Introduction to Database Systems
CSE 444

Lecture 8: Transactions in SQL
Where We Are

• What we have already learned
  – Relational model of data
  – Data manipulation language: SQL
  – Views and constraints
  – Database design (E/R diagrams & normalization)

• But what if I want to update my data?
• Today: transactions in SQL (Sec. 6.6)
  – Old edition: Sec. 8.6
Transactions

- **Problem**: An application must perform several writes and reads to the database, as a unit

- **Solution**: multiple actions of the application are bundled into one unit called *Transaction*

- **Very powerful concept**
  - *Database transactions* (that’s where they started)
  - *Transaction monitors*
  - *Transactional memory*
Turing Awards to Database Researchers

• Charles Bachman 1973 for CODASYL

• Edgar Codd 1981 for relational databases

• Jim Gray 1998 for transactions
The World Without Transactions

• Just write applications that talk to databases

• Rely on operating systems for scheduling, and for concurrency control

• What can go wrong?
  – Several famous anomalies
  – Other anomalies are possible (but not famous)
Lost Updates

Two people attempt to rent two movies for Fred, from two different terminals. What happens?
Unrepeatable Read

Client 1: rent-a-movie
\[
x = \text{SELECT} \ \text{rentals} \ \text{FROM} \ \text{Cust} \\
\quad \text{WHERE} \ \text{cname} = \text{‘Fred’}
\]

if (x < 5)
\{ UPDATE Cust \\
\quad \text{SET} \ \text{rentals} = \ \text{rentals} + 1 \\
\quad \text{WHERE} \ \text{cname} = \text{‘Fred’}
\}
else println(“Denied!”)

Client 2: rent-a-movie
\[
x = \text{SELECT} \ \text{rentals} \ \text{FROM} \ \text{Cust} \\
\quad \text{WHERE} \ \text{cname} = \text{‘Fred’}
\]

if (x < 5)
\{ UPDATE Cust \\
\quad \text{SET} \ \text{rentals} = \ \text{rentals} + 1 \\
\quad \text{WHERE} \ \text{cname} = \text{‘Fred’}
\}
else println(“Denied!”)

What’s wrong?
Inconsistent Read

Client 1: move from gizmo → gadget

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

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

Client 2: inventory…. 

SELECT sum(quantity)
FROM Product

What’s wrong?
Inconsistent Read

Client 1: rent-two-movies
x = SELECT rentals FROM Cust
   WHERE cname = 'Fred'

if (x < 4) { /* movie 1...*/
   UPDATE Cust
   SET rentals = rentals + 1
   WHERE cname = 'Fred'

   /* ...and movie 2 */
   UPDATE Cust
   SET rentals = rentals + 1
   WHERE cname = 'Fred'
}
else println("Denied!")

Client 2: rent-a-movie
x = SELECT rentals FROM Cust
   WHERE cname = 'Fred'

if (x < 5)
   { UPDATE Cust
     SET rentals = rentals + 1
     WHERE cname = 'Fred' }
else println("Denied!")

What’s wrong?
Dirty Reads

Client 1: transfer $100 acc1 → acc2
X = Account1.balance
Account2.balance += 100

If (X>=100) Account1.balance -=100
else { /* rollback ! */
    account2.balance -= 100
    println("Denied !")
}

Client 1: transfer $100 acc2 → acc3
Y = Account2.balance
Account3.balance += 100

If (Y>=100) Account2.balance -=100
else { /* rollback ! */
    account3.balance -= 100
    println("Denied !")
}

What’s wrong?
Some Famous anomalies

- **Dirty read (Write-Read conflict)**
  - T reads data written by T’ while T’ has not committed
  - What can go wrong: T’ writes more data (which T has already read) or T’ aborts
  - Inconsistent read: T sees some but not all changes made by T’

- **Unrepeatable read (Read-Write conflict)**
  - T reads the same value twice and gets two different results

- **Lost update (Write-Write conflict)**
  - 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’
Protection against crashes

Client 1:

\[
\text{UPDATE Accounts} \\
\text{SET balance} = \text{balance} - 500 \\
\text{WHERE name} = \text{‘Fred’}
\]

\[
\text{UPDATE Accounts} \\
\text{SET balance} = \text{balance} + 500 \\
\text{WHERE name} = \text{‘Joe’}
\]

Crash!

What’s wrong?
Enter Transactions

• Concurrency control
  – The famous anomalies and more…

• Recovery
Definition

• **A transaction** = one or more operations, which reflect a single real-world transition
  – Happens completely or not at all

• Examples
  – Transfer money between accounts
  – Rent a movie; return a rented movie
  – Purchase a group of products
  – Register for a class (either waitlisted or allocated)

• By using transactions, all previous problems disappear
Transactions in Applications

START TRANSACTION

[SQL statements]

COMMIT  or  ROLLBACK (=ABORT)

May be omitted: first SQL query starts txn
Transactions in Ad-hoc SQL

- Default: each statement = one transaction
Revised Code

Client 1: rent-a-movie

```sql
START TRANSACTION
x = SELECT rentals
    FROM Cust
    WHERE cname= 'Fred'

if (x < 5)
    { UPDATE Cust
        SET rentals= rentals + 1
        WHERE cname= 'Fred'
    }
else println("Denied !")

COMMIT
```

Client 2: rent-a-movie

```sql
START TRANSACTION
x = SELECT rentals
    FROM Cust
    WHERE cname= 'Fred'

if (x < 5)
    { UPDATE Cust
        SET rentals= rentals + 1
        WHERE cname= 'Fred'
    }
else println("Denied !")

COMMIT
```

Now it works like a charm
Revised Code

Client 1: transfer $100  acc1→ acc2
START TRANSACTION
X = Account1.balance;    Account2.balance += 100

If (X>=100) { Account1.balance -=100;  COMMIT }  
else {println(“Denied !”); ROLLBACK}

Client 1: transfer $100  acc2→ acc3
START TRANSACTION
X = Account2.balance;    Account3.balance += 100

If (X>=100) { Account2.balance -=100;  COMMIT }  
else {println(“Denied !”); ROLLBACK}
Using Transactions

Very easy to use:

• START TRANSACTION
• COMMIT
• ROLLBACK

But what EXACTLY do they mean?

• Popular culture: ACID
• Underlying theory: serializability
Transaction Properties
ACID

- **Atomic**
  - State shows either all the effects of txn, or none of them
- **Consistent**
  - Txn moves from a state where integrity holds, to another where integrity holds
- **Isolated**
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **Durable**
  - 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 tnx 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
• App can then decide to retry or abandon or…
Reasons for Rollback

- User changes their mind ("ctl-C"/cancel)
- Explicit in program, when app program finds a problem
  - E.g. when the # of rented movies > max # allowed
  - Use it freely in Project 2!!
- System-initiated abort
  - System crash
  - Housekeeping, e.g. due to timeouts