Databases

TDA357/DIT621– LP3 2023

Lecture 12

Ana Bove

(much of the material is based on material from both Thomas Hallgren and Jonas Duregård)

February 20th 2023

Recall Last Lecture

- **JSON**:
  - Validation and schema;
  - **JSON** path language;
  - **JSON** path language in PostgreSQL.

Overview of Today's Lecture

- Transactions:
  - ACID properties;
  - Interference problems;
  - Isolation levels;
- Authorisation and privileges;
- Indexes.

Database Design and System Reliability

So far:

- ER-modeling, FD and NF: high-level design on how the information in the database is organised;
- **SQL**: Within the DBMS:
  - Creating tables, inserting and modifying values, querying the database;
  - Constraints on types and attributes;
  - Requirements on changes (CASCADE);
  - More complex restrictions, for example involving several tables (ASSERTION and TRIGGER).

Today (but not in depth):

Transactions: Ensure that concurrent database access does not corrupt the data or return inconsistent results;

Authorization: Give (different) access to different users;

Indexes: Fast retrieval of data (belongs to design part).

Transactions

Some changes in a database requires more than one update but need to be executed as part of an *atomic transaction*.

Example: Transferring money from accounts:

```
UPDATE Accounts SET balance = balance - 2000 WHERE name = Alice;
UPDATE Accounts SET balance = balance + 2000 WHERE name = Bob;
```

What would happen if the system crashes between the two updates?

Transactions: Atomicity

The solution is to wrap the updates in a *transaction*.

The DBMS then guarantees that transactions are *atomic*, that is, we get an *all-or-nothing* result: all updates are performed or none of them are!

```
BEGIN;
  UPDATE Accounts SET balance = balance - 2000 WHERE name = Alice;
  UPDATE Accounts SET balance = balance + 2000 WHERE name = Bob;
COMMIT;
```

Note: By default each **SQL** statement is treated as a separate atomic transaction.

In particular, triggers and cascading are executed as part of an atomic transaction.

Transactions: Explicit Rollback

At any time during a transaction we can explicitly ROLLBACK.

This immediately ends the ongoing transaction and reverts all changes done since the transaction started.

We could then have an application (for example in JDBC) doing something like:

```
BEGIN transaction....
... do some stuff/modifications ...
IF <everything has gone fine>
   COMMIT
ELSE
   ROLLBACK
```

Concurrent Transactions

What if at the same time Bob wants to transfer money to Carl?

```
UPDATE Accounts SET balance = balance  - 2000 WHERE name = Alice;
UPDATE Accounts SET balance = balance + 2000 WHERE name = Bob;
```

```
UPDATE Accounts SET balance = balance  - 1000 WHERE name = Bob;
UPDATE Accounts SET balance = balance + 1000 WHERE name = Carl;
```

We expect

name	balance
Alice	30000
Bob	20000
Carl	1000

name	balance
Alice	28000
Bob	21000
Carl	2000

But what would happen if transactions are not atomic?

Update = Read + Write

Under the hood, each of the following updates

```
UPDATE Accounts SET balance = balance  - 2000 WHERE name = Alice;
UPDATE Accounts SET balance = balance + 2000 WHERE name = Bob;
```

are likely implemented as two database accesses:

```
b1 := READ balance FROM Accounts WHERE name = Alice;
WRITE balance = b1 - 2000;
b2 := READ balance FROM Accounts WHERE name = Bob;
WRITE balance = b2 + 2000;
```

Interleaved Access

```
RB1  :  b1 := READ balance FROM Accounts WHERE name = Alice;
WB1  :  WRITE balance = b1 - 2000;
RB2  :  b2 := READ balance FROM Accounts WHERE name = Bob;
WB2  :  WRITE balance = b2 + 2000;
RB3  :  b3 := READ balance FROM Accounts WHERE name = Bob;
WB3  :  WRITE balance = b3 - 1000;
RB4  :  b4 := READ balance FROM Accounts WHERE name = Carl;
WB4  :  WRITE balance = b4 + 1000;
```

