Lecture 8: Transactions in SQL

Wednesday, April 14, 2010
Outline

• Transactions in SQL (6.6)
Transactions

• The problem: An application must perform several writes and reads to the database, as a unity
• Solution: multiple actions of the application are bundled into one unit called Transaction
Transactions

• 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?
  – Three 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?
Inconsistent Read (1/2)

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

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

What’s wrong?

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

\[
\text{if} (x < 5) \\
\{ \text{UPDATE Cust} \\
\text{SET rentals} = \text{rentals} + 1 \\
\text{WHERE cname= 'Fred'} \} \\
\text{else println("Denied !")}
\]
Inconsistent Read (2/2)

Client 1: move from gizmo \(\rightarrow\) gadget

\[
\text{UPDATE Products} \\
\text{SET quantity} = \text{quantity} + 5 \\
\text{WHERE product = 'gizmo'}
\]

Client 2: inventory…. 

\[
\text{UPDATE Products} \\
\text{SET quantity} = \text{quantity} - 5 \\
\text{WHERE product = 'gadget'}
\]

\[
\text{SELECT sum(quantity)} \\
\text{FROM Product}
\]
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 !")
}

What’s wrong ?

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 !")
}
The Three Famous anomalies

• Dirty read
  – T reads data written by T’ while T’ has not committed
  – What can go wrong: T’ write more data (which T has already read), or 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’
Protection against crashes

Client 1:

```
UPDATE Accounts
SET balance = balance - 500
WHERE name = 'Fred'

UPDATE Accounts
SET balance = balance + 500
WHERE name = 'Joe'
```

Crash!
Transactions

Do two things:

• Concurrency control

• Recovery
Definition

• A **transaction** = one or more operations, which reflects 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

Dan Suciu -- 444 Spring 2010
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
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
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
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: Isolation

• A transaction executes concurrently with other transaction

• Isolation: the effect is as if each transaction executes in isolation of the others
ACID: Consistency

• The database satisfies 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:
  – Applications preserve consistency, assuming they run atomically, and they run in isolation
  – The system ensures atomicity and isolation
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

• Sometimes also means recovery
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 the # of rented movies > max # allowed
  – Use it freely in Project 2 !!
• System-initiated abort
  – System crash
  – Housekeeping, e.g. due to timeouts