CS145 Lecture Notes #11
SQL Transactions

Transactions are motivated by two of the properties of a DBMS (discussed way back in Lecture Notes #1):

- *Multiuser* access: most database systems run as servers where multiple clients can simultaneously operate on the same database
- *Safe* from system crashes

Example schema:

```
CREATE TABLE Account (number INTEGER PRIMARY KEY,
                      name CHAR(30),
                      balance FLOAT);
```

Example: concurrent withdrawals

```
-- let user input account number
SELECT balance INTO myBalance
FROM Account WHERE number = myNumber;
-- display current balance
-- let user input amount of withdrawal
myBalance := myBalance - withdrawal;
IF (myBalance >= 0) THEN
  UPDATE Account SET balance = myBalance
  WHERE number = myNumber;
END IF;
```

- Homer withdraws \$100 from account #123
- Marge withdraws \$50 from account #123
- Initial balance = \$400, final balance = ???

$\leadsto$ Interleaving concurrent operations may cause problems
$\leadsto$ But interleaving operations on different accounts is okay

Example: balance transfer

```
UPDATE Account SET balance = balance - 100.00
WHERE number = 123;
UPDATE Account SET balance = balance + 100.00
WHERE number = 456;
```

- DBMS crashes in the middle—what now?
- DBMS buffers pages and updates them in memory for efficiency; before they are written back to disk, DBMS crashes—what now?

Solution: transactions!

A *transaction* is a sequence of one or more SQL operations (interactive or embedded) treated as one unit:

- Transaction begins automatically when the client issues its first SQL command
- Transaction ends (and new one begins) when the client issues the command `COMMIT`
- Transactions obey the "ACID properties": Atomicity, Consistency, Isolation, Durability

ACID Properties

Isolation

- Transactions must behave as if they were executed in isolation from each other
- Isolation is obtained through *serializability*: operations within transactions may be interleaved (for efficiency), but execution must be equivalent to some serial order

$\rightsquigarrow$ Solves the problem of concurrent withdrawals

- How is this guarantee achieved?
  – Take CS245!
  – Locking, multiversion concurrency control, etc.

Durability

- If the DBMS crashes after a transaction commits, all effects of the transaction must remain in the database
  – Sounds obvious, but every DBMS manipulates data in memory

$\rightsquigarrow$ Solves the problem of system crash after balance transfer

- How is this guarantee achieved?
  – Take CS245!
  – Logging, and various other mechanisms

Atomicity

- Each transaction's operations are execute all-or-nothing, never left "half-done"
  – If the DBMS crashes before a transaction commits, no effects of this transaction should remain in the database—the transaction may start over when the DBMS comes back up
  – If an error or exception occurs during a transaction, partial effects of the transaction must be undone

- Transaction rollback (a.k.a. transaction abort):
  - Undoes partial effects of a transaction
  - May be system-initiated or client-initiated

    Example of client-initiated rollback:

```
-- get user input and execute SQL commands
-- confirm results with user
IF (confirmed) THEN COMMIT;
ELSE ROLLBACK;
END IF;
```

⇝ Solves the problem of system crash during balance transfer

- How is this guarantee achieved?
  - Take CS245!
  - Logging

Consistency

- Assume all database constraints are true at the start of every transaction, they should remain true at the end of every transaction
- How is this guarantee achieved?
  - Guaranteed by the transactions themselves and/or constraints and triggers declared in the DBMS

Isolation Levels

Serializable

- Strongest isolation level—SQL default

⇝ Weaker isolation levels increase performance by eliminating overhead and allowing higher degrees of concurrency

Read Uncommitted

- A data item is *dirty* if it is written by an uncommitted transaction
- Problem of reading dirty data written by another uncommitted transaction: what if that transaction eventually aborts?

Example: wrong average

⇝ `T2` may only care about approximate average — dirty reads okay

```
-- T1.begin:
  -- T1.step1:
  UPDATE Account
  SET balance = balance - 200.00
  WHERE number = 123;
-- T1.abort:
ROLLBACK;
```

```
-- T2.begin:
  -- T2.step1:
  SELECT AVG(balance)
  FROM Account;
-- T2.commit:
COMMIT;
```

Read Committed

- A read-committed transaction cannot read dirty data written by other uncommitted transactions
- But read-committed is still not necessarily serializable

Example: different averages

```
-- T1.begin:
  -- T1.step1:
  UPDATE Account
  SET balance = balance - 200.00
  WHERE number = 123;
-- T1.commit:
COMMIT;
```

```
-- T2.begin:
  -- T2.step1:
  SELECT AVG(balance)
  FROM Account;
  -- T2.step2:
  SELECT AVG(balance)
  FROM Account;
-- T2.commit:
COMMIT;
```

Repeatable Read

- In a repeatable-read transaction, if a tuple is read once, then the same tuple must be retrieved again if the query is repeated

$\rightsquigarrow$ Possible implementation: lock every tuple read by the transaction

Example: same average

```
-- T1.begin:
  -- T1.step1:
  UPDATE Account
  SET balance = balance - 200.00
  WHERE number = 123;
-- T1.commit:
COMMIT;
```

```
-- T2.begin:
  -- T2.step1:
  SELECT AVG(balance)
  FROM Account;
  -- T2.step2:
  SELECT AVG(balance)
  FROM Account;
-- T2.commit:
COMMIT;
```

- But repeatable-read is still not necessarily serializable!
- A repeatable-read transaction may see *phantom* tuples, which are inserted by other transactions while this transaction is executing

Example: different averages

```
-- T1.begin:
  -- T1.step1:
  INSERT INTO Account
  VALUES(456, 'Apu', 5000);
-- T1.commit:
COMMIT;
```

```
-- T2.begin:
  -- T2.step1:
  SELECT AVG(balance)
  FROM Account;
  -- T2.step2:
  SELECT AVG(balance)
  FROM Account;
-- T2.commit:
COMMIT;
```

Summary

```
SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED |
                                  READ COMMITTED |
                                  REPEATABLE READ |
                                  SERIALIZABLE };
```

From weakest to strongest:

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted			
Read Committed			
Repeatable Read			
Serializable			

It is also possible to tell DBMS that a transaction will not perform any writes:

- `SET TRANSACTION READ ONLY;`
- Many, many transactions and applications fall into this category
- DBMS will optimize concurrency control accordingly
  Example: if there are ten read-only transactions and no other transactions, what does the DBMS need to do to guarantee serializability?