Transactions in SQL

Outline

1. Motivation
2. Definition and Properties
3. Examples of Bad Interactions
4. Isolation Levels
5. JDBC

Why Transactions?

- Database systems are usually accessed by several users or processes at the same time.
  - Both queries and modifications.
- Like operating systems, which support interaction of processes, a DMBS needs to keep processes from troublesome interactions.

Example: Bad Interaction

- You and your domestic partner each take $100 from different ATM's at about the same time.
  - The DBMS better make sure one account deduction doesn't get lost.
- Compare: An OS allows two people to edit a document at the same time. It provides locks to prevent the loss of concurrent updates.

Why Transactions? (cont')

- Transactions are also required in order to preserve the consistency of modifications to the database in the event of crashes.
- Example: money transfer between accounts #123 and #456

update set bal = bal–100 where id = 123;

update set bal = bal+100 where id = 456;

Why Transactions? (cont')

- Money "vanishes" if the database or the user process crashes right after the first update...
- unless the two updates are executed in a single transaction: in the event of a crash the DBMS will abort the transaction and undo the first update.

Outline

1. Motivation
2. Definition and Properties
3. Examples of Bad Interactions
4. Isolation Levels
5. JDBC

Transactions

- *Transaction* = process involving database queries and/or modification.
- Normally with some strong properties regarding concurrency.
- Formed in SQL from single statements or explicit programmer control.

ACID Transactions

- *ACID transactions* are:
  - *Atomic* : Whole transaction or none is done.
  - *Consistent* : Database consistency preserved.
  - *Isolated* : It appears to the user as if only one process executes at a time, i.e. serially.
  - *Durable* : Effects of a process survive a crash.

ACID Transactions (2)

- Atomicity is the property we need in the event of crashes
- Isolation is the property we need when concurrent accesses occur
  - Note: several *isolation levels* are defined (see later)
- Consistency must be ensured by the programmer
  - e.g. withdraw and deposit the **same** amount

Transaction Control

- The SQL statement START TRANSACTION starts a new transaction
- You can end the transaction with either COMMIT or ROLLBACK

- These statements belong to the Transaction Control Language (TCL) of SQL

COMMIT

- The SQL statement COMMIT causes a transaction to complete.
  - Its database modifications are now permanent in the database.
  - Before COMMIT, the modifications were only tentative.

ROLLBACK

- The SQL statement ROLLBACK also causes the transaction to end, but by *aborting*.
  - No effects on the database: modifications are undone
- Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it.

Implicit Start of a Transaction

- A transaction is implicitly started on the first statement issued after:
  - opening a new connection to the DBMS.
  - the last transaction ends (i.e. commits or rolls back).

Implicit Commit of a Transaction

- A transaction is implicitly committed when:
  - a new transaction is started with START TRANSACTION…
  - a DDL statement like CREATE TABLE… is issued

Implicit Rollback of a Transaction

- A transaction is automatically aborted when:
  - a constraint is violated or a severe error occurs (like division by 0)
  - the connection to the database is closed abruptly without committing

Autocommit mode

- Each connection to the database has a property named Autocommit
- Autocommit is on by default whenever a new connection is opened
- Autocommit can be set to off using the following statement:

```
set autocommit=0;
```

Autocommit mode (2)

- When autocommit is off, transactions are managed by issuing START TRANSACTION, COMMIT, and ROLLBACK

- When on, the DBMS automatically executes a "COMMIT" right after each SQL statement

Outline

1. Motivation
2. Definition and Properties
3. Examples of Bad Interactions
4. Isolation Levels
5. JDBC

Example: Interacting Processes

- Assume the usual Sells(bar,beer,price) relation, and suppose that Joe's Bar sells only Bud for $2.50 and Miller for $3.00.
- Sally is querying Sells for the highest and lowest price Joe charges.
- Joe decides to stop selling Bud and Miller, but to sell only Heineken at $3.50.

Sally's Program

- Sally executes the following two SQL statements called (min) and (max) to help us remember what they do.

(max)
```
SELECT MAX(price) FROM Sells
WHERE bar = 'Joe''s Bar';
```

(min)
```
SELECT MIN(price) FROM Sells
WHERE bar = 'Joe''s Bar';
```

Joe's Program

- At about the same time, Joe executes the following steps: (del) and (ins).

(del) DELETE FROM Sells
WHERE bar = 'Joe"s Bar';

(ins) INSERT INTO Sells
VALUES('Joe"s Bar', 'Heineken', 3.50);

Interleaving of Statements

- Statements are executed in the order they are issued: (max) must come before (min), and (del) must come before (ins)
- But there are no other constraints on the order of these statements
- Unless we group Sally's and/or Joe's statements into transactions.

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min).

Joe's Prices:	{2.50,3.00}				{3.50}
Statement:		(max)	(del)	(ins)	(min)
Result:		3.00			3.50

- Sally sees MAX < MIN!

