TDA357/DIT621 – Databases

Lecture 13 – Remaining SQL topics: Transactions, Authorization

Jonas Duregård

Finalizing labs

- Since Wednesday last week every lab-related topic is covered
- I'm hoping many of you finish and demonstrate the lab this week.
- There may be a spike in demo requests on Friday, with long waiting times if a lot of people decide to demonstrate at the same time.
    - If you signed up early in a session (say 75 minutes before it ends) and don't get to demo let me know and I'll try to work something out.

- There are lab session next week, but demonstrating near the end of next week is risky: If you fail the demo you may not have time to retry!
- Next week, demoing will be prioritized over help requests in lab sessions
    - You can still get help when/if there are no demo requests

Authorization and privileges

- Every connection to a database (including from applications) is made by a user
- Users should not be allowed to do everything
- They should be allowed to do as little as possible for their purpose
- If an attacker compromises your web server, they should not automatically be able to read or erase all your data

Database privileges vs File system privileges

- A (UNIX) file system has:
  - Privileges on files
  - Three different privileges: read, write, execute
  - Three levels of access: owner, group, all
- A database has:
  - Privileges on schema elements (tables, views, triggers, etc.)
  - Nine different privileges
  - Unlimited levels of access – each user can be given different access

Granting privileges

- Syntax:

  GRANT *<privilege-list>* **ON** *<object>* **TO** *<user-list>*

- Examples:

```
GRANT SELECT
 ON Inventory
 TO webshop_user, marketing;
```

Allows two users to SELECT from Inventory

```
GRANT SELECT, DELETE, INSERT
 ON Registrations
 TO study_administrator;
```

Allows one user to SELECT, DELETE and INSERT on a table (or view?)

Basic table privileges

- SELECT [(<list of column names>)]
- DELETE
- INSERT [(<list of column names>)]
- UPDATE [(<list of column names>)]

- The square brackets '[' and ']', indicate optional syntax and are not part of the SQL-syntax
  - UPDATE(idnr,name) is a valid privilege, and so is simply UPDATE
  - Adding the column list limits the privilege to updating those columns

Less frequently used privileges

- EXECUTE
  - Allows users to execute a function/procedure
- REFERENCES [(<list of column names>)]
  - Allows users to create foreign keys
  - Rarely granted, since foreign keys are usually part of the design
- TRIGGER [(<list of column names>)]
  - Allows users to create triggers
  - Rarely granted for the same reason as above

Who can grant (and revoke) privileges?

- It is possible to grant privilege to grant privileges
- Just add WITH GRANT OPTION to the end of a GRANT statement
- In Postgres there is a root user (postgres) that can always grant and revoke any privileges, this is usually all you need

What about dropping tables?

- Rights to drop tables and other schema elements cannot be granted
- This right is exclusive to the owners of the schema element
    - So don't run your web server with the user that owns the tables! (to avoid some SQL injection attacks)

Grant all privileges

- You can replace the privilege list with ALL PRIVILEGES
    - Use judiciously

Revoking privileges

- Previously granted privileges can be revoked. Syntax:
  REVOKE *<list of privileges>*
    ON *<schema element>*
    FROM *<list of users>*;

Transactions

- Transactions are a fundamental concept in database usage
- In its basic form, it allows you to run several modifications as an *atomic* action
    - If any modification causes an error, the whole transaction is rolled back
    - Either all modifications are performed, or no modifications are performed

A very simple Postgres transaction

- This SQL code runs a sequence of inserts in a transaction
- If any INSERT fails, the whole transaction is rolled back
- The end result is either 8 successful inserts, or 0

Starts a transaction (Postgres syntax)

```
BEGIN;
INSERT INTO Taken VALUES(4444444444,'CCC111','5');
INSERT INTO Taken VALUES(4444444444,'CCC222','5');
INSERT INTO Taken VALUES(4444444444,'CCC333','5');
INSERT INTO Taken VALUES(4444444444,'CCC444','5');
INSERT INTO Taken VALUES(1111111111,'CCC111','3');
INSERT INTO Taken VALUES(1111111111,'CCC222','3');
INSERT INTO Taken VALUES(1111111111,'CCC333','3');
INSERT INTO Taken VALUES(1111111111,'CCC444','3');
COMMIT;
```

End the transaction and commit all changes to database

Explicitly rolling back

- It is possible to issue the `ROLLBACK;` statement any time during a transaction
- This immediately ends the transaction and reverts all changes
- Allows us to write applications (e.g. in JDBC) that do this kind of things:

```
BEGIN transaction
Do some modifications
If (everything is good?)
    COMMIT
Else
    ROLLBACK
```

Side note: Deferring constraints

TLDR: If you ever need to make two updates at the same time, with a constraint being temporarily violated after the first one – look up deferrable constraints!

