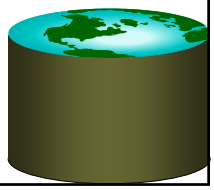
Transaction Overview and Concurrency Control

CS 186, Spring 2006,
Lectures 23-25
R & G Chapters 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

- **Very valuable properties of DBMSs**
  - without these, DBMSs would be much less useful
- **Based on concept of transactions with ACID properties**
- **Remainder of the lectures discuss these issues**

Concurrent Execution & Transactions

- **Concurrent execution essential for good performance.**
  - Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently.
- **A program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.**
- ***transaction*** **- DBMS's abstract view of a user program:**
  - a sequence of reads and writes.

Goal: The ACID properties

- **A tomicity: All actions in the transaction happen, or none happen.**
- **C onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.**
- **I solation: Execution of one transaction is isolated from that of all others.**
- **D urability: If a transaction commits, its effects persist.**

Atomicity of Transactions

- **A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.**
- **Atomic Transactions: a user can think of a transaction as always either executing all its actions, or not executing any actions at all.**
  - One approach: DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.
  - Another approach: *Shadow Pages*
  - Logs won because of need for audit trail and for efficiency reasons.

Transaction Consistency

- **"Consistency" - data in DBMS is accurate in modeling real world, follows integrity constraints**
- **User must ensure transaction consistent by itself**
- **If DBMS is consistent before transaction, it will be after also.**
- **System checks ICs and if they fail, the transaction rolls back (i.e., is aborted).**
  - DBMS enforces some ICs, depending on the ICs declared in CREATE TABLE statements.
  - Beyond this, DBMS does not understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

Isolation (Concurrency)

- **Multiple users can submit transactions.**
- **Each transaction executes as if it was running by itself.**
  - Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- **We will formalize this notion shortly.**
- **Many techniques have been developed. Fall into two basic categories:**
  - Pessimistic – don't let problems arise in the first place
  - Optimistic – assume conflicts are rare, deal with them *after* they happen.

Durability - Recovering From a Crash

- **System Crash - short-term memory lost (disk okay)**
  - This is the case we will handle.
- **Disk Crash - "stable" data lost**
  - ouch --- need back ups; raid-techniques can help avoid this.
- **There are 3 phases in Aries recovery (and most others):**
  - *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
  - *Redo*: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out.
  - *Undo*: The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, as found in the log), working backwards in the log.
- **At the end --- all committed updates and only those updates are reflected in the database.**
  - Some care must be taken to handle the case of a crash occurring during the recovery process!

Plan of attack (ACID properties)

- **First we'll deal with "I", by focusing on concurrency control.**
- **Then we'll address "A" and "D" by looking at recovery.**
- **What about "C"?**
- Well, if you have the other three working, and you set up your integrity constraints correctly, then you get this for free (!?).

Example

- **Consider two transactions (*Xacts*):**

```
T1:  BEGIN  A=A+100,   B=B-100   END
T2:  BEGIN  A=1.06*A,  B=1.06*B  END
```

- 1st xact transfers $100 from B's account to A's
- 2nd credits both accounts with 6% interest.
- Assume at first A and B each have $1000. What are the legal outcomes of running T1 and T2???
  - $2000 *1.06 = $2120
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect *must* be equivalent to these two transactions running serially in some order.

Example (Contd.)

- **Legal outcomes: A=1166,B=954 or A=1160,B=960**
- **Consider a possible interleaved *schedule*:**

```
T1:  A=A+100,              B=B-100
T2:            A=1.06*A,             B=1.06*B
```

❖ This is OK (same as T1;T2). But what about:

```
T1:  A=A+100,                        B=B-100
T2:            A=1.06*A, B=1.06*B
```

- **Result: A=1166, B=960; A+B = 2126, bank loses $6**
- **The DBMS's view of the second schedule:**

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

Scheduling Transactions

- ***Serial schedule:*** **A schedule that does not interleave the actions of different transactions.**
  - i.e., you run the transactions serially (one at a time)
- ***Equivalent schedules:*** **For any database state, the effect (on the set of objects in the database) and output of executing the first schedule is identical to the effect of executing the second schedule.**
- ***Serializable schedule*:** **A schedule that is equivalent to some serial execution of the transactions.**
  - Intuitively: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time.

