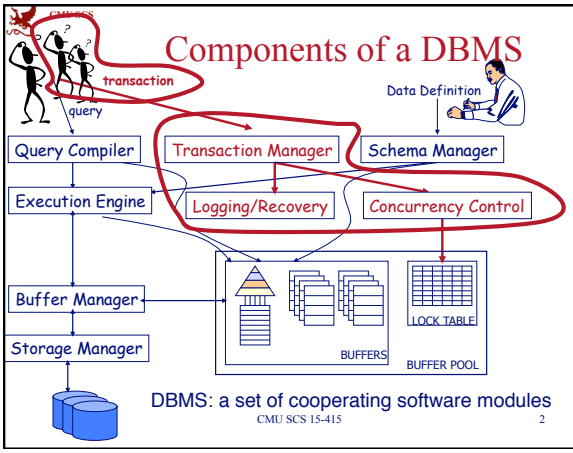
CMU SCS

Carnegie Mellon Univ.
Dept. of Computer Science
15-415 - Database Applications

Lecture #20: Overview of Transaction Management
(R&G ch. 16)

CMU SCS

Components of a DBMS



DBMS: a set of cooperating software modules

CMU SCS

Concurrency Control & Recovery

- Very valuable properties of DBMSs
- Based on concept of transactions with ACID properties
- Next lectures discuss these issues

CMU SCS

Overview

- Problem definition & 'ACID'
- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

CMU SCS

Transactions - dfn

= unit of work, eg.

move $10 from savings to checking

CMU SCS

Statement of Problem

- Concurrent execution of independent transactions (why do we want that?)

CMU SCS

Statement of Problem

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

CMU SCS

Statement of Problem

- Concurrent execution of independent transactions
  - utilization/throughput ("hide" waiting for I/Os.)
  - response time
- would also like:
  - correctness &
  - fairness
- Example: Book an airplane seat

CMU SCS

Example: 'Lost-update' problem

time

T1	T2
Read(N)	
	Read(N)
N=N-1	
	N= N-1
Write(N)	
	Write(N)

CMU SCS

Statement of problem (cont.)

- Arbitrary interleaving can lead to
  - Temporary inconsistency (ok, unavoidable)
  - "Permanent" inconsistency (bad!)
- Need formal correctness criteria.

CMU SCS

Definitions

- A program may carry out many operations on the data retrieved from the database
- However, the DBMS is only concerned about what data is read/written from/to the database.

CMU SCS

Definitions

- *database* - a fixed set of named data objects *(A, B, C, ...)*
- *transaction* - a sequence of read and write operations *(read(A), write(B), ...)*
  - DBMS's abstract view of a user program

CMU SCS

Correctness criteria: 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.

CMU SCS

Correctness criteria: The **ACID** properties

- **A** tomicity: 'all or nothing'
- **C** onsistency:
- **I** solation: 'as if alone'
- **D** urability: survive failures (e.g., power failure)

CMU SCS

Overview

- Problem definition & 'ACID'
- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

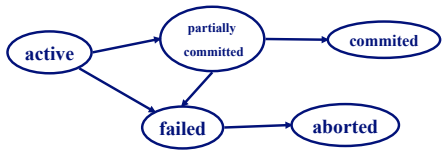
CMU SCS

Atomicity of Transactions

- Two possible outcomes of executing a transaction:
  - Xact might *commit* after completing all its actions
  - or it could *abort* (or be aborted by the DBMS) after executing some actions.
- DBMS guarantees that Xacts are *atomic*.
  - From user's point of view: Xact always either executes all its actions, or executes no actions at all.

CMU SCS

Transaction states



CMU SCS

Mechanisms for Ensuring Atomicity

- What would you do?

CMU SCS

Mechanisms for Ensuring Atomicity

- One approach: LOGGING
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.
- ~ like black box in airplanes …

CMU SCS

Mechanisms for Ensuring Atomicity

- Logging used by all modern systems.
- Q: why?

