Transactions

Overview

- *Transaction*: A sequence of database actions enclosed within special tags
- Properties:
  - ***A*tomicity**: Entire transaction or nothing
  - ***C*onsistency**: Transaction, executed completely, takes database from one consistent state to another
  - ***I*solation**: Concurrent transactions *appear* to run in isolation
  - ***D*urability**: Effects of committed transactions are not lost
- Consistency: Programmer needs to guarantee this
  - DBMS can do a few things, e.g., enforce constraints on the data
- Rest: DBMS guarantees

How does..

- .. this relate to *queries* that we discussed ?
  - Queries don't update data, so *durability* and *consistency* not relevant
  - Would want *concurrency*
    - Consider a query computing balance at the end of the day
  - Would want *isolation*
    - What if somebody makes a *transfer* while we are computing the balance
    - Typically not guaranteed for such long-running queries

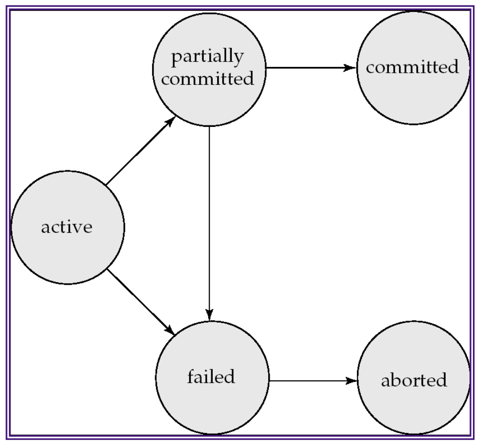
- TPC-C vs TPC-H
  - data entry vs decision support

Assumptions and Goals

- Assumptions:
  - The system can crash at any time
  - Similarly, the power can go out at any point
    - Contents of the main memory won't survive a crash, or power outage
  - BUT… **disks are durable. They might stop, but data is not lost.**
    - For now.
  - Disks only guarantee *atomic* sector *writes,* nothing more
  - Transactions are by themselves consistent
- Goals:
  - Guaranteed durability, atomicity
  - As much concurrency as possible, while not compromising isolation and/or consistency
    - Two transactions updating the same account balance… NO
    - Two transactions updating different account balances… YES

Transaction States

- active – initial state, while executing
- partially committed – after final statement
- failed – after discover that can not proceed
- aborted – after rolled back and DB restored
- committed – after successful completion



Next…

- Concurrency control schemes
  - A CC scheme is used to guarantee that concurrency does not lead to problems
  - For simplicity, we will ignore durability during this section
    - So no crashes
    - Though transactions may still abort
- Schedules
- When is concurrency okay ?
  - Serial schedules
  - Serializability

A Schedule

Transactions:

T1: transfers $50 from A to B

T2: transfers 10% of A to B

Database constraint: A + B is constant (*checking+saving accts)*

T1	T2
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)

Effect:		Before	After
	A	100	45
	B	50	105

Each transaction obeys the constraint.

The schedule does too.

Schedules

- A *schedule* is simply a (possibly interleaved) execution sequence of transaction instructions
- *Serial Schedule:* A schedule in which transactions appear one after the other
  - i.e., No interleaving
- Serial schedules satisfy isolation and consistency
  - Since each transaction by itself does not introduce inconsistency

Another serial schedule

T1	T2
	read(A) tmp = A*0.1 A = A – tmp write(A) read(B) B = B+ tmp write(B)
read(A) A = A -50 write(A) read(B) B=B+50 write(B)	

Effect:	Before	After
A	100	40
B	50	110

Consistent ?
Constraint is satisfied.

Since each Xion is consistent, any serial schedule must be consistent

Another schedule

T1	T2
read(A) A = A -50 write(A)	
	read(A) tmp = A*0.1 A = A – tmp write(A)
read(B) B=B+50 write(B)	
	read(B) B = B+ tmp write(B)

Is this schedule okay ?

Lets look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Consistent.
So this schedule is okay too.

Another schedule

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Is this schedule okay ?

Lets look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Further, the effect same as the serial schedule 1.

Called *serializable*

Example Schedules (Cont.)

A "bad" schedule

T1	T2
read(A)	
A = A -50	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
write(A)	
read(B)	
B=B+50	
write(B)	
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	50
B	50	60

Not consistent

Serializability

- A schedule is called *serializable* if:
  - *its final effect is the same as that of a serial schedule*
- Serializability → database remains consistent
  - Since serial schedules are fine
- Non-serializable schedules are unlikely to result in consistent databases
- We will ensure serializability
  - *Though typically relaxed in real high-throughput environments...*

Serializability

- Not possible to look at all *n!* serial schedules to check if the effect is the same
  - Instead ensure serializability by disallowing certain schedules
- Conflict serializability
- View serializability
  - allows more schedules

Conflict Serializability

- Two read/write instructions "conflict" if
  - They are by different transactions
  - They operate on the same data item
  - At least one is a "write" instruction
- Why do we care ?
  - If two read/write instructions don't conflict, they can be "swapped" without any change in the final effect
  - If they conflict they CAN'T be swapped

Equivalence by Swapping

T1	T2
read(A) A = A -50 write(A)	
	read(A) tmp = A*0.1 A = A – tmp write(A)
read(B) B=B+50 write(B)	
	read(B) B = B+ tmp write(B)

