EDAF75
Database Technology

Lecture 9

Christian.Soderberg@cs.lth.se

Feb 17, 2025

Today

- Transactions – handling concurrency and errors
- Triggers – keeping the database in a consistent state ’automatically’
- Understanding transactions and triggers will make the project much easier to solve – and we will require that you use them in your solution
- It will also make lab 3 easier to solve, but there you don’t have to use them (feel free to try, though!)

System structure

- For many databases today, concurrency is very, very important – some systems accept many thousands of concurrent requests (Google gets around 100 000 queries every second – I found that out by asking one of them)
- In an increasingly complex environment, more and more things can fail
- So, being able to handle concurrency and failure is getting more important by the day, *transactions* are a great tool to help us with both these issues

Concurrent access

What problems can occur when these statements are run concurrently:

```
UPDATE colleges
SET    enrollment = enrollment + 100
WHERE  c_name = 'LTH';

UPDATE colleges
SET    enrollment = enrollment + 150
WHERE  c_name = 'LTH';
```

- We can get any of three outcomes (add 100, 150, or 250 students)
- This is called *attribute level inconsistency*

Concurrent access

What problems can occur when these statements are run concurrently:

```
UPDATE students
SET    gpa = 3.85
WHERE  s_id = 123;

UPDATE students
SET    size_hs = 1200
WHERE  s_id = 123;
```

- We can get any of three outcomes (both attributes change, or any one of them change)
- This is called *tuple level inconsistency*

Concurrent access

What problems can occur when these statements are run concurrently:

```
UPDATE applications
SET    decision = 'Y'
WHERE  s_id IN (SELECT s_id
                FROM   students
                WHERE  gpa > 3.9);

UPDATE students
SET    gpa = min(1.1 * gpa, 4.0)
WHERE  size_hs > 2000;
```

- This is very fragile, it only works if the updates are not interleaved
- This is called *table level inconsistency*

Concurrent access

What problems can occur when these statements are run concurrently:

```
INSERT
INTO   archive
       SELECT *
       FROM   applications
       WHERE  decision = 'N';
DELETE
FROM   applications
WHERE  decision = 'N';

SELECT count(*)
FROM   applications;
SELECT count(*)
FROM   archive;
```

- This is also very fragile, and it only works if the sequences are not interleaved, it is called *multi-statement inconsistency*

Concurrency in databases

- We want to execute *sequences* of SQL statements so that each of the sequences *appear* to run in *isolation*
- We could do it by locking our DBMS
- Today we typically have
  - Multiprocessors
  - Multithreading
  - Server centers
  - Asynchronous I/O

  So we really want our DMBS to be as concurrent as possible, with as little blocking as possible

Handling failures

- Real systems today are *very* complicated, and we must be prepared for failures ("*Embrace Failure*")
- Common problems in a non-trivial system:
  - Disks crash
  - Processes crash
  - Network connections fail
  - DOS attacks
  - ...
- We want our code to do *all-or-nothing* updates – failures must not leave our DBMS in an inconsistent state

Transactions

- A *transaction* is a sequence of one or more SQL operations running as a unit
- It *appears* to run in *isolation*, i.e., without interference from anyone else
- Either all changes are reflected afterwards, or none
- It begins automatically on the first SQL statement
- A transaction normally ends with either
  - a `COMMIT`, when we try to commit our changes to the database
  - a `ROLLBACK`, when we reset our changes
- When the session ends, the current transaction ends
- In `AUTOCOMMIT` mode, each statement is a transaction
- *We use transactions both for handling concurrency, and for handling failures*

Using transactions in JDBC (and sqlite)

- We normally use transactions from either external programs (using e.g., `JDBC`), or in *stored procedures* (we'll come to that later)
- In `JDBC`, `Connection`-objects start out in *auto commit mode*, but we can call:

  ```
  conn.setAutoCommit(false);
  ```

  to start a transaction in a `Connection` object
- To finish a `JDBC` transaction, we call:

  ```
  conn.commit();
  ```

  or:

  ```
  conn.rollback();
  ```

Example of transactions in JDBC

We want to update sales and total sales for one kind of coffee bean, and make sure that we either add to both sales and totals, or add to none of them (i.e., we want atomicity), so we use a transaction.
JDBC code is often verbose, so we split it into several slides, here's the preparation:

```
var updateSalesStr =
    """
    UPDATE coffees
    SET    sales = ?
    WHERE  coffee_name = ?
    """;
var updateTotalsStr =
    """
    UPDATE coffees
    SET    total = total + ?
    WHERE  coffee_name = ?
    """;
var amount = 12000;
var bean = "Sidamo";
```