CMU SCS

Mechanisms for Ensuring Atomicity

Logging used by all modern systems.

- Q: why?
- A:
  - audit trail &
  - efficiency reasons

What other mechanism can you think of?

CMU SCS

A

Mechanisms for Ensuring Atomicity

- Another approach: SHADOW PAGES
  - (not as popular)

CMU SCS

Overview

- Problem definition & 'ACID'
- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

CMU SCS

C

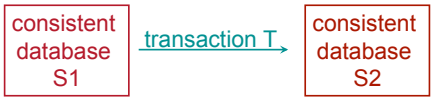
Transaction Consistency

- "Database consistency" - data in DBMS is accurate in modeling real world and follows integrity constraints

CMU SCS

Transaction Consistency

- "Transaction Consistency": if DBMS consistent before Xact (running alone), it will be after also
- Transaction consistency: **User's responsibility**
  - DBMS just checks IC



CMU SCS

Transaction Consistency (cont.)

- Recall: Integrity constraints
  - must be true for DB to be considered consistent
  - **Examples:**
    1. FOREIGN KEY R.sid REFERENCES S
    2. ACCT-BAL >= 0

CMU SCS

Transaction Consistency (cont.)

- System checks ICs and if they fail, the transaction rolls back (i.e., is aborted).
  - Beyond this, DBMS does not understand the semantics of the data.
  - e.g., it does not understand how interest on a bank account is computed
- Since it is the user's responsibility, we don't discuss it further

CMU SCS

Overview

- Problem definition & 'ACID'
- **A**tomicity
- **C**onsistency
- **I**solation ('as if alone')
- **D**urability

CMU SCS

I

Isolation of Transactions

- 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.
- Q: How would you achieve that?

CMU SCS

I

Isolation of Transactions

A: Many methods - two main categories:

- **Pessimistic** – don't let problems arise in the first place
- **Optimistic** – assume conflicts are rare, deal with them *after* they happen.

CMU SCS

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?

CMU SCS

Example

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

- many - but A+B should be: $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.

