *every* transaction in a schedule follows 2PL, then the schedule is *conflict serializable*.
- However, problems can occur with interpretation of when locks can be released.
  - Leading the cascading rollback problem.

Cascading Rollback

Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock($\mathbf{bal_x}$)		
t_3	read($\mathbf{bal_x}$)		
t_4	read_lock($\mathbf{bal_y}$)		
t_5	read($\mathbf{bal_y}$)		
t_6	$\mathbf{bal_x} = \mathbf{bal_y} + \mathbf{bal_x}$		
t_7	write($\mathbf{bal_x}$)		
t_8	unlock($\mathbf{bal_x}$)	begin_transaction	
t_9	⋮	write_lock($\mathbf{bal_x}$)	
t_{10}	⋮	read($\mathbf{bal_x}$)	
t_{11}	⋮	$\mathbf{bal_x} = \mathbf{bal_x} + 100$	
t_{12}	⋮	write($\mathbf{bal_x}$)	
t_{13}	⋮	unlock($\mathbf{bal_x}$)	
t_{14}	⋮	⋮	
t_{15}	rollback	⋮	
t_{16}		⋮	begin_transaction
t_{17}		⋮	read_lock($\mathbf{bal_x}$)
t_{18}		rollback	⋮
t_{19}			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, delay the release of *all* locks until end of transaction.
  - i.e Commit/Unlock or rollback/Unlock
- If we delay the release of all locks until after commit/rollback, it is called "Rigorous 2PL" .
- If we delay the release of only exclusive locks until after commit/rollback, it is known as "Strict 2PL"

Deadlock - potential problem when using locking

When transactions follow 2PL, an impasse may occur 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}	⋮	⋮

Deadlock

- There is One and Only one way to break a deadlock: abort one or more of the deadlocked transactions.
- Deadlock should be transparent to user, so DBMS should restart transaction(s).
- Three general techniques for managing 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 (least used)

- **DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.**
- **Could order transactions using transaction timestamps: for prevention of deadlocks**
  - **Wait-Die - *only an older transaction can wait for younger one (NOT a younger waiting for an older one)*, otherwise transaction (if younger &waiting) is aborted (*dies*) and restarted with same timestamp.(Why same timestamp? to be old so that it can wait).**
    - **Non-Pre-emptive**

Deadlock Prevention

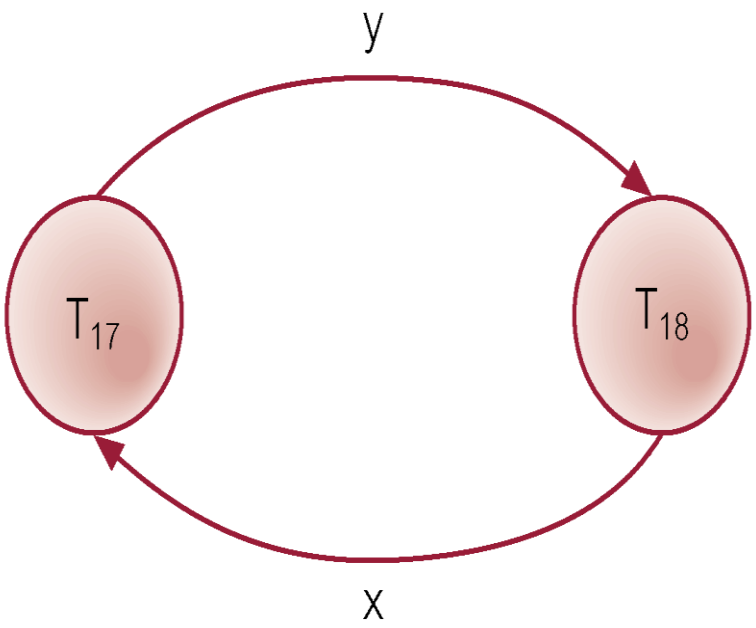
- **Wound-Wait** - *only a younger transaction can wait for an older one*. If older transaction requests lock held by younger one, younger one is aborted (*wounded*).
  - Pre-emptive
- A variant of 2PL, called conservative 2PL(static 2PL), can also be used to prevent deadlock.
  - In this approach, a transaction *obtains all its locks when it begins*, or it waits until all the locks are available.

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 is waiting to lock item locked by T_j.
- Deadlock exists if and only if WFG contains a cycle.
- WFG is created at regular intervals that will not make the system busy.