T1	T2
read(A) A = A -50 write(A)	
	read(A) tmp = A*0.1 A = A – tmp
read(B)	
	write(A)
B=B+50 write(B)	
	read(B) B = B+ tmp write(B)

Effect:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Equivalence by Swapping

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
	read(B)
write(B)	
	B = B+ tmp
	write(B)

Effect:	<u>Before</u>	<u>After</u>
A	100	45
B	50	105

! ==

Effect:	<u>Before</u>	<u>After</u>
A	100	45
B	50	55

Conflict Serializability

- Conflict-equivalent schedules:
  - If S can be transformed into S' through a series of swaps, S and S' are called *conflict-equivalent*
  - *conflict-equivalence guarantees same final effect on database*
- A schedule S is *conflict-serializable* if it is conflict-equivalent to a serial schedule

Equivalence by Swapping

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
read(B)	
B=B+50	
	write(A)
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Equivalence by Swapping

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

T1	T2
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Example Schedules (Cont.)

A “bad” schedule

T1	T2
read(A)	
A = A -50	
	Y
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
X	
write(A)	
read(B)	
B=B+50	
write(B)	
	B = B+ tmp
	write(B)

Can’t move Y below X
read(B) and write(B) conflict

Other options don’t work either

Not Conflict Serializable

View-Serializability

- Following not conflict-serializable

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

BUT, it is serializable
- The *conflicting write instructions don’t matter!* (in absence of reads)
- The final write is the only one that matters

- View-serializability, for S’ and S, and each datum Q:
  - if T_i reads initial value of Q in S, must also in S’
  - if T_i reads value written from T_j in S, must also in S’
  - if T_i performs final write to Q in S, must also in S’

Other notions of serializability

T_1	T_5
read(A) $A := A - 50$ write(A)	
	read(B) $B := B - 10$ write(B)
read(B) $B := B + 50$ write(B)	
	read(A) $A := A + 10$ write(A)

- Not conflict-serializable or view-serializable, but serializable
- Mainly because of the +/- only operations
  - Requires analysis of the actual operations, not just read/write operations
- Most high-performance transaction systems will allow these

Testing for conflict-serializability

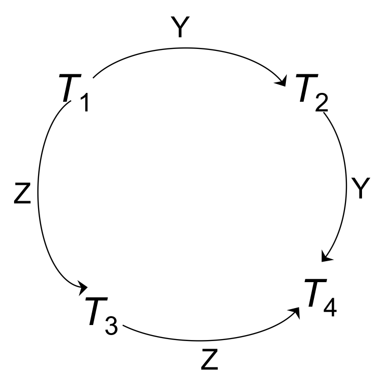
- Given a schedule, determine if it is conflict-serializable
- Draw a *precedence-graph* over the transactions
  - A directed edge from T1 to T2, if
    - they have conflicting instructions, and
    - T1's conflicting instruction comes first
- If there is a cycle in the graph, not conflict-serializable
  - Can be checked in at most $O(n+e)$ time, where n is the number of vertices, and e is the number of edges
- If there is none, conflict-serializable
- Whereas: testing for view-serializability is NP-hard.

Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Recap so far…

- We discussed:
  - Serial schedules, serializability
  - Conflict-serializability, view-serializability
  - How to check for conflict-serializability
- We haven't discussed:
  - How to guarantee serializability ?
    - Allowing transactions to run, and then aborting them if the schedules aren't serializable can be expensive
  - We can instead use schemes to guarantee that the schedule will be conflict-serializable
    - *Hint:* *locks*
  - Also, ***recoverability ?***

Recoverability

- Serializability is good for consistency
- What if transactions fail ?
  - T2 has already committed
    - A user might have been notified
  - Now T1 abort creates a problem
    - T2 has seen its effect, so just aborting T1 is not enough. T2 must be aborted as well (and possibly restarted)
    - But T2 is *committed*

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	COMMIT
read(B)	
B=B+50	
write(B)	
ABORT	

Recoverability

- *Recoverable* schedule: If T1 has read something T2 has written, T2 must commit before T1
  - Otherwise, if T1 commits, and T2 aborts, we have a problem

- Cascading rollbacks: If T10 aborts, T11 must abort, and hence T12 must abort and so on.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Recoverability

- *Dirty read*: Reading a value written by a transaction that hasn't committed yet

- *Cascadeless* schedules:
  - A transaction only reads *committed* values.
  - So if T1 has written A, but not committed it, T2 can't read it.
    - *No dirty reads*

- Cascadeless → No cascading rollbacks
  - That's good
  - We will try to guarantee that as well

Recap so far…

- We discussed:
  - Serial schedules, serializability
  - Conflict-serializability, view-serializability
  - How to check for conflict-serializability
  - Recoverability, cascade-less schedules

- We haven't discussed:
  - How to guarantee serializability ?
    - Allowing transactions to run, and then aborting them if the schedules aren't serializable can be expensive
  - We can instead use schemes to guarantee that the schedule will be conflict-serializable
    - *Hint: locks*

Concurrency Control

Approach, Assumptions etc..

