UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

CS411 - Transaction Management

illinois.edu

Review

- What is “non-quiescent checkpointing”?
- What is an advantage of undo/redo logging?
- Why do we care about scheduling?
- What is a “serial schedule”?
- What is a “serializable schedule”?

Example

<START T_1>

<T_1,A,4,5>

<START T_2>

<START T_3>

<COMMIT T_1>

<T_2,B,9,10>

<T_2,C,14,15>

<START T_4>

<T_3,D,19,20>

<T_4,E,30,31>

<COMMIT T_3>

<T_4,F,43,44>

Example

<START T_1>

<T_1,A,4,5>

<START T_2>

<COMMIT T_1>

<T_2,B,9,10>

<START CKPT (T_2)>

<T_2,C,14,15>

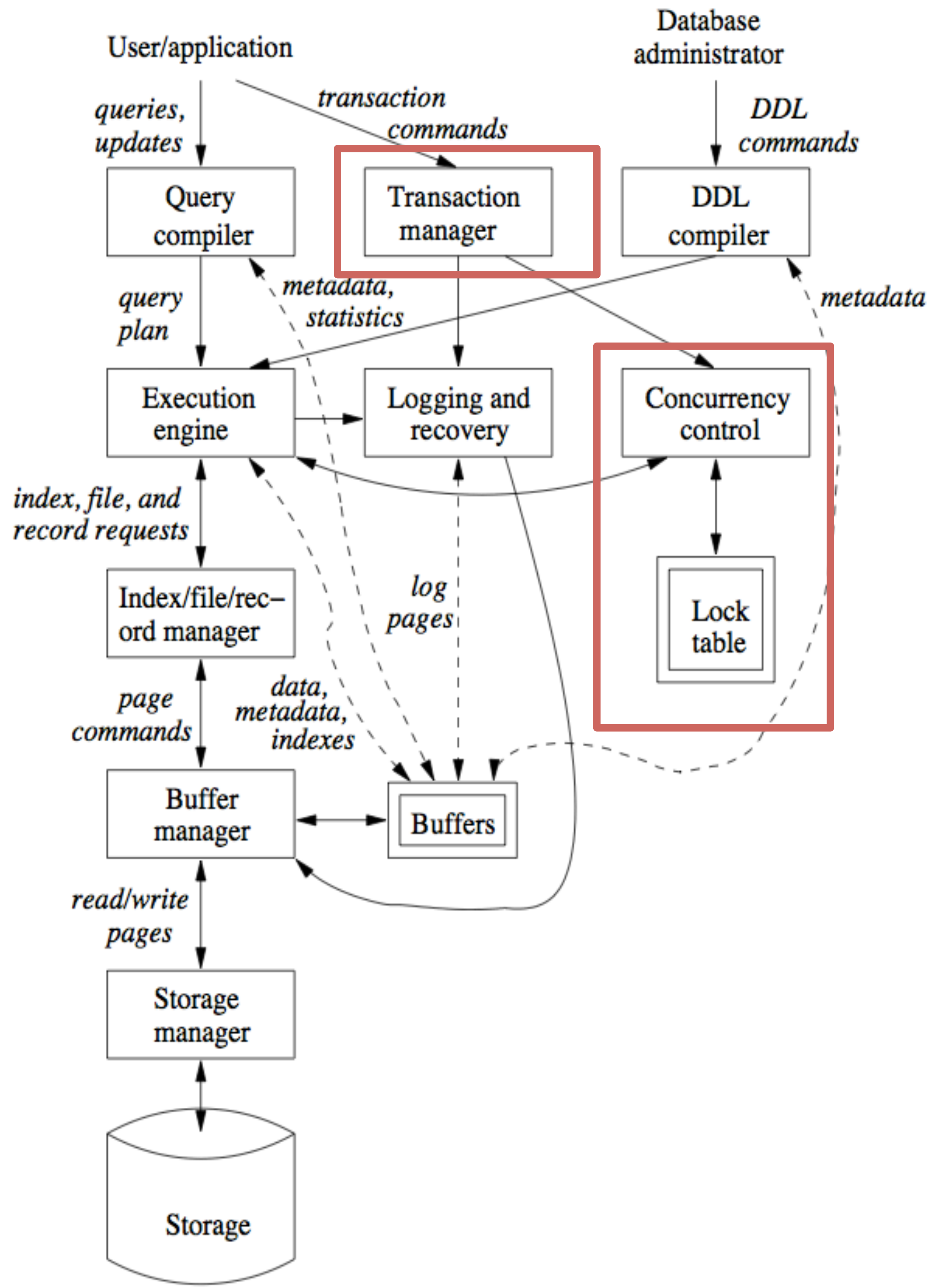
<START T_3>

<T_3,D,19,20>

<END CKPT>

<COMMIT T_2>

illinois.edu



illinois.edu

Serializable Schedule

- A schedule is *serializ**able*** if it has the same effect as some serial schedule

Notation

- We don’t care about values
- Just ***what*** is accessed, ***who*** accessed it, and ***how*** it is accessed
  - Read: $r_i(X)$ trasaction i reads element X
  - Write: $w_i(X)$ transaction i writes element X

Example

T1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;

T2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$;

S: $r_1(A)$; $w_1(A)$; $r_2(A)$; $w_2(A)$; $r_1(B)$; $w_1(B)$; $r_2(B)$; $w_2(B)$;

Conflicting swaps

- A pair of consecutive actions conflict if changing order changes the behavior of a transaction:
  1. two actions from the same transaction
