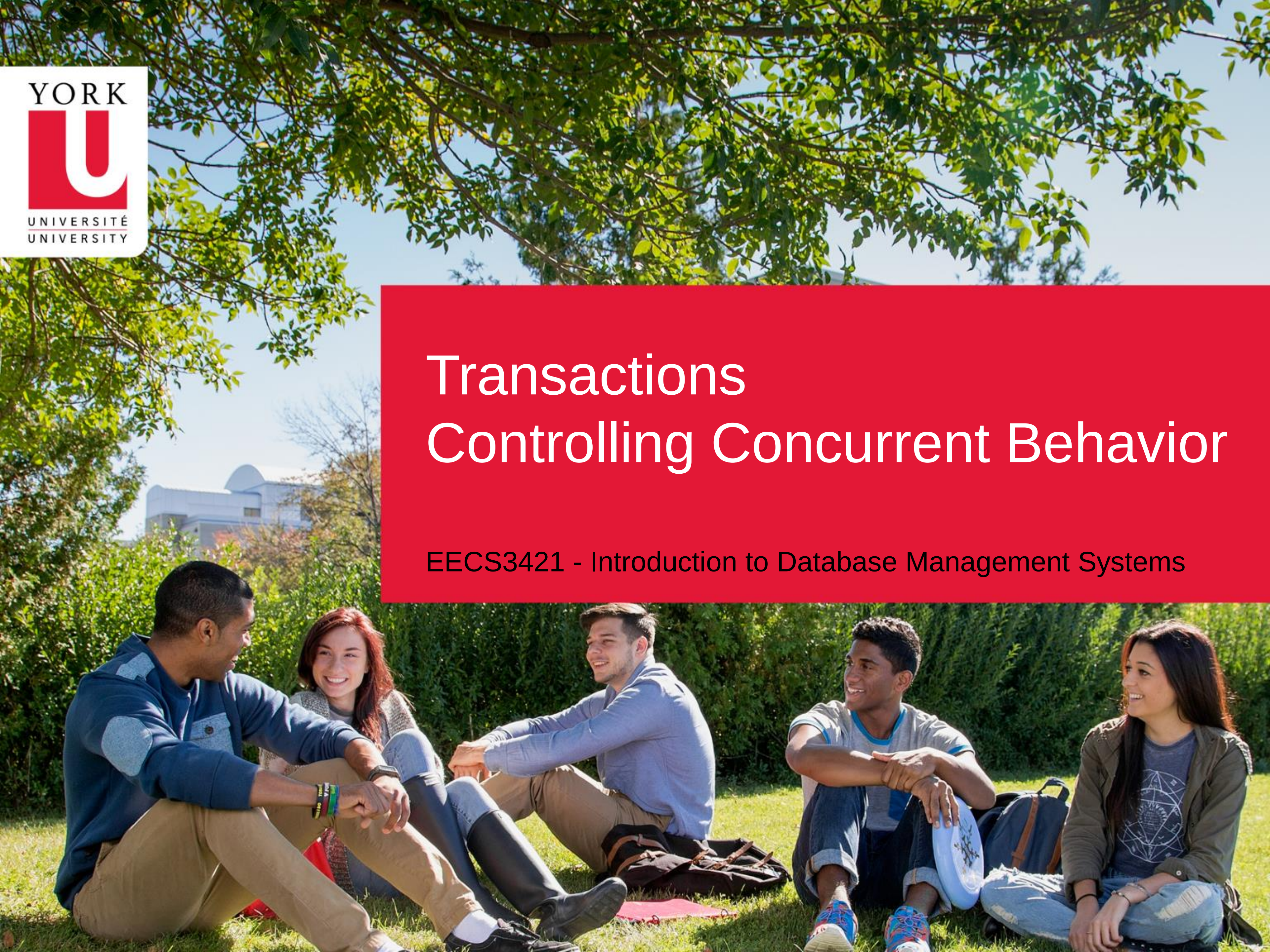

Transactions
Controlling Concurrent Behavior

EECS3421 - Introduction to Database Management Systems

Why Transactions?