Example of transactions in JDBC

Next we turn off auto commit (which means we get a transaction), and start our `try`-block (we use a `try`-with-resources statement):

```
conn.setAutoCommit(false);
try (updateSales = conn.prepareStatement(updateSalesStr);
     updateTotals = conn.prepareStatement(updateTotalsStr)) {
    updateSales.setInt(1, amount);
    updateSales.setString(2, bean);
    updateSales.executeUpdate();
    updateTotals.setInt(1, amount);
    updateTotals.setString(2, bean);
    updateTotals.executeUpdate();
    conn.commit();
```

When we commit in the last statement, both updates will be seen in the database (for everyone who is not in their own transaction), and any failure will be taken care of in the next slide...

Example of transactions in JDBC

If something bad happens (we may break some constraint), we get an exception, and we roll back our transaction.

```
} catch (SQLException e) {
    if (conn != null) {
        try {
            System.err.print("Transaction is being rolled back");
            conn.rollback();
        } catch(SQLException se) {
            System.err.print(se.toString());
        }
    }
}
conn.setAutoCommit(true);
```

Thanks to the try-with-resources statement, we know that `updateSales` and `updateTotal` will be closed automatically.

Example of transactions in JDBC

When we executed our updates, we could also have checked the number of effected rows (it's returned by our `INSERT`'s, and passed on through `executeUpdate()`), and maybe thrown an `SQLException` in case nothing gets inserted:

```
conn.setAutoCommit(false);
try (updateSales = conn.prepareStatement(updateSalesStr);
     updateTotals = conn.prepareStatement(updateTotalsStr)) {
    ...
    if (updateSales.executeUpdate() != 1) {
        throw new SQLException("Failed update into sales");
    }
    ...
    if (updateTotals.executeUpdate() != 1) {
        throw new SQLException("Failed update into totals");
    }
    conn.commit();
} ...
```

It's a bit of a hack, but it would have ensured that we rolled back the transaction in the `catch` statement

Transactions in Python with `pysqlite`

- SQLite3 operates in autocommit mode by default
- But `pysqlite` does not, so we must make sure to call `conn.commit()` after inserts, updates, and deletions, otherwise they will not be seen after our connection is closed
- Observe that we run `commit()` on our `Connection`, not our `Cursor`!
- We can configure a connection to use autocommit mode:

  ```
  con = sqlite3.connect(..., isolation_level=None)
  ```

  (we'll soon se what an isolation level is)
- BTW: as a shortcut we can also call `execute` directly on our connection

ACID

ACID properties:

- Atomicity
- Consistency
- Isolation
- Durability

ACID – Atomicity

- Handles failures during the execution of a transaction
- Guarantees that even in the face of failure, we get *all-or-nothing* – either all changes are made, or none
- Based on logging
- If a failure occurs, we need to restart the transaction if we want another try

Rolling back transactions

- A `ROLLBACK` will undo partial effects of a transaction
- Used to obtain atomicity
- Can be initiated by either the user or the system
- The rollback will only effect data in our tables
- Don't wait for long before doing `COMMIT`/`ROLLBACK` – you'll be locking parts of the database from others

A**C**ID – Consistency

- Guarantees that if all constraints hold when we begin a transaction, then they hold when the transaction finishes
- This, combined with serializability, means that we can make sure our constraints always hold (we always jump from one consistent state to another)
- We get consistancy due to the other three properties (so **AID** would have been a reasonable alternative acronym, maybe they thought ACID sounded 'cooler')

ACID – Isolation

Isolation:

- Transactions run without interference from other transactions
- Based on the concept of *serializability*: The operations may be interleaved, but the result of the execution must be equivalent to *some* sequential ordering of all transactions
- The DBMS must guarantee that the state of the system looks as if the transactions ran sequentially
- It does this by locking parts of the database – we happily leave the details to the DBMS

Transactions for concurrent access

Running the following two SQL-statements sequentially, in some order (using transactions), we'll solve our attribute level inconsistency (observe that T1 and T2 are just labels, they don't belong to the code):

```
T1: UPDATE college
    SET    enrollment = enrollment + 100
    WHERE  c_name = 'LTH'

T2: UPDATE college
    SET    enrollment = enrollment + 150
    WHERE  c_name = 'LTH'
```

- If T1 and T2 run sequentially, we either get T1;T2, or T2;T1, and the total number of students will increase by 250

Transactions for concurrent access

Running the following two SQL-statements sequentially, in some order, we'll solve our tuple level inconsistency:

```
T1: UPDATE students
    SET    gpa = 3.85
    WHERE  s_id = 123

T2: UPDATE students
    SET    size_hs = 1200
    WHERE  s_id = 123
```

