Schedule

- Nov. 6 (T) Transactions, Authorization.
  ◆ Read Sections 8.6-8.7.
- Nov. 8 (TH) Object-Oriented Database Design.
  ◆ Read Sections 4.1-4.4. Project Part 5 due on Sunday night.
- Nov. 13 (T) Object-Relational, Object Queries (OQL).
  ◆ Read Sections 4.5, 9.1. Assignment 6 due. (No office hours.)
- Nov. 15 (TH) More OQL.
  ◆ Read Sections 9.2-9.3. Project Part 6 due on Sunday night.
- Nov. 20 (T) Object-Relational Queries.
  ◆ Read Sections 9.4-9.5. Assignment 7 due (no late ones).
- Nov. 22 (TH) Thanksgiving - No class scheduled.
- Nov. 27 (T) Semistructured Data, XML.
  ◆ Read Sections 4.6-4.7. Assignment 8 due. Project Part 7 due.
TRANSACTION MANAGEMENT

Airline Reservations  many updates
Statistical Abstract of the US  many queries

Atomicity – all or nothing principle

Serializability – the effect of transactions as if they occurred one at a time

Items – units of data to be controlled
   fine-grained – small items
   course-grained – large items
   (granularity)

Controlling access by locks
   Read – sharable with other readers  shared
   Write – not sharable with anyone else  exclusive

Model – (item, locktype, transaction ID)
Transactions

= units of work that must be:

1. *Atomic* = either all work is done, or none of it.
2. *Consistent* = relationships among values maintained.
3. *Isolated* = appear to have been executed when no other DB operations were being performed.
   - Often called *serializable* behavior.
4. *Durable* = effects are permanent even if system crashes.
Commit/Abort Decision

Each transaction ends with either:

1. *Commit* = the work of the transaction is installed in the database; previously its changes may be invisible to other transactions.
2. *Abort* = no changes by the transaction appear in the database; it is as if the transaction never occurred.
   - ROLLBACK is the term used in SQL and the Oracle system.
   - In the ad-hoc query interface (*e.g.*, PostgreSQL psql interface), transactions are single queries or modification statements.
      - Oracle allows `SET TRANSACTION READ ONLY` to begin a multistatement transaction that doesn't change any data, but needs to see a consistent “snapshot” of the data.
   - In program interfaces, transactions begin whenever the database is accessed, and end when either a COMMIT or ROLLBACK statement is executed.
Example

\[ \text{Sells(bar, beer, price)} \]

- Joe's Bar sells Bud for $2.50 and Miller for $3.00.
- Sally is querying the database for the highest and lowest price Joe charges:
  1. \[ \text{SELECT MAX(price) FROM Sells WHERE bar = 'Joe's Bar'}; \]
  2. \[ \text{SELECT MIN(price) FROM Sells WHERE bar = 'Joe's Bar'}; \]
- At the same time, Joe has decided to replace Miller and Bud by Heineken at $3.50:
  3. \[ \text{DELETE FROM Sells WHERE bar = 'Joe's Bar' AND (beer = 'Miller' OR beer = 'Bud')}; \]
  4. \[ \text{INSERT INTO Sells VALUES('Joe's bar', 'Heineken', 3.50)}; \]
- If the order of statements is 1, 3, 4, 2, then it appears to Sally that Joe's minimum price is greater than his maximum price.
- Fix the problem by grouping Sally's two statements into one transaction, e.g., with one SQL statement.
Example: Problem With Rollback

- Suppose Joe executes statement 4 (insert Heineken), but then, during the transaction thinks better of it and issues a **ROLLBACK** statement.
- If Sally is allowed to execute her statement 1 (find max) just before the rollback, she gets the answer $3.50, even though Joe doesn't sell any beer for $3.50.
- Fix by making statement 4 a transaction, or part of a transaction, so its effects cannot be seen by Sally unless there is a **COMMIT** action.
SQL Isolation Levels

Isolation levels determine what a transaction is allowed to see. The declaration, valid for one transaction, is:

```
SET TRANSACTION ISOLATION LEVEL X;
```

where:

- $X = \text{SERIALIZABLE}$: this transaction must execute as if at a point in time, where all other transactions occurred either completely before or completely after.