- Approach
  - Guarantee conflict-serializability by limiting concurrency
    - Lock-based
- Assumptions:
  - Still ignoring durability
    - So no crashes
    - Though transactions may still abort
- Goal:
  - Serializability
  - Minimize the bad effect of aborts (cascade-less schedules only)

Lock-based Protocols

- Transactions must *acquire* locks before using data
- Two types:
  - *Shared* (S) locks (also called *read locks)*
    - Obtained if we want to only read an item
  - *Exclusive* (X) locks (also called *write locks)*
    - Obtained for updating a data item

Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

Example schedule:

T1	T2
read(B)	read(A)
B $\leftarrow$B-50	read(B)
write(B)	display(A+B)
read(A)	
A $\leftarrow$A + 50	
write(A)	

Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

Example schedule:

T1	T2
lock-X(B)	lock-S(A)
read(B)	read(A)
B $\leftarrow$B-50	unlock(A)
write(B)	lock-S(B)
unlock(B)	read(B)
	unlock(B)
lock-X(A)	display(A+B)
read(A)	
A $\leftarrow$A + 50	
write(A)	
unlock(A)	

Lock-based Protocols

- Lock requests are made to the *concurrency control manager*
  - It decides whether to *grant* a lock request
- Assume T2 holds lock, T1 asks for a lock on same:

Held lock	Lock wanted	Allow?
Shared	Shared	YES
Shared	Exclusive	NO
Exclusive	-	NO

- If *compatible,* grant the lock, otherwise T1 waits in a *queue.*

Lock instructions

Not enough to take minimum locks when you need to read/write something!

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

Example schedule:

T1		T2
lock-X(B)		lock-S(A)
read(B)		read(A)
B ← B-50		unlock(A)
write(B)		
unlock(B)	?	lock-S(B)
		read(B)
		unlock(B)
lock-X(A)		display(A+B)
read(A)		
A ← A + 50		
write(A)		
unlock(A)		

Not serializable

2-Phase Locking Protocol (2PL)

- Phase 1: Growing phase
  - Transaction may obtain locks
  - But may not release them
- Phase 2: Shrinking phase
  - Only release locks
- 2PL guarantees *conflict-serializability*
  - *lock-point*: the time at which a transaction acquired last lock
  - if *lock-point*(T1) < *lock-point*(T2), there can't be an edge from T2 to T1 in the *precedence graph*

T1

lock-X(B)
read(B)
B ←B-50
write(B)
unlock(B)

lock-X(A)
read(A)
A ←A + 50
write(A)
unlock(A)

2 Phase Locking

- Example: T1 in 2PL

	T1
Growing phase	lock-X(B)
	read(B)
	B ← B - 50
	write(B)
	lock-X(A)
	read(A)
	A ← A - 50
	write(A)
Shrinking phase	unlock(B)
	unlock(A)

2 Phase Locking

- Guarantees *conflict-serializability*, but not cascade-less recoverability

T1	T2	T3
lock-X(A), lock-S(B) read(A) read(B) write(A) unlock(A), unlock(B)		
	lock-X(A) read(A) write(A) unlock(A) commit	
		lock-S(A) read(A) commit
<xction fails>		

2 Phase Locking

- Guarantees *conflict-serializability*,
  - but *not cascade-less recoverability*
- Guaranteeing just recoverability:
  - If T2 performs a dirty read from T1 (i.e., T1 has not committed), then T2 can't commit unless T1 either commits or aborts
  - If T1 commits, T2 can proceed with committing
  - If T1 aborts, T2 must abort
    - So cascades still happen

Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort

T1	T2	T3
lock-X(A), lock-S(B)		
read(A)		
read(B)		
write(A)		
unlock(A), unlock(B)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
	Commit	lock-S(A)
		read(A)
		Commit
<xction fails>		

Strict 2PL will not allow that

Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort

T1	T2	T3
lock-X(A), lock-S(B)		
read(A)		
read(B)		
write(A)		
unlock(A), unlock(B)		
commit		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
	commit	
		lock-S(A)
		read(A)
		commit

Works. *Guarantees cascade-less and recoverable schedules.*

Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort
  - Read locks are ignored
- Rigorous 2PL: Release both *exclusive and read* locks only at the very end
  - Makes serializability order === the commit order
  - More intuitive behavior for the users
    - No difference for the system

Strict 2PL

- Lock conversion:
  - Transaction might not be sure what it needs a write lock on
  - Start with a S lock
  - *Upgrade* to an X lock later if needed
  - Doesn't change any of the other properties of the protocol

Implementation of Locking

- A separate process, or a separate module
- Uses a *lock table* to keep track of currently assigned locks and the requests for locks
  - Read up in the book

Recap so far…

- Concurrency Control Scheme
  - A way to guarantee serializability, recoverability etc
- Lock-based protocols
  - Use *locks* to prevent multiple transactions accessing the same data items
- 2 Phase Locking
  - Locks acquired during *growing phase,* released during *shrinking phase*
- Strict 2PL, Rigorous 2PL

More Locking Issues: Deadlocks

- No xction proceeds:

Deadlock
- T1 waits for T2 to unlock A
- T2 waits for T1 to unlock B

T1	T2
lock-X(B)	
read(B)	
B ← B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Rolling back transactions can be costly...

Deadlocks

- 2PL does not prevent deadlock
  - Strict doesn't either

T1	T2
lock-X(B)	
read(B)	
B ← B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Rolling back transactions can be costly...

