# Integrity Constraints, Triggers, Transactions and Procedures

## Database and Web Applications Laboratory

João Correia Lopes

INESC TEC, Faculdade de Engenharia, Universidade do Porto

# GOALS

By the end of this class, the student should be able to:

- Describe how to implement data business rules using SQL
- Describe how to maintain the consistency of the database in the presence of concurrent accesses
- Write database user-defined functions
- Describe some of the main features of the PostgreSQL DBMS

# PREVIOUSLY IN LBAW

- ER: Requirements specification and User interfaces
- EBD: Database specification
  - A4: Conceptual Data Model
  - A5: Relational Schema, validation and schema refinement
  - A6: Indexes, . . .

⇒ `http://web.fe.up.pt/~jlopes/doku.php/teach/lbaw/artefacts`

# Now in LBAW

- EBD: Database specification
  - A4: Conceptual Data Model
  - A5: Relational Schema, validation and schema refinement
  - A6: Indexes, **triggers, transactions, user defined functions**
  - A6: Database populated with data
- EAP: Architecture specification and Prototype
  - ...

# Now in LBAW

- EBD: Database specification
  - A4: Conceptual Data Model
  - A5: Relational Schema, validation and schema refinement
  - A6: Indexes, **triggers, transactions, user defined functions**
  - A6: Database populated with data
- EAP: Architecture specification and Prototype
  - ...

# NOW IN LBAW

- EBD: Database specification
  - A4: Conceptual Data Model
  - A5: Relational Schema, validation and schema refinement
  - A6: Indexes, **triggers, transactions, user defined functions**
  - A6: Database populated with data
- EAP: Architecture specification and Prototype
  - ...

# INDEXES, TRIGGERS, USER FUNCTIONS AND POPULATION (A6)

## A6

This artefact contains the physical schema of the database, the identification and characterization of the indexes, the support of **data integrity rules** with triggers, the definition of the database user-defined functions, and the identification and characterization of the database transactions

This artefact also **includes the complete database creation script, including all SQL necessary to define all integrity constraints, indexes, triggers and transactions**.

# OUTLINE

# INTEGRITY CONSTRAINTS

- Constraints and triggers are **database active elements**
- Integrity constraints enforce, globally, the business rules
  - Defined in expressions or commands that are written once and stored in the database and run in response to certain events
- Integrity constraints specified as part of a database schema:
  - Key Constraints
  - Foreign key constraints
  - Attribute constraints
  - Tuple (record) constraints
  - Domain constraints
- Triggers
- Assertions (N/A)

# KEY CONSTRAINTS

- **Entity integrity**
- Restrict the allowed values for certain attributes (c.f. keys)
- **One** `PRIMARY KEY`
- **Several** candidate keys with `UNIQUE + NOT NULL`
- Out of SQL standard, some implementations automatically create an (hash) index for the primary key fields, to improve performance

```
CREATE TABLE staff(
   id          CHAR(20),
   name        CHAR(50),
   age         INTEGER,
   CONSTRAINT name_age_u UNIQUE (name, age),
   CONSTRAINT staff_pk PRIMARY KEY (id)
);
```

In PostgreSQL use always **lower case** and the snake_case convention

# FOREIGN KEY CONSTRAINTS

- **Referential integrity**
- Foreign keys are declared to force certain values to make sense
- We can declare an attribute (or attributes) of a table as a foreign key referencing attribute(s) of a second table:
  - The attributes of the second table must be a primary key or have a restriction UNIQUE
  - Any value for the attributes of the foreign key, must also be present in the corresponding attributes of the referenced table

```
... REFERENCES <table> (<attribute>)

... CONSTRAINT name_x_fk FOREIGN KEY <attributes>
                         REFERENCES <table> (<attributes>)
```