  Example: Suppose Sally's statements 1 and 2 are one transaction and Joe's statements 3 and 4 are another transaction. If Sally's transaction runs at isolation level SERIALIZABLE, she would see the Sells relation either before or after statements 3 and 4 ran, but not in the middle.
• $X = \text{READ COMMITTED}$: this transaction can read only committed data.
  ◆ Example: if transactions are as above, Sally could see the original $\text{Sells}$ for statement 1 and the completely changed $\text{Sells}$ for statement 2.

• $X = \text{REPEATABLE READ}$: if a transaction reads data twice, then what it saw the first time, it will see the second time (it may see more the second time).
  ◆ Example: If 1 is executed before 3, then 2 must see the Bud and Miller tuples when it computes the min, even if it executes after 3. But if 1 executes between 3 and 4, then 2 may see the Heineken tuple.
• $X = \text{READ UNCOMMITTED}$: essentially no constraint, even on reading data written and then removed by a rollback.
  
  ◆ Example: 1 and 2 could see Heineken, even if Joe rolled back his transaction.
T1                      T2                      start with A = 5
Read A                              A on disk   A in T1       A in T2
Read A                              5            5              5
A:= A + 1                        A:= 2* A                        Write A
Write A
<table>
<thead>
<tr>
<th>Operation</th>
<th>THEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLOCK A</td>
<td>NO</td>
</tr>
<tr>
<td>WLOCK A</td>
<td>NO</td>
</tr>
<tr>
<td>UNLOCK A</td>
<td>US</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>US</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>W</td>
<td>OK</td>
<td>bad</td>
</tr>
</tbody>
</table>

RLOCK  ->  UNLOCK  can enclose a read
WLOCK  ->  UNLOCK  can enclose a write or read
T1
WLOCK A
Read A
A := A + 1
Write A
UNLOCK A

T2
WLOCK A

granted
Read A
A := 2 * A
Write A
UNLOCK A
T1
RLOCK A
Read A

A := A + 1

WLOCK A
wait

T2
RLOCK A
Read A

A := 2*A

WLOCK A

upgrade lock request

upgrade lock request

wait

waits

Deadlock!
T1
WLOCK A
WLOCK B
wait
UNLOCK A
UNLOCK B

T2
WLOCK B
WLOCK A
wait
deadlock
UNLOCK B
UNLOCK A
Deadlock

AND

1. Wait and hold hold some locks while you wait for others
2. Circular chain of waiters

   wait-for graph

   T1 ← T4 → T3 ← T2

3. No pre-emption

We can avoid deadlock by doing at least ONE of:
1. Get all your locks at once
2. Apply an ordering to acquiring locks
3. Allow preemption (for example, use timeout on waits)
Serializability of schedules

T1
Read (A) A:= A-50 Write (A)
Read (B) B:= B+50 Write (B)

T2
Read (A) temp:= A * 0.1 A:= A + temp
Write (A) B:= B - temp Write (B)

Schedule is serializable if effect is the same as a serial schedule

T1 -> T2
A=
B=

T2 -> T1
A=
B=

Ignore arithmetic. What is important is the same sequence of operations.

Conflicts
Read-write
Write-read
Write-write

Two schedules $S_1$, $S_2$ are equivalent if
1. Set of transactions in $S_1$ and $S_2$ are the same
2. For each data item $Q$,
   if in $S_1$, $T_i$ executes Read ($Q$)
   and the value of $Q$ read by $T_i$ was written by $T_j$
   then the same is true in $S_2$
3. For each data item $Q$,
   if in $S_1$, transaction $T_i$ executes last write ($Q$),
   then same is true in $S_2$

A schedule is serializable if it is equivalent to a serial schedule.
Non-fatal errors
   Not recognizing that a schedule is serializable
Fatal errors
   Thinking that a schedule is serializable when it is not
Algorithm: Testing serializability of a schedule
Input: Schedule S for transactions T1,..., Tk
Output: Determination of whether S is serializable, and if so, an equivalent serial schedule.
Method: Create a directed graph G (called a serialization graph)
    Create a node for each transaction and label with transaction ID
    Create an edge for each Ti: UNLOCK Am followed by Tj: LOCK Am
    (where lock modes conflict).
    The edge Ti --> Tj labeled Am (the data item)

