Transactions Intro & Concurrency Control

R & G Chaps. 16/17

There are three side effects of acid.
Enhanced long term memory,
decreased short term memory,
and I forget the third.
- Timothy Leary

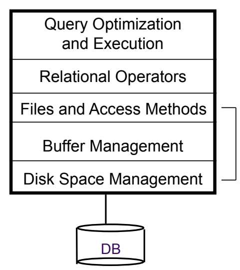
Concurrency Control & Recovery

- **Concurrency Control**
  - Provide correct and highly available data access in the presence of concurrent access by many users
- **Recovery**
  - Ensures database is fault tolerant, and not corrupted by software, system or media failure
  - 24x7 access to mission critical data
- **A boon to application authors!**
  - Existence of CC&R allows applications to be written without explicit concern for concurrency and fault tolerance

Roadmap

- **Overview (Today)**
- **Concurrency Control (Today/Thurs)**
- **Recovery (Next Week)**

Structure of a DBMS



These layers must consider concurrency control and recovery (Transaction, Lock, Recovery Managers)

Transactions and Concurrent Execution

- **Transaction ("xact"): DBMS's abstract view of a user program (or activity)**
  - A sequence of reads and writes of database objects.
  - Batch of work that must commit or abort as an atomic unit
- **Transaction Manager controls execution of transactions.**
- **User's program logic is invisible to DBMS!**
  - Arbitrary computation possible on data fetched from the DB
  - The DBMS only sees data read/written from/to the DB.
- **Challenge: provide atomic xacts to concurrent users!**
  - Given only the read/write interface.

Concurrency: Why bother?

- **The *latency* argument**
- **The *throughput* argument**
- **Both are critical!**

ACID properties of Transaction Executions

- **A tomicity:** All actions in the Xact happen, or none happen.
- **C onsistency:** If the DB starts consistent, it ends up consistent at end of Xact.
- **I solation:** Execution of one Xact is isolated from that of other Xacts.
- **D urability:** If a Xact commits, its effects persist.

Atomicity and Durability

A.C.I.D.

- **A transaction ends in one of two ways:**
  - *commit* after completing all its actions
    - "commit" is a contract with the caller of the DB
  - *abort* (or be aborted by the DBMS) after executing some actions.
    - Or *system crash* while the xact is in progress; treat as abort.
- **Two important properties for a transaction:**
  - *Atomicity* : Either execute all its actions, or none of them
  - *Durability* : The effects of a committed xact must survive failures.
- **DBMS ensures the above by *logging* all actions:**
  - *Undo* the actions of aborted/failed transactions.
  - *Redo* actions of committed transactions not yet propagated to disk when system crashes.

Transaction Consistency

A.C.I.D.

- **Transactions preserve DB *consistency***
  - Given a consistent DB state, produce another consistent DB state
- **DB Consistency expressed as a set of declarative Integrity Constraints**
  - CREATE TABLE/ASSERTION statements
- **Transactions that violate ICs are aborted**
  - That's all the DBMS can automatically check!

Isolation (Concurrency)

A.C.I.D.

- **DBMS interleaves actions of many xacts**
  - Actions = reads/writes of DB objects
- **DBMS ensures xacts do not "step on" each other.**
- **Each xact executes <u>as if</u> it were running by itself.**
  - Concurrent accesses have no effect on a Transaction's behavior
  - Net effect *must be* identical to executing all transactions for *some* serial order.
  - Users & programmers think about transactions in isolation
    - Without considering effects of other concurrent transactions!

Today: Focus on Isolation

- **Serial schedules safe but slow**
- **Try to find schedules *equivalent* to serial …**

Conflicting Operations

- **Need a tool to decide if 2 schedules are equivalent**
- **Use notion of "conflicting" operations**
- **<u>Definition</u>: Two operations conflict if:**
  - They are by different transactions,
  - they are on the same object,
  - and at least one of them is a write.