Preventing deadlocks

- Graph-based protocols
  - Acquire locks only in a well-known order

bad

T1	T2
lock-X(B)	
read(B)	
B $\leftarrow$ B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

good

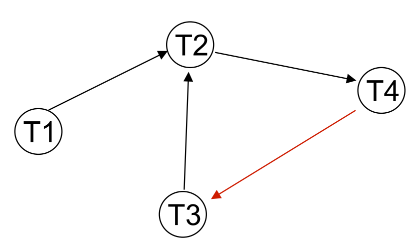
T1	T2
lock-X(A)	
lock-X(B)	
read(B)	
B $\leftarrow$ B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)

- Might not know locks in advance

Detecting existing deadlocks

- Timeouts (local information)
- *waits-for graph* (global information):
  - edge $T_i \rightarrow T_j$ when T_i waiting for T_j

T1	T2	T3	T4
		X(Z)	
	X(V)		
			X(W)
S(V)			
	S(W)		
		S(V)	



Suppose T4 requests lock-S(Z)....

Dealing with Deadlocks

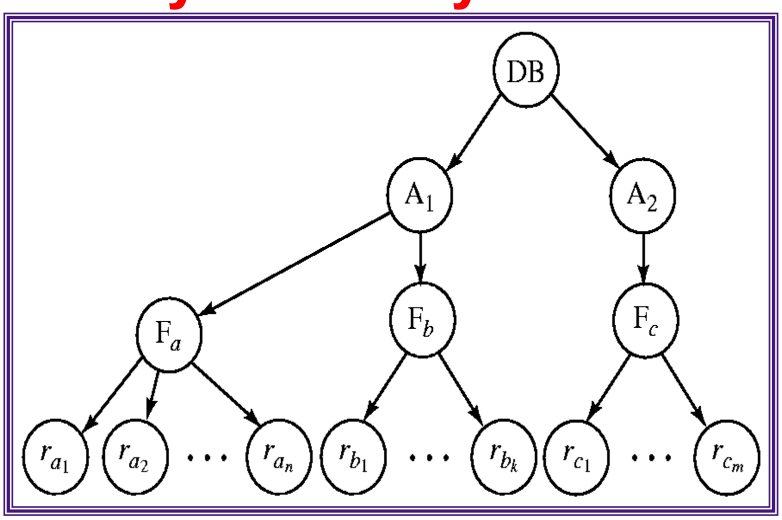
- Deadlock detected, now what ?
  - Will need to abort some transaction
- Victim selection
  - Use time-stamps; say T1 is older than T2
  - *wait-die scheme:* T1 will wait for T2. T2 will not wait for T1; instead it will abort and restart
  - *wound-wait scheme:* T1 will *wound* T2 (force it to abort) if it needs a lock that T2 currently has; T2 will wait for T1.
- Issues
  - Prefer to prefer transactions with the most work done
  - Possibility of starvation
    - If a transaction is aborted too many times, it may be given priority in continuing

Locking granularity

Locking granularity (not always done)

- Locking granularity
  - What are we taking locks on ? Tables, tuples, attributes ?
- Coarse granularity
  - e.g. take locks on tables
  - less overhead (the number of tables is not that high)
  - very low concurrency
- Fine granularity
  - e.g. take locks on tuples
  - much higher overhead
  - much higher concurrency
  - What if I want to lock 90% of the tuples of a table ?
    - Prefer to lock the whole table in that case

Granularity Hierarchy



The highest level in the example hierarchy is the entire database.

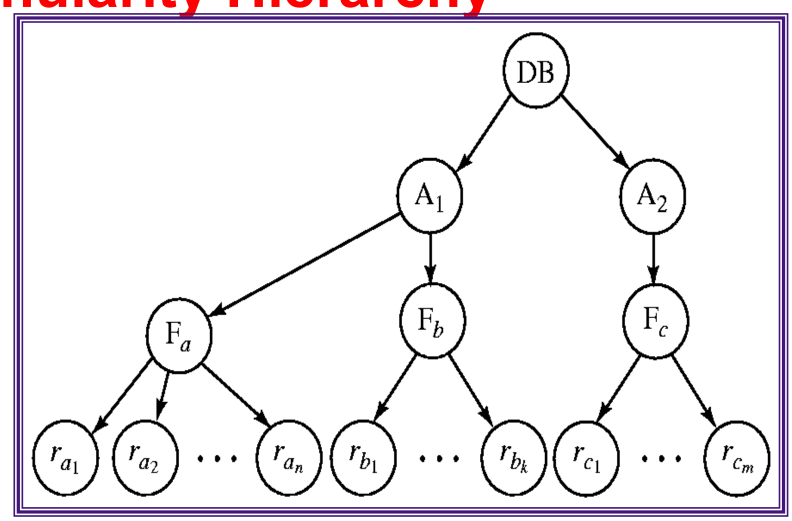
The levels below are of type *area*, *file or relation* and *record* in that order.