- We get T1;T2, or T2;T1, and in both cases, both attributes will be updated

Transactions for concurrent access

Running the following two SQL-statements sequentially, in some order, we'll partly solve our table level inconsistency:

```
T1: UPDATE applications
    SET    decision = 'Y'
    WHERE  s_id IN (SELECT s_id
                    FROM   students
                    WHERE  gpa > 3.9)

T2: UPDATE students
    SET    gpa = min(1.1 * gpa, 4.0)
    WHERE  size_hs > 2000
```

- Once again, we get T1;T2, or T2;T1 – so we get a consistent result, but we don't know for sure which one (which means that a student from a big high school whose gpa is just under 3.9 might get accepted, or not, it depends on in which order the transactions were run – but everyone gets treated the same way)

Transactions for concurrent access

Running the following two sequences of SQL-statements sequentially, in some order, we'll partly solve our multi-statement inconsistency:

```
T1: INSERT
    INTO   archive
           SELECT *
           FROM   applications
           WHERE  decision = 'N';
    DELETE
    FROM   applications
    WHERE  decision = 'N';

T2: SELECT count(*)
    FROM   applications;
    SELECT count(*)
    FROM   archive;
```

- Once again, we get `T1;T2`, or `T2;T1` – so we get a consistent result, but we don't know for sure which one: either we archive first, and get a bigger archive and a smaller `applications` table, or vice versa

ACID – Durability

Durability:

- If a crash occurs after a commit, all effects of the transactions remain in the database
- The implementation is based on logging

ACID – More about isolation

- Isolation involves some extra work, and reduces concurrency
- There are weaker levels of isolation:
  - `READ UNCOMMITTED`
  - `READ COMMITTED`
  - `REPEATABLE READ`
  - `SERIALIZABLE`
- They give progressively more overhead, and less concurrency – but more consistency guarantees

Isolation levels

- Isolation levels are set per transaction
- Simultaneous transactions can have different isolation level

Dirty Reads

- Items written by uncommitted transactions are called “dirty”

```
T1: UPDATE college
    SET    enrollment = enrollment + 100
    WHERE  c_name = 'LTH';
    COMMIT;

T2: SELECT avg(enrollment)
    FROM   college;
```

- Without isolation, T2 might read `enrollment` before it is committed by T1, and if it’s never committed, T2 will use a value which was not supposed to be in the database at all

Isolation level READ UNCOMMITTED

- When we set the isolation level to READ UNCOMMITTED, our transaction may perform dirty reads:

```
T1: UPDATE students
    SET    gpa = min(1.1 * gpa, 4.0)
    WHERE  size_hs > 2000;
    COMMIT;

T2: SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
    SELECT avg(gpa)
    FROM students;
```

- Here we have no guarantees that T1 and T2 run serially, so we could get some errors in our gpa-calculation
- We get higher concurrency, paid for by lower accuracy

Isolation level READ COMMITTED

- Using READ COMMITTED, we won’t get dirty reads:

```
T1: UPDATE students
    SET    gpa = min(1.1 * gpa, 4,0)
    WHERE  size_hs > 2000;
    COMMIT;

T2: SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
    SELECT avg(gpa) FROM students;
    SELECT max(gpa) FROM students;
```

- There are still no guarantees that T1 and T2 run serially, so the average could be calculated before the gpa-update, and the max calculated after
- So, once again we get better performance, but risk getting some inaccuracies

Isolation level REPEATABLE READ

- Using REPEATABLE READ, we won’t get dirty reads, and *reading the same value several times will always yield the same value*
- This means that we would use the same gpa’s for avg and max:

```
T1: UPDATE students
    SET    gpa = min(1.1 * gpa, 4.0)
    WHERE  size_hs > 2000;
    COMMIT

T2: SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
    SELECT avg(gpa) FROM students;
    SELECT max(gpa) FROM students;
```

Isolation level REPEATABLE READ

```
T1: UPDATE students
    SET    gpa = min(1.1 * gpa, 4.0);
    UPDATE students
    SET    size_hs = 1500
    WHERE  s_id = 123;
    COMMIT;

T2: SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
    SELECT avg(gpa) FROM students;
    SELECT avg(size_hs) FROM students;
```

- Not serializable, we might calculate avg(gpa) before updating the gpa's, and avg(size_hs) after we've updated Amy (s_id = 123)