If there is a cycle then schedule is non-serializable.
If there is no cycle, then (it is a DAG)
    do a topological sort to get a serial schedule

DAG implies some partial order.
Any total order consistent with the partial order is an equivalent serial schedule.
If no progress is possible, then there is a cycle
T1
LOCK A
UNLOCK A
LOCK B
UNLOCK B

T2
LOCK A
UNLOCK A
LOCK B
UNLOCK B

A

B
ABORT CAN CAUSE CASCADING ROLLBACK

T1	T2
LOCK A
Read A
change A
Write A
UNLOCK A

LOCK A
Read A
change A
Write A
UNLOCK A

LOCK B
Read B
Discover problem
ABORT

Need to undo the change to A
CASCADING ABORT
How to avoid cascading rollback.

Make decision early
Defer commit of dependent transaction
Hold locks until abort no longer possible
2PL  2-Phase Locking

Phase I: All requesting of locks precedes
Phase II: Any releasing of locks

Theorem: Any schedule for 2-phase locked transaction is serializable
Time

Data items

T1
T2
T3
T4
commit
Commit or Abort occurs in between Phase I and Phase II

Effects are permanent

Effects are not visible

Abort → roll back
Read and write locks

Edges:
1. Ti read locks or write locks A
   Tj is next transaction to write lock A
   i ≠ j  edge Ti --> Tj  
   
   R-W, W-W conflict

2. Ti write locks A
   then  ∀ transactions Tk that readlock A after Ti but
   before another
   transaction write locks A
   edge Ti  --> Tk's  
   W-R conflict
Ti  WLOCK A

Tk1  RLOCK A
Tk2  RLOCK A

Tj  WLOCK A
<table>
<thead>
<tr>
<th>N</th>
<th>R</th>
<th>W</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Read</td>
<td>OK</td>
<td>OK</td>
<td>bad</td>
</tr>
<tr>
<td>Write</td>
<td>OK</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Increment</td>
<td>OK</td>
<td>bad</td>
<td>bad</td>
</tr>
</tbody>
</table>

T1

INCR (A)

INCR (A)
READ (B)
WRITE (B)

READ (B)
WRITE (B)
if increment locks

if no increment locks
Authorization in SQL

• File systems identify certain access privileges on files, e.g., read, write, execute.
• In partial analogy, SQL identifies six access privileges on relations, of which the most important are:
  1. **SELECT** = the right to query the relation.
  2. **INSERT** = the right to insert tuples into the relation – may refer to one attribute, in which case the privilege is to specify only one column of the inserted tuple.
  3. **DELETE** = the right to delete tuples from the relation.
  4. **UPDATE** = the right to update tuples of the relation – may refer to one attribute.
Granting Privileges

• You have all possible privileges to the relations you create.
• You may grant privileges to any user if you have those privileges “with grant option.”
  ♦ You have this option to your own relations.

Example

1. Here, Sally can query Sells and can change prices, but cannot pass on this power:
   ```sql
   GRANT SELECT ON Sells,
       UPDATE(price) ON Sells
   TO sally;
   ```

2. Here, Sally can also pass these privileges to whom she chooses:
   ```sql
   GRANT SELECT ON Sells,
       UPDATE(price) ON Sells
   TO sally
   WITH GRANT OPTION;
   ```
Revoking Privileges

- Your privileges can be revoked.
- Syntax is like granting, but REVOKE ... FROM instead of GRANT ... TO.
- Determining whether or not you have a privilege is tricky, involving “grant diagrams” as in text. However, the basic principles are:
  a) If you have been given a privilege by several different people, then all of them have to revoke in order for you to lose the privilege.
  b) Revocation is transitive. if A granted $P$ to $B$, who then granted $P$ to $C$, and then $A$ revokes $P$ from $B$, it is as if $B$ also revoked $P$ from $C$. 