Can lock at any level in the hierarchy

Granularity Hierarchy

- New lock mode, called *intentional* locks
  - Declare an intention to lock parts of the subtree below a node
  - IS: *intention shared*
    - The lower levels below may be locked in the shared mode
  - IX: *intention exclusive*
  - *SIX: shared and intention-exclusive*
    - The entire subtree is locked in the shared mode, but I might also want to get exclusive locks on the nodes below
- Protocol:
  - If you want to acquire a lock on a data item, all the ancestors must be locked as well, at least in the intentional mode
  - So you always start at the top *root* node

Granularity Hierarchy



(1) Want to lock *F_a* in shared mode, *DB* and *A1* must be locked in at least IS mode (but IX, SIX, S, X are okay too)

(2) Want to lock *rc1* in exclusive mode, *DB, A2,Fc* must be locked in at least IX mode (SIX, X are okay too)

Compatibility Matrix with Intention Lock Modes

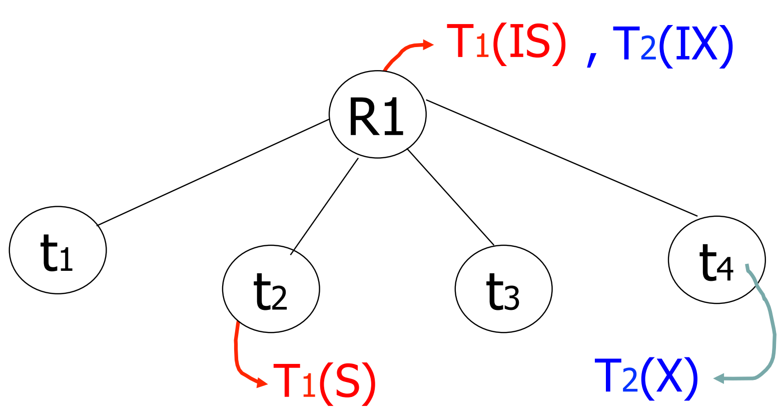
- Locks from different transactions:

requestor

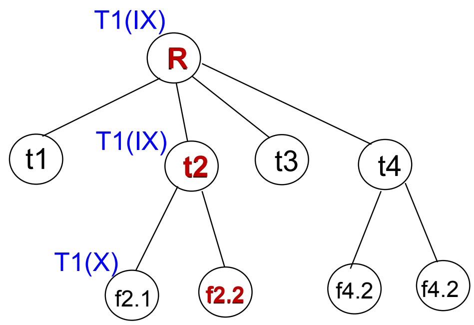
holder

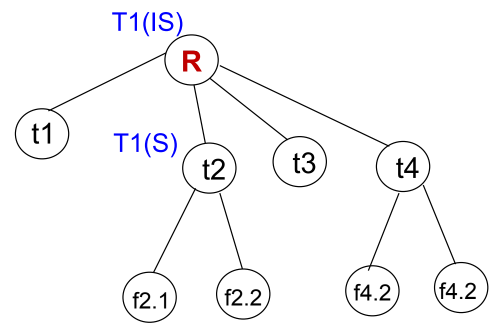
	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

Example

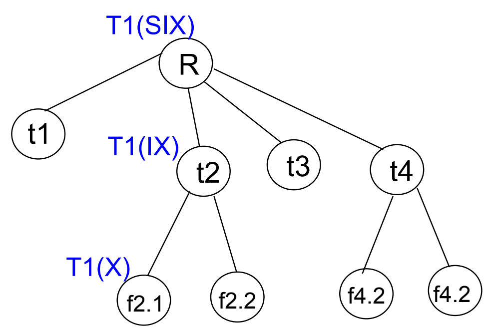


Examples





Can T2 access object f2.2 in X mode?
What locks will T2 get?



Other CC Schemes

- Time-stamp based
  - Transactions are issued time-stamps when they enter the system
  - The time-stamps determine the *serializability* order
  - So if T1 entered before T2, then T1 should be before T2 in the serializability order
  - Say *timestamp(T1) < timestamp(T2)*
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read that data item or written it, then the write is *rejected* and T1 is aborted
  - Aborted transaction are restarted with a new timestamp
    - Possibility of *starvation*

Other CC Schemes

- Time-stamp based
  - Example

T_1	T_2	T_3	T_4	T_5
				read(X)
	read(Y)			
read(Y)				
		write(Y)		
		write(Z)		
				read(Z)
	read(X)			
	abort			
read(X)				
		write(Z)		
		abort		
				write(Y)
				write(Z)

ACID, cont

Other CC Schemes

- Time-stamp based
  - As discussed here, has too many problems
    - Starvation
    - Non-recoverable
    - Cascading rollbacks required
  - Most can be solved fairly easily
    - Read up
  - Remember: We can always put more and more restrictions on what the transactions can do to ensure these things
    - The goal is to find the minimal set of restrictions to as to not hinder concurrency

Other CC Schemes

- Optimistic concurrency control
  - Also called validation-based
  - Intuition
    - Let the transactions execute as they wish
    - At the very end when they are about to commit, check if there might be any problems/conflicts etc
      - If no, let it commit
      - If yes, abort and restart
  - Optimistic: The hope is that there won't be too many problems/aborts

Isolation Levels: Snapshot Isolation

- Very popular scheme, used as the primary scheme by many systems including Oracle, PostgreSQL etc…
  - Several others support this in addition to locking-based protocol
- A type of "optimistic concurrency control"
- Key idea:
  - For each object, maintain past "versions" of the data along with timestamps
    - Every update to an object causes a new version to be generated

Isolation Levels: Snapshot Isolation

- Read queries:
  - Let "t" be the "time-stamp" of the query, i.e., the time at which it entered the system
  - When the query asks for a data item, provide a version of the data item that was latest as of "t"
    - Even if the data changed in between, provide an old version
  - No locks needed, no waiting for any other transactions or queries
  - The query executes on a consistent snapshot of the database
- Update queries (transactions):
  - Reads processed as above on a snapshot
  - Writes are done in private storage
  - At commit time, for each object that was written, check if some other transaction updated the data item since this transaction started
    - If yes, then abort and restart
    - If no, make all the writes public simultaneously (by making new versions)

