Lecture 14: Transactions in SQL

Wednesday, February 8, 2006

Overview

• Midterm review

• Chapter 8.6
• Note: this is an easy introduction to transactions; more details when we discuss implementations
Midterm

- Friday, 11:30, this room (in class)
  - 50’

- Open book
  - Notes, books, lectures, everything you want
  - But no computers

Midterm

- SQL

- E/R Diagrams

- Functional Dependencies

- XML/Xpath/XQuery
SQL

- Know the basics: SFW, GROUP-BY, HAVING…
- When are two queries equivalent?
  - Eliminating subqueries
  - Eliminating joins
  - Be aware of duplicates
- Insert/delete, especially more than one tuple
- Constraints in SQL

E/R Diagrams

- Good design (don’t make stupid mistakes)
- Translation to relations
  - Many-many v.s. many-one relationships
- Subtleties:
  - Inheritance
  - Union types
  - Weak entity sets
Functional Dependencies

• Know the definition of $X \rightarrow Y$
  – Does a given table satisfy $X \rightarrow Y$?
• Understand inference
  – If $A \rightarrow B$, $B \rightarrow C$, does it follow that $C \rightarrow A$?
  Why? Why not?
• Understand closure: $X^+$
• Understand BCNF (no 3NF)

XML

• Basics in XPath and Xquery
• In what sense is XML “semistructured”? 
Midterm

How to prepare:

• Read lecture notes
• Read from the textbook
• Review the homeworks
• Try to solve exercise (book, past exams)
• Make sure you understand

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

Multiple users: single statements

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?
Multiple users: multiple statements

Client 1:  
\[
\text{INSERT INTO SmallProduct(name, price) SELECT pname, price FROM Product WHERE price } \leq 0.99
\]
\[
\text{DELETE Product WHERE price } \leq 0.99
\]

Client 2:  
\[
\text{SELECT count(*) FROM Product}
\]
\[
\text{SELECT count(*) FROM SmallProduct}
\]

What’s wrong?

Protection against crashes

Client 1:  
\[
\text{INSERT INTO SmallProduct(name, price) SELECT pname, price FROM Product WHERE price } \leq 0.99
\]
\[
\text{DELETE Product WHERE price } \leq 0.99
\]

What’s wrong?

Crash!
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

- 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 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

Famous anomalies

- **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’
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

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
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

Are dirty reads OK here?

What if we switch the two updates?

Isolation Level: Read Committed

Stronger than READ UNCOMMITTED

It is possible to read twice, and get different values

SET ISOLATION LEVEL READ COMMITED

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 ≠ 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 = '555555'

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

Let y = SELECT Account.amount
       FROM Account
       WHERE Account.number = '777777'

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

Isolation Level: Serializable

Strongest level

Default

```
SET ISOLATION LEVEL SERIALIZABLE

. . .
```