# MAINTAINING FOREIGN KEY CONSTRAINTS

- The declaration of a foreign key, means that any non-zero value of this key has to appear in the corresponding attributes of the referenced table
- The foreign key constraint is **checked at the end** of each SQL statement able to violate it[1]:
  - Inserts (INSERT queries) and updates (UPDATE queries) in the table that violate the constraint are forbidden
  - The deletions (DELETE queries) and updates (UPDATE queries) in the referenced table that violate the constraint integrity, are treated in accordance with the chosen action:

    `CASCADE` — propagate deletions and changes to the table
    `SET NULL` — puts null values in attributes of the foreign key
    `SET DEFAULT` — puts default values in the attributes
    `RESTRICT` or `NO ACTION` — (default) prohibits such updates or deletions (rollback the transaction)

[1] Checked automatically by the DBMS

# ATTRIBUTE CONSTRAINTS

- **More bussiness rules**
- Restrict the allowed values for the attributes
  - using **CHECK constraints** in their definition
  - using constraints in a domain used in their definition

```
CREATE TABLE actors (
    ... ,
    sex CHAR(1) CONSTRAINT sex_ck CHECK (sex IN ('F','M')),
    ...
```

```
CREATE DOMAIN sex_type CHAR(1)
              CHECK (VALUE IN ('F','M')) DEFAULT 'F';

CREATE TABLE actors (
    ... ,
    sex sex_type,
    ...
```

# ATTRIBUTE CONSTRAINTS

- **More bussiness rules**
- Restrict the allowed values for the attributes
  - using **CHECK constraints** in their definition
  - using constraints in a domain used in their definition

```
CREATE TABLE actors (
    ... ,
    sex CHAR(1) CONSTRAINT sex_ck CHECK (sex IN ('F','M')),
    ...
```

```
CREATE DOMAIN sex_type CHAR(1)
              CHECK (VALUE IN ('F','M')) DEFAULT 'F';

CREATE TABLE actors (
    ... ,
    sex sex_type,
    ...
```

# MAINTAINING ATTRIBUTE CONSTRAINTS

- If a constraint is violated, the SQL statement is undone (**rollback**) and there is a run-time error
- The CHECK condition includes the values to be verified
  - and may include WHERE predicates as in a SQL query: other attributes from the same record, other records or even other tables
- The constraint is checked only in changes to the attributes defined in the schema
  - it is not checked in changes to the other attributes in the condition
- Is this constraint equivalent to a FOREIGN KEY constraint?

```
CREATE TABLE studio (
  ... ,
  president INTEGER CONSTRAINT president_ck CHECK (
     president IN (SELECT ssn FROM staff))
  ...
```

# MAINTAINING ATTRIBUTE CONSTRAINTS

- If a constraint is violated, the SQL statement is undone (**rollback**) and there is a run-time error
- The CHECK condition includes the values to be verified
  - and may include WHERE predicates as in a SQL query: other attributes from the same record, other records or even other tables
- The constraint is checked only in changes to the attributes defined in the schema
  - it is not checked in changes to the other attributes in the condition
- Is this constraint equivalent to a FOREIGN KEY constraint?

```
CREATE TABLE studio (
  ... ,
  president INTEGER CONSTRAINT president_ck CHECK (
     president IN (SELECT ssn FROM staff))
  ...
```

# TUPLE CONSTRAINTS

- **More bussiness rules**
- Generic constraints may involve various attributes or even several tables; there are 2 ways:
  - based in records (using CHECK conditions)
  - generic assertions to express constraints involving various tables or several records variables taking values in the same table (c.f. triggers)
- The condition may be an arbitrary predicate in SQL
- The condition is checked every time a record is inserted or updated

```
CREATE TABLE actors (
   name        CHAR(30) UNIQUE,
   sex         CHAR(1),
   CONSTRAINT male_ck CHECK (sex = 'F' OR name NOT LIKE 'Ms.%')
);
```

