

# SQL: Transactions

Introduction to Databases

CompSci 316 Fall 2018



**DUKE**  
COMPUTER SCIENCE

# Announcements (Tue., Oct. 16)

- **Midterm** graded
  - Sample solution already posted on Sakai
- **Two Homework 2 problems** due tonight!
- **Homework 3** to be assigned on Thursday
- **Project Milestone #1** feedback by email this weekend

# 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
- **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
  - You can turn it on/off through the API
    - Examples later in this lecture
  - For PostgreSQL:
    - psql command-line processor turns it on by default
    - You can turn it off at the psql prompt by typing:  
`\set AUTOCOMMIT 'off'`

# 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
  - Some systems roll back only this statement and let the transaction continue; others roll back the whole transaction
- How is atomicity achieved?
  - Logging (to support undo)

# Durability

- DBMS accesses data on stable storage by bringing data into memory
- 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
  - 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
- 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 User  
SET pop = 0.99  
WHERE uid = 142;  
  
ROLLBACK;

- -- T2:  
  
SELECT AVG(pop)  
FROM User;  
  
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:

```
UPDATE User
SET pop = 0.99
WHERE uid = 142;
COMMIT;
```

- -- T2:

```
SELECT AVG(pop)
FROM User;
```

```
SELECT AVG(pop)
FROM User;
COMMIT;
```

# REPEATABLE READ

- Reads are repeatable, but may see **phantoms**
- Example: different average (still!)

- -- T1:

```
INSERT INTO User
VALUES(789, 'Nelson',
      10, 0.1);
COMMIT;
```

- -- T2:

```
SELECT AVG(pop)
FROM User;
```

```
SELECT AVG(pop)
FROM User;
COMMIT;
```

# Summary of SQL isolation levels

Isolation level/anomaly	Dirty reads	Non-repeatable reads	Phantoms
<b>READ UNCOMMITTED</b>	Possible	Possible	Possible
<b>READ COMMITTED</b>	Impossible	Possible	Possible
<b>REPEATABLE READ</b>	Impossible	Impossible	Possible
<b>SERIALIZABLE</b>	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**
- PostgreSQL defaults to **READ COMMITTED**

# Transactions in programming

Using psycopg2 as an example:

```
conn = psycopg2.connect(dbname='beers')
```

```
conn.set_session(isolation_level='SERIALIZABLE',  
                 read_only=False,  
                 autocommit=True)
```

- `isolation_level` defaults to `READ COMMITTED`
  - `read_only` defaults to `False`
  - `autocommit` defaults to `False`
- When `autocommit` is `False`, `commit/abort` current transaction as follows:

```
conn.commit()
```

```
conn.rollback()
```

# ANSI isolation levels are lock-based

- **READ UNCOMMITTED**
  - **Short-duration locks**: lock, access, release immediately
- **READ COMMITTED**
  - **Long-duration write locks**: 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



# Isolation levels not based on locks?

## Snapshot isolation in Oracle

- Based on **multiversion concurrency control**
  - Used 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

- 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!

👉 To avoid write skew, when committing, ensure the transaction didn't *read* any object others wrote and committed after this transaction started

# Bottom line

- Group reads and dependent writes into a transaction in your applications
  - E.g., enrolling a class, booking a ticket
- Anything less than **SERIALIZABLE** is potentially very dangerous
  - Use only when performance is critical
  - **READ ONLY** makes weaker isolation levels a bit safer