Isolation level REPEATABLE READ

```
T1: INSERT
    INTO   students  ...100 tuples...;
    COMMIT;

T2: SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
    SELECT avg(gpa) FROM students;
    SELECT max(gpa) FROM students;
```

- It turns out that the implementation of repeatable locks allows us to read values which have been inserted after the previous read (the lock is only for the values which was there during our last read)
- We sometimes call the added rows *phantom tuples*
- Deletions will not be allowed in this example

Read Only transactions

- Independent of isolation level
- Helps the DBMS optimize performance

```
T1: SET TRANSACTION READ ONLY;
    SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
    SELECT avg(gpa) FROM students;
    SELECT max(gpa) FROM students;
```

Isolation Levels, cheat sheet

	dirty reads	nonrepeatable reads	phantoms
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

Using table constraints

- When we define attributes in our tables, we can add default values:

```
balance DECIMAL(14,2) DEFAULT (0.0),
```

- We can also add checks:

```
balance DECIMAL(14,2) DEFAULT (0.0) CHECK (balance > 0),
```

- We can even add table constraints:

```
CONSTRAINT non_negative CHECK (balance >= 0)
                        ON CONFLICT ROLLBACK
```

More about table constraints

- Some options for conflicts are (there might be others, depending on which DBMS we use):
  - `ROLLBACK`
  - `ABORT` – aborts the current statement, but keeps other changes in the current transaction, and keeps the transaction alive
  - `IGNORE` – ignores any update to the affected row, but continues the statement

Exercise

Exercise

Make sure no one enters an invalid gpa for a student in the `students` table.

Exercise

Exercise

First try: make sure no one lowers the gpa of a student in the `students` table.

Triggers

- A *trigger* makes sure some code is run if a given action is performed, and some optional condition hold
- The actions which can trigger a trigger are insertions, updates, and deletions
- Triggers are useful for:
  - moving logic for monitoring from application to DBMS
  - making sure constraints hold, beyond simple constraints in our table definitions
- Part of SQL standard, but substantial variation between DMBS's

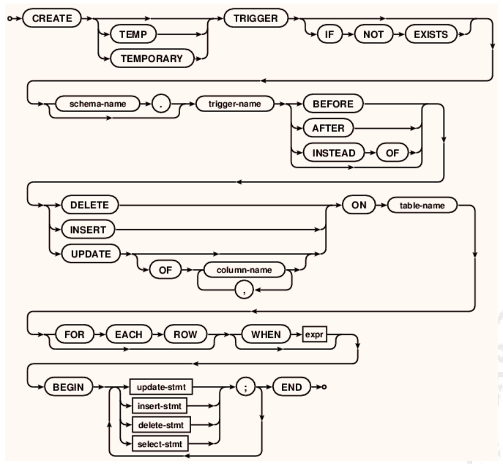
Triggers in SQLite3

- A trigger in SQLite3:

```
CREATE TRIGGER <name>
  BEFORE|AFTER|INSTEAD OF <event>
  [OF <column name>]
  ON <table>
  [WHEN <condition>]
BEGIN
  <actions>
END
```

- The event can be INSERT, UPDATE, or DELETE
- The referencing variables refer to the old (OLD) and/or the new (NEW) values (either rows or tables)

CREATE TRIGGER statement in SQLite3



Exercise

Exercise

Second try: make sure no one lowers the gpa of a student in the `students` table.

Exercise

Exercise

Make sure no student applies for more than two majors at any given college.

Exercise

Exercise

Add some logging into our database, so we can see when students have their gpa's changed.

More about foreign keys

- When we use foreign keys, the database helps us to check that the values we reference are actually there
- We can tell the database what should happen if a foreign key is removed or updated – some alternatives, among them:
  - our row with the foreign key could be updated or removed
  - we can restrict the change in the referenced table
  - we can set the foreign key to a default value
  - we can ignore the removal/update (we'll continue in an inconsistent state)

Exercise

Exercise

Make sure all applications for a student are removed when the student is removed.

Issues with triggers

- Several triggers can be active the same time
- Some, but not all, DBMS's lets us choose between row-level or statement-level
- Be careful: triggers may activate each other, or themselves

A slide from the first lecture

- Q: How would you implement a program which keeps track of your friends' addresses, phone numbers, and twitter handles?
- Food for thought:
  - how do we save data between sessions?
  - how do we allow several people to ask for information at the same time?
  - how do we handle simultaneous updates?
  - how do we handle failing hard disk and broken network connections?
  - what do we do if we realize an update is going awry?
  - how do we save our data efficiently?
  - how do we query our data in a simple way?
  - how do we make our queries fast?
- For a serious programmer, using a database system to handle the data is a no-brainer