Anomalies with Interleaved Execution

Unrepeatable Reads:

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

Reading Uncommitted Data ("dirty reads"):

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

Overwriting Uncommitted Data:

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

Conflict Serializable Schedules

- **We need a formal notion of equivalence that can be implemented efficiently...**
- **Two operations conflict if they are by different transactions, they are on the same object, and at least one of them is a write.**
- **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
- **Schedule S is conflict serializable if S is conflict equivalent to some serial schedule.**
- **Note, some "serializable" schedules are NOT conflict serializable.**
  - This is the price we pay for efficiency.

Conflict Equivalence - Intuition

- **If you can transform an interleaved schedule by swapping consecutive *non-conflicting* operations of different transactions into a serial schedule, then the original schedule is *conflict serializable*.**
- ***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 Equivalence (Continued)

- **Here's another example:**

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

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

NOT!

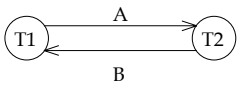
Dependency Graph

- ***Dependency graph*: One node per Xact; edge from *Ti* to *Tj* if an operation of Ti conflicts with an operation of Tj and Ti's operation appears earlier in the schedule than the conflicting operation of Tj.**
- **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)	



Dependency graph

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

View Serializability – an Aside

- **Alternative (weaker) notion of serializability.**
- **Schedules S1 and S2 are view equivalent if:**
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2
- **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 Conflict Serializability

- **Conflict Serializability doesn't allow all schedules that you would consider correct.**
  - This is because it is strictly *syntactic* - it doesn't consider the meanings of the operations or the data.
- **In practice, Conflict Serializability is what gets used, because it can be done efficiently.**
  - Note: in order to allow more concurrency, some special cases do get implemented, such as for travel reservations, etc.
- **Two-phase locking (2PL) is how we implement it.**

Locks

- **We use "locks" to control access to items.**
- **Shared (S) locks – multiple transactions can hold these on a particular item at the same time.**
- **Exclusive (X) locks – only one of these and no other locks, can be held on a particular item at a time.**

Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

Two-Phase Locking (2PL)

- **Locking Protocol**
  - Each transaction must obtain
    - a S (*shared*) lock on object before reading,
    - and an X (*exclusive*) lock on object before writing.
  - A transaction can not request additional locks once it releases any locks.
- **Thus, there is a "growing phase" followed by a "shrinking phase".**
- **2PL on its own is sufficient to guarantee conflict serializability.**
  - Doesn't allow cycles in the dependency graph!
- **But, it's susceptible to "cascading aborts"…**

Avoiding 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:
  - All locks held by a transaction are released only when the transaction completes

Strict 2PL (continued)

All locks held by a transaction are released only when the transaction completes

- **Strict 2PL allows only schedules whose precedence graph is acyclic, but it is actually stronger than needed for that purpose.**
- **In effect, "shrinking phase" is delayed until:**
  a) Transaction has committed (commit log record on disk), or
  b) Decision has been made to abort the transaction (then locks can be released after rollback).

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)	

Is it a 2PL schedule? Strict 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)

Is it a 2PL schedule? 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)

Is it a 2PL schedule? Strict 2PL?

Lock Management

- **Lock and unlock requests are handled by the Lock Manager.**
- **LM contains an entry for each currently held lock.**
- **Lock table entry:**
  - Ptr. to list of transactions currently holding the lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- **When lock request arrives see if anyone else holding a conflicting lock.**
  - If not, create an entry and grant the lock.
  - Else, put the requestor on the wait queue
- **Locking and unlocking have to be atomic operations**
- **Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock**
  - Can cause deadlock problems

Example: Output = ?

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

Is it a 2PL schedule? Strict 2PL?

Deadlocks

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

Deadlock Prevention

- **Assign priorities based on timestamps. Assume 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
- **If a transaction re-starts, make sure it gets its original timestamp**
  - Why?