# CONSTRAINTS IN POSTGRESQL

- NOT all SQL standard supported

⇒ PostgreSQL 13 Documentation, 5.4. Constraints

# WHAT WE'VE LEARNT

- Integrity constraints enforce business rules globally (specified as part of the DB Schema)

# OUTLINE

# USER DEFINED FUNCTIONS

- **Advantages** of using stored procedures:
  - Reduce the number of round trips between application and database server
    - All SQL statements are wrapped inside a function stored in the database server so the application only has to issue a function call to get the result back
  - Increase the application performance
    - because user-defined functions are pre-compiled and stored in the database server and uses its full-power
  - Be able to be reused in many applications
    - Once a function is developed, any application can use it eliminating the need to write SQL code

*R. Ramakrishnan, J. Gehrke. Stored Procedures. In Database Management Systems. McGRAW-Hill International Editions, 3rd Edition, 2003, Section 6.5.*

# USER DEFINED FUNCTIONS

- **Disadvantages** of stored procedures:
  - Slow software development because it requires specialised skills that many developers do not possess (PL/SQL)
  - Make it difficult to manage versions and hard to debug
  - **Code not portable** to other database management systems (MySQL, SQL Server, PostgreSQL, Oracle, DB2)

*R. Ramakrishnan, J. Gehrke. Stored Procedures. In Database Management Systems. McGRAW-Hill International Editions, 3rd Edition, 2003, Section 6.5.*

# UDF SYNTAX

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $name$
  DECLARE
    declaration;
    [...]
  BEGIN
    < function_body >
    [...]
    RETURN { variable_name | VALUE }
  END;
  $name$ LANGUAGE plpgsql;
```

⇒ PostgreSQL 13 Documentation, CREATE FUNCTION

⇒ PostgreSQL 13 Documentation, Chapter 42. PL/pgSQL - SQL Procedural Language

# EXAMPLE 1

```
CREATE OR REPLACE FUNCTION totalRecords ()
RETURNS INTEGER AS $total$
DECLARE
   total INTEGER;
BEGIN
   SELECT COUNT(*) INTO total FROM company;
   RETURN total;
END;
$total$ LANGUAGE plpgsql;

SELECT totalRecords();
```

⇒ Tutorialspoint. PostgreSQL - Functions

# EXAMPLE 2

```
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        RAISE EXCEPTION 'Unable to create a new job: a job is currently running';
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN
            -- don't worry if it already exists
    END;
END;
$$ LANGUAGE plpgsql;
```

⇒ PostgreSQL 13 Documentation, 42.13. Porting from Oracle PL/SQL

# WHAT WE'VE LEARNT

- User-defined functions in the database server may perform complex computations more efficiently and may be reused

# OUTLINE

# TRIGGERS IN SQL:1999

- An *active database* has a set of associated triggers
- Triggers are **Event-Condition-Action** rules:

  Event — a change to the database that **activates** the trigger
  Condition — a query or test that is run when the trigger is activated
  Action — a procedure that is executed when the trigger is activated and its condition is true

- The action can be executed before, after or instead of the trigger event
- The action may refer the new values and old values of records inserted, updated or deleted in the trigger event that fires it
- The programmer specifies that the action is performed:
  - once for each modified record (`FOR EACH ROW`)
  - once for all records that are changed on a database operation

# TRIGGER PROCEDURES IN POSTGRESQL

- PL/pgSQL can be used to define trigger procedures
- A trigger procedure is created with the CREATE FUNCTION command with no arguments and a return type of trigger
- The function must be declared with no arguments, even if it expects to receive arguments specified in CREATE TRIGGER

⇒ PostgreSQL 13 Documentation, 40.10. Trigger Procedures

# EXAMPLE OF TRIGGER IN PG

- To maintain a table called AUDIT where log messages will be inserted whenever there is an entry in COMPANY table for a new record

```
CREATE TABLE audit(
  emp_id INT NOT NULL,
  entry_date TEXT NOT NULL);
```

```
CREATE TRIGGER example_trigger AFTER INSERT ON COMPANY
  FOR EACH ROW EXECUTE PROCEDURE auditlogfunc();
```