  2. $r_i(X), w_j(X)$ or $w_i(X), r_j(X)$
  3. $w_i(X), w_j(X)$

Conflict-Serializable

- Making a series of non-conflicting swaps to a schedule, we can produce a serial schedule
- Non-conflicting swaps don't change behavior, so conflict-serializable $\Longrightarrow$ serializable

Example

r1(A); w1(A); r2(A); w2(A); r1(B); w1(B); r2(B); w2(B);

r1(A); w1(A); r2(A); r1(B); w2(A); w1(B); r2(B); w2(B);

r1(A); w1(A); r1(B); r2(A); w2(A); w1(B); r2(B); w2(B);

r1(A); w1(A); r1(B); r2(A); w1(B); w2(A); r2(B); w2(B);

r1(A); w1(A); r1(B); w1(B); r2(A); w2(A); r2(B); w2(B);

Conflict-Serializable

- Conflict-Serializable schedules are a subset of serializable schedules
  - Conflict-serializable is a "stronger" definition
- Example:
  - $w_1(A); w_2(A); w_2(B); w_1(B); w_3(B);$
  - Executing $T_1, T_2, T_3 \Longrightarrow$
    - A gets value from T_2
    - B gets value from T_3

Precedence Graphs

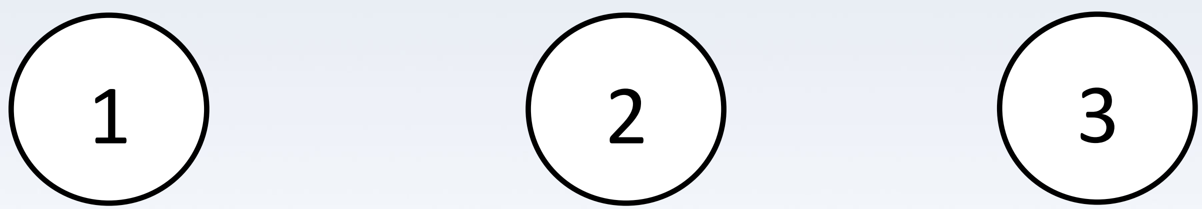
- Checking a schedule to see if it is conflict-serializable
  - Graph based approach
  - A node for each transaction
  - A directed edge from T_i to T_j if $T_i <_S T_j$
    - T_i takes precedence over T_j in schedule S

Precedence Graphs

- $T_i <_S T_j$ if there are actions A_i and A_j:
  - A_i is in T_i and A_j is in T_j
  - A_i happens before A_j
  - A_i and A_j both involve the same database element
  - Either A_i or A_j is a write action
- In other words, T_i and T_j have some conflicting action