Conflict Serializable Schedules

- **Definition: Two schedules are conflict equivalent iff:**
  - They involve the same actions of the same transactions, and
  - every pair of conflicting actions is ordered the same way
- **Definition: Schedule S is conflict serializable if:**
  - S is conflict equivalent to some serial schedule.
- **Note, some "serializable" schedules are NOT conflict serializable**
  - A price we pay to achieve efficient enforcement.

Conflict Serializability – Intuition

- **A schedule S is conflict serializable if:**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions.
- ***Example:***

R(A) W(A)		R(B) W(B)	
	R(A) W(A)		R(B) W(B)

≡

R(A) W(A) R(B) W(B)	
	R(A) W(A) R(B) W(B)

Conflict Serializability (Continued)

- **Here's another example:**

R(A)		W(A)
	R(A) W(A)	

- **Serializable or not????**

NOT!

Dependency Graph

$$T_i \longrightarrow T_j$$

- ***Dependency graph*:**
  - One node per Xact
  - Edge from Ti to Tj if:
    - An operation Oi of Ti conflicts with an operation Oj of Tj and
    - Oi appears earlier in the schedule than Oj.
- **Theorem: Schedule is conflict serializable *if and only if* its dependency graph is acyclic.**

Example

- **A schedule that is not conflict serializable:**

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

$$T1 \underset{B}{\overset{A}{\rightleftarrows}} T2$$

Dependency graph

- **The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.**

An Aside: View Serializability

- **Alternative (weaker) notion of serializability.**
- **Schedules S1 and S2 are view equivalent if:**
  1. If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2 (*same initial reads*)
  2. If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2 (*same dependent reads*)
  3. If Ti writes final value of A in S1, then Ti also writes final value of A in S2 (*same winning writes*)
- **Basically, allows all conflict serializable schedules + "blind writes"**

T1:	R(A)	W(A)	
T2:	W(A)		
T3:			W(A)

view ≡

T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)

Notes on Serializability Definitions

- **View Serializability allows (slightly) more schedules than Conflict Serializability does.**
  - But V.S. is difficult to enforce efficiently.
- **Neither definition allows all schedules that you would consider "serializable".**
  - Because they don't understand the meanings of the operations or the data.
- **In practice, Conflict Serializability is what gets used, because it can be enforced efficiently.**
  - To allow more concurrency, some special cases do get handled separately, such as for travel reservations, etc.

Two-Phase Locking (2PL)

- **The most common scheme for enforcing conflict serializability**
- **"Pessimistic"**
  - Sets locks for fear of conflict
  - The alternative scheme is called Optimistic Concurrency Control
    - Not today

Two-Phase Locking (2PL)

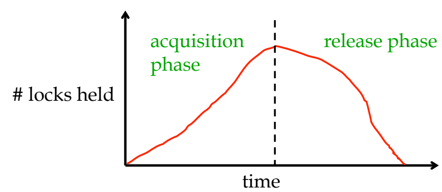
Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

rules:

- Xact must obtain a **S** (*shared*) lock before reading, and an **X** (*exclusive*) lock before writing.
- Xact cannot get new locks after releasing any locks.

Two-Phase Locking (2PL), cont.



2PL guarantees conflict serializability

But, does *not* prevent **Cascading Aborts**.

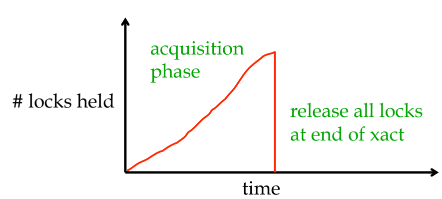
Strict 2PL

- ***Problem:*** **Cascading Aborts**
- ***Example:*** **rollback of T1 requires rollback of T2!**

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A)	

- **Strict Two-phase Locking (Strict 2PL) protocol:**
  Same as 2PL, except:
  Locks released only when transaction completes
  i.e., either:
  (a) transaction has committed (commit record on disk), or
  (b) transaction has aborted and rollback is complete.