```
CREATE OR REPLACE FUNCTION auditlogfunc() RETURNS TRIGGER AS $example_table$
BEGIN
  INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID, current_timestamp);
  RETURN NEW;
END;
$example_table$ LANGUAGE plpgsql;
```

⇒ Tutorialspoint. PostgreSQL - Triggers

# EXAMPLE OF TRIGGER IN PG

- To maintain a table called AUDIT where log messages will be inserted whenever there is an entry in COMPANY table for a new record

```
CREATE TABLE audit(
  emp_id INT NOT NULL,
  entry_date TEXT NOT NULL);
```

```
CREATE TRIGGER example_trigger AFTER INSERT ON COMPANY
  FOR EACH ROW EXECUTE PROCEDURE auditlogfunc();
```

```
CREATE OR REPLACE FUNCTION auditlogfunc() RETURNS TRIGGER AS $example_table$
BEGIN
  INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID, current_timestamp);
  RETURN NEW;
END;
$example_table$ LANGUAGE plpgsql;
```

⇒ Tutorialspoint. PostgreSQL - Triggers

# EXAMPLE OF TRIGGER IN PG

- To maintain a table called AUDIT where log messages will be inserted whenever there is an entry in COMPANY table for a new record

```
CREATE TABLE audit(
  emp_id INT NOT NULL,
  entry_date TEXT NOT NULL);
```

```
CREATE TRIGGER example_trigger AFTER INSERT ON COMPANY
  FOR EACH ROW EXECUTE PROCEDURE auditlogfunc();
```

```
CREATE OR REPLACE FUNCTION auditlogfunc() RETURNS TRIGGER AS $example_table$
BEGIN
  INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID, current_timestamp);
  RETURN NEW;
END;
$example_table$ LANGUAGE plpgsql;
```

⇒ Tutorialspoint. PostgreSQL - Triggers

# WHAT WE'VE LEARNT

- Integrity constraints enforce business rules globally (specified as part of the DB Schema)
- Triggers enforce any business rules at the database level

# OUTLINE

# TRANSACTIONS

- Concurrent execution of user programs is essential for good DBMS performance
  - because disk access is frequent, and relatively slow, it is important to keep the CPU busy working on several programs user programs concurrently
- **DANGER!! DANGER!!** this may lead to inconsistencies!
- A user program can perform many operations on data from the database, but the DBMS is only concerned with the data that is read/written to/from the database
- A **transaction** is the DBMS's abstract view of a user program: a sequence of reads and writes operations
- A transaction leaves the database in a consistent state, i.e. respecting all integrity constraints (ICs).

*R. Ramakrishnan, J. Gehrke. Overview of Transaction Management. In Database Management Systems. McGRAW-Hill International Editions, 3rd Edition, 2003, Chapter 16.*

# CONCURRENCY IN A DBMS

- Users submit transactions, and can think of each transaction as executing by itself
  - Concurrency is **achieved by the DBMS**, which interweaves actions (reads/writes of database objects) or various transactions
  - Each transaction must leave the database in a consistent state, if the database is consistent when the transaction begins
    - The DBMS checks some integrity constraints, e.g. ICs declared in CREATE TABLE
    - In addition, the DBMS does not understand the semantics of the data (e.g. it does not understand how the interest on a bank account is computed)
- **Issues**: effect of **interleaving** transactions (Concurrency) and **crashes** (Recovery)

*R. Ramakrishnan, J. Gehrke. Overview of Transaction Management. In Database Management Systems. McGRAW-Hill International Editions, 3rd Edition, 2003, Chapter 16.*

# ACID PROPERTIES (RECAP)

- **Atomicity**
  the result of all operations is reflected in the BD or none
- **Consistency**
  isolated execution of the transaction (i.e. it is the only running) preserves the consistency of the database
- **Isolation**
  even if multiple transactions are in concurrent execution, for a pair $T_i$ and $T_j$, $T_i$ considers that $T_j$ finished before $T_i$ started or $T_j$ begins after $T_i$ ends
- **Durability**
  after a successful completion of the transaction, all changes made to the database persist (even after system failures)
- **Concurrency control**: ensures Consistency and Isolation, given Atomicity
- **Recovery**: guarantees Atomicity and Durability

# ACID PROPERTIES (RECAP)

- **Atomicity**
  the result of all operations is reflected in the BD or none
- **Consistency**
  isolated execution of the transaction (i.e. it is the only running) preserves the consistency of the database
- **Isolation**
  even if multiple transactions are in concurrent execution, for a pair $T_i$ and $T_j$, $T_i$ considers that $T_j$ finished before $T_i$ started or $T_j$ begins after $T_i$ ends
- **Durability**
  after a successful completion of the transaction, all changes made to the database persist (even after system failures)
- **Concurrency control**: ensures Consistency and Isolation, given Atomicity
- **Recovery**: guarantees Atomicity and Durability

# EXAMPLE: BANK TRANSFER