- Database systems are normally being accessed by many users or processes at the same time
  - Both queries and modifications
- Unlike operating systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions

Example: Troublesome Interaction

- **Example**: Two people withdraw $100 from the same account using different ATM's at about the same time
  - The DBMS better make sure one account deduction doesn't get lost
- Compare with OS processes: An OS allows two people to edit a document at the same time; If both write, one's changes get lost

Transactions

- **Transaction** = process involving database queries and/or modification
- Normally with some strong properties regarding concurrency
- Formed in SQL from single statements or embedded in code

ACID Transactions

- **ACID transactions** are:
  - **Atomic**: Whole transaction or none is done
  - **Consistent**: Database constraints preserved
  - **Isolated**: It appears to the user as if only one process executes at a time
  - **Durable**: Effects of a transaction survive a crash
- Optional: weaker forms of transactions are often supported as well

COMMIT

- The SQL statement **COMMIT** causes a transaction to complete
  - Its database modifications are now permanent in the database

ROLLBACK

- The SQL statement **ROLLBACK** also causes the transaction to end, but by *aborting*
  - No effects on the database
- Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it

Example: Interacting Processes

- Assume the usual Sells(bar,beer,price) relation, and suppose that Joe’s Bar sells only Bud for $2.50 and Miller for $3.00
- Sally is querying Sells for the highest and lowest price Joe charges
- Joe decides to stop selling Bud and Miller, but to sell only Heineken at $3.50

Sally’s Program

Sally executes the following two SQL statements called (min) and (max) to help us remember what they do

(max) SELECT MAX(price) FROM Sells
WHERE bar = ‘Joe’’s Bar’;

(min) SELECT MIN(price) FROM Sells
WHERE bar = ‘Joe’’s Bar’;

Joe’s Program

At about the same time, Joe executes the following steps: (del) and (ins)

```
(del)    DELETE FROM Sells
         WHERE bar = ‘Joe’’s Bar’;

(ins)    INSERT INTO Sells
         VALUES(‘Joe’’s Bar’, ‘Heineken’, 3.50);
```

Problem: Interleaving of Statements

- Although (max) must come before (min), and (del) must come before (ins), there are no other constraints on the order of these statements, unless we group Sally's and/or Joe's statements into transactions

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min)

Joe's Prices:	{2.50,3.00}	{2.50,3.00}		{3.50}
Statement:	(max)	(del)	(ins)	(min)
Result:	3.00			3.50

- Sally sees MAX < MIN!

Fixing the Problem by Using Transactions

- If we group Sally's statements (max)(min) into one transaction, then she cannot see this inconsistency
- Sally sees Joe's prices at some fixed time
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices

Another Problem: Rollback

- Suppose Joe executes (del)(ins), not as a transaction, but after executing these statements, decides to cancel it and issues a ROLLBACK statement
- If Sally executes her statements after (ins) but before the rollback, she sees a value, 3.50, that never existed in the database

Solution

- If Joe executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT
  - If the transaction executes ROLLBACK instead, then its effects can *never* be seen

Isolation Levels

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time
- Only one level ("serializable") = ACID transactions
- Each DBMS implements transactions in its own way

Choosing the Isolation Level

Within a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL *X*

where *X* =

1. SERIALIZABLE
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED

Serializable Transactions

- If Sally = (max)(min) and Joe = (del)(ins) are each transactions, and Sally runs with isolation level SERIALIZABLE, then she will see the database either before or after Joe runs, but not in the middle

Isolation Level Is Personal Choice

- Your choice, e.g., run serializable, affects only how *you* see the database, not how others see it
- Example: If Joe runs serializable, but Sally doesn't, then Sally might see no prices for Joe's Bar
  - i.e., it looks to Sally as if she ran in the middle of Joe's transaction

Read-Commited Transactions

- If Sally runs with isolation level READ COMMITTED, then she can see only committed data, but not necessarily the same data each time
- Example: Under READ COMMITTED, the interleaving (max)(del)(ins)(min) is allowed, as long as Joe commits
  - Sally sees MAX < MIN

Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time
  - But the second and subsequent reads may see *more* tuples as well

Example: Repeatable Read

- Suppose Sally runs under REPEATABLE READ, and the order of execution is (max)(del)(ins)(min)
  - (max) sees prices 2.50 and 3.00
  - (min) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by (max)

Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never)
- Example: If Sally runs under READ UNCOMMITTED, she could see a price 3.50 even if Joe later aborts