Overview

- How to deal with situations where multiple users access and modify entries in the database at the same time?
- Problems that can occur (conflicts)
  - How to formally represent multiple database accesses?
  - What are the typical problems?
- Ways of addressing conflicts
  - Locking
  - Serialisation

Transactions

- An action or a series of actions carried out by a single user or application program, which read/update the contents of the DB
- A transaction is a logical unit of DB processing, consisting of one or more DB access operations
- Transaction boundaries may be specified implicitly or explicitly
- Transactions are recorded in system log, kept on disk

Example Transactions

<u>Successful</u>

```
begin_transaction;
read_item(X);
X:=X-10;
write_item(X);read_item(Y);
Y:=Y+5;
write_item(Y);
end_transaction;
commit;
```

<u>Unsuccessful</u>

```
begin_transaction;
read_item(X);
X:=X-10;
write_item(X);read_item(Y);
[transaction fails]
abort;
```

Concurrency

- A number of operations that are executed while overlapping in time
- Concurrency is particularly important if the different operations use the same data
- Conflicts can arise
- A number of solutions
  - Serialisation
  - Locking

Transaction Properties [1]

- Must hold for every transaction for the DB to remain stable
- **ACID** properties
  - **A**tomicity
    - A transaction should be treated as an indivisible unit
    - Managed by transaction recovery subsystem
  - **C**onsistency preservation
    - A transaction must transform the database from one consistent state to another consistent state - only valid data will be written to the DB
    - Managed by programmers / DBMS module

Transaction Properties [2]

- **I**solation
  - Transactions should execute independently of each other
  - Managed by concurrency control subsystem
- **D**urability
  - Effects of a successful transaction must be permanently recorded in the DB
  - Managed by recovery subsystem

Concurrency Control

- Concurrency control is necessary because of
  - Lost update problem
  - Uncommitted dependency (or dirty read) problem
  - Inconsistent analysis problem

Lost Update Problem

- An apparently completed update by one user can be overridden by another user
- While T1 (transaction 1) reads the value of an item, the value of that item is changed by T2 (transaction 2)
- This can lead to situations where one of the changes (updates) of one transaction are disregarded (lost)

EXAMPLE

Time	T1	T2	X
t_1		`begin_transaction`	100
t_2	`begin_transaction`	`read(X)`	100
t_3	`read(X)`	`X:=X+100`	100
t_4	`X:=X-10`	`write(X)`	200
t_5	`write(X)`	`commit`	90
t_6	`commit`		90

Uncommitted Dependency Problem

- The uncommitted dependency problem occurs when T3 is allowed to use the result of another transaction (T4) before T4 has committed the changes
- The reason for not committing vary (cancelled by the user, connection problems, system crashes, etc.)
- Failure to commit causes a rollback, but other transactions are unaware of the rollback

EXAMPLE

Time	T3	T4	X
t_1		`begin_transaction`	100
t_2		`read(X)`	100
t_3		`X:=X+100`	100
t_4	`begin_transaction`	`write(X)`	200
t_5	`read(X)`	...	200
t_6	`X:=X-10`	rollback	100
t_7	`write(X)`		190
t_8	`commit`		190

Inconsistent Analysis Problem

- If one transaction (T6) reads in several items from the database while some of the values are updated by another transaction (T5)
- This leads to a situation where some of the values T6 uses are not (yet) updated by T5 but some of the items have been updated

EXAMPLE

Time	T5	T6	X	Y	Z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum=0	100	50	25	0
t_3	read(X)	read(X)	100	50	25	0
t_4	X:=X-10	sum:=sum+X	100	50	25	100
t_5	write(X)	read(Y)	90	50	25	100
t_6	read(Z)	sum:=sum+Y	90	50	25	150
t_7	Z:=Z+10		90	50	25	150
t_8	write(Z)		90	50	35	150
t_9	commit	read(Z)	90	50	35	150
t_{10}		sum:=sum+Z	90	50	35	185
t_{11}		commit	90	50	35	185

Schedules and Serialisability

- The objective of a concurrency control is to schedule transactions in such a way as to avoid any interference between them
- One solution is to allow only one transaction at a time
- Other solution is for scheduled transactions to work concurrently. How do we order them in time?