```
READ(A); A:= A - 5000; WRITE(A);
READ(B); B:= B + 5000; WRITE(B);
```

- Consistency
  - The sum (balance of A + balance B) remains unchanged
  - If the database is in a consistent state before $T_i$ run, it will also be in a consistent state after $T_i$ run
- Atomicity
  - What happens if there is a fault after WRITE(A) and before WRITE(B)?
  - The idea is to keep track of write operations and undo them or redo them again (recovery)
- Durability
  - Changes are also written to disk
  - The existing information, written in the disk, about the modifications is sufficient to allow subsequent recovery in case of failure
- Isolation
  - Serialisability concept

# EXAMPLE: BANK TRANSFER

```
READ(A); A:= A - 5000; WRITE(A);
READ(B); B:= B + 5000; WRITE(B);
```

- Consistency
  - The sum (balance of A + balance B) remains unchanged
  - If the database is in a consistent state before $T_i$ run, it will also be in a consistent state after $T_i$ run
- Atomicity
  - What happens if there is a fault after WRITE(A) and before WRITE(B)?
  - The idea is to keep track of write operations and undo them or redo them again (recovery)
- Durability
  - Changes are also written to disk
  - The existing information, written in the disk, about the modifications is sufficient to allow subsequent recovery in case of failure
- Isolation
  - Serialisability concept

# EXAMPLE: BANK TRANSFER

```
READ(A); A:= A - 5000; WRITE(A);
READ(B); B:= B + 5000; WRITE(B);
```

- Consistency
  - The sum (balance of A + balance B) remains unchanged
  - If the database is in a consistent state before $T_i$ run, it will also be in a consistent state after $T_i$ run
- Atomicity
  - What happens if there is a fault after WRITE(A) and before WRITE(B)?
  - The idea is to keep track of write operations and undo them or redo them again (recovery)
- Durability
  - Changes are also written to disk
  - The existing information, written in the disk, about the modifications is sufficient to allow subsequent recovery in case of failure
- Isolation
  - Serialisability concept

# EXAMPLE: BANK TRANSFER

```
READ(A); A:= A - 5000; WRITE(A);
READ(B); B:= B + 5000; WRITE(B);
```

- Consistency
  - The sum (balance of A + balance B) remains unchanged
  - If the database is in a consistent state before $T_i$ run, it will also be in a consistent state after $T_i$ run
- Atomicity
  - What happens if there is a fault after WRITE(A) and before WRITE(B)?
  - The idea is to keep track of write operations and undo them or redo them again (recovery)
- Durability
  - Changes are also written to disk
  - The existing information, written in the disk, about the modifications is sufficient to allow subsequent recovery in case of failure
- Isolation
  - Serialisability concept

# EXAMPLE: BANK TRANSFER