Example

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$

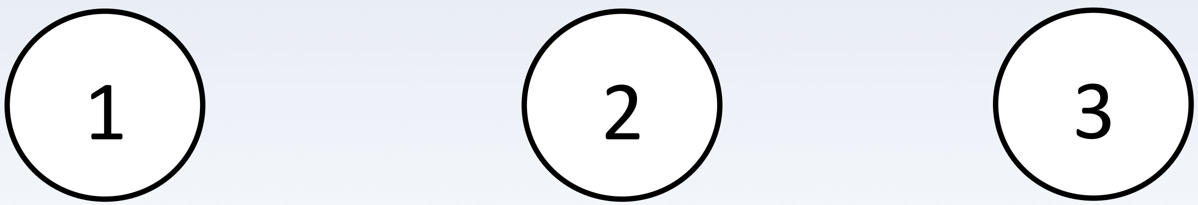


Precedence Graphs

- S is conflict-serializable iff its precedence graph has no cycles
  - Intuitively: a loop means the transactions in the loop share actions that can't be swapped to create a serial schedule
  - Formal proof is in the book

Example

$$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$$



Precedence Graphs

- Useful for humans to prove things about schedules
- Not very helpful for a DBMS, though
  - examine all possible schedules?
  - search for schedule with no cycles?

Scheduler

- Assures serializability in real time by:
  - Viewing actions of incoming transactions
  - "Locking" database elements to indicate they are in use: $l_1(A)$
  - "Unlocking" them when transaction finishes with them: $u_1(A)$
  - Delaying actions requiring locked elements

illinois.edu

Two-Phase Locking

- ***All*** locks for a transaction ***must*** come before ***all*** unlocks
  - Scheduler inserts locks before all reads and rights
  - Scheduler inserts unlocks after transaction commits or aborts

Example

$l_1(A)$; $w_1(A)$; $l_2(B)$; $r_2(B)$; $w_1(A)$; $u_1(A)$; $l_2(A)$; $w_2(A)$; $u_2(A)$; $u_2(B)$;

Serializability

- A two-phase locking scheduler ***guarantees*** conflict-serializability
  - Intuitively:
    - all actions for a transaction could be moved just before unlock phase
    - unlocks happen serially, so schedule is serializable
  - Again, we won't prove this

Deadlock

- Example:
  - T_1 has a lock on A, waiting for B
  - T_2 has a lock on B, waiting for A
- Two-phase lock scheduler can't stop this
  - Deadlock resolution is covered enough in other classes
  - We won't cover it

Lock Modes

- Practical schemes have multiple modes
  - Shared lock: $sl_1(A)$
    - ***Multiple transactions*** can have a shared lock on one element
    - Can only ***read*** the element, though
  - Exclusive lock: $xl_1(A)$
    - Only ***one transaction*** can have an exclusive lock
    - No shared locks allowed on the element
    - Can ***read or write*** the element

Compatibility Matrix

Lock Requested

Lock Held

	S	X
S	Yes	No
X	No	No

illinois.edu

Example

$sl_1(A)$; $w_1(A)$;

$sl_2(A)$; $r_2(A)$;

$sl_2(B)$; $r_2(B)$;