- Sometimes a transaction will intermediately place the database in an inconsistent state (violating some constraint)
- By default this will cause an error and roll back the transaction
- You can set constraints as deferrable in SQL to defer checking them until the end of a transaction
- The constraint is still guaranteed, the difference is just when it is checked
- This is sometimes necessary, e.g. when you have cyclic references

The canonical transaction example

- Here is a semi-realistic example of a bank performing two updates:

```sql
-- Transfer 100$ from account 101 to account 42
UPDATE Accounts SET amount=amount-100 WHERE id=101;
UPDATE Accounts SET amount=amount+100 WHERE id=42;
```

- Two things are important here:
  - The two updates should be treated as an atomic transaction!
    (If one update succeeds and the other fails, someone will lose money)
  - The transaction should only commit if both updates affect exactly one row (otherwise, it should roll back)
    - Can be implemented in JDBC or similar, not in a plain SQL file

Pseudocode for transaction example

```
Prepare two statements:
 s1=UPDATE Accounts SET amount=amount-? WHERE id=?;
 s2=UPDATE Accounts SET amount=amount+? WHERE id=?;
Fill all holes (question marks) with user data
Start transaction
If (s1.executeUpdate() != 1)
  rollback and throw exception
If (s2.executeUpdate() != 1)
  rollback and throw exception

Commit and return
```

"If s1 affected more or less than one row"

Here we know that both updates affected one row each successfully

Note: I'm assuming errors in either update automatically throws an exception and rolls back

Every query is a transaction

- As we already know, single queries and modifications are atomic
- One way to think about it:
  If a SQL query is executed outside a transaction, a transaction is automatically started and committed after the query is successful
- This transaction includes all triggers, cascading etc. executed by the query

Concurrent transactions

- If the database only has a single user, running all their transactions in sequence, the transaction story would end here
  - This is rarely the case in practice
- Concurrent access to the database makes transactions a lot more difficult

Example: Scheduling lectures again

- Suppose two separate processes, P1 and P2, are connected to a database containing lecture times
- Each process runs two queries (P1:ins;del P2:max;min) before terminating
- The database initially contains a single lecture, Monday 13-15
- The database alternates between serving the different processes
- What possible results can P2 give?

Process P1

ins: Insert a new lecture Monday 10-12
del: Delete the lecture Monday 13-15

Process P2

min: Select the start-time of the first lecture on Monday
max: Select the end-time of the last lecture on Monday
Process Result: max-min

Scheduling, continued

ins: add 10-12 lecture
del: remove 13-15 lecture

(ins) is always run before (del) and (min) before (max), but other than that, *scheduling* between processes depends on chance

P1: (ins) (del)
P2: (min) (max)

What are the results of min and max?

```
(ins) (del) (min) (max)    min=10, max=12, result=2
(ins) (min) (del) (max)    min=10, max=12, result=2
(ins) (min) (max) (del)    min=10, max=15, result=5 (?)
(min) (ins) (del) (max)    min=13, max=12, result=-1 (???)
(min) (ins) (max) (del)    min=13, max=15, result=2
(min) (max) (ins) (del)    min=13, max=15, result=2
```

Serializability

- Two processes/transactions run in serial if one ends before the other starts
- They are serializable if they give the same result as if they were run in serial
- Note that for two processes P1 and P2, there are two possible serializable results (P1 before P2 and P2 before P1)
- In our previous example:

Serializable schedules:

```
(ins) (del) (min) (max)
(ins) (min) (del) (max)
(min) (ins) (max) (del)
(min) (max) (ins) (del)
```

Non-serializable schedules:

```
(ins) (min) (max) (del)
(min) (ins) (del) (max)
```

ACID Transactions

- A DBMS is expected to support ”ACID transactions”, which are

  Atomic: Either the whole transaction is run, or nothing.