Selected interleaved access	**[A: 30000, B: 20000, C: 1000]**
(RB1, WB1, RB2, WB2, RB3, WB3, RB4, WB4)	[A: 28000, B: 21000, C: 2000]
(RB1, WB1, RB2, RB3, WB2, WB3, RB4, WB4)	[A: 28000, B: 19000, C: 2000]
(RB1, WB1, RB3, RB2, WB3, WB2, RB4, WB4)	[A: 28000, B: 22000, C: 2000]
(RB1, RB3, WB1, WB3, RB2, RB4, WB2, WB4)	[A: 28000, B: 21000, C: 2000]
(RB3, RB1, WB3, WB1, RB4, RB2, WB4, WB2)	[A: 28000, B: 21000, C: 2000]
(RB3, WB3, RB1, RB4, WB1, WB4, RB2, WB2)	[A: 28000, B: 21000, C: 2000]
(RB3, WB3, RB4, RB1, WB4, WB1, RB2, WB2)	[A: 28000, B: 21000, C: 2000]
(RB3, WB3, RB4, WB4, RB1, WB1, RB2, WB2)	[A: 28000, B: 21000, C: 2000]

Serialisability

Two processes/transactions P_1 and P_2 run in serial if one ends before the other starts.

(Note that this could mean that P_1 competely runs before P_2 but also that P_2 completely runs before P_1!)

On the other hand, they are *serialisable* if their outcome is the same as if they were run in serial.

ACID Properties

Database transactions are expected to have the following properties:
(coined in 1983 as a golden standard for Database transaction properties.)

Atomicity:	Transactions are atomic (all-or-nothing).
Consistency:	Database constraints are preserved.
Isolation:	Concurrent transactions do not interfere with each other.
Durability:	Once a transaction has been committed, the changes are permanent, not matter what (for example a system crash).

- Consistency and durability are dealt with “under the hood” by most DBMS;
- Doing changes within BEGIN/COMMIT gives us atomicity;
- Isolation is the tricky part and what caused problems in our example.
  Just running all transactions in a serial way is not a realistic solution!

Interference Problems: Dirty Read

When one transaction is allowed to read a value before the value has been committed.

order	transaction 1	transaction 2
1		INSERT v
2	READ v	
3		ROLLBACK

- **Transaction 1** performs a *dirty read*: reads a value that never actually *existed*;
- There is a violation of the atomicity of **transaction 2**.

Interference Problems: Non-repeatable Read

When one transaction reads the same data twice and gets different results because of another concurrent transaction.

order	Transaction 1	Transaction 2
1	READ v	
2		UPDATE v = v'; COMMIT
3	READ v	

Note: Similar when **Transaction 2** deletes v instead of just updating it.

- **Transaction 1** performs a *non-repeatable read*;
- There is a violation of the atomicity of **transaction 1** since it can observe changes taking place *outside*.

Interference Problems: Phantom

The same query gives different results beacuse of data committed in another concurrent transaction.

order	Transaction 1	Transaction 2
1	SELECT * FROM A	
2		INSERT INTO A ... ; COMMIT
3	SELECT * FROM A	

Note: In non-repeatable reads, rows could have changed or disappeared.

- **Transaction 1** has the *phantom* problem;
- There is a violation of the atomicity of **transaction 1** since it can observe changes taking place *outside*.

Isolation Levels

The isolation level decides how other processes are allowed to interfere.

```
START TRANSACTION isolation_level;
```

Allowed isolations vs level of isolation (from highest to lowest):

	Dirty reads	Non-repeatable reads	Phantoms
SERIALIZABLE	NO	NO	NO
REPEATABLE READ	NO	NO	YES
READ COMMITTED	NO	YES	YES
READ UNCOMMITTED	YES	YES	YES

Recall:

Dirty read:	Reads data that is later rolled back.
Non-repeatable read:	Reads data that is deleted/modified during transaction.
Phantom:	New rows appear in a query result during transaction.

Isolation Levels: Summary

SERIALIZABLE: I accept no interference from other transactions.
Default in **SQL**.

REPEATABLE READ: I accept that others make changes as long as they do not modify or delete data I have already looked at.

Example: When I need to update the price of some products.

READ COMMITTED: I accept that others make any changes to the database.
Default in PostgreSQL.

Example: Removing half of the money in each back account; each update does not depend on other data.

READ UNCOMMITTED: I accept that some data I get might actually never have been in the database (!!).
Not supported in PostgreSQL .

Transactions with JDBC in Java (questions on this in the lab session!)

To use transactions run (only once) `conn.setAutoCommit(false);` where `conn` is your connection object.

End each transaction with `conn.commit();` or `conn.rollback();`, which also starts a new transaction.