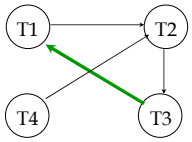
Deadlock Detection

- **Alternative is to allow deadlocks to happen but to check for them and fix them if found.**
- **Create a waits-for graph:**
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- **Periodically check for cycles in the waits-for graph**

Deadlock Detection (Continued)

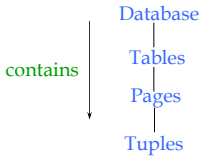
Example:

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



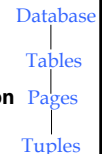
Multiple-Granularity Locks

- **Hard to decide what granularity to lock (tuples vs. pages vs. tables).**
- **Shouldn't have to make same decision for all transactions!**
- **Data "containers" are nested:**



Solution: New Lock Modes, Protocol

- **Allow Xacts to lock at each level, but with a special protocol using new "intention" locks:**
- **Still need S and X locks, but before locking an item, Xact must have proper intension 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?

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

Multiple Granularity Lock Protocol

- **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 SIX on parent? S 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.

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 courser-grained locks when too many low level locks acquired

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

Dynamic Databases – The "Phantom" Problem

- **If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL (on individual items) will not assure serializability:**
- **Consider T1 – "Find oldest sailor"**
  - T1 locks all records, and finds oldest sailor (say, *age* = 71).
  - Next, T2 inserts a new sailor; *age* = 96 and commits.
  - T1 (within the same transaction) checks for the oldest sailor again and finds sailor aged 96!!
- **The sailor with age 96 is a "phantom tuple" from T1's point of view --- first it's not there then it is.**
- **No serial execution where T1's result could happen!**

The "Phantom" Problem – example 2

- **Consider T3 – "Find oldest sailor for each rating"**
  - T3 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
  - Next, T4 inserts a new sailor; *rating* = 1, *age* = 96.
  - T4 also deletes oldest sailor with rating = 2 (and, say, *age* = 80), and commits.
  - T3 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- **T3 saw only part of T4's effects!**
- **No serial execution where T3's result could happen!**

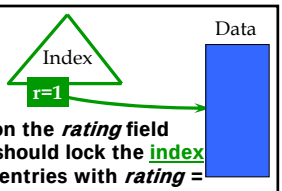
The Problem

- **T1 and T3 implicitly assumed that they had locked the set of all sailor records satisfying a predicate.**
  - Assumption only holds if no sailor records are added while they are executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- **Examples show that conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed!**

Predicate Locking (not practical)

- **Grant lock on all records that satisfy some logical predicate, e.g. *age > 2*salary*.**
- **Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.**
  - What is the predicate in the sailor example?
- **In general, predicate locking has a lot of locking overhead.**

Index Locking



- **If there is a dense index on the *rating* field using Alternative (2), T3 should lock the index page containing the data entries with *rating* = 1.**
  - If there are no records with *rating* = 1, T3 must lock the index page where such a data entry *would* be, if it existed!
- **If there is no suitable index, T3 must lock all pages, and lock the file/table to prevent new pages from being added, to ensure that no records with *rating* = 1 are added or deleted.**

Transaction Support in SQL-92

- **SERIALIZABLE – No phantoms, all reads repeatable, no "dirty" (uncommited) reads.**
- **REPEATABLE READS – phantoms may happen.**
- **READ COMMITTED – phantoms and unrepeatable reads may happen**
- **READ UNCOMMITTED – all of them may happen.**

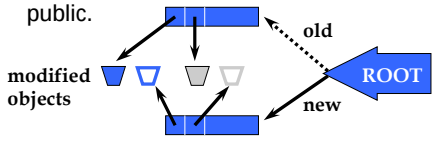
Optimistic CC (Kung-Robinson)

Locking is a conservative approach in which conflicts are prevented. Disadvantages:

- **Lock management overhead.**
- **Deadlock detection/resolution.**
- **Lock contention for heavily used objects.**
- **Locking is "pessimistic" because it assumes that conflicts will happen.**
- **If conflicts are rare, we might get better performance by not locking, and instead checking for conflicts at commit.**

Kung-Robinson Model

- **Xacts have three phases:**
  - READ: Xacts read from the database, but make changes to private copies of objects.
  - VALIDATE: Check for conflicts.
  - WRITE: Make local copies of changes public.



Details on Opt. CC

- **Students not responsible for these details for Spring 06 class.**
  - You *do* need to understand the basic difference between "pessimistic" approaches and "optimistic" ones.
  - You should understand the tradeoffs between them.
  - Optimistic concurrency control is a very elegant idea, and I highly recommend that you have a look.