CMU SCS

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
```

CMU SCS

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)	

CMU SCS

'Correctness'?

- Q: How would you judge that a schedule is 'correct'?
- ('schedule' = 'interleaved execution')

CMU SCS

'Correctness'?

- Q: How would you judge that a schedule is 'correct'?
- A: if it is equivalent to **some** serial execution

CMU SCS

Formal Properties of Schedules

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules:* For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule. (*)

(*) no matter what the arithmetic e.t.c. operations are!

CMU SCS

Formal Properties of Schedules

- *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.)

CMU SCS

Anomalies with interleaved execution:

- R-W conflicts
- W-R conflicts
- W-W conflicts
- (why not R-R conflicts?)

CMU SCS

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	

CMU SCS

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	

CMU SCS

Anomalies with Interleaved Execution

- Unrepeatable Reads (RW Conflicts):

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

CMU SCS

Anomalies with Interleaved Execution

- Unrepeatable Reads (RW Conflicts):

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

CMU SCS

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

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

CMU SCS

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

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

CMU SCS

Solution?

- Q: How could you guarantee that all resulting schedules are correct (= serializable)?

CMU SCS

Answer

- (Part of the answer:) use locks!

CMU SCS

Answer

- (Full answer:) use locks; keep them until commit ('strict 2 phase locking')
- Let's see the details

CMU SCS

Lost update problem - no locks

T1	T2
Read(N)	
	Read(N)
N = N -1	
	N = N -1
Write(N)	
	Write(N)

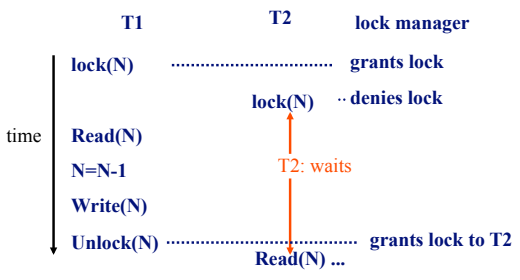
CMU SCS

Solution – part 1

- with locks:
- lock manager: grants/denies lock requests

CMU SCS

Lost update problem – with locks



CMU SCS

Locks

- Q: I just need to read 'N' - should I still get a lock?

CMU SCS

Solution – part 1

- Locks and their flavors
  - exclusive (or write-) locks
  - shared (or read-) locks
  - <and more ... >
- compatibility matrix

T2 wants / T1 has	S	X
S		
X		

CMU SCS

Solution – part 1

- Locks and their flavors
  - exclusive (or write-) locks
  - shared (or read-) locks
  - <and more ... >
- compatibility matrix

T2 wants / T1 has	S	X
S	Yes	
X		

CMU SCS

Solution – part 1

- transactions request locks (or upgrades)
- lock manager grants or blocks requests
- transactions release locks
- lock manager updates lock-table

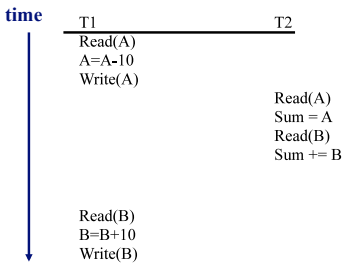
CMU SCS

Solution – part 2

locks are not enough – eg., 'inconsistent analysis'

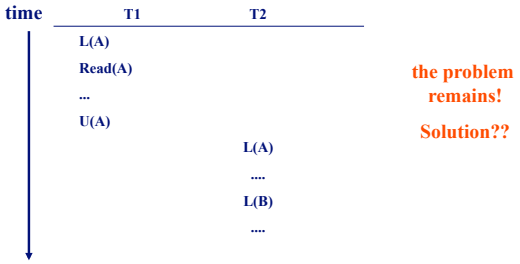
CMU SCS

'Inconsistent analysis'



CMU SCS

'Inconsistent analysis' – w/ locks



the problem remains!

Solution??

CMU SCS

General solution:

- Protocol(s)
- Most popular protocol: 2 Phase Locking (2PL)

CMU SCS

2PL

X-lock version: transactions issue no lock requests, after the first 'unlock'

THEOREM: if all transactions obey 2PL -> all schedules are serializable

CMU SCS

2PL – example

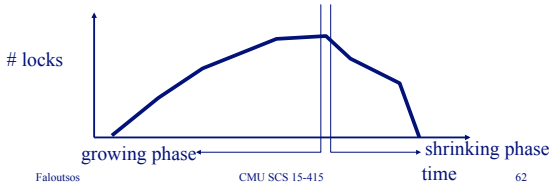
- 'inconsistent analysis' – how does 2PL help?
- how would it be under 2PL?
- (answer: on the chalk-board)

CMU SCS

2PL – X/S lock version

transactions issue no lock/upgrade request, after the first unlock/downgrade