```
READ(A); A:= A - 5000; WRITE(A);
READ(B); B:= B + 5000; WRITE(B);
```

- Consistency
  - The sum (balance of A + balance B) remains unchanged
  - If the database is in a consistent state before $T_i$ run, it will also be in a consistent state after $T_i$ run
- Atomicity
  - What happens if there is a fault after WRITE(A) and before WRITE(B)?
  - The idea is to keep track of write operations and undo them or redo them again (recovery)
- Durability
  - Changes are also written to disk
  - The existing information, written in the disk, about the modifications is sufficient to allow subsequent recovery in case of failure
- Isolation
  - Serialisability concept

# ATOMICITY OF TRANSACTIONS

- A transaction can **commit** after completing all its actions, or may abort (**rollback**) (or be aborted by the DBMS) after running only part of these actions
- A very important property guaranteed by the DBMS for all transactions is that they are atomic. That is, a user can think of a transaction as always executing all its actions in a single step, or perform any of them
  - The DBMS records in a **log** all actions to be able to undo the actions of aborted transactions
- Note that **individual SQL DDL statements are atomic** and therefore there's no need to define transactions for them
  - DBMSs issues an implicit COMMIT before and after any data definition language (DDL) statement

# EXAMPLE

```
T1:  BEGIN  A=A+100,   B=B-100    END
T2:  BEGIN  A=1.05*A,  B=1.05*B   END
```

- Intuitively, the 1st transaction transfers 100 from account B to account A. The 2nd credits in both accounts a 5% interest
- There is no guarantee that T1 will execute before T2 or vice versa, if both are submitted together. However, the net effect is equivalent to the transactions being run in series, in any order — $(A + B) * 1.05$ constant
- Considering two possible intertwined schedules:

```
T1:  A=A+100,             B=B-100
T2:            A=1.05*A,            B=1.05*B
```

```
T1:  A=A+100, /* R(A),W(A) */     B=B-100 /* R(B),W(B) */
T2:            A=1.05*A, B=1.05*B
```

# SCHEDULING TRANSACTIONS

- **Serial schedule**: Schedule that does not interleave the actions of different transactions
- **Equivalent schedules**: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule
- **Serialisable schedule**: A schedule that is equivalent to some serial execution of the transactions

(Note: If each transaction preserves consistency, every serialisable schedule preserves consistency.)

*R. Ramakrishnan, J. Gehrke. Overview of Transaction Management. In Database Management Systems. McGRAW-Hill International Editions, 3rd Edition, 2003, Chapter 16.*

# ANOMALIES WITH INTERLEAVED EXECUTION

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

```
T1:  R(A), W(A),                    R(B), W(B), Abort
T2:                R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts):

```
T1:  R(A),                    R(A), W(A), C
T2:         R(A), W(A), C
```

- Overwriting Uncommitted Data (WW Conflicts):

```
T1:  W(A),                    W(B), C
T2:         W(A), W(B), C
```

*R. Ramakrishnan, J. Gehrke. Overview of Transaction Management. In Database Management Systems. McGRAW-Hill International Editions, 3rd Edition, 2003, Chapter 16.*

# ANOMALIES WITH INTERLEAVED EXECUTION

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

```
T1:  R(A), W(A),                    R(B), W(B), Abort
T2:                R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts):

```
T1:  R(A),                    R(A), W(A), C
T2:         R(A), W(A), C
```

- Overwriting Uncommitted Data (WW Conflicts):

```
T1:  W(A),                    W(B), C
T2:         W(A), W(B), C
```

*R. Ramakrishnan, J. Gehrke. Overview of Transaction Management. In Database Management Systems. McGRAW-Hill International Editions, 3rd Edition, 2003, Chapter 16.*

# ANOMALIES WITH INTERLEAVED EXECUTION

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

