Unit 9: Data security management.

2022/2023

Contents

9.1. Failures and fault recovery.
9.2. Transactions and concurrency.
9.3. Dumping and restoring data. Backup and recovery PITR.
9.4. Database integrity verification tools.
9.5. Postgresql configuration files.
9.6. Extensions.
9.7. Migration of data between management systems.
9.8. Documentation of security measures and policies.
9.9. SQLi.

9.1. Failures and fault recovery.

9.1.1. Data protection.

The main data protection regulation in Spain is the Organic Law 3/2018, of December 5, on the Protection of Personal Data and Guarantee of Digital Rights (**LOPDGDD**), which implements the General Data Protection Regulation (**GDPR**) in the country.

The LOPDGDD regulates the processing of personal data, including their **collection, use, storage, and transfer, by individuals, companies, and public bodies**.

9.1.1. Data protection.

Under the **LOPDGDD**, individuals have the right to **access, rectify, erase, restrict, and object to the processing** of their personal data.

They also have the right to **data portability**, which allows them to receive their data in a structured and commonly used format and transfer it to another controller.

Organizations must obtain **explicit consent** from individuals before processing their personal data, and must provide **information about the purposes of the processing, the categories of data being processed, and the rights of the data subjects**.

9.1.1. Data protection.

Organizations must also implement appropriate **technical and organizational measures to ensure the security** of personal data and to prevent **unauthorized access, disclosure, or destruction**.

Violations of the LOPDGDD can result in significant fines and penalties, including fines of up to €20 million or 4% of the organization's global annual revenue, whichever is higher. Additionally, individuals have the right to file complaints with the Spanish Data Protection Agency (AEPD) and to seek compensation for damages resulting from violations of their data protection rights.

9.1.1. Data protection.

In case of a cyber attack that results in a **breach of the LOPDGDD and/or GDPR**, it is important to take immediate action to mitigate the damage and comply with the legal requirements. The following steps should be taken:

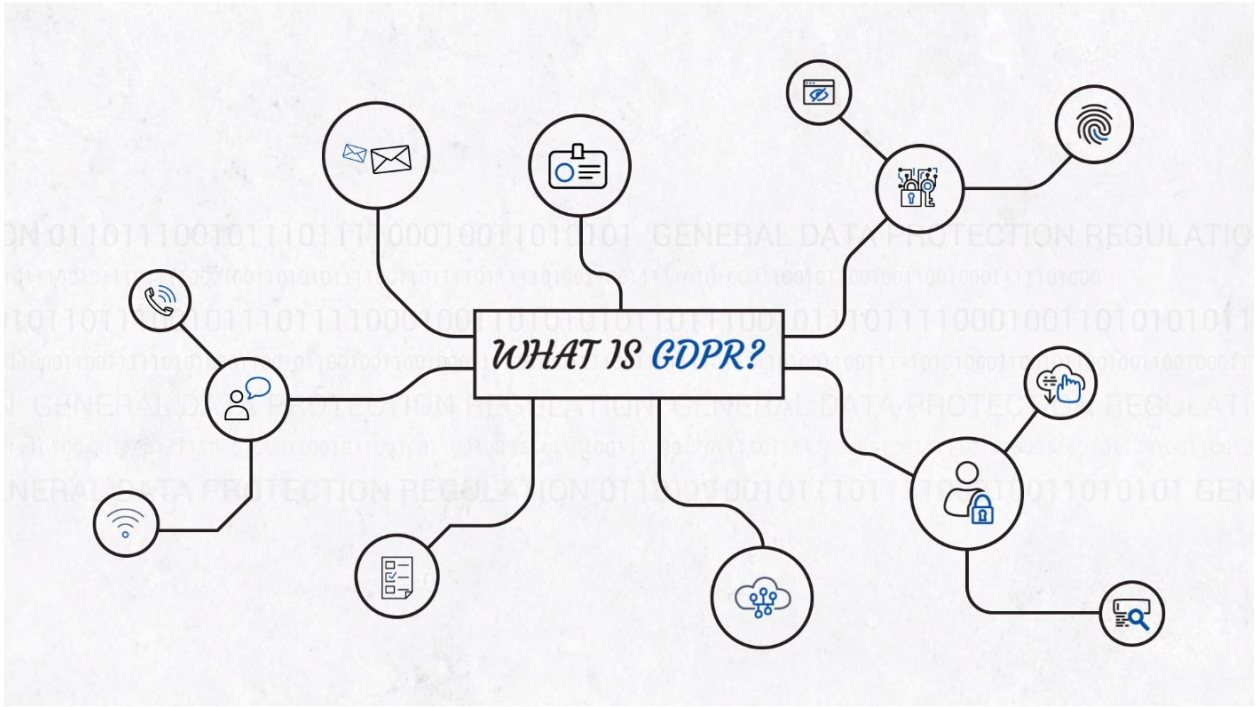
1. **Assess the extent of the breach**: Determine the scope of the breach, what information was affected, and who may have been impacted.
2. **Notify the relevant authorities**: In Spain, the Spanish Data Protection Agency (AEPD) must be notified of any data breaches within 72 hours of becoming aware of the incident.
3. **Notify affected individuals**: If the breach is likely to result in a high risk to the rights and freedoms of individuals, they must be notified as soon as possible.
4. **Implement remedial measures**: Take steps to prevent further damage and prevent similar incidents from occurring in the future.
5. **Document the incident**: Keep a detailed record of the breach, including the steps taken to address it, in order to comply with legal requirements and demonstrate due diligence.

9.1.1. Data protection.

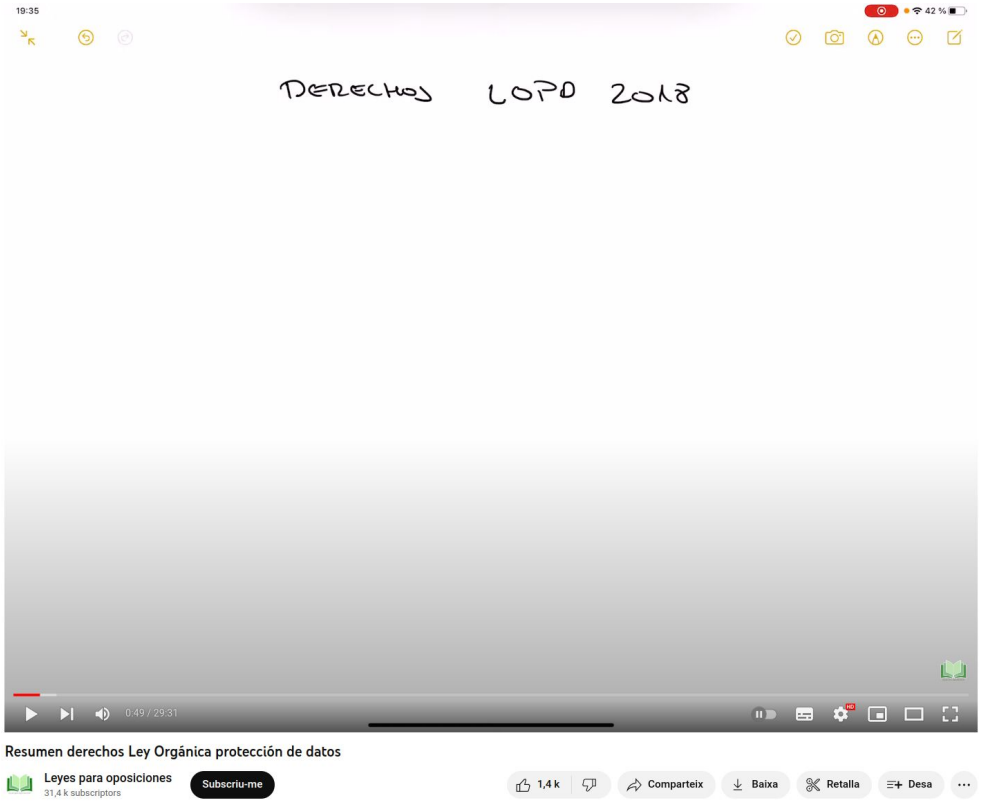
It is important to note that: failure to comply with the LOPDGDD and/or GDPR can result in significant fines and penalties, so it is crucial to take these regulations seriously and ensure that appropriate measures are in place to prevent data breaches and respond effectively if they do occur.