Example - Wait-For-Graph (WFG)

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}	⋮	⋮



Recovery from Deadlock Detection

- **Several issues:**
  - **choice of deadlock victim;**
  - **how far to roll a transaction back;**
  - **Avoiding starvation on some transaction.**
    - **Transaction starvation is similar to "Livelock"**
      - **What is "Livelock"? (reading assignment)**

Timestamping

- A concurrency control protocol where by transactions are ordered globally so that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.
- *For any operation to proceed, Last update on a data item* must be carried out by an older transaction
- Conflict is resolved by rolling back and restarting transaction(s).
- No locks so, no deadlock.
- Like the 2PL, this also guarantees the serializability of a schedule

Basic Timestamping Protocol

Timestamp

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

- Can be generated by using system clock at time when transaction has started, or by incrementing a logical counter every time a new transaction starts.
- Basic Timestamping guarantees that transactions are *conflict serializable*, and the results are equivalent to a serial schedule in which the transactions are executed *in chronological order of the timestamps*

Basic Timestamping - procedure

- **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(latter) timestamp. Why new?**
- **Also timestamps for data items:**
  - **read-timestamp - timestamp of last transaction to read item;**
  - **write-timestamp - timestamp of last transaction to write item.**

Basic Timestamping - Read(x)

- Consider a transaction T with timestamp ts(T):
- Check the last write on the Data Item

ts(T) < WTS(x)

  - x already updated by younger (later) transaction.
  - Transaction must be aborted and restarted with a new timestamp.
- Otherwise the Read continues and the RTS(X) will be set to the max of (RTS(x) and ts(T)). Why Max?

Basic Timestamping - Write(x)

Check the last read and write

If ts(T) < RTS(x)

- **X is already read by a younger transaction**
- **Hence error to update now and restarted with new timestamp**

If ts(T) < WTS (x)

- **x already written by younger transaction.**
  - This means that transaction T is attempting to write an obsolete value of data item x. Transaction T should be rolled back and restarted using a new timestamp
- **Otherwise, operation is accepted and executed.**
  - **WTS(x) is set to ts(T)**

Modifications to Basic Timestamping

- A modification to the basic timestamp ordering protocol that *relaxes conflict serializability* can be used to provide greater **concurrency** by **rejecting obsolete write operations**
  - The extension(modification) is known as **Thomas's write rule**
  - **In this case , if an older transaction tries to write into an item that was already written by a younger one, then its write is ignored and it does not need to be restarted.**
  - **Reduces the number of re-works(re-starting)**

Example –Timestamp Ordering

Time	Op	T_{19}	T_{20}	T_{21}
t_1		begin_transaction		
t_2	read(bal_x)	read(bal_x)		
t_3	$bal_x = bal_x + 10$	$bal_x = bal_x + 10$		
t_4	write(bal_x)	write(bal_x)	begin_transaction	
t_5	read(bal_y)		read(bal_y)	
t_6	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	begin_transaction
t_7	read(bal_y)			read(bal_y)
t_8	write(bal_y)		write(bal_y)+	
t_9	$bal_y = bal_y + 30$			$bal_y = bal_y + 30$
t_{10}	write(bal_y)			write(bal_y)
t_{11}	$bal_z = 100$			$bal_z = 100$
t_{12}	write(bal_z)			write(bal_z)
t_{13}	$bal_z = 50$	$bal_z = 50$		commit
t_{14}	write(bal_z)	write(bal_z)‡	begin_transaction	
t_{15}	read(bal_y)	commit	read(bal_y)	
t_{16}	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	
t_{17}	write(bal_y)		write(bal_y)	
t_{18}			commit	

+ At time t_8, the write by transaction T_{20} violates the first timestamping write rule described above and therefore is aborted and restarted at time t_{14}.

‡ At time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

Optimistic Techniques

- Based on the assumption that conflict is rare and it is more efficient to let transactions proceed without delays to ensure serializability.
- At commit, check is made to determine whether conflict has occurred.
- If there is a conflict, transaction must be rolled back and restarted.
- Potentially allows greater concurrency than traditional protocols.

Optimistic Techniques

- **Three phases:**
  - **Read**
  - **Validation**
  - **Write (Only for an Update Transaction- involving any DB modification)**

Optimistic Techniques - Read Phase

- **Extends from start until immediately before commit.**

- **Transaction reads values from database and stores them in local variables (buffer). Updates are applied to a local copy of the data (the buffer).**

Optimistic Techniques - Validation Phase

- Follows the read phase.
- For read-only transaction, checks that data read are still current values. If no interference, transaction is committed, else aborted and restarted.
- For update transaction, checks transaction leaves database in a consistent state, with serializability maintained.