Isolation Levels: Snapshot Isolation

- Advantages:
  - Read query don't block at all, and runs very fast
  - As long as conflicts are rare, update transactions don't abort either
  - Overall better performance than locking-based protocols
- Major disadvantage:
  - Not serializable
  - Inconsistencies may be introduced
  - See the wikipedia article for more details and an example
    - http://en.wikipedia.org/wiki/Snapshot_isolation

The "Phantom" problem

- An interesting problem that comes up for dynamic databases
- Schema: *accounts(acct_no, balance, zipcode, …)*
- Transaction 1: Find the number of accounts in *zipcode = 20742,* and divide $1,000,000 between them
- Transaction 2: Insert *<acctX, …, 20742, …>*
- Execution sequence:
  - T1 locks all tuples corresponding to "zipcode = 20742", finds the total number of accounts (= num_accounts)
  - T2 does the insert
  - T1 computes 1,000,000/num_accounts
  - When T1 accesses the relation again to update the balances, it finds one new ("phantom") tuple (the new tuple that T2 inserted)
- Not serializable

Time-stamp based CC

- Transactions are issued time-stamps
  - When they enter the system
  - Time-stamps determine the *serializability* order
  - If T1 entered before T2,
    Then T1 before T2 in the serializability order
- Say *timestamp(T1) < timestamp(T2)*
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read or written that data item, then the write is *rejected* and T1 is aborted
  - Aborted transaction are restarted with a new timestamp
    - Possibility of *starvation*

Time-stamp based CC

$TS(T_1) < TS(T_2) < TS(T_3) < TS(T_4) < TS(T_5)$

- Example

T_1	T_2	T_3	T_4	T_5
				write(X)
	read(Y)			
read(Y)				
		write(Y)		
		write(Z)		
				read(Z)
	read(X)			
read(X)				
		write(Z)		
				write(Y)
				write(Z)

Time-stamp based CC

$TS(T_1) < TS(T_2) < TS(T_3) < TS(T_4) < TS(T_5)$

- Example

T_1	T_2	T_3	T_4	T_5
				write(X)
	read(Y)			
read(Y)				
		write(Y) write(Z)		
				read(Z)
	read(X) ***abort***			
read(X) ***abort***				
		write(Z) ***abort***		
				write(Y) write(Z)

Time-stamp based CC

- The following set of instructions is not conflict-serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- As discussed before, not even *view-serializabile:*
  - if T_i reads initial value of Q in S, must also in S'
  - if T_i reads value written from T_j in S, must also in S'
  - if T_i performs final write to Q in S, must also in S'

Time-stamp based CC

- Thomas' Write Rule
  - *Ignore obsolete writes*

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

ignored

- Say *timestamp(T1) < timestamp(T2)*
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read ~~or written~~ that data item, then the write is *rejected* and T1 is aborted
    - *If a transaction with larger time-stamp already written that data item, then the write is ignored*

Other CC Schemes

- Time-stamp based
  - Many potential problems
    - Starvation
    - Non-recoverable
    - Cascading rollbacks required
  - Most can be solved fairly easily
    - Read up
  - Remember: We can always put more and more restrictions on what the transactions can do to ensure these things
    - The goal is to *find the minimal set of restrictions to as to not hinder concurrency*

Other CC Schemes

- Optimistic concurrency control
  - Also called validation-based
  - Intuition
    - Let the transactions execute as they wish
    - At the very end when they are about to commit, check if there might be any problems/conflicts etc
      - If no, let it commit
      - If yes, abort and restart
  - Optimistic: The hope is that there won't be too many problems/ aborts

Recovery

Context

- ACID properties:
  - We have talked about Isolation and Consistency
  - How do we guarantee Atomicity and Durability ?
    - Atomicity: Two problems
      - Part of the transaction is done, but we want to cancel it
        - ABORT/ROLLBACK
      - System crashes during the transaction. Some changes made it to the disk, some didn't.
    - Durability:
      - Transaction finished. User notified. But changes not sent to disk yet (for performance reasons). System crashed.
- Essentially similar solutions

Reasons for crashes

- Transaction failures
  - Logical errors, deadlocks
- System crash
  - Power failures, operating system bugs etc
- Disk failure
  - Head crashes; ***for now we will assume***
    - ***STABLE STORAGE:* Data *never* lost. Can approximate by using RAID and maintaining geographically distant copies of the data**

Approach, Assumptions etc..

- Approach:
  - Guarantee A and D:
    - by controlling how the disk and memory interact,
    - by storing enough information during normal processing to recover from failures
    - by developing algorithms to recover the database state
- Assumptions:
  - System may crash, but the *disk is durable*
  - The only *atomicity* guarantee is that *a disk block write* is *atomic*
- Obvious naïve solutions exist that work, but are too expensive.
  - E.g. A *shadow copy* solution
    - Make a copy of the database; do the changes on the copy; do an atomic switch of the *dbpointer* at commit time
  - Goal is to do this as efficiently as possible

Buffer Management

- Buffer manager
  - sits between DB and disk
  - writing every operation to disk, as it occurs, too slow…
  - ideally only write a block to disk at commit
    - aggregates updates
    - trans might not commit
- Bottom line
  - want to *decouple* data writes from DB operations

STEAL vs NO STEAL, FORCE vs NO FORCE

- STEAL:
  - The buffer manager *can steal* a (memory) page from the database
    - ie., it can write an arbitrary page to the disk and use that page for something else from the disk
    - In other words, the database system doesn't control the buffer replacement policy
  - Why a problem ?
    - The page might contain *dirty writes,* ie., writes/updates by a transaction that hasn't committed
  - But, we must allow *steal* for performance reasons.
- NO STEAL:
  - Stealing not allowed. More control, but less flexibility for the buffer manager ➔ poor performance.