9.1.1. Data protection.

9.1.1. Data protection.



9.1.1. Data protection.



9.1.1. Data protection.

Conclusion:

Data protection regulation in Spain is important to **protect individual rights** and ensure that organizations process their **personal data** properly and securely. The **LOPDGDD and the GDPR set clear standards for the processing of personal data**, and violations of these regulations can have serious consequences for organizations.

9.1.2. Tablespaces.

Allow database administrators to control the **physical location of database objects** such as tables, indexes, and stored procedures.

By default, all database objects are stored in the "pg_global" tablespace, which is located in the PostgreSQL data directory.

Using tablespaces, database administrators can **move database objects to different physical locations**, such as separate disks or storage devices, to improve performance or manage disk space.

Tablespaces can also be used to **distribute data across multiple disks for load balancing**.

9.1.2. Tablespaces.

To create a new tablespace, the "CREATE TABLESPACE" command is used, specifying the location where the tablespace should be created. For example:

```
CREATE TABLESPACE mytablespace LOCATION '/path/to/mytablespace';
```

Once the tablespace is created, it can be **assigned to a database object** using the "ALTER TABLE" or "ALTER INDEX" command. For example:

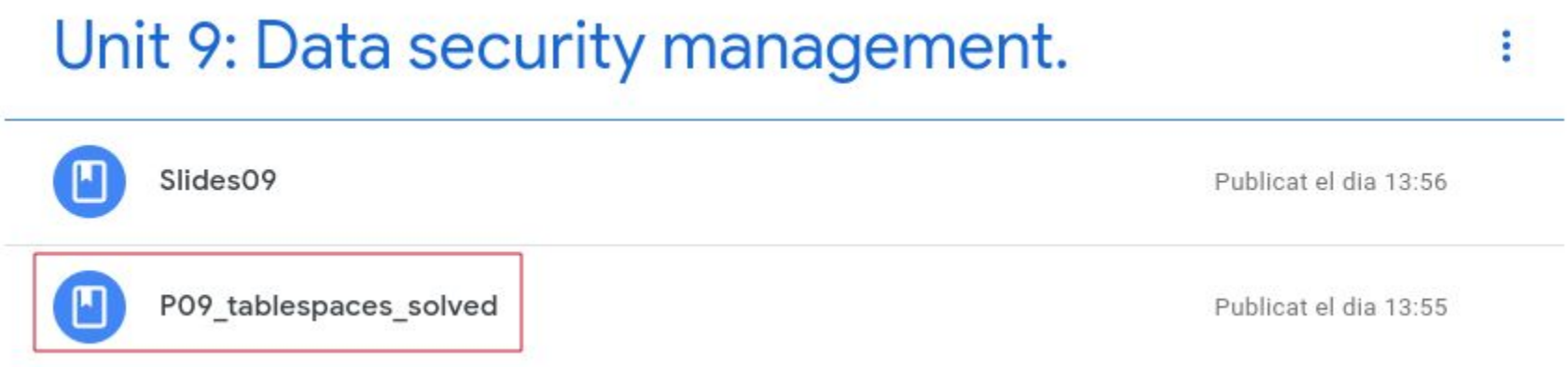
```
ALTER TABLE mytable SET TABLESPACE mytablespace;
```

9.1.2. Tablespaces.

Tablespaces can also be used **to manage backup and recovery operations**, as individual tablespaces can be backed up and restored independently of the main database (not recommended by PsotgreSQL).

It is important to note that **tablespaces are only available in the PostgreSQL server**, and not in the client applications that interact with the server. Therefore, applications must be designed to work with the tablespace configuration of the server.

9.1.2. Tablespaces.



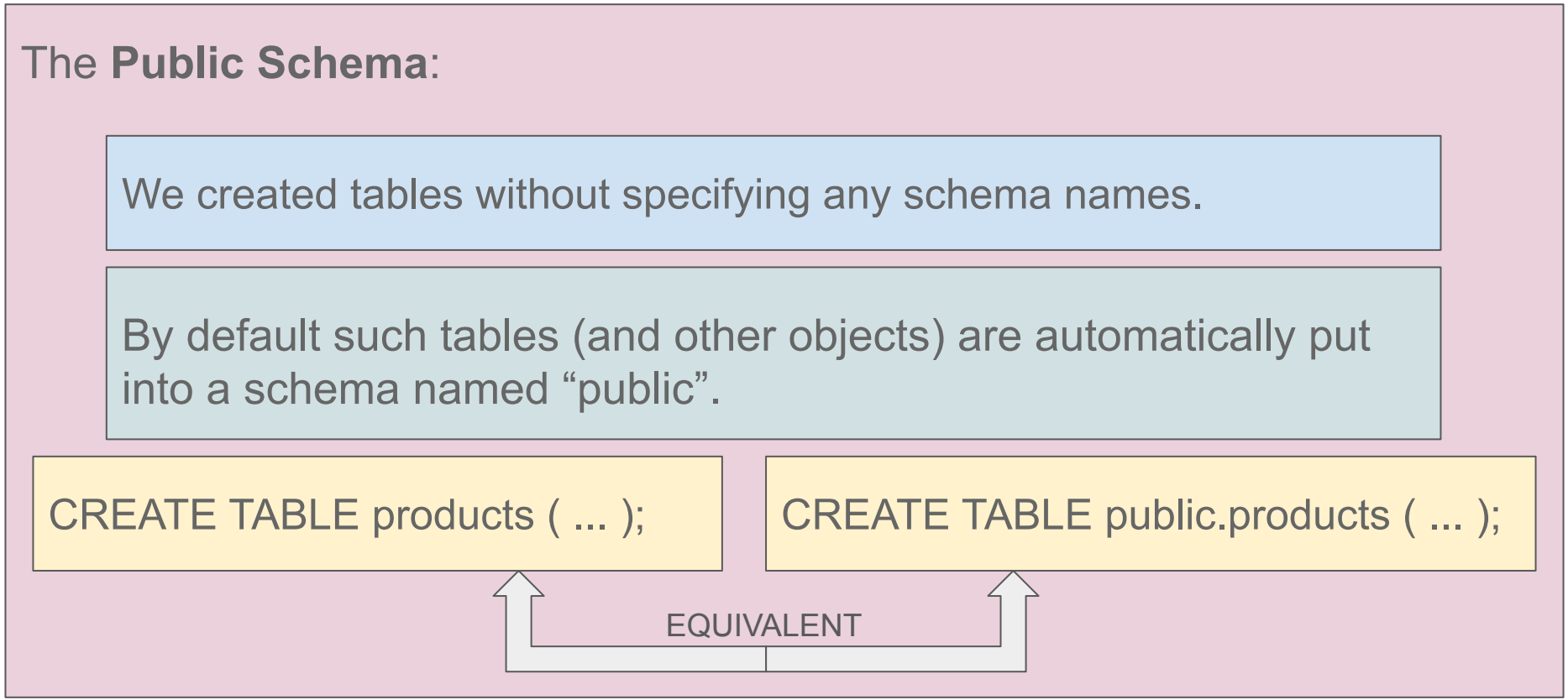
Problems solved: better performance and hard disk out of space

9.1.3. Schemas.

Overall, dividing a database into schemas can provide many **benefits**, including:

- improved organization and management
- easier scalability
- better performance
- enhanced security
- easier customization

9.1.3. Schemas.


The **Public Schema**:
We created tables without specifying any schema names.
By default such tables (and other objects) are automatically put into a schema named “public”.
CREATE TABLE products (...);
CREATE TABLE public.products (...);
EQUIVALENT

9.1.3. Schemas.

Dividing a database into schemas can provide several advantages, including:

1. **Improved organization and management**: Dividing a database into schemas can make it easier to manage, as database objects can be logically grouped together by schema. This can help **simplify administration tasks, such as backups, security, and maintenance**.
2. **Enhanced security**: Schemas can be used to control access to specific database objects, allowing database administrators to **grant or revoke access at the schema level**. This can help improve security by restricting access to sensitive data or operations.
3. **Easier customization**: Schemas can be used to customize database objects for specific users or applications. For example, a **schema** could be created **for a specific department within an organization**, allowing them to have their own **customized views** of the data.
4. **Better performance**: By separating large tables or frequently accessed data into separate schemas, database performance can be improved. This is because smaller tables can be cached more efficiently, and queries can be optimized for specific schemas.
5. **Easier scalability**: Schemas can be used to **partition data across multiple servers or databases**, allowing for easier scaling of the system. This can help improve performance and accommodate growing amounts of data.