Schedules

- A sequence of operations by a set of concurrent transactions that preserves the order of the operations in each individual transaction
- Two operations conflict if
  - They belong to different transactions AND
  - They access the same item (e.g. `X`, `Y`) AND
  - At least one of the operations is a `write_item(X)`

Schedule Criteria [1]

- Complete
  - All operations from each of the transactions is present (read, write, etc.)
  - Commit or abort operation must be last in each transaction
  - Any pair of operations from same transaction appear in correct order
  - For any pair of conflicting operations, one must occur before the other in the schedule
- Partial order of operations
  - Two non-conflicting operations may occur simultaneously

Schedule Criteria [2]

- Recoverability
  - Should not be necessary to rollback, i.e. undo write operations to DB after commit point
  - Transaction only commits when all other transactions that are writing to common data item have committed
- Avoidance of cascading rollback
  - All transactions only read items written by committed transactions
- Strictness
  - Transactions cannot read or write items until last transaction to write item has committed or aborted

Example of Nonrecoverable Schedule

Time	T1	T2
t_1	begin_transaction	
t_2	read(X)	
t_3	X:=X+10	
t_4	write(X)	begin_transaction
t_5		read(X)
t_6		X:=X*1.1
t_7		write(X)
t_8		read(Y)
t_9		Y:=Y*1.1
t_{10}		write(Y)
t_{11}	read(Y)	commit
t_{12}	Y:=Y-100	
t_{13}	write(Y)	
t_{14}	commit	

Types of Schedules

- Serial schedule
  - All operations from each transaction are executed consecutively, without operations from different transactions interleaving
- Non-serial schedule
  - Operations from different transactions are interleaved

Schedule Equivalence

- Result equivalent
  - Produces same final DB state
- View equivalent:
- For two schedules S and S' it is the case that
  - If T in S reads the initial value of X then T in S' reads the initial value of X
  - If T1 reads the value of X, written to by T2 then T1 must also read the value of X written to by T2 in S'
  - If the last write operation to X in S was done by T1, then T1 also to be the last transaction to write to X in S'

Example of Serial Schedule

Time	T1	T2
t_1	begin_transaction	
t_2	read(X)	
t_3	write(X)	
t_4	read(Y)	
t_5	write(Y)	
t_6	commit	
t_7		begin_transaction
t_8		read(X)
t_9		write(X)
t_{10}		read(Y)
t_{11}		write(Y)
t_{12}		commit

Example of Non-Serial, Equivalent Schedules

Time	T1	T2	T1	T2
t_1	begin_trans		begin_trans	
t_2	read(X)		read(X)	
t_3	write(X)		write(X)	
t_4		begin_trans		begin_trans
t_5		read(X)		read(X)
t_6		write(X)	read(Y)	
t_7	read(Y)			write(X)
t_8	write(Y)		write(Y)	
t_9	commit		commit	
t_{10}		read(Y)		read(Y)
t_{11}		write(Y)		write(Y)
t_{12}		commit		commit

Serialisable Schedules [1]

- A non-serial schedule is called serialisable if
  - there is a way in which the transactions can be executed concurrently without interfering with one another
  - and thereby produce a database state that could be produced by a serial execution
- If a set of transactions executes concurrently, we say that the schedule is correct if it produces the same results as some serial execution

Serialisable Schedules [2]

- In Serialisability, the ordering of read and write operations is important
  - If two transactions only read an item, they do not conflict and the **order is not important**
  - If two transactions either read or write completely separate items, they do not conflict and **the order is not important**
  - If one transactions writes a item and another reads or writes the same item, they could conflict and **the order is important**

Conflict Serialisability

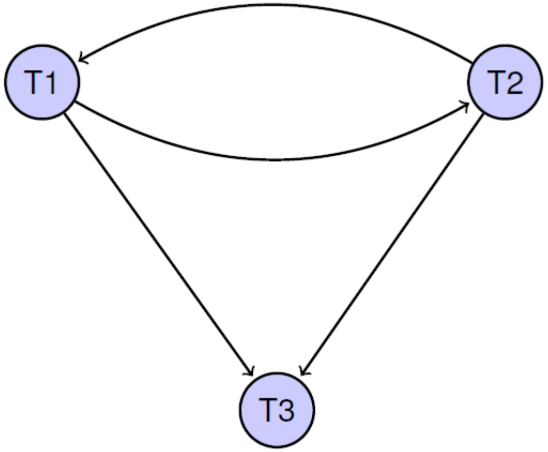
- Two transactions T1 and T2 are in conflict if an operation in T1 and an operation in T2 forces a temporal order between the transactions
- If two transaction can be serialised while maintaining these conflicts (temporal orders), they are called conflict serialisable

Precedence Graphs

- Precedence graphs allow one to visualize conflicts between transactions
- A precedence graph has:
  - A node for each transaction
  - A directed edge Ti →Tj , if Tj reads the value of an item written to by Ti
  - A directed edge Ti →Tj , if Tj writes to an item after it has been read by Ti
  - A directed edge Ti →Tj , if Tj writes to an item after it has been written to by Ti
- If there is a cycle in the graph, the transactions are not conflict serialisable

Example of Precedence Graph

Time	T1	T2	T3
t_1	`begin_trans`		
t_2	`read(X)`		
t_3		`begin_trans`	
t_4		`write(X)`	
t_5		`commit`	
t_6	`write(X)`		
t_7	`commit`		
t_8			`begin_trans`
t_9			`write(X)`
t_{10}			`commit`