Validation

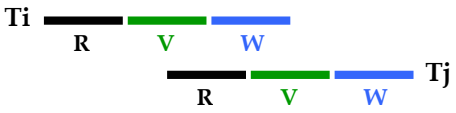
- **Test conditions that are sufficient to ensure that no conflict occurred.**
- **Each Xact is assigned a numeric id.**
  - Just use a **timestamp**.
- **Xact ids assigned at end of READ phase, just before validation begins.**
- **ReadSet(Ti): Set of objects read by Xact Ti.**
- **WriteSet(Ti): Set of objects modified by Ti.**

Test 1

- **For all i and j such that Ti < Tj, check that Ti completes before Tj begins.**

Test 2

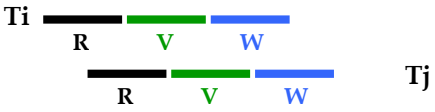
- **For all i and j such that Ti < Tj, check that:**
  - Ti completes before Tj begins its Write phase **+**
  - WriteSet(Ti) ∩ ReadSet(Tj) is empty.



Does Tj read dirty data? Does Ti overwrite Tj's writes?

Test 3

- **For all i and j such that Ti < Tj, check that:**
  - Ti completes Read phase before Tj does **+**
  - WriteSet(Ti) ∩ ReadSet(Tj) is empty **+**
  - WriteSet(Ti) ∩ WriteSet(Tj) is empty.



Does Tj read dirty data? Does Ti overwrite Tj's writes?

Applying Tests 1 & 2: Serial Validation

- **To validate Xact T:**

```
valid = true;
// S = set of Xacts that committed after Begin(T)
//    (above defn implements Test 1)
//The following is done in critical section
< foreach  Ts in S do {
  if ReadSet(T) intersects WriteSet(Ts)
       then valid = false;
  }
  if valid then { install updates; // Write phase
                  Commit T } >
            else Restart T
```

start of critical section

end of critical section

Comments on Serial Validation

- **Applies Test 2, with T playing the role of Tj and each Xact in Ts (in turn) being Ti.**
- **Assignment of Xact id, validation, and the Write phase are inside a** critical section**!**
  - Nothing else goes on concurrently.
  - So, no need to check for Test 3 --- can't happen.
  - If Write phase is long, major drawback.
- **Optimization for Read-only Xacts:**
  - Don't need critical section (because there is no Write phase).

Overheads in Optimistic CC

- **Must record read/write activity in ReadSet and WriteSet per Xact.**
  - Must create and destroy these sets as needed.
- **Must check for conflicts during validation, and must make validated writes ``global".**
  - Critical section can reduce concurrency.
  - Scheme for making writes global can reduce clustering of objects.
- **Optimistic CC restarts Xacts that fail validation.**
  - Work done so far is wasted; requires clean-up.

Other Techniques

- Timestamp CC: **Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each Xact a timestamp (TS) when it begins:**
  - If action ai of Xact Ti conflicts with action aj of Xact Tj, and TS(Ti) < TS(Tj), then ai must occur before aj. Otherwise, restart violating Xact.
- Multiversion CC: **Let writers make a "new" copy while readers use an appropriate "old" copy.**
  - Advantage is that readers don't need to get locks
  - Oracle uses a simple form of this.

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 directly implement the notions of conflict.**
  - The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- **Must be careful if objects can be added to or removed from the database ("phantom problem").**
- **Index locking common, affects performance significantly.**
  - Needed when accessing records via index.
  - Needed for locking logical sets of records (index locking/predicate locking).

Summary (Contd.)

- **Multiple granularity locking reduces the overhead involved in setting locks for nested collections of objects (e.g., a file of pages);**
  - should not be confused with tree index locking!
- **Tree-structured indexes:**
  - Straightforward use of 2PL very inefficient.
  - Idea is to use 2PL on data to ensure serializability and use other protocols on tree to ensure structural integrity.
  - We didn't cover this, but if you're interested, it's in the book.

Summary (Contd.)

- **Optimistic CC aims to minimize CC overheads in an ``optimistic'' environment where reads are common and writes are rare.**
- **Optimistic CC has its own overheads however; most real systems use locking.**
- **There are many other approaches to CC that we don't cover here. These include:**
  - timestamp-based approaches
  - multiple-version approaches
  - semantic approaches