Uncommitted changes might be on disk after crash…

STEAL vs NO STEAL, FORCE vs NO FORCE

- FORCE:
  - The database system *forces* all the updates of a transaction to disk before committing
  - Why ?
    - To make its updates permanent before committing
  - Why a problem ?
    - Most probably random I/Os, so poor response time and throughput
    - Interferes with the disk controlling policies
- NO FORCE:
  - Don't do the above. Desired.
  - Problem:
    - Guaranteeing durability becomes hard
  - We might still have to *force* some pages to disk, but minimal.

Committed changes might NOT be on disk after crash…

STEAL vs NO STEAL, FORCE vs NO FORCE

	No Steal	Steal
No Force		Desired
Force	Trivial	

What if NO STEAL, FORCE ?

- Only updates from committed transaction are written to disk (since no steal)
- Updates from a transaction are forced to disk before commit (since force)
  - A minor problem: how do you guarantee that all updates from a transaction make it to the disk atomically ?
    - Remember we are only guaranteed an atomic *block write*
    - What if some updates make it to disk, and other don't ?
  - Can use something like shadow copying/shadow paging
- No atomicity/durability problems.

What if STEAL, NO FORCE ?

- After crash:
  - Disk might have DB data from uncommitted transactions
  - Disk might not have DB data from committed transactions
- How to recover?

  "Log-based recovery"

Log-based Recovery

- Most commonly used recovery method
- A log is a record of everything the database system does
- For every operation done by the database, a *log record* is generated and stored *typically on a different (log) disk*
  - *<T1, START>*
  - *<T2, COMMIT>*
  - *<T2, ABORT>*
  - *<T1, A, 100, 200>*
    - T1 modified A; old value = 100, new value = 200

Log

- **Example transactions** T_0 **and** T_1 **(**T_0 **executes before** T_1**):**

T_0: **read** (*A*)
A: - A - 50
write (*A*)
read (*B*)
B:- B + 50
write (*B*)

T_1 : **read** (*C*)
C:- C- 100
write (*C*)

- Log:

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

Log-based Recovery

- Assumptions:
  1. Log records are ***immediately pushed to the disk*** as soon as they are generated
  2. Log records are written to disk in the order generated
  3. A log record is generated *before* the actual data value is updated
  4. *Strict two-phase locking*
     - The first assumption can be relaxed
     - As a special case, a *transaction is considered committed only after <T1, COMMIT> has been pushed to the disk*
- Also:
  - Log writes are *sequential*
  - They are also typically on a different disk
  - LFS == log-structured file system, and basis of *journaling* file systems

Recovery

STEAL is allowed, so changes of a transaction may have made it to the disk

- UNDO(T1):
  - Procedure executed to *rollback/undo* the effects of a transaction
  - E.g.
    - *<T1, START>*
    - *<T1, A, 200, 300>*
    - *<T1, B, 400, 300>*
    - *<T1, A, 300, 200>* *[[note: second update of A]]*
    - T1 decides to abort
  - Any of the changes might have made it to the disk

Using the log to *abort/rollback*

- UNDO(T1):
  - Go *backwards* in the *log* looking for log records belonging to T1
  - Restore the values to the old values
  - NOTE: Going backwards is important.
    - *A* was updated twice
  - In the example, we simply:
    - Restore A to 300
    - Restore B to 400
    - Restore A to 200
  - Note: No other transaction could have changed A or B in the meantime
    - *Strict two-phase locking*

Using the log to *recover*

- We don't require FORCE, so a change made by the committed transaction may not have made it to the disk before the system crashed
  - BUT, the log record did (recall our assumptions)
- REDO(T1):
  - Procedure executed to recover a committed transaction
  - E.g.
    - *<T1, START>*
    - *<T1, A, 200, 300>*
    - *<T1, B, 400, 300>*
    - *<T1, A, 300, 200>*   *[[note: second update of A]]*
    - *<T1, COMMIT>*
  - By our assumptions, all the log records made it to the disk (since the transaction committed)
  - But any or none of the changes to A or B might have made it to disk

Using the log to *recover*

- REDO(T1):
  - Go *forwards* in the *log* looking for log records belonging to T1
  - Set the values to the new values
  - NOTE: Going forwards is important.
  - In the example, we simply:
    - Set A to 300
    - Set B to 300
    - Set A to 200

Idempotency

- Both redo and undo are required to *idempotent*
  - *F is idempotent, if F(x) = F(F(x)) = F(F(F(F(…F(x)))))*
- Multiple applications shouldn't change the effect
  - This is important because we don't know exactly what made it to the disk, and we can't keep track of that
  - E.g. consider a log record of the type
    - <T1, A, *incremented by 100*>
    - Old value was 200, and so new value was 300
  - But the on disk value might be 200 or 300 (since we have no control over the buffer manager)
  - So we have no idea whether to apply this log record or not
  - Hence, ***value based logging*** is used (also called *physical)*, not operation based (also called *logical)*