9.1.3. Schemas.

To create a new schema, you can use the "**CREATE SCHEMA**" command, followed by the name of the schema you want to create. For example, to create a schema called "sales":

```
CREATE SCHEMA sales;
```

You can also specify the owner of the schema using the "**AUTHORIZATION**" clause. For example, to create a schema called "sales" owned by a user named "john":

```
CREATE SCHEMA sales AUTHORIZATION john;
```

9.1.3. Schemas.

Once you have created a schema, you can create tables, views, indexes, and other objects within that schema. To create a new table within a schema, you can use the "CREATE TABLE" command, specifying the schema name and table name. For example, to create a table called "customers" within the "sales" schema:

```
CREATE TABLE sales.customers (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT NOT NULL
);
```

database.schema.table

9.1.3. Schemas.

To specify a schema for an existing table, you can use the "**ALTER TABLE**" command, followed by the table name and the "**SET SCHEMA**" clause. For example, to move a table called "orders" to the "sales" schema:

```
ALTER TABLE orders SET SCHEMA sales;
```

You can also grant permissions to a schema using the "**GRANT**" command. For example, to grant read and write access to a user named "jane" for the "sales" schema:

```
GRANT USAGE, CREATE, TEMPORARY ON SCHEMA sales TO jane;
```

9.1.3. Schemas.

The variable **SEARCH_PATH**, which can be set at the cluster level or at user level, indicates a list of schemas where objects are searched in case the name of the owner is not given.

```
SET search_path TO my_schema, public;
ALTER USER user_name SET search_path = my_schema, public;
```

You can see your **search path** with:

```
SHOW search_path;
```

To **drop a schema** including all contained objects, use:

```
DROP SCHEMA myschema CASCADE;
```

9.1.3. Schemas.

schema named “public” (described as “standard public schema”)

Access privileges list: is of interest here. The format of the privilege information provides three items: the privilege grantee, the privileges, and privilege grantor(format “grantee=privileges/grantor”).

```
sampledb=# \dn+
                                  List of schemas
  Name  |  Owner   |  Access privileges   |   Description
--------+----------+----------------------+-----------------------
 public | postgres | postgres=UC/postgres+| standard public schema
        |          | =UC/postgres         |
(1 row)
```

owned by the role “postgres”

U=Usage, C=Create

An empty string is found here. This is how privileges granted to all users (PUBLIC).

Granting usage and create privileges on the public schema to all users is viewed by some as possibly contrary to general security principles best practices

9.1.3. Schemas.

- sampledb=# REVOKE USAGE ON SCHEMA public FROM PUBLIC;
- sampledb=# REVOKE CREATE ON SCHEMA public FROM PUBLIC;

```
sampledb=# \dn+
                          List of schemas
  Name  |  Owner   |  Access privileges   |      Description
--------+----------+----------------------+------------------------
 public | postgres | postgres=UC/postgres | standard public schema
(1 row)
```

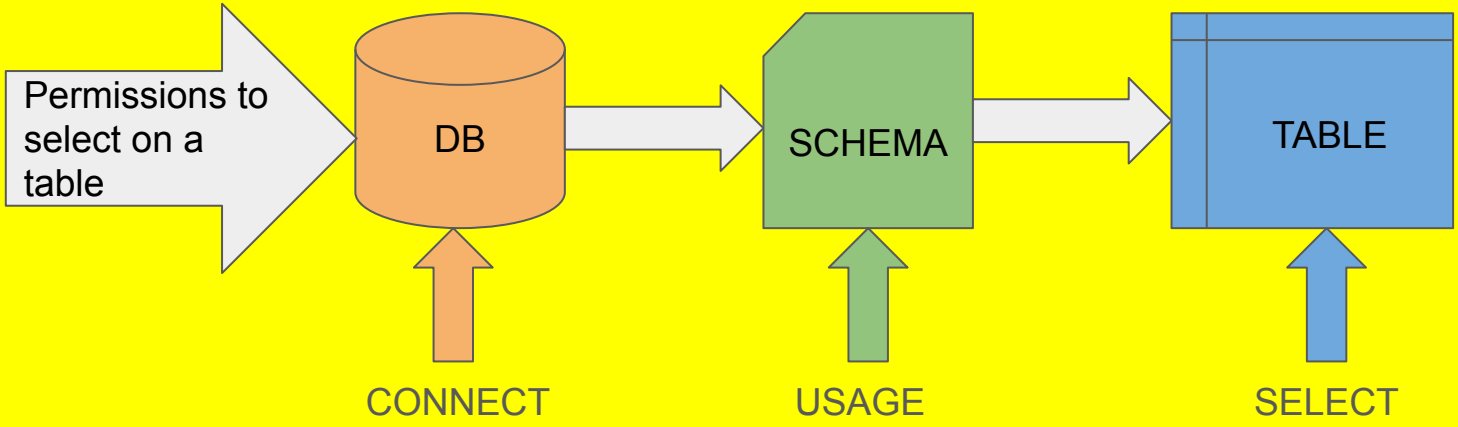
- sampledb=# REVOKE ALL PRIVILEGES ON SCHEMA public FROM PUBLIC;

```
sampledb=# \dn+
                          List of schemas
  Name  |  Owner   | Access privileges |      Description
--------+----------+-------------------+------------------------
 public | postgres |                   | standard public schema
(1 row)
```

9.1.3. Schemas.

GRANTing on a database doesn't GRANT rights to the schema within.

Similarly, GRANTing on a schema does not grant rights on the tables within.



9.1.3. Schemas.

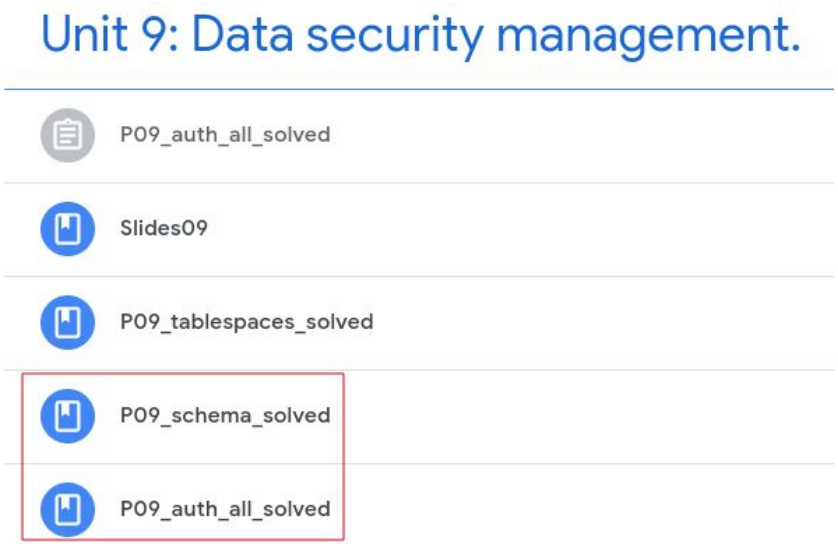
- Read this article: https://dba.stackexchange.com/questions/117109/how-to-manage-default-privileges-for-users-on-a-database-vs-schema/117661#117661

9.1.3. Schemas.

Overall, creating schemas in PostgreSQL is a straightforward process that can help you organize and manage your database more effectively.

https://www.postgresql.org/docs/current/ddl-schemas.html

9.1.3. Schemas.



Problems solved: security of data (unauthorized accesses)

9.1.4. Sequences.

A **sequence** is a special type of database object created **to generate unique numeric identifiers** in the PostgreSQL database.

The **sequence objects** (also known as sequence generators or simply sequences) are **single-row tables** created via a command from the command line: **`CREATE SEQUENCE`**.

