COMP 430
Intro. to Database Systems
Transactions, concurrency, & ACID
Motivating transactions

/* Transfer $100 from one account to another */
UPDATE Account
SET balance = balance – 100
WHERE id = 1234;

UPDATE Account
SET balance = balance + 100
WHERE id = 5678;

Both parties unhappy, since money transfer incomplete.
Motivating transactions

/* Transfer $100 from one account to another */
UPDATE Account
SET balance = balance – 100
WHERE id = 1234;

UPDATE Account
SET balance = balance + 100
WHERE id = 5678;

Want transfer to execute as a single all-or-nothing transaction.
Motivating *transactions* with concurrency

/* Transfer $100 from one account to another */
UPDATE Account
SET balance = balance – 100
WHERE id = 1234;

UPDATE Account
SET balance = balance + 100
WHERE id = 5678;

/* Get paid interest */
UPDATE Account
SET balance = balance * 1.02;
Timelines for account updates

Both allowed.
Timelines for account updates

Also allowed because it corresponds to one of the serial orders – *serializable.*
Timelines for account updates

Not allowed because it doesn’t correspond to one of the serial orders – not *serializable*. 
Motivations *transactions* with errors

If there’s an error, may want to abort transaction’s changes. *Rollback* to state before transaction.

Somewhat expected “errors”:
- Negative balance
- Bad user input

Unexpected errors:
- SQL statement returns error code – `@@ERROR <> 0`
- Inconsistent data found

Not best handled by transaction rollback.
Transaction properties: ACID

<table>
<thead>
<tr>
<th>Atomic</th>
<th>All-or-nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistent</td>
<td>Start &amp; end in consistent state</td>
</tr>
<tr>
<td>Isolated</td>
<td>Serializable</td>
</tr>
<tr>
<td>Durable</td>
<td>Effects permanent after commit</td>
</tr>
</tbody>
</table>
Transaction syntax

BEGIN TRANSACTION transfer;

/* Transfer $100 from one account to another */
UPDATE Account
SET balance = balance – 100
WHERE id = 1234;

UPDATE Account
SET balance = balance + 100
WHERE id = 5678;

COMMIT TRANSACTION transfer;

Can also ROLLBACK a transaction.
Transaction syntax on single statements

/* Get paid interest */
UPDATE Account
SET balance = balance * 1.02;

BEGIN TRANSACTION interest;
/* Get paid interest */
UPDATE Account
SET balance = balance * 1.02;
COMMIT TRANSACTION interest;

Advantages:
• Explicit that want transaction
• Correct even when auto-commit turned off
• Adding code to transaction is less error-prone
• Allows use of other transaction options

Advantage:
• Brief
Understanding some consequences
Why interleave transactions?

After all, interleaving can lead to anomalous outcomes. Need to check serializability.

Real-time performance:
- Don’t want fast transactions to wait on slow ones

Parallel resources:
- Don’t block transactions that use different resources – CPU, disk, …

See COMP 322, re: read-write conflicts
ACID or not?

Want these properties even in the presence of
• Power failures
• User aborting / rollbacks – logging
• Many concurrent users – locks

But still want quick performance.

Some systems relax ACID properties for improve performance.

<table>
<thead>
<tr>
<th>Atomic</th>
<th>All-or-nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistent</td>
<td>Start &amp; end in consistent state</td>
</tr>
<tr>
<td>Isolated</td>
<td>Serializable</td>
</tr>
<tr>
<td>Durable</td>
<td>Effects permanent after commit</td>
</tr>
</tbody>
</table>
Logging

- Basic idea: A list of each data change, written to disk
  - Transaction ID, what was updated, old value, new value
  - Sufficient to undo changes on rollback

- Handled automatically by DBMS
  - Separate from any user-level logging.
Why do we need logging?

Why not update data **only** at end of transaction?

Temporary results stored in memory until transaction done.
  - Transaction could have lots of temporary results.
  - Transaction could take a long time.

What if failure occurs during this end-of-transaction write?
Locking

Standard approach to reserve exclusive access to a variable or other resource.

Steps done by DBMS:
1. Acquire a lock
2. Use the resource
3. Release the lock

Much more info in COMP 322, COMP 421.
What should you do?

Don’t unnecessarily use transactions.
Don’t unnecessarily make transactions too big.
- Large transactions harder to schedule.

Possibly break single operations on large amounts of data into separate transactions.
- One motivation for cursors – a transaction per record instead of per table