

SQL: Transactions

CPS 116
Introduction to Database Systems

Announcements (October 2)

- ❖ Project milestone #1 due in 1½ weeks
 - Come to my office hours if you want to chat about project ideas
- ❖ Midterm in class this Thursday
 - Open book, open notes
 - Format similar to the sample midterm
 - Covers everything up to today's lecture
 - Emphasizes materials exercised in homeworks
- ❖ Sample solutions to sample midterm and Homework #2 available

Transactions

- ❖ A transaction is a sequence of database operations with the following properties (ACID):
 - Atomic: Operations of a transaction are executed all-or-nothing, and are never left "half-done"
 - Consistency: Assume all database constraints are satisfied at the start of a transaction, they should remain satisfied at the end of the transaction
 - Isolation: Transactions must behave as if they were executed in complete isolation from each other
 - Durability: If the DBMS crashes after a transaction commits, all effects of the transaction must remain in the database when DBMS comes back up

SQL transactions

- ❖ A transaction is automatically started when a user executes an SQL statement
- ❖ Subsequent statements in the same session are executed as part of this transaction
 - Statements see changes made by earlier ones in the same transaction
 - Statements in other concurrently running transactions do not see these changes
- ❖ COMMIT command commits the transaction
 - Its effects are made final and visible to subsequent transactions
- ❖ ROLLBACK command aborts the transaction
 - Its effects are undone

Fine prints

- ❖ Schema operations (e.g., CREATE TABLE) implicitly commit the current transaction
 - Because it is often difficult to undo a schema operation
- ❖ Many DBMS support an AUTOCOMMIT feature, which automatically commits every single statement
 - For DB2:
 - db2 command-line processor turns it on by default
 - You can turn it off with option +c
 - More examples to come when we cover database API's

Atomicity

- ❖ Partial effects of a transaction must be undone when
 - User explicitly aborts the transaction using ROLLBACK
 - E.g., application asks for user confirmation in the last step and issues COMMIT or ROLLBACK depending on the response
 - The DBMS crashes before a transaction commits
- ❖ Partial effects of a modification statement must be undone when any constraint is violated
 - However, only this statement is rolled back; the transaction continues
- ❖ How is atomicity achieved?
 - Logging (to support undo)

Durability

7

- ❖ Effects of committed transactions must survive DBMS crashes
- ❖ How is durability achieved?
 - Forcing all changes to disk at the end of every transaction?
 - Too expensive: DBMS manipulates data in memory
 - Logging (to support redo)

Consistency

8

- ❖ Consistency of the database is guaranteed by constraints and triggers declared in the database and/or transactions themselves
 - Whenever inconsistency arises, abort the statement or transaction, or (with deferred constraint checking or application-enforced constraints) fix the inconsistency within the transaction

Isolation

9

- ❖ Transactions must appear to be executed in a serial schedule (with no interleaving operations)
- ❖ For performance, DBMS executes transactions using a serializable schedule
 - In this schedule, operations from different transactions can interleave and execute concurrently
 - But the schedule is guaranteed to produce the same effects as a serial schedule
- ❖ How is isolation achieved?
 - Locking, multi-version concurrency control, etc.

SQL isolation levels

10

- ❖ Strongest isolation level: **SERIALIZABLE**
 - Complete isolation
 - SQL default
- ❖ Weaker isolation levels: **REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED**
 - Increase performance by eliminating overhead and allowing higher degrees of concurrency
 - Trade-off: sometimes you get the “wrong” answer

READ UNCOMMITTED

11

- ❖ Can read “dirty” data
 - A data item is dirty if it is written by an uncommitted transaction
- ❖ Problem: What if the transaction that wrote the dirty data eventually aborts?
- ❖ Example: wrong average
 - -- T1: UPDATE Student SET GPA = 3.0 WHERE SID = 142; ROLLBACK;
 - -- T2: SELECT AVG(GPA) FROM Student; COMMIT;

READ COMMITTED

12

- ❖ No dirty reads, but non-repeatable reads possible
 - Reading the same data item twice can produce different results
- ❖ Example: different averages
 - -- T1: UPDATE Student SET GPA = 3.0 WHERE SID = 142; COMMIT;
 - -- T2: SELECT AVG(GPA) FROM Student; COMMIT;

REPEATABLE READ 13

❖ Reads are repeatable, but may see phantoms

❖ Example: different average (still!)

```
▪ -- T1:                -- T2:
                        SELECT AVG(GPA)
                        FROM Student;

INSERT INTO Student
VALUES(789, 'Nelson', 10, 1.0);
COMMIT;

                        SELECT AVG(GPA)
                        FROM Student;
COMMIT;
```

Summary of SQL isolation levels 14

	Dirty reads	Non-repeatable reads	Phantoms
READ UNCOMMITTED			
READ COMMITTED			
REPEATABLE READ			
SERIALIZABLE			

❖ Syntax: At the beginning of a transaction,
SET TRANSACTION ISOLATION LEVEL
isolation_level [READ ONLY|READ WRITE];

- READ UNCOMMITTED can only be READ ONLY

Transactions in programming (JDBC) 15

- ❖ Set isolation level for the current transaction
 - `con.setTransactionIsolationLevel(l);`
 - Where *l* is one of TRANSACTION_SERIALIZABLE (default), TRANSACTION_REPEATABLE_READ, TRANSACTION_READ_COMMITTED, and TRANSACTION_READ_UNCOMMITTED
- ❖ Set the transaction to be read-only or read/write (default)
 - `con.setReadOnly(true|false);`
- ❖ Turn on/off AUTOCOMMIT (commits every single statement)
 - `con.setAutoCommit(true|false);`
- ❖ Commit/rollback the current transaction (when AUTOCOMMIT is off)
 - `con.commit();`
 - `con.rollback();`

ANSI isolation levels are lock-based 16

- ❖ READ UNCOMMITTED
 - Short-duration locks: lock, access, release immediately
- ❖ READ COMMITTED
 - Long-duration write lock: do not release write locks until commit
- ❖ REPEATABLE READ
 - Long-duration locks on all data items accessed
- ❖ SERIALIZABLE
 - Lock ranges to prevent insertion as well

An isolation level not based on locks 17

Snapshot isolation

- ❖ Based on multiversion concurrency control
- ❖ Available in Oracle, PostgreSQL, MS SQL Server, etc.
- ❖ How it works
 - Transaction *X* performs its operations on a private snapshot of the database taken at the start of *X*
 - *X* can commit only if it does not write any data that has been also written by a transaction committed after the start of *X*
- ❖ Avoids all ANSI anomalies
- ❖ But is NOT equivalent to SERIALIZABLE because of write skew anomaly

Write skew example 18

- ❖ Constraint: combined balance $A+B \geq 0$
- ❖ $A = 100, B = 100$
- ❖ T_1 checks $A+B - 200 \geq 0$, and then proceeds to withdraw 200 from *A*
- ❖ T_2 checks $A+B - 200 \geq 0$, and then proceeds to withdraw 200 from *B*
- ❖ Possible under snapshot isolation because the writes (to *A* and to *B*) do not conflict
- ❖ But $A+B = -200 < 0$ afterwards!

Bottom line

- ❖ Group reads and dependant writes into a transaction in your applications
 - E.g., enrolling a class, booking a ticket

- ❖ Anything less than **SERIALABLE** is potentially very dangerous
 - Use only when performance is critical
 - **READ ONLY** makes weaker isolation levels a bit safer