Serial Vs Sequence

Examples of sequences:

- {1,2,3,4,5}
- {5,4,3,2,1}
- {5,10,15,20,25}
- {10,20,30,40}

9.1.4. Sequences.

- **Sequences**:
  - CREATE SEQUENCE [IF NOT EXISTS] sequence_name

[AS { SMALLINT | INT | BIGINT }]
[INCREMENT [BY] increment]
[MINVALUE minvalue | NO MINVALUE]
[MAXVALUE maxvalue | NO MAXVALUE]
[START [WITH] start]
[CACHE cache]
[[NO] CYCLE]
[OWNED BY { table_name.column_name | NONE }]

allows the sequence to wrap around when the ***maxvalue*** or ***minvalue*** has been reached by an ascending or descending sequence respectively

data type

+1, +5, etc

Is there a minimum value?

Is there a maximum value?

allows the sequence to begin anywhere (the default starting value is ***minvalue*** for ascending sequences and ***maxvalue*** for descending ones)

how many sequence numbers are to be preallocated and stored in memory for faster access

sequence associated with a specific table column

https://www.PostgreSQL.org/docs/current/sql-createsequence.html

9.1.4. Sequences.

The **sequence manipulation functions** offer a simple and safe multi-user methods to work with sequence objects.

```
SELECT setval('foo', 42);
-- Next nextval will return 43
SELECT setval('foo', 42, true);
-- Same as above
SELECT setval('foo', 42, false);
-- Next nextval will return 42
```

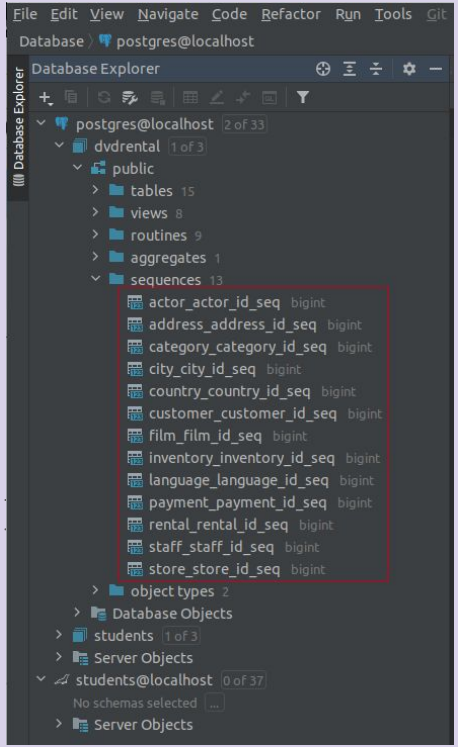
Function	Return Type	Description
`currval(regclass)`	`bigint`	Return value most recently obtained with `nextval` for specified sequence
`lastval()`	`bigint`	Return value most recently obtained with `nextval` for any sequence
`nextval(regclass)`	`bigint`	Advance sequence and return new value
`setval(regclass, bigint)`	`bigint`	Set sequence's current value
`setval(regclass, bigint, boolean)`	`bigint`	Set sequence's current value and `is_called` flag

9.1.4. Sequences.

Listing all sequences in a database:

```
SELECT
    relname sequence_name
FROM
    pg_class
WHERE
    relkind = 'S';
```

```
dvdrental=# SELECT
    relname sequence_name
FROM
    pg_class
WHERE
    relkind = 'S';
       sequence_name
---------------------------
 customer_customer_id_seq
 actor_actor_id_seq
 category_category_id_seq
 film_film_id_seq
 address_address_id_seq
 city_city_id_seq
 country_country_id_seq
 inventory_inventory_id_seq
 language_language_id_seq
 payment_payment_id_seq
 rental_rental_id_seq
 staff_staff_id_seq
 store_store_id_seq
(13 rows)
```



Deleting sequences:

```
DROP SEQUENCE [ IF EXISTS ] sequence_name [, ...
[ CASCADE | RESTRICT ];
```

9.1.4. Sequences.

Example 1:

```
CREATE SEQUENCE mysequence
INCREMENT 5
START 100;
```

```
postgres@192:sequences> SELECT nextval('mysequence');
+-----------+
| nextval   |
|-----------|
| 100       |
+-----------+
SELECT 1
Time: 0.030s
postgres@192:sequences> SELECT nextval('mysequence');
+-----------+
| nextval   |
|-----------|
| 105       |
+-----------+
SELECT 1
Time: 0.018s
```

Everytime that you call “nextval’ you get the next value of the sequence.

9.1.4. Sequences.

Example 2:

```
CREATE SEQUENCE three
INCREMENT -1
MINVALUE 1
MAXVALUE 3
START 3
CYCLE;
```

Cycle option is activated

```
postgres@192:sequences> SELECT nextval('three');
+-----------+
| nextval   |
|-----------|
| 3         |
+-----------+
SELECT 1
Time: 0.018s
postgres@192:sequences> SELECT nextval('three');
+-----------+
| nextval   |
|-----------|
| 2         |
+-----------+
SELECT 1
Time: 0.017s
postgres@192:sequences> SELECT nextval('three');
+-----------+
| nextval   |
|-----------|
| 1         |
+-----------+
SELECT 1
Time: 0.018s
postgres@192:sequences> SELECT nextval('three');
+-----------+
| nextval   |
|-----------|
| 3         |
+-----------+
SELECT 1
Time: 0.020s
```

9.1.4. Sequences.

Example 3:

```
CREATE SEQUENCE department_num
START 10
INCREMENT 10
MINVALUE 10;

create table departments(
    num INTEGER DEFAULT
nextval('department_num') NOT NULL
PRIMARY KEY,
    name text NOT NULL,
    town text
);
```

Sequence to define a PK

```
INSERT INTO departments (name, town)
VALUES ('ACCOUNTING','SEVILLA');
INSERT INTO departments (name, town)
VALUES ('RESEARCH','MADRID');
INSERT INTO departments (name, town)
VALUES ('SALES','BARCELONA');
INSERT INTO departments (name, town)
VALUES ('PRODUCTION','BILBAO');
```

```
postgres@192:sequences> select * from departments;
+-------+------------+-----------+
| num   | name       | town      |
|-------+------------+-----------|
| 10    | ACCOUNTING | SEVILLA   |
| 20    | RESEARCH   | MADRID    |
| 30    | SALES      | BARCELONA |
| 40    | PRODUCTION | BILBAO    |
+-------+------------+-----------+
SELECT 4
Time: 0.022s
```

9.1.4. Sequences.

Example 3:

```
postgres@192:sequences> select * from departments;
+-------+------------+-----------+
| num   | name       | town      |
|-------+------------+-----------|
| 10    | ACCOUNTING | SEVILLA   |
| 20    | RESEARCH   | MADRID    |
| 30    | SALES      | BARCELONA |
| 40    | PRODUCTION | BILBAO    |
+-------+------------+-----------+
SELECT 4
Time: 0.020s
postgres@192:sequences> select lastval();
+-----------+
| lastval   |
|-----------|
| 40        |
+-----------+
SELECT 1
Time: 0.017s
```

Returns value most recently obtained with nextval for any sequence

```
postgres@192:sequences> select setval('department_num',20);
+----------+
| setval   |
|----------|
| 20       |
+----------+
SELECT 1
Time: 0.019s
postgres@192:sequences> select nextval('department_num');
+-----------+
| nextval   |
|-----------|
| 30        |
+-----------+
SELECT 1
Time: 0.020s
postgres@192:sequences> select nextval('department_num');
+-----------+
| nextval   |
|-----------|
| 40        |
+-----------+
SELECT 1
Time: 0.019s
postgres@192:sequences> select currval('department_num');
+-----------+
| currval   |
|-----------|
| 40        |
+-----------+
SELECT 1
Time: 0.024s
```

Set sequence current value

Return value most recently obtained with nextval for specified sequence

9.1.4. Sequences.

Creating a sequence associated with a table column

Example 4:

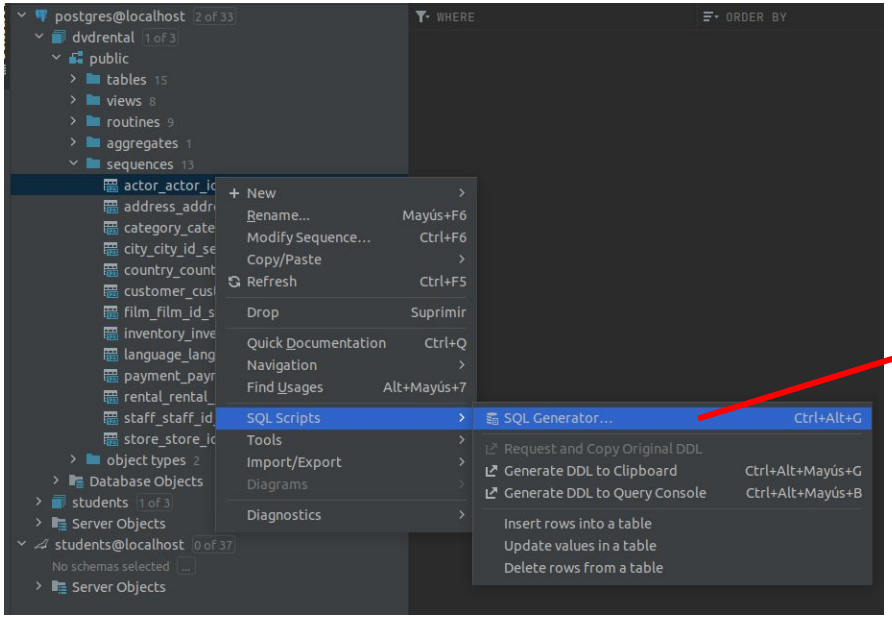
```
CREATE TABLE order_details(
    order_id SERIAL,
    item_id INT NOT NULL,
    item_text text NOT NULL,
    price DEC(10,2) NOT NULL,
    PRIMARY KEY(order_id, item_id)
);
```

```
CREATE SEQUENCE order_item_id
START 10
INCREMENT 10
MINVALUE 10
OWNED BY order_details.item_id;

INSERT INTO order_details
(order_id, item_id, item_text, price)
VALUES
(100, nextval('order_item_id'),'DVD Player',100),
(100, nextval('order_item_id'),'Android TV',550),
(100, nextval('order_item_id'),'Speaker',250);
```

```
tests=# select * from order_details;
 order_id | item_id | item_text  | price
----------+---------+------------+--------
      100 |      10 | DVD Player | 100.00
      100 |      20 | Android TV | 550.00
      100 |      30 | Speaker    | 250.00
(3 rows)
```

9.1.4. Sequences.




```
create sequence public.order_item_id
    minvalue 10
    increment by 10;

alter sequence public.order_item_id owner to postgres;

alter sequence public.order_item_id owned by public.order_details.item_id;
```

9.1.4. Sequences.

Problems solved:

1. **Auto-incrementing Primary Keys**: Sequences are often used to automatically generate values for primary key columns.
2. **Generating Unique Identifiers**: Sequences provide a reliable and efficient way to generate unique identifiers for database records.
3. **Concurrent Access and Concurrency Control**: Sequences are designed to handle concurrent access in a multi-user environment.
4. **Order and Sorting**: Sequences ensure that the generated values follow a specific order.
5. **Replication and Data Synchronization**: When replicating or synchronizing databases, sequences play a crucial role in ensuring consistent data across different instances. By using the same sequence generator, each database instance generates values in the same order, maintaining data.

9.1.4. Sequences.

P09_sequences_solved

Publicat el dia 13:36

9.2. Transactions and concurrency.

Problems solved: To maitain integrity of the database.

9.2.1. Transactions in PostgreSQL.

A transaction is a series of database operations that are executed as a single unit of work.

Transactions are used to ensure data consistency and integrity by allowing multiple database operations to be treated as a single, atomic operation.

This means that either all the operations in the transaction are completed successfully, or none of them are.

```
BEGIN;
INSERT INTO users (id, name) VALUES (1, 'John');
INSERT INTO orders (id, user_id, product) VALUES (1, 1, 'Widget');
COMMIT;
ROLLBACK;
```

Statement starts a new transaction

If either of the two "INSERT" statements were to fail, the entire transaction would be rolled back, and no changes would be made to the database.

Commit the operations

To explicitly revert the changes

9.2.1. Transactions in PostgreSQL.

```
import psycopg2

# Connect to the database
conn = psycopg2.connect(
    host="localhost",
database="bank",
user="bankapp",
password="alualualu"
)

# Create a cursor object
cursor = conn.cursor()

# Begin a transaction
cursor.execute("BEGIN")
```

```
try:
    # Perform the updates
    cursor.execute(
        "UPDATE app.accounts SET balance = balance - 500 WHERE account_number = '12345678'"
    )
    cursor.execute(
        "UPDATE app.accounts SET balance = balance + 500 WHERE account_number = '98765432'"
    )
    # Commit the transaction
    conn.commit()
    print("Transaction completed successfully!")
except Exception as e:
    # Rollback the transaction if an error occurs
    conn.rollback()
    print(f"An error occurred: {str(e)}")
finally:
    # Close the connection
    conn.close()
```

9.2.1. Transactions in PostgreSQL.

```
CREATE user bankapp WITH PASSWORD 'alualualu';
CREATE DATABASE bank; -- WITH OWNER bankapp
ALTER DATABASE bank OWNER TO bankapp;
\c bank
REVOKE ALL PRIVILEGES ON SCHEMA public FROM PUBLIC;
CREATE SCHEMA app;
ALTER SCHEMA app OWNER TO bankapp;
GRANT USAGE, CREATE ON SCHEMA app TO bankapp;
CREATE TABLE app.accounts (
 account_number VARCHAR(9) PRIMARY KEY,
 balance decimal(10,2) NOT NULL
);
ALTER TABLE app.accounts OWNER TO bankapp;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA app TO bankapp;
insert into app.accounts (account_number, balance) values
('123456789', 8020.25),
('987654321', 999.10),
('111111111', 100.25),
('222222222', 12400.25);
```

9.2.2. Concurrency in PostgreSQL.

Concurrency refers to the ability of **multiple users or applications to access and modify the database at the same time**.

To ensure that data is consistent and accurate, PostgreSQL uses a variety of techniques to manage concurrency, including **locks and isolation levels**.

```
-- User 1:
BEGIN;
UPDATE
 products
SET
 quantity = quantity - 1
WHERE
 id = 1;
COMMIT;
```

```
-- User 2:
BEGIN;
UPDATE
 products
SET
 quantity = quantity - 1
WHERE
 id = 1;
COMMIT;
```

9.2.2. Concurrency in PostgreSQL.

To ensure integrity with concurrent access in a scenario where multiple clients are running the same code, you can employ various techniques and strategies. Here are a few approaches you can consider:

1. Database-level locking.
2. Transactions with isolation levels.
3. Optimistic concurrency control.
4. Pessimistic concurrency control.
5. Conflict resolution strategies.
6. Connection pooling.
7. Comprehensive testing.

9.2.2. Concurrency in PostgreSQL.

1. Database-level locking: Use explicit locks provided by the database to control concurrent access to critical resources. For example, you can use row-level locking or table-level locking to prevent conflicts when multiple clients try to access and update the same bank account simultaneously.

```
import psycopg2
import time

# Connect to the database
conn = psycopg2.connect(
    host="localhost", database="bank", user="bankapp",
password="alualualu"
)


# Create a cursor object
cursor = conn.cursor()

# Acquire an exclusive lock on the accounts table
cursor.execute("LOCK TABLE app.accounts IN EXCLUSIVE
MODE")
```

```
# Perform updates within the locked section
try:
    # Deduct $500 from account with account number '123456789'
    cursor.execute(
        "UPDATE app.accounts SET balance = balance - 500 WHERE account_number = '123456789'"
    )
    time.sleep(1)
    # Add $500 to account with account number '987654321'
    cursor.execute(
        "UPDATE app.accounts SET balance = balance + 500 WHERE account_number = '987654321'"
    )

    # Commit the transaction
    conn.commit()

    print("Updates were successfully applied!")
except Exception as e:
    # Rollback the transaction if an error occurs
    conn.rollback()
    print(f"An error occurred: {str(e)}")
finally:
    # Close the connection
    conn.close()
```

9.2.2. Concurrency in PostgreSQL.

We will focus solely on approach 1 as this issue involves a higher level of complexity…

9.3. Dumping and restoring data. Backup and recovery PITR.

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
);

CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

What does it lack?