Strict 2PL (continued)



Next …

- **A few examples**

Non-2PL, A= 1000, B=2000, Output =?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Unlock(A)	
	Read(A)
	Unlock(A)
	Lock_S(B)
Lock_X(B)	
	Read(B)
	Unlock(B)
	PRINT(A+B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

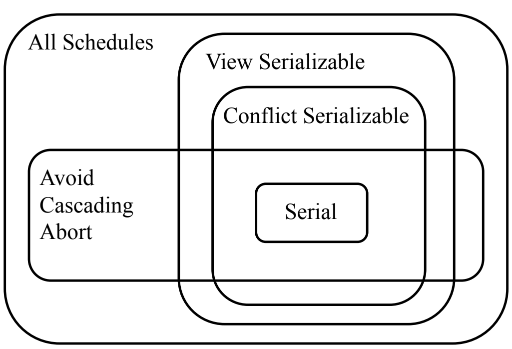
2PL, A= 1000, B=2000, Output =?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Unlock(A)	
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

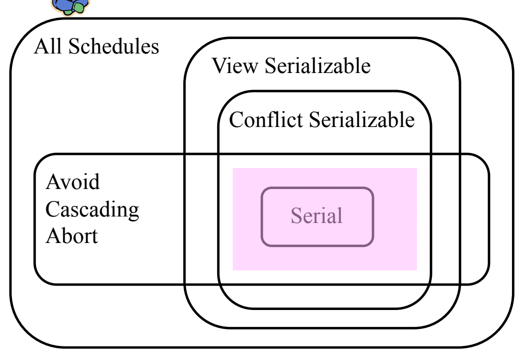
Strict 2PL, A= 1000, B=2000, Output =?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	
	Read(A)
	Lock_S(B)
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)

Venn Diagram for Schedules



Which schedules does Strict 2PL allow?



Lock Management

- **Lock and unlock requests handled by Lock Manager**
- **LM keeps an entry for each currently held lock.**
- **Entry contains:**
  - List of xacts currently holding lock
  - Type of lock held (shared or exclusive)
  - Queue of lock requests

Lock Management, cont.

- **When lock request arrives:**
  - Does any other xact hold a conflicting lock?
    - If no, grant the lock.
    - If yes, put requestor into wait queue.
- **Lock upgrade:**
  - xact with shared lock can request to upgrade to exclusive

Example

Lock_X(A)	
	Lock_S(B)
	Read(B)
	Lock_S(A)
Read(A)	
A: = A-50	
Write(A)	
Lock_X(B)	

Deadlocks

- **Deadlock: Cycle of transactions waiting for locks to be released by each other.**
- **Two ways of dealing with deadlocks:**
  - prevention
  - avoidance
  - detection
- **Many systems just punt and use Timeouts**
  - What are the dangers with this approach?

Deadlock Prevention

- **Common technique in operating systems**
- **Standard approach: resource ordering**
  - Screen < Network Card < Printer
- **Why is this problematic for Xacts in a DBMS?**

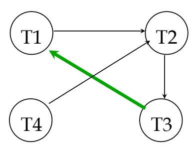
Deadlock Detection

- **Create and maintain a "waits-for" graph**
- **Periodically check for cycles in graph**

Deadlock Detection (Continued)

Example:

T1: S(A), S(D), S(B)
T2: X(B) X(C)
T3: S(D), S(C), X(A)
T4: X(B)



Deadlock Avoidance

- **Assign priorities based on timestamps.**
- **Say Ti wants a lock that Tj holds**

  Two policies are possible:

  Wait-Die: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts

  Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits

- **Why do these schemes guarantee no deadlocks?**
- **Important detail: If a transaction re-starts, make sure it gets its original timestamp. -- Why?**

