Chapter 20

Transaction Management

Chapter 20 - Objectives

- Function and importance of transactions.
- Properties of transactions.
- Concurrency Control
  - Deadlock and how it can be resolved.
  - Granularity of locking.

Chapter 20 - Objectives

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

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)

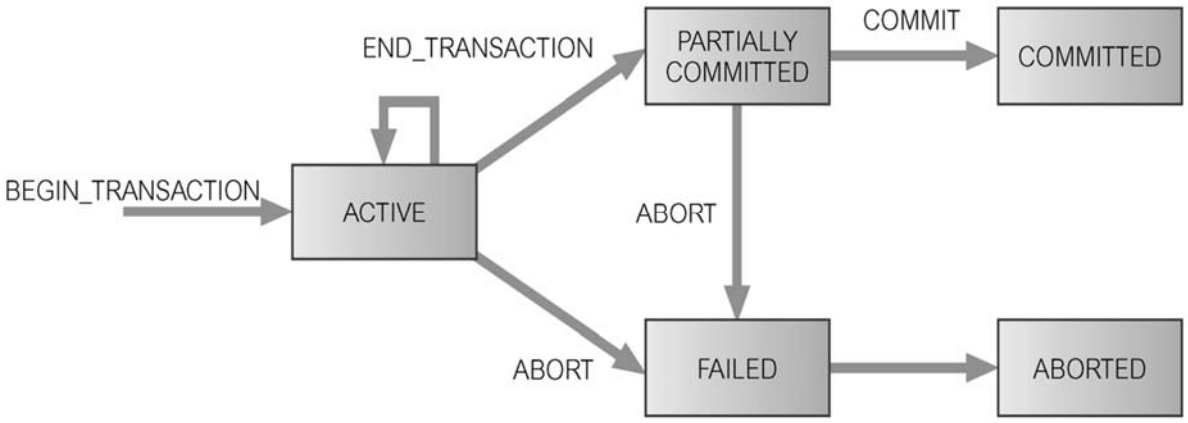
```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(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 of a transaction are:**

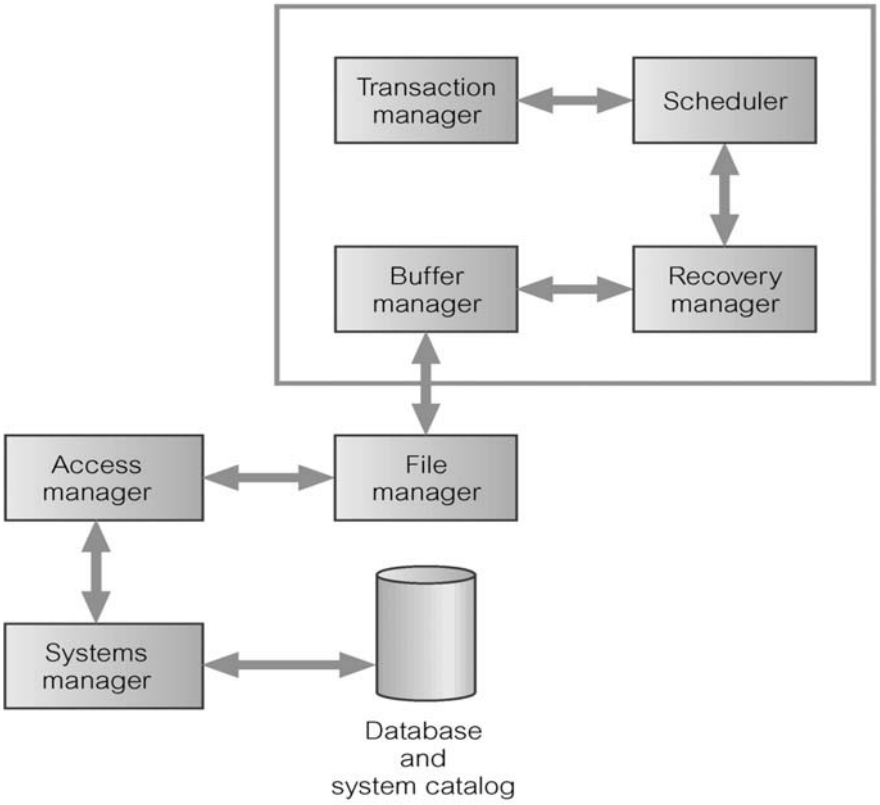
Atomicity **'All or nothing' property.**

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.**

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 (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

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.**

Inconsistent Analysis Problem

- **Occurs when transaction reads several values but second transaction updates some of them during execution of first.**
- **Sometimes referred to as *fuzzy read* or *unrepeatable read*.**
- **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_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.

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

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 two transactions above, a valid schedule using these 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

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	

- 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.
- 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_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

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}	⋮	⋮

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.**

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.

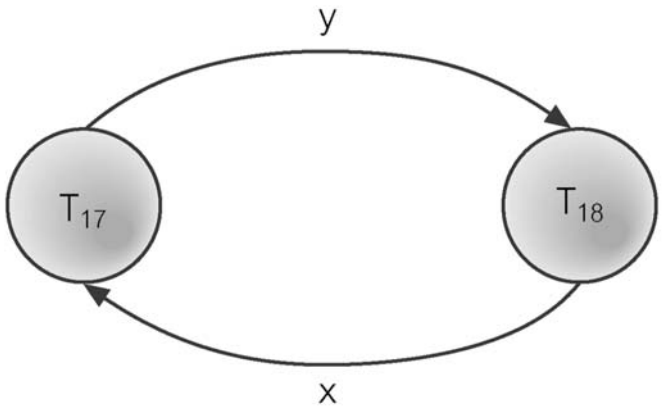
Deadlock Prevention

– **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*).**

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 -> 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.**

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.**

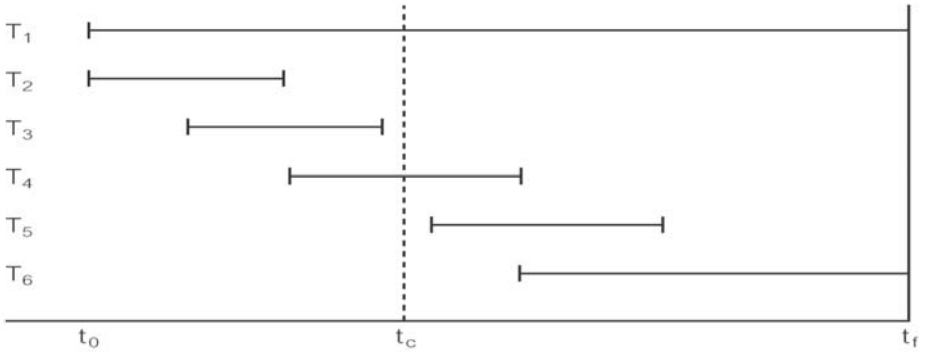
Transactions and Recovery

- **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 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.