9.3. Dumping and restoring data. Backup and recovery PITR.

COPY FROM:

- **COPY <table-name> FROM <file-path>**
  - **COPY weather FROM '~/weather.csv' DELIMITER ',' CSV HEADER;**

COPY TO:

- **COPY (<select-query-here>) TO <file-path>;**
  - **COPY (SELECT * FROM weather) TO '/home/alumne/weather.txt'**
  - **COPY (SELECT * FROM weather) TO '~/weather.csv' CSV DELIMITER ',' HEADER;**
  - **COPY weather TO '~/weather2.csv' CSV DELIMITER ',' HEADER;**

- **Very important: COPY** can be used only by DBA user. For normal users must use **\copy**…

- More info here.
- Official documentation here.

Exercise.

Always be user postgres (DBA)

1. Create a database “weatherdb” for user “weather” (you must use the commands ‘createuser’ and ‘createdb’).

```
postgres@ginjol:/mnt/c/Windows/System32$ man createuser
postgres@ginjol:/mnt/c/Windows/System32$ createuser -P weather
Enter password for new role:
Enter it again:
postgres@ginjol:/mnt/c/Windows/System32$ createdb weatherdb -O weather
postgres@ginjol:/mnt/c/Windows/System32$
```

Not a good DB design...

2. Create the table weather as described before importing it from a text file.

```
postgres@ginjol:/mnt/c/Windows/System32$ psql -h localhost -U weather weatherdb
Password for user weather:
psql (12.14 (Ubuntu 12.14-0ubuntu0.20.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

weatherdb=> CREATE TABLE weather (
weatherdb(>     city            varchar(80),
  temp_weatherdb(>     temp_lo         int,           -- low temperature
weatherdb(>     temp_hi         int,           -- high temperature
weatherdb(>     prcp            real,          -- precipitation
weatherdb(>     date            date
weatherdb(> );
CREATE TABLE cities (
    name            varchar(80),
    location        point
);CREATE TABLE
weatherdb=>
weatherdb=> CREATE TABLE cities (
weatherdb(>     name            varchar(80),
weatherdb(>     location        point
weatherdb(> );
CREATE TABLE
weatherdb=>
```

3. Import [weather.csv] into the table using the command copy (copy is only for user postgres, other users must use \copy).

```
weatherdb=# COPY weather FROM '/home/sergi/Descargas/weather.csv' DELIMITER ',' CSV HEADER;
COPY 5
```

4. Copy the table into a csv file.

```
weatherdb=> \copy weather FROM '/mnt/c/Users/sergi/Downloads/weather.csv' DELIMITER ',' CSV HEADER;
COPY 5
```

5. Insert data into cities:

INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)'), ('Hayward','(37.7, -122.1)');

```
weatherdb=> INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)'),
weatherdb-> ('Hayward','(37.7, -122.1)');
INSERT 0 2
```

6. Add the PKs and the FK:

ALTER TABLE weather ADD PRIMARY KEY (city, date);
ALTER TABLE cities ADD PRIMARY KEY (name);
ALTER TABLE weather ADD FOREIGN KEY (city) references cities (name);

```
weatherdb=# ALTER TABLE weather ADD PRIMARY KEY (city, date);
ALTER TABLE
```

```
weatherdb=> ALTER TABLE weather ADD PRIMARY KEY (city, date);
ABLE cities ADD PRIMARY KEY (name);
ALTER TABLE weather ADD FOREIGN KEY (city) references cities (name);
ALTER TABLE
weatherdb=> ALTER TABLE cities ADD PRIMARY KEY (name);
ALTER TABLE
weatherdb=> ALTER TABLE weather ADD FOREIGN KEY (city) references cities (name);
ALTER TABLE
```

7. Do you know how to copy from a table to a file?

9.3. Dumping and restoring data. Backup and recovery PITR.

You can export the database schema with pg_dump (in bash shell):

- **pg_dump -s databasename** ← *Only schema*
- **pg_dump -a databasename** ← *Only data*
- **pg_dump --inserts databasename**
- **pg_dump --inserts databasename > file.txt**

Schema and inserts

Examples:

- **pg_dump -s weatherdb**
- **pg_dump -a weatherdb**
- **pg_dump --inserts weatherdb**
- **pg_dump --inserts weatherdb > file.txt**

9.3. Dumping and restoring data. Backup and recovery PITR.

pg_dumpall: To backup all databases and server globals (only for user postgres).

Examples:

- **pg_dumpall > backup.sql**
- **pg_dumpall --schema-only > schema_dump.sql**

It will generate a file named "schema_dump.sql" containing only the schema definitions.

It will generate a SQL script containing the definitions and data of all databases, including roles and other global objects. The output is redirected to a file named "backup.sql" in the current directory.

9.3. Dumping and restoring data. Backup and recovery PITR.

The whole process of dumping and restoring:

- **pg_dump -U your_username -d your_database_name -f your_dump_file.sql**
- **dropdb samplecompany**
- **psql -U your_username < your_dump_file.sql**

Example:

- **pg_dump -U postgres -d samplecompany -C -f samplecompany.sql**
- **dropdb samplecompany**
- **psql -U postgres < samplecompany.sql**

9.3. Dumping and restoring data. Backup and recovery PITR.

You can import the database schema in a binary dump filewith pg_restore (in bash shell):

- **pg_dump -F c mydb > db.dump**
- **dropdb mydb**
- **createdb -U postgres mydb**
- **pg_restore -U postgres -d postgres db.dump**

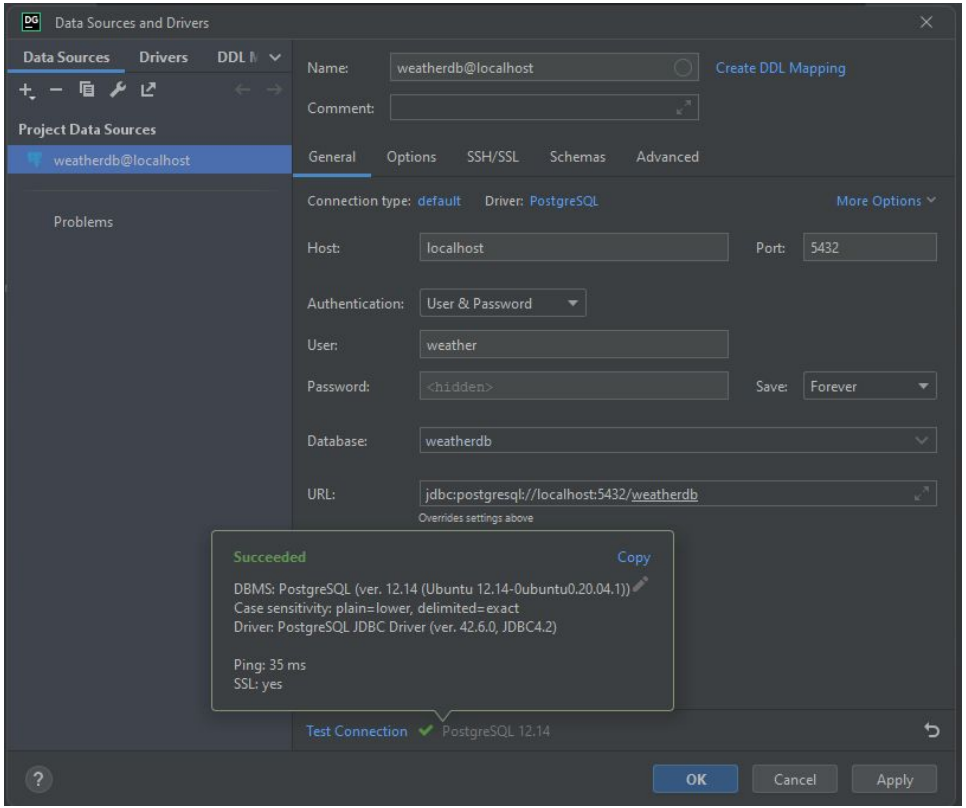
Format custom

dbname

Example:

- **pg_dump -F c weatherdb > weatherdb.dump**
- **dropdb weatherdb**
- **createdb -U postgres weatherdb**
- **pg_restore -U postgres -d weatherdb weatherdb.dump**

9.3. Dumping and restoring data. Backup and recovery PITR.



21:29 - Data extractors
22:54 - Export to another database
24:02 - Quick table backup
...
25:01 - SQL generator
25:39 - Generate DDL files for the schema
26:08 - Dump data for PostgreSQL and MySQL

9.3. Dumping and restoring data. Backup and recovery PITR.

Point-In-Time Recovery (PITR) is a feature in PostgreSQL that allows you to **restore the database to a specific point in time**, rather than just restoring to the point at which the backup was taken.