To set the isolation level, at the start of the transaction, use `conn.setTransactionIsolation(Connection.<isolation_level>)`.

If the connection closes without committing, the transaction is rolled back.

See the corresponding sections in the JDBC manual.

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 only be allowed to do the things they need for their purpose.

If an attacker gets hold of a username/password, the damage they can do will be limited by the privileges granted to that user.

Databases vs File Systems Privileges/Permissions

In a Unix files system we have

- Three different permissions on files and directories:
  read (r), write (w), execute (x);
- Permissions on three levels: for the owner, for users in the same group, for everyone else.

In an **SQL** database we have:

- Privileges on schema elements (tables, views, triggers, etc);
- Nine different privileges (SELECT, INSERT, UPDATE, DELETE, . . .);
- Privileges are controlled separately for each user;
- Privileges can also be granted to roles that are given to users.

Granting and Revoking Privileges

Initially, all objects (tables, views, etc) are owned by the user who created them.

The owner has all privileges on his/her objects, other user have no access at all.

Privileges can be granted and revoked:

- The owner can *grant* certain privileges to other users;
- One can even *grant* the option to grant privileges to other users;
- Granted privileges can be *revoked*;
- Only the owner of a table can drop the table (and similarly for other schema elements).
  This is a privilege that cannot be granted to other users.

Roles

Users can be given *roles* (which are like groups in Unix).

Privileges can be granted to roles.

This can simplify the administration of privileges.

Example: The roles in a student portal could be: student, teacher, study_administrator.

Granting and Revoking Privileges in Action

- Allows two users/roles to SELECT from Products:

```
GRANT SELECT ON Products TO webshop_users, marketing;
```

- Allows one user to SELECT, DELETE and INSERT on Registrations:

```
GRANT SELECT, DELETE, INSTERT ON Registrations TO study_admin;
```

- Restricts SELECT privilege to certain attributes (can also be done for INSERT and UPDATE privileges):

```
GRANT SELECT (name, email, office) ON Employees TO PUBLIC;
```

- Grants all privileges! (not to be used too often :-):

```
GRANT ALL PRIVILEGES ON ...;
```

- Removes UPDATE and DELETE privileges from a user in Transfers:

```
REVOKE UPDATE, DELETE ON Transfers FROM staff;
```

- Grants a privilege and passes the right to grant the privilege to others:

```
GRANT UPDATE (salary) ON Employees TO Bob WITH GRANT OPTION;
```

Less Common 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.)

Indexes

Consider a table/relation with 50000 rows and the schema

Employees (<u>name</u>, email, office, salary)

How fast are these queries?

```sql
SELECT email FROM Employees WHERE name = 'Ana';
```

```sql
SELECT name FROM Employees WHERE office = '6116';
```

- Finding rows by using primary keys is expected to be fast;
- This is because the DBMS automatically creates an *index* for the primary key;
- Finding rows based on the values of other attributes might require *all rows* to be retrieved and compared.

What is an Index?

An *index* is a data structure that allows values to be found quickly.

PostgreSQL uses *B-Trees* for indexes by default.
It is also possible to use hash tables.

A *Self-balancing tree (B-tree)* is a data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.

They are a generalisation of binary search trees, allowing nodes to have more than two children.

They are optimised for systems that read and write large blocks of data.

What is Indexed?

There is always an index for the primary key.

An index is also added for attributes with a UNIQUE constraints.

Additional indexes can be added and dropped.
Not part of standard **SQL** though.

```
CREATE INDEX index_name ON Table_name (attribute₁, ..., attributeₙ);
```

```
DROP INDEX index_name;
```

Advantages and Disadvantages

Advantages:

- Quicker to find rows matching the values when using WHERE;
- Easier to check that uniqueness constrains are still valid in INSERT and UPDATE operations;
- Faster to find matching rows in JOIN operations;
- Produce sorted/grouped output without sorting when using ORDER BY, GROUP BY;
- Certain comparisons can be optimised using indexes.

Disadvantages:

- Extra work (time) is needed to update the index when data is inserted, updated or deleted;
- Storing the index takes extra space (sometime more than the actual table!).

Overview of Next Lecture

- Relational algebra;
- Correspondence between **SQL** and relational algebra;
- A glance into query optimisation.

Reading:

Book:	chapters 2.4–2.5, 5 and 16.2–16.3
Notes:	chapter 6.1–6.5