  Consistent: Database constraints are preserved.

  Isolated: Different transactions may not interact with each other.

  Durable: Effects of a transaction are not lost in case of a system crash.
- Consistency and durability are non-issues for applications
  - Dealt with "under the hood" by most DBMS
- By just adding BEGIN/COMMIT we get atomicity
- Isolation is the tricky part
  - Lack of isolation is what caused us problems in the last example

Just run everything in serial?

- One solution to the problem is to just run transactions in serial, meaning essentially we only allow these two schedules if the processes use transactions:

  `(ins) (del) (min) (max)`

  `(min) (max) (ins) (del)`

- Keep in mind that when the transactions start, the DBMS do not know what queries the transactions will run, so it cannot schedule in advance
  - It cannot allow other schedules like **`(ins) (min) (del) (max)`**, because it cannot be sure of the result of (min) before the other process commits
  - For e.g. **`(ins) (min) (ROLLBACK) (max)`** min gives incorrect result, that is valid neither before or after P1

The problem of long transactions

- In the lectures example, the transactions where presumably done quickly
    - The DBMS may loose performance from running the queries in serial, but not catastrophically so
    - This is not always the case
- Consider the example of a booking system: Each process first lists the available bookings (list) then the user selects one of them and books it (book)
    - Bad schedules would be things like:

```
P1: (list)          (book)
P2:           (list)          (book)
```

    - Showing the same room (two lists in sequence) may lead to double booking
    - Here, each transaction would wait for user input, which can take minutes
    - A serial schedule would mean extreme delays for listing

Transaction isolation levels and interference

- Each transaction can choose an isolation level
- The isolation level decides <u>how other processes are allowed to interfere</u>
- Standard SQL defines four transaction isolation levels:
    - READ UNCOMMITTED
    - READ COMMITTED
    - REPEATABLE READ
    - SERIALIZABLE
- Isolation levels are defined by which of three common kinds of interference are allowed: Dirty reads, Non-repeatable reads and Phantoms

Dirty read

- Transaction T1 modifies/creates a data item
- Transaction T2 then reads that data item while T1 is still running
- T1 then performs a ROLLBACK, T2 has read a data item that was never committed, and so never really existed.

- This is a violation of the atomicity of T1
- T2 performs the dirty read, to prevent it T2 needs to change its isolation level

Non-repeatable read

- Transaction T1 reads a data item.
- Another transaction T2 then modifies or deletes that data item and commits.
- If T1 then attempts to re-read (<u>or modify</u>) the data item, it receives a modified value or discovers that the data item has been deleted.

- This violates the atomicity of T1, since it can observe outside changes
- T1 performs the non-repeatable read. To prevent it, T1 needs to change its isolation level

Phantoms

- T1 performs a SELECT and gets a number of rows
- T2 performs at least one INSERT and commits
- T1 repeats the SELECT and gets the same rows as before, but also some additional rows

- Different from a non-repeatable read, where rows could also have disappeared or changed
- This violates the atomicity of T1
- T1 has the phantom problem, and can prevent it by changing isolation level

Isolation levels and possible interference

Remember: A transactions isolation level answers "how do I allow others to interfere with me?

Allowed interferences from other transactions

Isolation levels	Dirty reads	Non-repeatable reads	Phantoms
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Increasing isolation strictness

- Dirty read: Reads stuff that is later rolled back
- Non-repeatable read: Reads stuff that is deleted/modified during transaction
- Phantoms: New rows appear in query result during transactions

Another summary of isolation levels

When I start a transation using isolation level …

- … serializable, I accept no interference at all
- … repeatable read, I accept that others make changes as long as they don't modify or delete data that I have already looked at
- … read commited, I accept that other make any changes to the database
- … read uncommited, I accept that some of the data I get might actually never even be in the database (!)

When should I use Read uncommitted?

- When you only care about never waiting for other transactions
  (for maximal performance)
  - You are not worried about reading incorrect values now and then
  - You just want a transaction that can be rolled back

When should I use Read committed?