Log-based recovery

- Log is maintained
- If during the normal processing, a transaction needs to abort
  - UNDO() is used for that purpose
- If the system crashes, then we need to do recovery using both UNDO() and REDO()
  - Some transactions that were going on at the time of crash may not have completed, and must be *aborted/undone*
  - Some transactions may have committed, but their changes didn't make it to disk, so they must be *redone*
  - Called *restart recovery*

Restart Recovery (after a crash)

- After restart, go backwards into the log, and make two lists
  - How far ?? For now, assume till the beginning of the log.
- undo_list: A list of transactions that must be undone
  - *<Ti, START>* record is in the log, but no *<Ti, COMMIT>*
- redo_list: A list of transactions that need to be redone
  - Both *<Ti, START> and <Ti, COMMIT>* records are in the log
- After that:
  - UNDO all the transactions on the undo_list one by one
  - REDO all the transaction on the redo_list one by one
  - *this is different than the recovery algorithm in 16.4*

Restart Recovery (after a crash)

- Must do the UNDOs first before REDO
  - *<T2, A, 10, 30>*
  - *<T1, A, 10, 20>*
  - *<T1, abort>* *[[so A was restored back to 10]]*
  - *<T2, commit>*
- If we do UNDO(T1) first, and then REDO(T2), it will be okay
- Trying to do other way around doesn't work

Checkpointing

- How far should we go back in the log while constructing redo and undo lists ??
  - It is possible that a transaction made an update at the very beginning of the system, and that update never made it to disk
    - very very unlikely, but possible (because we don't do force)
  - For correctness, we have to go back all the way to the beginning of the log
  - Bad idea !!

- Checkpointing is a mechanism to reduce this

Checkpointing

- Periodically, the database system writes out everything in the memory to disk
  - Goal is to get the database in a state that we know (not necessarily consistent state)
- Steps:
  - Stop all other activity in the database system
  - Write out the entire contents of the memory to the disk
    - Only need to write updated pages, so not so bad
    - Entire === all updates, whether committed or not
  - Write out all the log records to the disk
  - Write out a special log record to disk
    - *<CHECKPOINT LIST_OF_ACTIVE_TRANSACTIONS>*
    - The second component is the list of all active transactions in the system right now
  - Continue with the transactions again

Restart Recovery w/ checkpoints

- Key difference: Only need to go back till the last checkpoint
- Steps:
  - undo_list:
    - Go back till the checkpoint as before.
    - Add all the transactions that were active at that time, and that didn't commit
      - e.g. possible that a transactions started before the checkpoint, but didn't finish till the crash
  - redo_list:
    - Similarly, go back till the checkpoint constructing the redo_list
    - Add all the transactions that were active at that time, and that did commit
  - Do UNDOs and REDOs as before

Recap so far …

- Log-based recovery
  - Uses a *log* to aid during recovery
- UNDO()
  - Used for normal transaction abort/rollback, as well as during restart recovery
- REDO()
  - Used during restart recovery
- Checkpoints
  - Used to reduce the restart recovery time

Other issues

- ARIES: Considered *the canonical description of log-based recovery*
  - Used in most systems
  - Has many other types of log records that simplify recovery significantly
- Loss of disk:
  - Can use a scheme similar to checkpointing to periodically dump the database onto *tapes* or *optical storage*
  - Techniques exist for doing this while the transactions are executing (called *fuzzy dumps)*
- Shadow paging:
  - Read up

Recap

- STEAL vs NO STEAL, FORCE vs NO FORCE
  - We studied how to do STEAL and NO FORCE through log-based recovery scheme

	No Steal	Steal
No Force		**Desired**
Force	**Trivial**	

	No Steal	Steal
No Force	**REDO**	**REDO UNDO**
Force		**UNDO**

Write-ahead logging

- We assumed that log records are written to disk as soon as generated
  - Too restrictive
- Write-ahead logging:
  - Before an update on a data item (say A) makes it to disk, the log records referring to the update must be forced to disk
  - How ?
    - Each log record has a log sequence number (LSN)
      - Monotonically increasing
    - For each page in the memory, we maintain the LSN of the *last log record* that updated a record on this page
      - *pageLSN*
    - If a page *P* is to be written to disk, all the log records till *pageLSN(P)* are forced to disk

Write-ahead logging

- Write-ahead logging (WAL) is sufficient for all our purposes
  - All the algorithms discussed before work
- Note the special case:
  - A transaction is not considered committed, unless the <T, commit> record is on disk

Other issues

- The system halts during checkpointing
  - Not acceptable
  - Advanced recovery techniques allow the system to continue processing while checkpointing is going on
- System may crash during recovery
  - Our simple protocol is actually fine
  - In general, this can be painful to handle
- B+-Tree and other indexing techniques
  - Strict 2PL is typically not followed (we didn't cover this)
  - So physical logging is not sufficient; must have logical logging
    - Read 16.7 if interested.

Recap

- ACID Properties
  - Atomicity and Durability :
    - Logs, undo(), redo(), WAL etc
  - Consistency and Isolation:
    - Concurrency schemes
  - Strong interactions:
    - We had to assume Strict 2PL for proving correctness of recovery