Validation phase Rules

- Each transaction T is assigned a timestamp at the start of its execution, start(T), one at the start of its validation phase, validation(T), and one at its finish time, finish(T), including its write phase, if any. To pass the validation test, one of the following must be true:
- **(1)** All transactions S with earlier timestamps must have finished before transaction T started; that is, finish(S) < start(T).(serial)
- **(2)** If transaction T starts before an earlier one(S) finishes, then:
  - (a) the set of data items written by the earlier transaction are not the ones read by the current transaction; and
  - (b) the earlier transaction completes its write phase before the current transaction enters its validation phase, that is start(T) < finish(S) < validation(T).

Optimistic Techniques - Write Phase

- **Follows successful validation phase for update transactions.**
- **Updates made to local copy are applied to the database.**

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 (**sometimes called an area or database space – a section of physical disk in which relations are stored**).**
  - **A record.**
  - **A field value of a record.**

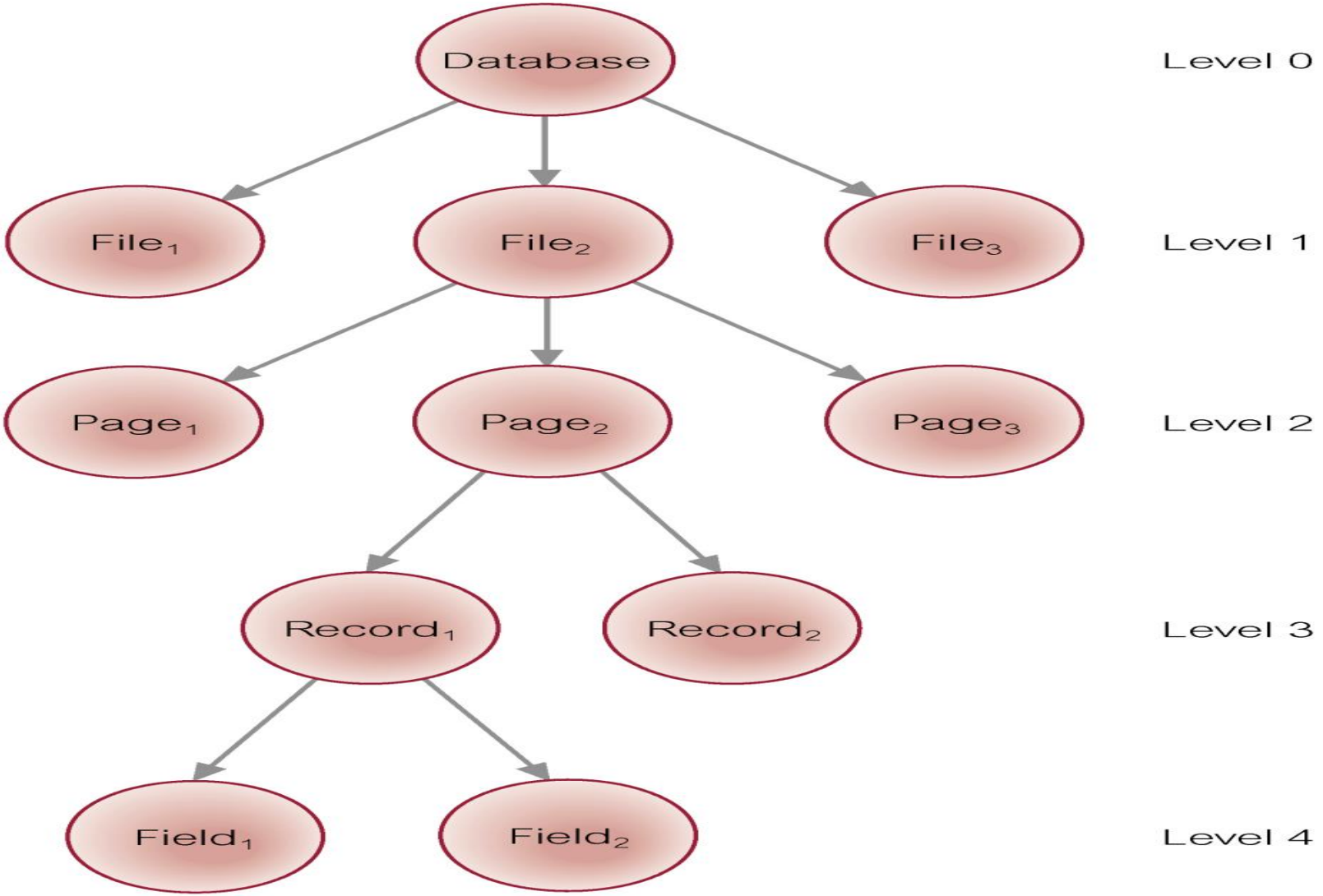
Granularity of Data Items

- **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/nature 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.
- When a node is locked an intension lock is placed on its ancestors.
- DBMS should check hierarchical path before granting lock.

