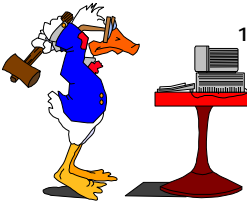
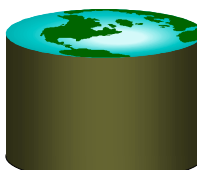


Transaction Management Overview

15-415, Spring 2003, Lecture 21
R & G Chapter 16



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



Definitions

- Database
 - a fixed set of named resources (entities)
- Consistency constraints
 - must be true for DB to be considered consistent
 - **Example:**

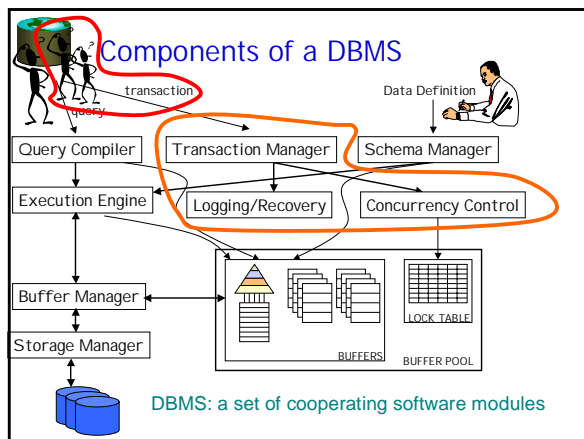
$$\Sigma(\text{ACCT-BALS}) = \Sigma(\text{ASSETS})$$

$$\text{ACCT-BAL} \geq 0$$
- Key point

consistent database
S1

→

consistent database
S2



Statement of Problem

- Concurrent execution of independent transactions
 - utilization/throughput ("hide" waiting for I/Os.)
 - response time
 - fairness
- Example:

	T1:		T2:
t0:	tmp1 := read(X)		tmp2 := read(X)
t1:			
t2:	tmp1 := tmp1 - 20		tmp2 := tmp2 + 10
t3:			
t4:	write tmp1 into X		
t5:			write tmp2 into X

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

Statement of problem (cont.)

- Arbitrary interleaving can lead to
 - Temporary inconsistency (ok, unavoidable)
 - "Permanent" inconsistency
- Need correctness criteria:
 - **schedule:** a particular action sequencing for a set of transactions
 - **consistent schedule:** each transaction sees consistent view of DB



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.



Transaction Consistency

- "Consistency" - data in DBMS is accurate in modeling real world, follows integrity constraints
- **User must ensure transaction consistent by itself**
 - I.e., if DBMS consistent before Xact, 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).



Goal: The ACID properties

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



Isolation (Concurrency)

- **Users submit transactions, and**
- **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.



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.
- A very important property guaranteed by the DBMS for all transactions is that they are **atomic**. That is, a user can think of a Xact 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.



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:	$R(A), W(A), R(B), W(B)$	



Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

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



Scheduling Transactions

- Serial schedule: Schedule that does not interleave the actions of different transactions.
- Equivalent schedules: For any database state, the effect (on the set of objects in the database) 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.
(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)



Lock-Based Concurrency Control

- Here's a simple way to allow concurrency but avoid the anomalies just described...
- Strict Two-phase Locking (Strict 2PL) Protocol:
 - Each Xact must obtain a *S (shared)* lock on object before reading, and an *X (exclusive)* lock on object before writing.
 - System can obtain these locks *automatically*
 - Two phases: acquiring locks, and releasing them
 - no lock is ever acquired after one has been released
 - All locks held by a transaction are released when the transaction completes
 - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.



Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, "dirty reads"):

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

- Unrepeatable Reads (RW Conflicts):

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



Aborting a Transaction (i.e., Rollback)

- If a xact T_i aborted, all actions must be undone.
 - Also, if T_j reads object last written by T_i , T_j must be aborted!
- Most systems avoid such cascading aborts by releasing locks only at EOT (i.e., strict locking).
 - If T_i writes an object, T_j can read this only after T_i finishes.
- In order to undo actions of an aborted transaction, DBMS maintains log which records every write. Log also used to recover from system crashes: all active Xacts at time of crash are aborted when system comes back up.



The Log

- Log consists of "records" that are written sequentially.
 - Typically chained together by Xact id
 - Log is often *duplexed* and *archived* on stable storage.
- Need for UNDO and/or REDO depend on Buffer Mgr.
 - UNDO required if uncommitted data can overwrite stable version of committed data (STEAL buffer management).
 - REDO required if xact can commit before all its updates are on disk (NO FORCE buffer management).
- The following actions are recorded in the log:
 - *if Ti writes an object, write a log record with:*
 - If UNDO required need "before image"
 - If REDO required need "after image".
 - *Ti commits/aborts:* a log record indicating this action.



Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Concurrency control is automatic.
 - System automatically inserts lock/unlock requests and schedules actions of different Xacts in such a way as to ensure that the resulting execution is equivalent to executing the Xacts one after the other in some order.
- Write-ahead logging (WAL) and the recovery protocol are used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.



Logging Continued

- Write Ahead Logging protocol
 - Log record must go to disk *before* the changed page!
 - implemented via a handshake between log manager and the buffer manager.
 - All log records for a transaction (including it's commit record) must be written to disk before the transaction is considered "Committed".
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.



Durability - Recovering From a Crash

- 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 and written to disk.
 - *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!