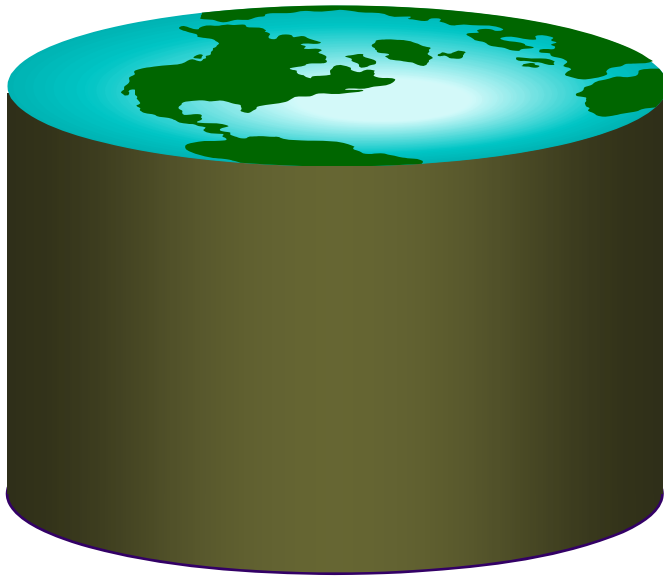
CS 405G: Introduction to Database Systems



Today's Topic

- Transaction

Transactions

- A program may carry out many operations on the data retrieved from the database
- However, the DBMS is only concerned about what data is read/written from/to the database.
- *database* - a fixed set of relations *(A, B, C, …)*
- *transaction* - a sequence of read and write operations *(read(A), write(B), …)*
  - DBMS's abstract view of a user program

Correctness criteria: The ACID properties

- A tomicity: All actions in the Xact happen, or none happen.
- C onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- I solation: Execution of one Xact is isolated from that of other Xacts.
- D urability: If a Xact commits, its effects persist.

An Example about SQL Transaction

- Consider two transactions (*Xacts*):

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- 1st xact transfers $100 from B's account to A's
- 2nd credits both accounts with 6% interest.
- Assume at first A and B each have $1000. What are the legal outcomes of running T1 and T2???
  - $1100 *1.06 = $1166
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect *must* be equivalent to these two transactions running serially in some order.

Example (Contd.)

- Legal outcomes: A=1166,B=954 or A=1160,B=960
- Consider a possible interleaved *schedule*:

T1:	A=A+100,		B=B-100
T2:		A=1.06*A,	B=1.06*B

❖ This is OK (same as T1;T2). But what about:

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

- **Result: A=1166, B=960; A+B = 2126, bank loses $6**
- **The DBMS's view of the second schedule:**

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

SQL transactions

- Syntax in SQL:

  `BEGIN`

  <database operations>

  `COMMIT [ROLLBACK]`

- A transaction is automatically started when a user executes an SQL statement (begin is optional)
- 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 (flushing the update to disk)
- `ROLLBACK` command aborts the transaction (all effects are undone)

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)

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, only those operations that can be interleaved are executed concurrently
  - Those that can not be interleaved are in a serialized way
  - The schedule guarantees to produce the same effects as a serial schedule

SQL isolation levels

- Strongest isolation level: `SERIALIZABLE`
  - Complete isolation
  - Usually use as 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:
    UPDATE Student
    SET GPA = 3.0
    WHERE SID = 142;

    ROLLBACK;
    ```

    ```
    -- T2:


    SELECT AVG(GPA)
    FROM Student;

    COMMIT;
    ```
    ```

READ COMMITTED

- All reads see a snapshot of the database (including all committed transactions) right before the *beginning of the query*
  - 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;

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

REPEATABLE READ

- Reads are repeatable, but may see phantoms
  - Do not allow the modification of existing values
  - New rows may be inserted in the mean time
- 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;
    ```
    ```

SERIALIZABLE READ

- Reads see the snapshot of the database right before the *beginning* of the transaction
- Example: the same average
  - -- T1:

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

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

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`

Summary of SQL features covered so far

- Query
- Modification
- Constraints
- Triggers
- Views
- Transaction

☞ Next: Database programming