



# THE LECTURE 9

## TRANSACTIONS

# TRANSACTIONS

- Major component of database systems
- Critical for most applications; arguably more so than SQL
- Turing awards to database researchers:
  - Charles Bachman 1973
  - Edgar Codd 1981 for inventing relational dbs
  - Jim Gray 1998 for inventing transactions

# WHY DO WE NEED TRANSACTIONS

- Concurrency control
- Recovery

# CONCURRENCY CONTROL

- Dirty read
  - T reads data written by T' while T' is running
  - Then T' aborts
- Lost update
  - Two tasks T and T' both modify the same data
  - T and T' both commit
  - Final state shows effects of only T, but not of T'
- Inconsistent read
  - One task T sees some but not all changes made by T'

# DIRTY READS

Client 1:

/\* transfer \$100 from account 1 to account 2 \*/

UPDATE Accounts

SET balance = balance + 100

WHERE accountNo = '1111'

X = SELECT balance

FROM Accounts

WHERE accountNo = '2222'

If X < 100 /\* abort .... \*/

then UPDATE Accounts

SET balance = balance - 100

WHERE accountNo = '1111'

Else UPDATE Accounts

SET balance = balance - 100

WHERE accountNo = '2222'

Client 2:

/\* withdraw \$100 from account 1 \*/

X = SELECT balance

FROM Accounts

WHERE accountNo = '1111'

If X > 100

then UPDATE Accounts

SET balance = balance - 100

WHERE accountNo = '1111'

..... Dispense cash .... Cli

# LOST UPDATES

Client 1:

```
UPDATE Product  
SET Price = Price - 1.99  
WHERE pname = 'Gizmo'
```

Client 2:

```
UPDATE Product  
SET Price = Price*0.5  
WHERE pname='Gizmo'
```

Two managers attempt to do a discount.  
Will it work ?

# INCONSISTENT READ

Client 1:

```
UPDATE Products  
SET quantity = quantity + 5  
WHERE product = 'gizmo'
```

```
UPDATE Products  
SET quantity = quantity - 5  
WHERE product = 'gadget'
```

Client 2:

```
SELECT sum(quantity)  
FROM Product
```

What's wrong ?

# PROTECTION AGAINST CRASHES

Client I:

```
UPDATE Products  
SET quantity = quantity + 5  
WHERE product = 'gizmo'
```

```
UPDATE Products  
SET quantity = quantity - 5  
WHERE product = 'gadget'
```



Crash !

What's wrong ?




## DEFINITION

- **A transaction** = one or more operations, which reflects a single real-world transition
  - In the real world, this happened completely or not at all
- Examples
  - Transfer money between accounts
  - Purchase a group of products
  - Register for a class (either waitlist or allocated)
- If grouped in transactions, all problems in previous slides disappear

# TRANSACTIONS IN SQL

- In “ad-hoc” SQL:
  - Default: each statement = one transaction
- In a program:  
START TRANSACTION  
[SQL statements]  
COMMIT or ROLLBACK (=ABORT)



May be omitted:  
first SQL query  
starts txn

## REVISED CODE

```
Client 1: START TRANSACTION
UPDATE Product
SET Price = Price - 1.99
WHERE pname = 'Gizmo'
COMMIT
```

```
Client 2: START TRANSACTION
UPDATE Product
SET Price = Price*0.5
WHERE pname='Gizmo'
COMMIT
```

Now it works like a charm

# TRANSACTION PROPERTIES

## ACID

- **A**tomic
  - State shows either all the effects of txn, or none of them
- **C**onsistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- **I**solated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **D**urable
  - Once a txn has committed, its effects remain in the database

# ACID:ATOMICITY

- Two possible outcomes for a transaction
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made
- That is, transaction's activities are all or nothing

## ACID: CONSISTENCY

- The state of the tables is restricted by integrity constraints
  - Account number is unique
  - Stock amount can't be negative
  - Sum of *debits* and of *credits* is 0
- Constraints may be explicit or implicit
- How consistency is achieved:
  - Programmer makes sure a txn takes a consistent state to a consistent state
  - The system makes sure that the txn is atomic

