SQL: Transactions

CPS 116

Introduction to Database Systems

Announcements (September 27)

- Homework #2 due this Thursday
  - Sample solution available next Tuesday
- Graded Homework #1 back on Thursday
- Midterm next Thursday in class
  - Sample midterm available this Thursday
  - Solution to sample midterm available next Tuesday
  - Review session next week to be scheduled
- Project milestone #1 due in 2½ weeks

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

- 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

- 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

- 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

- 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

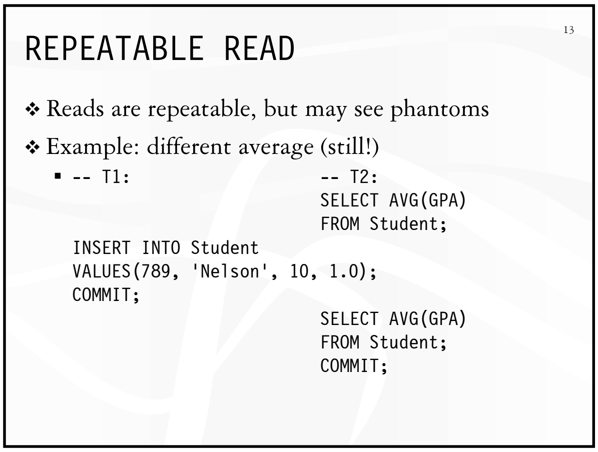
- Can read "dirty" data
  - A data item is dirty if it is written by an uncommitted transaction
- Problem: What if the transaction that wrote the dirty data eventually aborts?
- Example: wrong average

```
-- T1:                    -- T2:
UPDATE Student
SET GPA = 3.0
WHERE SID = 142;          SELECT AVG(GPA)
                          FROM Student;
ROLLBACK;
                          COMMIT;
```

READ COMMITTED

- No dirty reads, but non-repeatable reads possible
  - Reading the same data item twice can produce different results
- Example: different averages

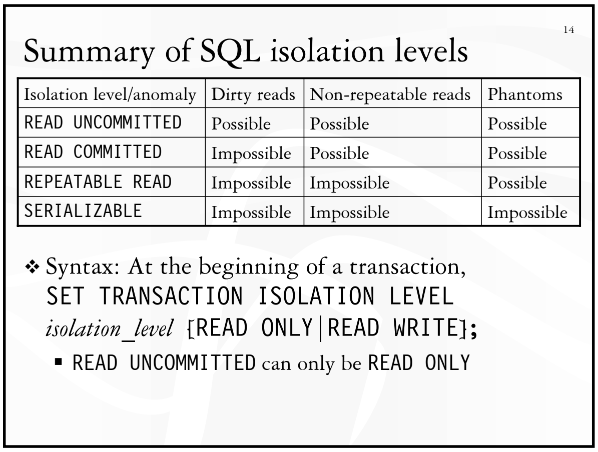
```
-- T1:                    -- T2:
                          SELECT AVG(GPA)
                          FROM Student;
UPDATE Student
SET GPA = 3.0
WHERE SID = 142;
COMMIT;
                          SELECT AVG(GPA)
                          FROM Student;
                          COMMIT;
```



REPEATABLE READ

- 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

Isolation level/anomaly	Dirty reads	Non-repeatable reads	Phantoms
`READ UNCOMMITTED`	Possible	Possible	Possible
`READ COMMITTED`	Impossible	Possible	Possible
`REPEATABLE READ`	Impossible	Impossible	Possible
`SERIALIZABLE`	Impossible	Impossible	Impossible

- Syntax: At the beginning of a transaction, `SET TRANSACTION ISOLATION LEVEL` *isolation_level* `[READ ONLY|READ WRITE];`
  - `READ UNCOMMITTED` can only be `READ ONLY`