Locking Granularity

- **Hard to decide what granularity to lock (tuples vs. pages vs. tables).**
- ***why?***

Multiple-Granularity Locks

- **Shouldn't have to make same decision for all transactions!**
- **Data "containers" are nested:**

contains

Database
Tables
Pages
Tuples

Solution: New Lock Modes, Protocol

Database
Tables
Pages
Tuples

- **Allow Xacts to lock at each level, but with a special protocol using new "intent" locks:**
- **Still need S and X locks, but before locking an item, Xact must have proper intent locks on all its ancestors in the granularity hierarchy.**

  - IS – Intent to get S lock(s) at finer granularity.
  - IX – Intent to get X lock(s) at finer granularity.
  - SIX mode: Like S & IX at the same time. Why useful?

Multiple Granularity Lock Protocol

Database
Tables
Pages
Tuples

- **Each Xact starts from the root of the hierarchy.**
- **To get S or IS lock on a node, must hold IS or IX on parent node.**
  - What if Xact holds S on parent? SIX on parent?
- **To get X or IX or SIX on a node, must hold IX or SIX on parent node.**
- **Must release locks in bottom-up order.**

Protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.

Lock Compatibility Matrix

	IS	IX	SIX	S	X
IS	√	√	√	√	-
IX	√	√	-	-	-
SIX	√	-	-	-	-
S	√	-	-	√	-
X	-	-	-	-	-

Database
Tables
Pages
Tuples

- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time.

Examples – 2 level hierarchy

Tables
Tuples

- **T1 scans R, and updates a few tuples:**
  - T1 gets an SIX lock on R, then get X lock on tuples that are updated.
- **T2 uses an index to read only part of R:**
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- **T3 reads all of R:**
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can use lock escalation to decide which.
  - Lock escalation dynamically asks for coarser-grained locks when too many low level locks acquired

	IS	IX	SIX	S	X
IS	√	√	√	√	
IX	√	√			
SIX	√				
S	√			√	
X					

Just so you're aware: Optimistic CC

- **Basic idea: let all transactions run to completion**
  - Make tentative updates on private copies of data
  - At commit time, check schedules for serializability
  - If you can't guarantee it, restart transaction else "install" updates in DBMS
- **Pros & Cons**
  - No waiting or lock overhead in serializable cases
  - Restarted transactions waste work, slow down others
- **OCC a loser to 2PL in traditional DBMSs**
  - Plays a secondary role in some DBMSs
- **Generalizations:**
  - Multi-version and Timestamp CC manage the multiple copies in a permanent way

Just So You're Aware: Indexes

- **2PL on B+-tree pages is a rotten idea.**
  - Why?
- **Instead, do short locks (latches) in a clever way**
  - Idea: Upper levels of B+-tree just need to direct traffic correctly. Don't need to be serializably handled!
  - Different tricks to exploit this
- **Note: this is pretty complicated!**

Just So You're Aware: Phantoms

- **Suppose you query for sailors with rating between 10 and 20, using a B+-tree**
  - Tuple-level locks in the Heap File
- **I insert a Sailor with rating 12**
- **You do your query again**
  - Yikes! A phantom!
  - Problem: Serializability assumed a static DB!
- **What we want: lock the *logical* range 10-20**
  - Imagine that lock table!
- **What is done: set locks in indexes cleverly**

Summary

- **Correctness criterion for isolation is "serializability".**
  - In practice, we use "conflict serializability," which is somewhat more restrictive but easy to enforce.
- **Two Phase Locking and Strict 2PL: Locks implement the notions of conflict directly.**
  - The lock manager keeps track of the locks issued.
  - **Deadlocks** may arise; can either be prevented or detected.
- **Multi-Granularity Locking:**
  - Allows flexible tradeoff between lock "scope" in DB, and locking overhead in RAM and CPU
- **More to the story**
  - Optimistic/Multi-version/Timestamp CC
  - Index "latching", phantoms