Granularity (Levels) of Locking



Database Recovery

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

- **The Need for Recovery Control**
  - **Two types of storage: volatile (main memory) and non-volatile.**
  - **Volatile storage does not survive system crashes.**
  - **Stable storage represents information that has been replicated in several non-volatile storage media with independent failure modes like in RAID technology.**

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 (intentional corruption or destruction of data, hardware, or software Facilities).

Transactions and Recovery

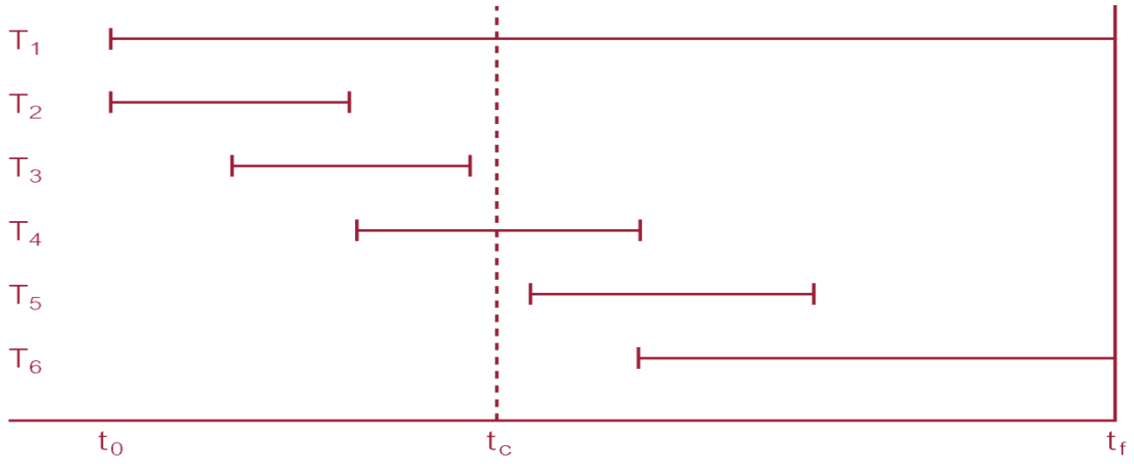
- Transactions represent basic unit of Work and also recovery.
- The explicit writing of the buffers to secondary storage is known as force-writing
- Recovery manager is responsible for Atomicity and Durability of the ACIDity properties.
  - "I"- taken care of by Scheduler, "C" both by DBMS and Programmer
- 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 active 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 which are 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 (two types of records are maintained)
  - Transaction records.
  - Checkpoint records.
- Often used for other purposes too (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 Such as pointer to the next and previous log record.

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 (multiple copies maintained).
- Log file sometimes split into two separate random-access files.
- Potential bottleneck; critical in determining overall performance.

Checkpointing

- The information in the log file is used to recover from a database failure.
- One difficulty with this scheme is that when a failure occurs we may not know how far back in the log to search and we may end up redoing transactions that have been safely written to the database.
- To limit the amount of searching and subsequent processing that we need to carry out on the log file, we can use a technique called **checkpointing**

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* at the time of checkpointing**
- **When failure occurs, redo all transactions that committed since the checkpoint and undo all transactions active at time of crash.**

Checkpointing

- In previous example(slide 75?), with checkpoint at time t_c, changes made by T_2 and T_3 have already been written to secondary storage.
- Thus:
  - only redo T_4 and T_5,
  - undo transactions T_1 and T6.

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 – log based**
  - **Immediate Update -log based**
  - **Shadow Paging -Non- Log based Scheme**

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.
- Redo Operations use the after image values to rollforward.

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 is done. *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*. (we use before image values to restore)**
- **If there is a "*transaction commit*" log record then we redo the transaction.**
- **Redo Operations are done *in the order they were written to log* using the after image values**

Shadow Paging

- Maintain two page tables during the life of a transaction: *current* page and *shadow* page table.
- When transaction starts, the two pages are the same.
- Shadow page table is never changed thereafter and is used to restore database in event of failure.
- During transaction processing, current page table records all updates to database.
- When transaction completes, current page table becomes shadow page table.
- Better than the log based; since no log management and undo/redo operation but disadvantageous in that it may introduce disk fragmentation and routine garbage collection to reclaim inaccessible disk blocks