- When you are performing several queries, but the later do not really depend on the results of the first still being accurate
  - They have to have been accurate at some point (no dirty reads)
  - The "insert lots of grades at once"-example could be Read committed
- Basically, whenever a query encounters a row that has been modified by another still active transaction, it will wait for that transaction to commit or roll back, then proceed based on the outcome
  - Possibly harmful for transactions that take long time

When should I use Repeatable read?

- When you run several queries and the later depend on the data from the former to still be accurate, but not on it being *complete*
- An example: Read some rows from a table, then run a separate query for some of those rows
  - It's important that none of the rows we read have been modified or deleted
  - It's not important that no new rows have been added, even if they would be in our initial selection

When should I use serializable?

- When you are running multiple queries and it's essential that no one else modifies the parts of the database you are using in any way while the transaction is running
  - You are not too worried about locking others out from the database

Example

- Suppose I have a program that does the following:
  - Query 1: SELECT the price of a product
  - Compute a new price based on the old price
  - Query 2: UPDATE the price of the product to the new price
- Which isolation level should I use?
  - Repeatable read should be OK
  - Reasoning: Inserts on the product table should not cause any problems, but other modifications of the price may be bad
  - Note: I never actually repeat the read, but when I make the insert, I assume the price is the same as when I first fetched it
  - The important thing is: *if* I had re-read the price, it would not have changed

Example

What is a suitable transaction level for the example we saw earlier?

```
BEGIN;
UPDATE Accounts SET amount=amount-100 WHERE id=101;
UPDATE Accounts SET amount=amount+100 WHERE id=42;
COMMIT;
```

- Read committed should be fine since the two queries act on separate rows and no hidden outside state is carried between them
  - If another query sneaks in and updates the amount for account 42, we should still end up in a consistent state
  - If someone deletes account 101, we should still end up in the same situation as if it where deleted after the transaction
- Read <u>un</u>committed would be bad, as it would allow us to update rows based on changes that are later rolled back (a kind of dirty read)

Which isolation level is correct for booking rooms?

"*Each process first lists the available bookings (list) then the user selects one of them and books it (book)*"

- Repeatable read would prevent double bookings (phantoms should not be a problem, but non-repeatable reads could mean someone has already booked the selected room)
- Arguably, transactions should not be used here
- Having long transactions that wait for user input risks locking up the database
- A solution that gives an error when two users try to book the room is enough
  - Constraints in the database should already prevent double booking, if they do not, transactions can be used to "double check" that the room is still free before booking (but after collecting user input)
  - Like running (list)(check)(book), where (check) and (book) is a transaction

What actually happens in the booking case?

- If we run two booking processes at the same time in SERIALIZABLE transactions, will the (list) operation wait for the other process to end?
- No, in practice a schedule like this is possible:

```
P1: (list)          (book)
P2:         (list)          (book)
```

- Both processes get the same list, but when the slower process (P2) tries to commit it will fail and roll back
- At least, that's how it works in Postgres…
  - A lot of these things are implementation-specific ☹

A warning about long transactions

- In general, it seems safe to have transactions on the form: (some selects)…time consuming stuff…(modifications)(commit)
- Like in the booking example, where this works out nicely
- If you have more complex transaction with delays after modifications, other modifications may have to wait a long time
- This can cause deadlocks where your whole database is locked waiting for some commit/rollback that may never even happen
- Also makes your application vulnerable to denial of service attacks, where an attacker deliberately causes your database to deadlock

Can I have a long read uncommitted transaction?

- A read uncommitted transactions seems a lot less likely to cause waiting, so are long read uncommitted transactions unproblematic?
- Generally, no – they are still problematic.
- Remember that read uncommitted means others are allowed to interfere with you, but your interference may still cause others to lock
- So rather, you can have a long transaction if you know all other transactions are read uncommitted…

Transactions in JDBC

- To use transactions, run **`conn.setAutoCommit(false);`**
  - Where conn is your Connection object
  - Only done once for the connection
- End each transaction with **`conn.commit();`** or **`conn.rollback();`**
  - This also starts a new transaction
- To set isolation level, run something like:

```
conn.setTransactionIsolation(
                    Connection.TRANSACTION_REPEATABLE_READ);
```

  - This must be done at the start of the transaction (e.g. after last commit)
- If the connection closes without committing, the transaction is rolled back