Concurrency Control Techniques

- Needed to make a schedule
- Locking
- Deadlocks
- Timestamps

Locking [1]

- A lock prevents several transactions from accessing and/or changing items while another transaction accesses and/or changes it
- Lock
  - Variable describing status of an item
  - Information held in a lock table
- Danger of deadlock
  - Each transaction in a set of transactions is waiting for an item locked by another transaction in the same set

Locking [2]

- Types of locks
  - Binary
    - Two possible states: locked or unlocked
    - Two operations: `lock_item(X),unlock_item(X)`
  - Shared /Exclusive
    - Multiple states
    - Three operations: `read_lock(X),write_lock(X),unlock(X)`

Binary Locks

- Transaction must lock data item before `read_item` or `write_item` operations
- Transaction must unlock data item after finishing with it
- No two transactions can access same data item concurrently

Shared /Exclusive Locks

- Shared lock: If a transaction has a shared lock on an item, it can read the item but not update it
- Exclusive Lock: If a transaction has an exclusive lock on an item, it can both read and update it
- Some systems allow shared locks to be upgraded to exclusive locks
- Similarly, sometimes exclusive locks can be downgraded to shared locks

Locking Schedule

Time	T1	T2
$t_{1,2}$	write_lock(X);read(X)	
t_3	X:=X+100	
$t_{4,5}$	write(X);unlock(X)	begin_transaction
$t_{6,7}$		write_lock(X);read(X)
t_8		X:=X*1.1
$t_{9,10}$		write(X);unlock(X)
$t_{11,12}$		write_lock(Y);read(Y)
t_{13}		Y:=Y*1.1
$t_{14,15}$		write(Y);unlock(Y)
$t_{16,17}$	write_lock(Y);read(Y)	commit
t_{18}	Y:=Y-100	
$t_{19,20}$	write(Y);unlock(Y)	
t_{21}	commit	

If X=100 and Y=400, if $T_1 < T_2$ then X=220, Y=330, and if $T_2 < T_1$ then X=210, Y=340, but the schedule above gives: X=220 and Y=340

Two-phase Locking (2PL)

- The previous example shows that locking does not always guarantee serialisability
- 2PL does ensure serialisability
- Principle of 2PL
  - Every transaction must lock an item (read or write) before accessing it
  - Once a lock has been released, no new items can be locked
- 2PL has two phases: the growing phase and the shrinking phase
- Growing phase: all locks are acquired but cannot release any
- Shrinking phase: all locks are released but cannot acquire any

Preventing the Lost Update Problem

Time	T1	T2	X
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(X)	100
t_3	write_lock(X)	read(X)	100
t_4	WAIT	X:=X+100	100
t_5	WAIT	write(X)	200
t_6	WAIT	commit/unlock(X)	200
t_7	read(X)		200
t_8	X:=X-10		200
t_9	write(X)		190
t_{10}	commit/unlock(X)		190

Preventing the Uncommitted Dependency Problem

Time	T3	T4	X
t_1		`begin_transaction`	100
t_3		`write_lock(X)`	100
t_4		`read(X)`	100
t_5	`begin_transaction`	`X:=X+100`	100
t_4	`write_lock(X)`	`write(X)`	200
t_5	WAIT	`rollback/unlock(X)`	100
t_4	`read(X)`		100
t_6	`X:=X-10`		100
t_7	`write(X)`		90
t_8	`commit/unlock(X)`		90

What are Deadlocks?

- A deadlock occurs if several transaction are waiting for each other to unlock a data item
- General solutions to deadlocks
  - Timeouts
  - Deadlock *prevention*
  - Deadlock *detection* and *recovery*

Example of Deadlocks

Time	T1	T2
t_1	`begin_transaction`	
t_2	`write_lock(X)`	`begin_transaction`
t_3	`read(X)`	`write_lock(Y)`
t_4	`X:=X-10`	`read(Y)`
t_5	`write(X)`	`Y:=Y+100`
t_6	`write_lock(Y)`	`write(Y)`
t_7	WAIT	`write_lock(X)`
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	...	...

Deadlock Prevention

- Timeouts
  - A transaction waits a specified amount of time for a lock
  - If the lock is not granted, the transaction is aborted
- Deadlock prevention
  - Order transactions by time, abort younger one, while keeping track of the original time the younger transaction was started
  - Conservative 2PL: All data items have to be locked at the beginning of a transaction
  - Disadvantage of conservative 2PL: locks are held longer; time consuming to determine whether all required locks are free; longer waits for all required locks

Timestamping

- A timestamp is a unique identifier (created by the DBMS) that indicates the relative starting time of transactions
- Timestamping is a concurrency protocol that order transactions, such that transactions with smaller timestamps get priority in the event of a conflict

Summary

- Transactions
- Concurrency problems (lost update problem etc.)
- Schedules (serial, non-serial, serialisability)
- Conflict serialisability and precedence graphs
- Locking (2PL)
- Deadlocks