```
T1:  R(A), W(A),                    R(B), W(B), Abort
T2:               R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts):

```
T1:  R(A),                   R(A), W(A), C
T2:         R(A), W(A), C
```

- Overwriting Uncommitted Data (WW Conflicts):

```
T1:  W(A),                   W(B), C
T2:         W(A), W(B), C
```

*R. Ramakrishnan, J. Gehrke. Overview of Transaction Management. In Database Management Systems. McGRAW-Hill International Editions, 3rd Edition, 2003, Chapter 16.*

# TRANSACTION SUPPORT IN SQL2

- A transaction begins implicitly with any query or command that manipulates the database
- The COMMIT command ends the transaction successfully
- The ROLLBACK command aborts the transaction and undoes the changes
- We can declare that the (next) transaction is read only:

```
SET TRANSACTION READ ONLY;
```

- The serialisability is the default, but the programmer may specify a different behaviour with ISOLATION LEVEL:

```
SET TRANSACTION READ WRITE ISOLATION LEVEL SERIALIZABLE;
```

# TRANSACTION SUPPORT IN SQL2 (2)

- Other consistency levels instead of SERIALIZABLE:
  - REPEATABLE READ — transactions can only read committed records and between two reads the transactions cannot modify the record
  - READ COMMITED — transactions can only read committed records (between two reads the record may have been modified)
  - READ UNCOMMITED — records still uncommitted can be read

| Isolation Level | Dirty Read | Nonrepeatable Read | Phantom Read |
|---|---|---|---|
| **Read uncommitted** | Possible | Possible | Possible |
| **Read committed** | Not possible | Possible | Possible |
| **Repeatable read** | Not possible | Not possible | Possible |
| **Serializable** | Not possible | Not possible | Not possible |

⇒ PostgreSQL 13 Documentation, 13.2. Transaction Isolation

# EXPLICIT LOCKING

- The DBMS performs locking automatically
- In certain cases, however, locking must be controlled manually by specifying a transaction's lock type and scope
- If ONLY is not specified, the table and all its descendant tables (if any) are locked
- If no lock mode is specified, then ACCESS EXCLUSIVE, the most restrictive mode, is used
- Once obtained, the lock is held for the remainder of the current transaction (there's no UNLOCK command)

```
LOCK [ TABLE ] [ ONLY ] <table> IN <lock_mode> [ NOWAIT ]
```

⇒ PostgreSQL 13 Documentation, LOCK

# TRANSACTIONS IN POSTGRESQL

```
SET TRANSACTION transaction_mode

where transaction_mode is one of:

  ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ |
                    READ COMMITTED | READ UNCOMMITTED }
  READ WRITE | READ ONLY
  [ NOT ] DEFERRABLE
```

⇒ PostgreSQL 13 Documentation, 3.4. Transactions

⇒ PostgreSQL 13 Documentation, 13.2. Transaction Isolation

# WHAT WE'VE LEARNT

- Concurrency control and recovery are among the most important functions provided by a DBMS
- Concurrent execution of user programs is essential for good DBMS performance and **transactions must be carefully designed for performance and consistency**
- Users (SQL programmers) need not worry about concurrency, as long as they specify the **transactions isolation level**

# NOW IN LBAW

- **EBD**: Database specification
  - A4: Conceptual Data Model
  - A5: Relational Schema, validation and schema refinement
  - **A6**: Indexes, triggers, transactions, user defined functions
  - **A6**: The SQL script to create in the database all the above
  - **A6**: The SQL script to populate a database with test data with an amount of tuples suitable for testing and with plausible values for the fields from the database

# Now in LBAW

- **EBD**: Database specification
  - A4: Conceptual Data Model
  - A5: Relational Schema, validation and schema refinement
  - **A6**: Indexes, triggers, transactions, user defined functions
  - **A6**: The SQL script to create in the database all the above
  - **A6**: The SQL script to populate a database with test data with an amount of tuples suitable for testing and with plausible values for the fields from the database