This is a complex issue:
https://www.postgresql.org/docs/current/continuous-archiving.html
https://wiki.postgresql.org/images/c/ce/PGCONF-PITR_Mark_Jones_2015-10-28.pdf

9.3. Dumping and restoring data. Backup and recovery PITR.

pg_dump + crontab (only Linux)

```
sergi@ginjol:~$ sudo su - postgres
[sudo] contraseña para sergi:
postgres@ginjol:~$ psql
psql (14.7 (Ubuntu 14.7-0ubuntu0.22.04.1))
Type "help" for help.

postgres=# SHOW hba_file;
              hba_file
-------------------------------------
 /etc/postgresql/14/main/pg_hba.conf
(1 row)
```

vi /etc/postgresql/14/main/pg_hba.conf

```
# Noninteractive access to all databases is required during automatic
# maintenance (custom daily cronjobs, replication, and similar tasks).
#
# Database administrative login by Unix domain socket
local   all             postgres                                peer

# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     trust
# IPv4 local connections:
host    all             all             127.0.0.1/32            trust
# IPv6 local connections:
host    all             all             ::1/128                 trust
# Allow replication connections from localhost, by a user with the
# replication privilege.
local   replication     all                                     peer
host    replication     all             127.0.0.1/32            scram-sha-256
host    replication     all             ::1/128                 scram-sha-256
```

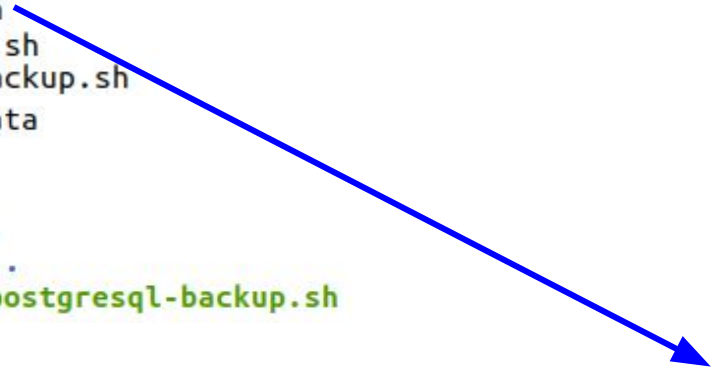
sudo systemctl restart postgresql

https://www.postgresql.org/docs/current/auth-pg-hba-conf.html

9.3. Dumping and restoring data. Backup and recovery PITR.

pg_dump + crontab (only Linux)

```
sergi@ginjol:~$ sudo systemctl restart  postgresql
sergi@ginjol:~$ sudo mkdir /data
sergi@ginjol:~$ sudo vi /data/postgresql-backup.sh
sergi@ginjol:~$ chmod a+x /data/postgresql-backup.sh
sergi@ginjol:~$ sudo chmod a+x /data/postgresql-backup.sh
sergi@ginjol:~$ sudo chown -R postgres:postgres /data
sergi@ginjol:~$ ls -al /data
total 12
drwxr-xr-x  2 postgres postgres 4096 may 18 20:22 .
drwxr-xr-x 21 root     root     4096 may 18 20:16 ..
-rwxr-xr-x  1 postgres postgres  130 may 18 20:22 postgresql-backup.sh
```



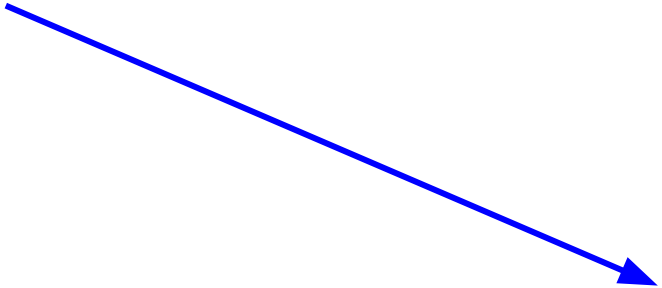
```
#!/bin/bash

BACKUP_DIR="/data/"
FILE_NAME=$BACKUP_DIR`date +%d-%m-%Y-%I-%M-%S-%p`.sql
pg_dump -U postgres weather > $FILE_NAME
~
```

9.3. Dumping and restoring data. Backup and recovery PITR.

pg_dump + crontab (only Linux)

```
sergi@ginjol:/data$ sudo su postgres
postgres@ginjol:/data$ crontab -e
```



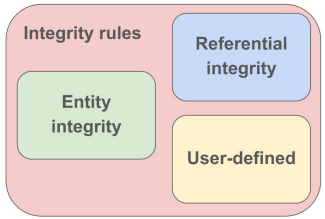
```
# m h  dom mon dow   command
# Add below content in the crontab. It will take the backup for every minute!
# PostgreSQL backup
* * * * * /data/postgresql-backup.sh
-- INSERT --
```

9.4. Database integrity verification tools.

3.7. Relational integrity.

Every relational system must follow the next integrity rules:

- **The entity integrity rule**
- **The referential integrity rule**
- **Additionally: user-defined integrity rules** (that is, specific integrity rules of a specific DB).



3.7. Relational integrity.

THE ENTITY INTEGRITY RULE

No component of the primary key of a base relation is allowed to accept nulls.

Remember that:

- **NULL** may mean "**property does not apply**". For example, the supplier may be a country, in which case the attribute CITY has a null value because such property does not apply.
- **NULL** may mean "**value is unknown**". For example, if the supplier is a person, then a null value for CITY attribute means we do not know the location of this supplier.
- **NULLs cannot be in primary keys**, **but can be in foreign keys**.
- EXAMPLE: DNI_partner can be NULL if you are not married.

3.7. Relational integrity.

THE REFERENTIAL INTEGRITY RULE

The database can not contain inconsistent foreign key values.

In other words, the **valid values of a foreign key** are:

- Existing values in the primary key of reference.
- NULL values.

Another way of saying it:

- The database must not contain any unmatched foreign key values.

Not possible!!

Students

Id	Name	Surname	E-mail
1	Sergi	González	sgonzalezr@iesfbmoll.org
2	Antonio	Banderas	abanderas@noemail.org
3	Pep	Noguera	pnoguera@noemail.org

StudentsEnrolled

IdStudent*	IdCourse*	Date
1	1	12/9/18
1	4	12/9/18
2	2	13/9/18
3	1	16/9/18

Courses

Id	Name
1	Java
2	PHP
3	SQL

9.4. Database integrity verification tools.

3.7. Relational integrity.

WHY ARE FOREIGN KEYS IMPORTANT?

- **Foreign-to-primary-key** matching are **the "glue"** which holds the database together.

Another way of saying it:

- Foreign keys provide the "links" between two relations.

Remember that a relation's foreign key can refer to the same relation:

- PERSON (DNI, name, DNI_partner*)

3.7. Relational integrity.

What should the DBMS do if you try to change the primary key referenced by a foreign key?

- **RESTRICT:** It is not allowed to delete the primary key.
- **CASCADE:** Delete the tuple (row) of the foreign keys referencing that primary key.
- **SET NULL:** Set the value of the foreign key to NULL.
- **SET DEFAULT:** Sets the value of the foreign key to a value.
- **Execute a TRIGGER:** Execute a code to do something.

```
CUSTOMER (NIF, name, address, phone, fax, …)
INVOICE (num_inv, data, …, NIF*)
CREATE TABLE INVOICE (
        NUM_INV    DOMAIN (NUM_DOCUMENT),
        DATE DOMAIN (DATE)
        NIF     DOMAIN (NIF),
  PRIMARY KEY (NUM_INV),
  FOREIGN KEY (NIF) REFERENCES
CUSTOMER
        NULLS NOT ALLOWED
        DELETE OF CUSTOMER RESTRICTED
        UPDATE OF CUSTOMER RESTRICTED);
```

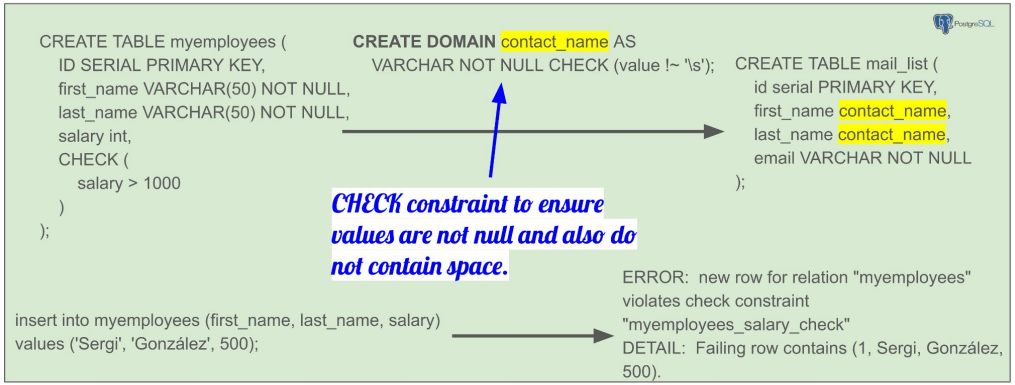
9.4. Database integrity verification tools.