## ACID: ISOLATION

- A transaction executes concurrently with other transaction
- Isolation: the effect is as if each transaction executes in isolation of the others

## ACID: DURABILITY

- The effect of a transaction must continue to exist after the transaction, or the whole program has terminated
- Means: write data to disk



# ROLLBACK

- If the app gets to a place where it can't complete the transaction successfully, it can execute ROLLBACK
- This causes the system to “abort” the transaction
  - The database returns to the state without any of the previous changes made by activity of the transaction

# REASONS FOR ROLLBACK

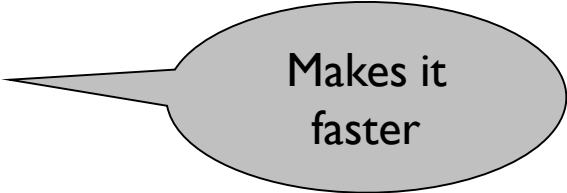
- User changes their mind (“ctl-C”/cancel)
- Explicit in program, when app program finds a problem
  - e.g. when qty on hand < qty being sold
- System-initiated abort
  - System crash
  - Housekeeping
    - e.g. due to timeouts

# READ-ONLY TRANSACTIONS

Client 1:    **START TRANSACTION**  
              **INSERT INTO** SmallProduct(name, price)  
              **SELECT** pname, price  
              **FROM** Product  
              **WHERE** price <= 0.99

**DELETE** Product  
                      **WHERE** price <=0.99  
              **COMMIT**

Client 2:    **SET TRANSACTION READ ONLY**  
              **START TRANSACTION**  
              **SELECT** count(\*)  
              **FROM** Product  
  
              **SELECT** count(\*)  
              **FROM** SmallProduct  
              **COMMIT**



Makes it  
faster

# ISOLATION LEVELS IN SQL

1. “Dirty reads”

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. “Committed reads”

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. “Repeatable reads”

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions (default):

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

# ISOLATION LEVEL: DIRTY READS

Plane seat  
allocation

What can go  
wrong ?

What can go  
wrong if only  
the function  
AllocateSeat  
modifies Seat ?

```
function AllocateSeat( %request)
```

```
SET ISOLATION LEVEL READ UNCOMMITTED
```

```
START TRANSACTION
```

```
Let x = SELECT Seat.occupied  
FROM Seat  
WHERE Seat.number = %request
```

```
If (x == 1) /* occupied */ ROLLBACK
```

```
UPDATE Seat  
SET occupied = 1  
WHERE Seat.number = %request
```

```
COMMIT
```

Are dirty reads  
OK here ?

What if we  
switch the  
two updates ?

```
function TransferMoney( %amount, %acc1, %acc2)
```

```
START TRANSACTION
```

```
Let x =  SELECT Account.balance  
        FROM Account  
        WHERE Account.number = %acc1
```

```
If (x < %amount) ROLLBACK
```

```
UPDATE Account  
SET balance = balance+%amount  
WHERE Account.number = %acc2
```

```
UPDATE Account  
SET balance = balance-%amount  
WHERE Account.number = %acc1
```

```
COMMIT
```

# ISOLATION LEVEL: READ COMMITTED

Stronger than  
READ UNCOMMITTED

It is possible  
to read twice,  
and get different  
values

## SET ISOLATION LEVEL READ COMMITTED

```
Let x =  SELECT Seat.occupied  
        FROM Seat  
        WHERE Seat.number = %request
```

```
/* ..... More stuff here .... */
```

```
Let y =  SELECT Seat.occupied  
        FROM Seat  
        WHERE Seat.number = %request
```

```
/* we may have  $x \neq y$  ! */
```

## ISOLATION LEVEL: REPEATABLE READ

Stronger than  
READ COMMITTED

May see incompatible  
values:

another txn transfers  
from acc. 55555 to  
77777

SET ISOLATION LEVEL REPEATABLE READ

```
Let x =  SELECT Account.amount
         FROM Account
         WHERE Account.number = '55555'
```

```
/* ..... More stuff here .... */
```

```
Let y =  SELECT Account.amount
         FROM Account
         WHERE Account.number = '77777'
```

```
/* we may have a wrong x+y ! */
```