In general: 'growing' and 'shrinking' phase

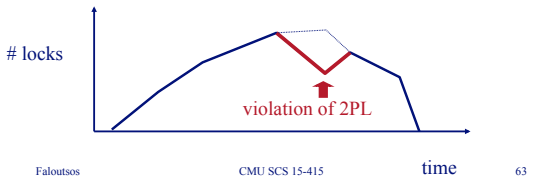


CMU SCS

2PL – X/S lock version

transactions issue no lock/upgrade request, after the first unlock/downgrade

In general: 'growing' and 'shrinking' phase



CMU SCS

2PL – observations

- limits concurrency
- may lead to deadlocks
- **strict 2PL** (a.k.a. 2PLC): keep locks until 'commit'
  - avoids 'dirty reads' etc
  - but limits concurrency even more
  - (and still may lead to deadlocks)

CMU SCS

Aborting a Transaction (i.e., Rollback)

- If an xact *Ti* aborted, all actions must be undone.
- On 'dirty reads': cascading aborts
- strict 2PL: avoids 'dirty reads' (why?)

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

CMU SCS

Aborting a Transaction (i.e., Rollback)

- 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.

CMU SCS

(Review) 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.

CMU SCS

(Review) 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 ura What happens if system **crashes** between *commit* and *flushing modified data to disk* ? st.

CMU SCS

D

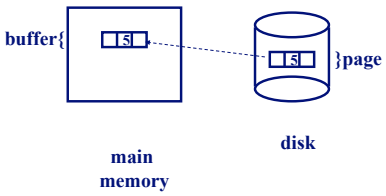
Problem definition

- Records are on disk
- for updates, they are copied in memory
- and flushed back on disk, *at the discretion of the O.S.!* (unless forced-output: 'output (B)' = fflush())

CMU SCS

Problem definition - eg.:

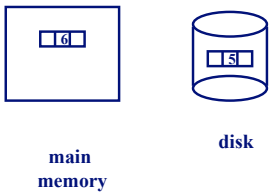
→ read(X)
X=X+1
write(X)



CMU SCS

Problem definition - eg.:

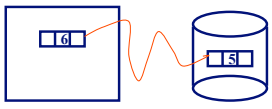
read(X)
→ X=X+1
write(X)



CMU SCS

Problem definition - eg.:

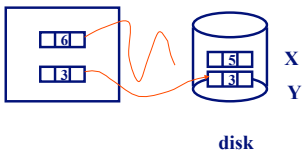
read(X)
X=X+1
→ write(X)



buffer joins an ouput queue,
but it is NOT flushed immediately!
Q1: why not?
Q2: so what?

Problem definition - eg.:

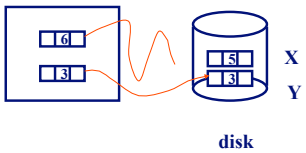
read(X)
read(Y)
→ X=X+1
Y=Y-1
write(X)
write(Y)



Q2: so what?

Problem definition - eg.:

read(X)
read(Y)
X=X+1
Y=Y-1
write(X)
→ write(Y)



Q2: so what?

Q3: how to guard against it?

Solution: W.A.L.

- redundancy, namely
- write-ahead log, on 'stable' storage
- Q: what to replicate? (not the full page!!)
- A:
- Q: how exactly?

CMU SCS

W.A.L. - intro

- replicate intentions: eg:
  <T1 start>
  <T1, X, 5, 6>
  <T1, Y, 4, 3>
  <T1 commit> (or <T1 abort>)

CMU SCS

W.A.L. - intro

- in general: <transaction-id, data-item-id, old-value, new-value> (or similar)
- each transaction writes a log record first, **before** doing the change
- when done, DBMS
  - writes a <commit> record on the log
  - makes sure that all log records are flushed, &
  - lets xact exit

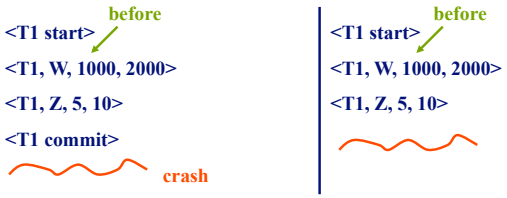
CMU SCS

W.A.L.

- After a failure, DBMS "replays" the log:
  - undo uncommited transactions
  - redo the committed ones

CMU SCS

W.A.L.



REDO T1 **UNDO T1**

CMU SCS

Logging (cont.)

- All logging and CC-related activities are handled transparently by the DBMS.

CMU SCS

Durability - Recovering From a Crash

- 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!

CMU SCS

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
  - Property ensured: resulting execution is equivalent to executing the Xacts one after the other in some order.

CMU SCS

Summary

- Write-ahead logging (WAL) and the recovery protocol are used to:
  1. undo the actions of aborted transactions, and
  2. restore the system to a consistent state after a crash.

CMU SCS

ACID properties:

Atomicity (all or none)
Consistency
Isolation (as if alone)
Durability