$xl_1(B)$; **DENIED**

$u_2(A)$; $u_2(B)$

$xl_1(A)$; $r_1(B)$; $w_1(B)$

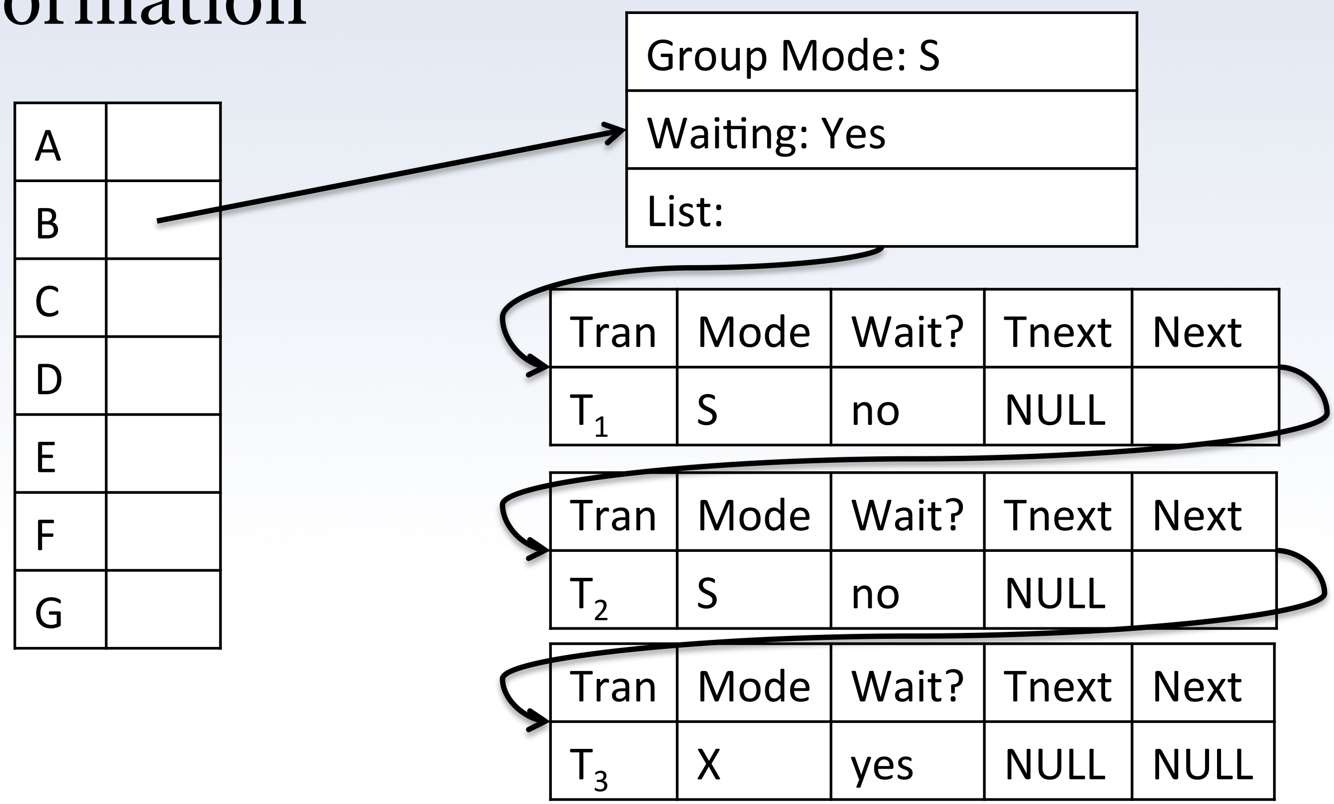
$u_1(A)$; $u_2(B)$

Scheduler Architecture

- How is this scheme implemented in a DBMS?
  - need to track locking and unlocking of elements
  - need to decide when to grant or deny a lock
  - need to delay transactions whose locks are denied

Scheduler Architecture

- Locks are entered in a ***lock table***
  - A relation of database elements and lock information



Scheduler Architecture

- Group Mode: current lock mode for B
- Waiting: is anyone waiting to lock B
- List: (next slide)

A	
B	
C	
D	
E	
F	
G	

Group Mode: U
Waiting: Yes
List:

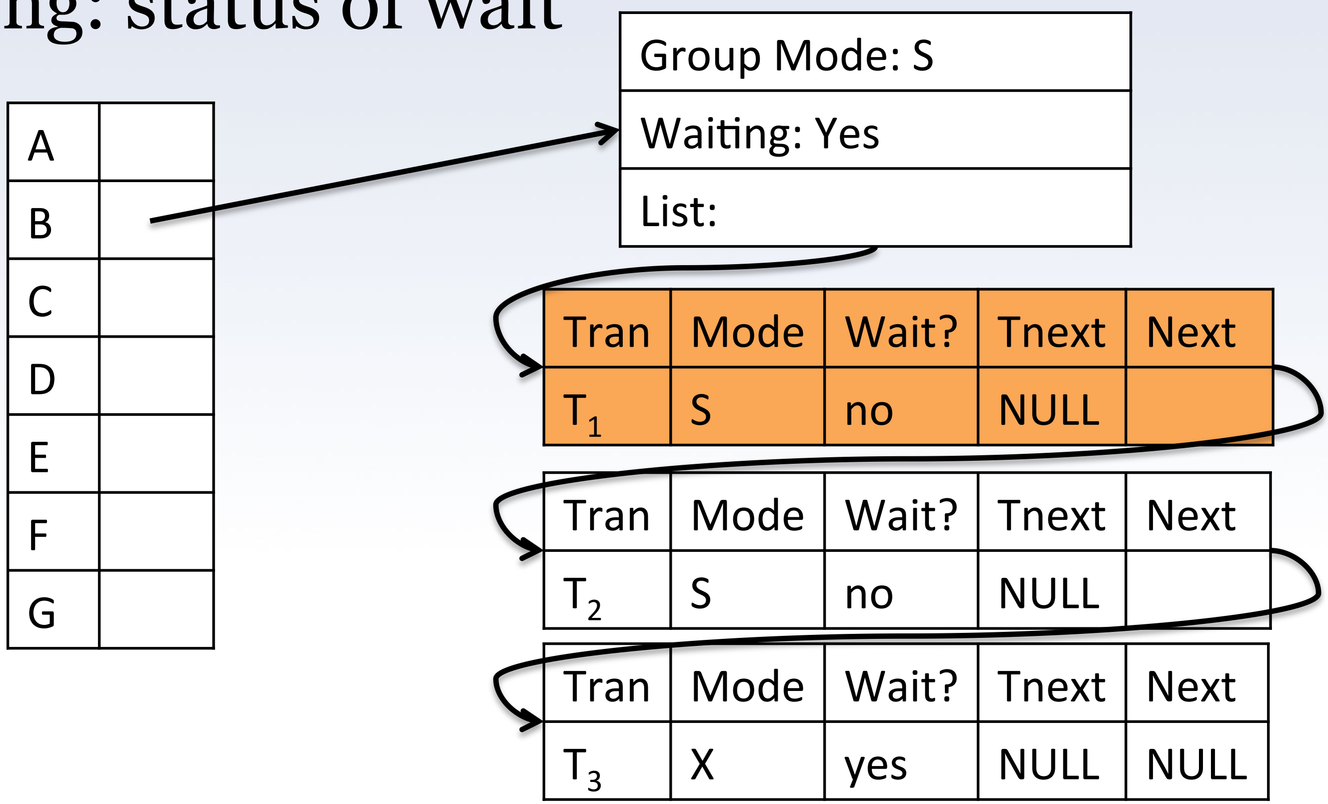
Tran	Mode	Wait?	Tnext	Next
T_1	S	no	NULL	

Tran	Mode	Wait?	Tnext	Next
T_2	S	no	NULL	

Tran	Mode	Wait?	Tnext	Next
T_3	X	yes	NULL	NULL

Scheduler Architecture

- Tran: transaction for this entry
- Mode: lock mode for transaction
- Waiting: status of wait



Handling locks

- If transaction T needs a lock on A
  - No lock-table entry: grant lock and create entry
  - Lock-table entry: check group mode and grant/deny as appropriate
    - Add to list and indicate waiting or not waiting