Fixing the Problem by Using Transactions

- If we group Sally's statements (max)(min) into one transaction, then she cannot see this inconsistency.
- She sees Joe's prices at some fixed time.
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.

Another Problem: Rollback

- Suppose Joe executes (del)(ins), but after executing these statements, thinks better of it and issues a ROLLBACK statement.
- If Sally executes her statements after (ins) but before the rollback, she sees a value, 3.50, that never existed in the database.

Solution

- If Joe executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT.
  - If the transaction executes ROLLBACK instead, then its effects can *never* be seen.

Outline

1. Motivation
2. Definition and Properties
3. Examples of Bad Interactions
4. Isolation Levels
5. JDBC

Definition

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time.
- Each DBMS implements transactions in its own way:
  - pessimistic concurrency control, with locks (the most common way)
  - optimistic concurrency control, with timestamps

Serializable and Concurrency

- ACID transactions use the highest isolation level: serializable
  - Transactions *appears* as if executed one after another (hence the name)
- Often, the only way to ensure this property is to *make* transactions execute one after another, using locks
  - Consistency is guaranteed, but at the expense of concurrency

The Rationale for Isolation Levels

- Question: can we trade isolation for concurrency? Answer: yes, we can lower isolation level to achieve more concurrency
  - some transactions do not need the serializable level, e.g. read transactions
  - and thus can achieve better throughput by choosing a lower isolation level

Default Isolation Level

- Each DBMS defines its own default isolation level. This default is usually lower than Serializable, because Serializable degrades performances too much.
- SET TRANSACTION ISOLATION LEVEL allows you to choose the level you need. Different transactions may run at different isolation levels.

Choosing the Isolation Level

- At the beginning of a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL *X*

where *X* =

1. SERIALIZABLE (ACID transaction)
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED

Choosing the Isolation Level (2)

- The isolation level of a transaction must be set before executing the first statement of the transaction.

- It cannot be changed afterward.

Serializable Transactions

- If Sally = (max)(min) and Joe = (del)(ins) are each transactions, and Sally runs with isolation level SERIALIZABLE, then she will see the database either before or after Joe runs, but not in the middle.

Isolation Level Is Personal Choice

- Your choice, e.g., run serializable, affects only how *you* see the database, *not* how *others* see it.
- Example: If Joe Runs serializable, but Sally doesn't, then Sally might see no prices for Joe's Bar.
  - i.e., it looks to Sally as if she ran in the middle of Joe's transaction.

Read-Commited Transactions

- If Sally runs with isolation level READ COMMITTED, then she can see only committed data, but not necessarily the same data each time.
- Example: Under READ COMMITTED, the interleaving (max)(del)(ins)(min) is allowed, as long as Joe commits.
  - Sally sees MAX < MIN.

Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time.
  - But the second and subsequent reads may see *more* tuples as well.

Example: Repeatable Read

- Suppose Sally runs under REPEATABLE READ, and the order of execution is (max)(del)(ins)(min).
  - (max) sees prices 2.50 and 3.00.
  - (min) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by (max).

Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never).
- Example: If Sally runs under READ UNCOMMITTED, she could see a price 3.50 even if Joe later aborts.

SQL Read Phenomena

- The previous examples illustrate what SQL calls read phenomena that can affect a transaction:
  - dirty reads: the transaction reads data that another transaction has modified but not committed yet
  - non-repeatable (or fuzzy) reads: the transaction reads the same row twice, but don't get the same values for that row

SQL Phenomena (2)

- phantom reads: the transaction executes two identical queries, but the collection of rows returned by the second query is different from the first.
  - For example, the second query returns all the tuples returned by the first query, plus additional tuples that were inserted (and committed) by another transaction in the meantime

Isolation Levels VS Phenomena

	Dirty Reads	Fuzzy Reads	Phantom Reads
Read Uncommitted	possible	possible	possible
Read Committed	not possible	possible	possible
Repeatable Read	not possible	not possible	possible
Serializable	not possible	not possible	not possible

Note: Serializable is more than the absence of undesirable phenomena: it ensures that transactions appear to execute one at a time

Outline

1. Motivation
2. Definition and Properties
3. Examples of Bad Interactions
4. Isolation Levels
5. JDBC

Transaction Control and JDBC

- With JDBC, the following TCL statements must NOT be executed using the execute or executeUpdate methods:
  - START TRANSACTION
    - start transactions implicitly instead
  - SET TRANSACTION ISOLATION LEVEL
    - use the Connection.setTransactionIsolation(int) method instead

Transaction Control and JDBC (2)

- COMMIT
  - use the Connection.commit() method instead
- ROLLBACK
  - use the Connection.rollback() method instead

SQLException Handling

- Many JDBC methods throw an SQLException when something goes wrong
  - e.g a key constraint is violated, the connection to the database is lost, etc.
- If an SQException occurs, your code must call Connection.rollback() to make sure the current transaction is aborted