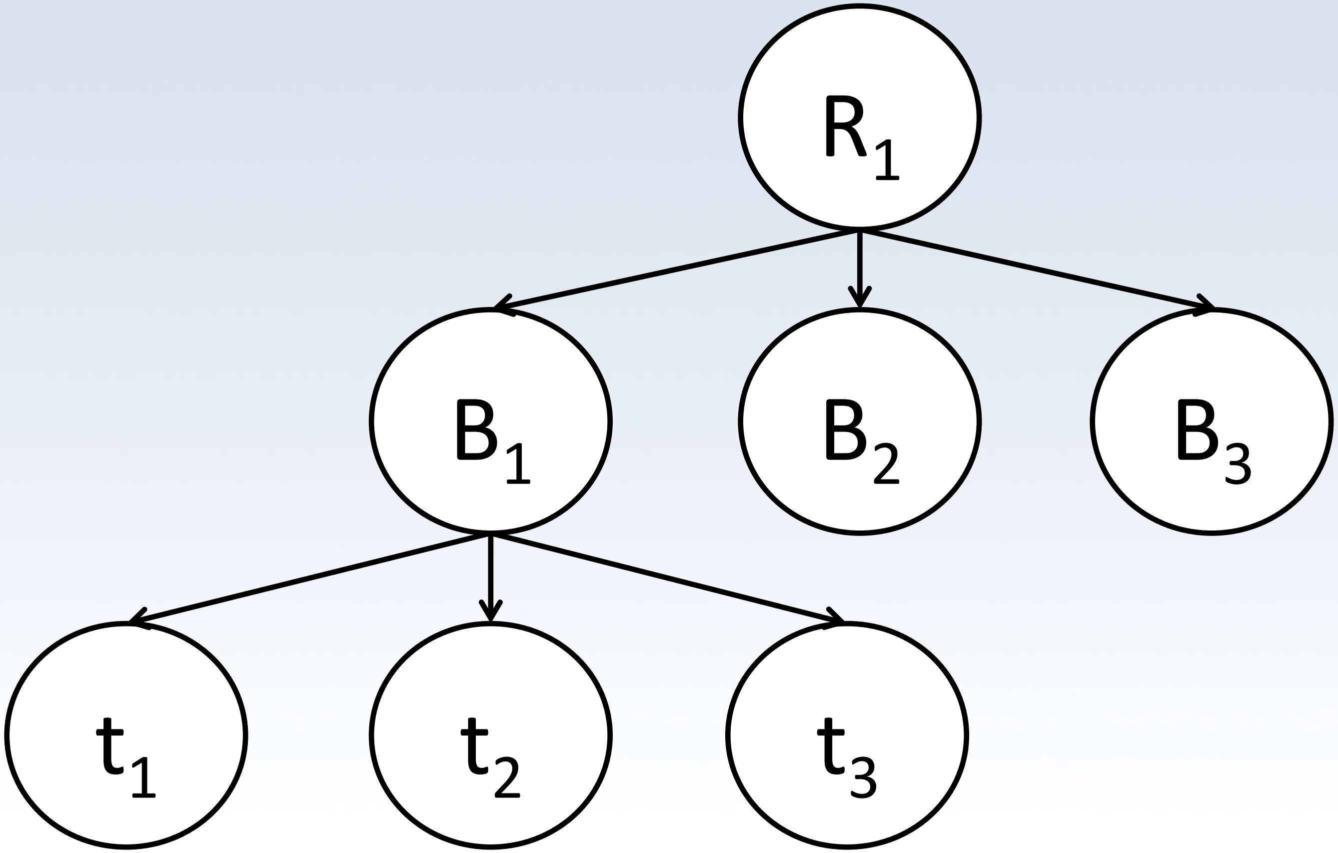
Handling unlocks

- If transaction T unlocks A
  - Delete T from list
  - Update group mode by examining remaining entries
  - If waiting is "yes", search list to see if any waiting locks can be granted

Locking Elements

- “Elements” is vague
- What are we locking?
  - Tables?
  - Blocks?
  - Tuples?
- Yes.

Hierarchical Locking



Warning Locks

- To lock an element with S or X, work down the hierarchy tree
- If element we want to lock is down the tree, request IS or IX lock on this node
  - If granted, go lower. If not, wait.
- When we get to element we want, request S or X lock

Compatibility Matrix

Lock Requested

Lock Held

	IS	IX	S	X
IS	Yes	Yes	Yes	No
IX	Yes	Yes	No	No
S	Yes	No	Yes	No
X	No	No	No	No

Example

- Page 925

Read phenomena

- Three types:
  1. Phantom reads
  2. Non-repeatable reads
  3. Dirty reads

Phantom Reads

- Can’t lock tuples that haven’t been inserted yet
  - If another transaction inserts a new tuple, it won’t know about our read locks
  - We might see part of the insert
- To avoid this, we can lock the entire table

Non-repeatable Reads

- Release read locks before committing
  - Other transactions might modify the data before our transaction commits
  - But we won't see data that hasn't been committed

Dirty reads

- Don't bother getting read locks
  - Just get whatever the current value is
  - Other transactions could modify the data
  - Those transactions might even be rolled back

Further reading

- Chapter 19 covers:
  - interaction between logging and transaction management
  - how to deal with deadlocks
- http://msdn.microsoft.com/en-us/library/aa213039(v=sql.80).aspx
- http://dev.mysql.com/doc/refman/5.0/en/internal-locking.html