We saw in UNIT 5:

CHECK

NOT NULL

5.6. Data types in SQL.



9.4. Database integrity verification tools.

What is database integrity?

Database integrity is the **state of a database** in which the **data is consistent and accurate**. It is important to maintain database integrity to ensure that the data is reliable and can be trusted.

What are database integrity verification tools?

Database integrity verification tools are used to check the integrity of a database. These **tools** can be used **to identify and fix errors in the data**, as well as to prevent errors from occurring in the first place.

9.4. Database integrity verification tools.

Delete VS Truncate:

- **`TRUNCATE`** `table_a;` **or** **`DELETE`** `FROM table_a;` is the same: **table_a will be emptied**.
- With **DELETE**s, **dead rows remain in database pages** and their dead pointers are still present in indices (you can solve it running: 'VACUUM (FULL, ANALYZE) table_a;').
- TRUNCATE, on the other hand, keeps the table “clean” (the resulting table looks almost identical internally to a newly created table).
- TRUNCATE is all or nothing: it doesn’t have WHERE!
- Delete is DML, but Truncate is DDL. As you now, it has implications: https://stackoverflow.com/questions/139630/whats-the-difference-between-truncate-and-delete-in-sql
- So, maybe it’s a good idea if you change Delete for Truncate in Postgres.
- More information: https://stackoverflow.com/questions/11419536/postgresql-truncation-speed/11423886#11423886

9.4. Database integrity verification tools.

What is database bloat?

Database bloat is disk space that was used by a table or index and is available for reuse by the database but has not been reclaimed. Bloat is created when deleting or updating tables and indexes.

What are some database integrity verification tools available in Postgresql 14?

- **pg_vacuum:** This tool can be used to remove dead rows from a database to free up space and improve performance.

VACUUM is the built-in command for performing vacuuming in PostgreSQL, pg_vacuum is an extension that adds extra features and control over the vacuuming process, such as parallel vacuuming and enhanced monitoring capabilities.

9.4. Database integrity verification tools.

What are some database integrity verification tools available in Postgresql 14?

- **pg_repack:** This is an extension for PostgreSQL that allows you to perform online reorganization and optimization of database tables without requiring exclusive locks or disrupting ongoing database operations. It's designed **to reclaim wasted space, improve query performance, and reduce table bloat**.
- **pg_dump:** This tool can be used **to create a backup** of a database. This can be useful for restoring the database to a previous state if errors are found. PITR is very important.

9.5. Postgresql configuration files.

Three main configuration files (Obe and Hsu, 2012):

postgresql.conf | pg_hba.conf | pg_ident.conf

postgresql.conf: General settings. How much memory to allocate, default storage location for new databases, which IPs PostgreSQL listens on, where logs are stored, etc.

pg_hba.conf: Controls security. It manages access to the server, dictating which users can login into which databases, which IPs or groups of IPs are permitted to connect and the authentication scheme expected.

pg_ident.conf: The mapping file that maps an authenticated OS login to a PostgreSQL user. This file is used less often, but allows you to map a server account to a PostgreSQL account. E.g., people sometimes map the OS root account to the postgre's superuser account. Each authentication line in pg_hba.conf can use a different pg_ident.conf file.

9.5. Postgresql configuration files.

Location of configuration files:

- **SELECT name**, setting
  FROM pg_settings
  WHERE category = 'File Locations';

```
postgres=# SELECT name, setting
 FROM pg_settings
 WHERE category = 'File Locations';
       name        |                  setting
-------------------+-----------------------------------------
 config_file       | /etc/postgresql/14/main/postgresql.conf
 data_directory    | /var/lib/postgresql/14/main
 extension_destdir |
 external_pid_file | /var/run/postgresql/14-main.pid
 hba_file          | /etc/postgresql/14/main/pg_hba.conf
 ident_file        | /etc/postgresql/14/main/pg_ident.conf
(6 rows)
```

9.5. Postgresql configuration files.

The postgresql.conf file:

```
listen_addresses = '*'                  # what IP address(es) to listen on;
                                        # comma-separated list of addresses;
                                        # defaults to 'localhost'; use '*' for all
                                        # (change requires restart)
port = 5432                             # (change requires restart)
max_connections = 100                   # (change requires restart)
#superuser_reserved_connections = 3     # (change requires restart)
```

It is the maximum number of concurrent connections allowed.

Default port for PostgeSQL is 5432.

It tells PostgreSQL which IPs to listen on. This usually defaults to localhost, but many people change it to *, meaning all available IPs.

9.5. Postgresql configuration files.

The postgresql.conf file:

shared_buffers defines the amount of memory you have shared across all connections to store recently accessed pages. This setting has the most effect on query performance. You want this to be fairly high, probably at least 25% of your on- board memory.

effective_cache_size is an estimate of how much memory you expect to be available in the OS and PostgreSQL buffer caches. It has no effect on actual allocation, but is used only by the PostgreSQL query planner to figure out whether plans under consideration would fit in RAM or not.

```
shared_buffers = 128MB                  # min 128kB
                                        # (change requires restart)

#effective_cache_size = 4GB

#work_mem = 4MB                         # min 64kB
#maintenance_work_mem = 64MB            # min 1MB
```

maintenance_work_mem is the total memory allocated for housekeeping activities like vacuuming (getting rid of dead records). This shouldn't be set higher than about 1 GB.

Work_mem controls the maximum amount of memory allocated for each operation such as sorting, hash join, and others.

9.5. Postgresql configuration files.

The postgresql.conf file:

To see the settings configured with this file:

```
SELECT name, context, unit,
            setting, boot_val, reset_val
 FROM pg_settings
 WHERE name in('listen_addresses','max_connections',
              'shared_buffers','effective_cache_size',
              'work_mem', 'maintenance_work_mem')
 ORDER BY context,name;
```

1.- If context is set to postmaster, it means changing this parameter requires a restart of the postgresql service. If context is set to user, changes require at least a reload.
2.- Unit of measurement of the column 'setting'.
3.-, 4.-, 5.- setting is the currently running setting in effect; boot_val is the default setting; reset_val is the new value if you were to restart or reload. You want to make sure that after any change you make to postgresql.conf the setting and reset_val are the same. If they are not, it means you still need to do a reload.

```
postgres=# SELECT name, context, unit,
                setting, boot_val, reset_val
 FROM pg_settings
 WHERE name in('listen_addresses','max_connections',
                'shared_buffers','effective_cache_size',
                'work_mem', 'maintenance_work_mem')
 ORDER BY context,name;
         name          |  context   | unit |  setting  | boot_val  | reset_val
-----------------------+------------+------+-----------+-----------+-----------
 listen_addresses      | postmaster |      | localhost | localhost | localhost
 max_connections       | postmaster |      | 100       | 100       | 100
 shared_buffers        | postmaster | 8kB  | 16384     | 1024      | 16384
 effective_cache_size  | user       | 8kB  | 524288    | 524288    | 524288
 maintenance_work_mem  | user       | kB   | 65536     | 65536     | 65536
 work_mem              | user       | kB   | 4096      | 4096      | 4096
(6 rows)
```

9.5. Postgresql configuration files.

The postgresql.conf file:

- **What does it happen if you edit one of those files and now the server is broken?**
  - The easiest way to figure out what you did wrong is to look at the log file:

```
listen_addresses = 'kkk'                # what IP address(es) to listen on;
```

error generated by me inside postgresql.conf

```
sergi@sergi-VirtualBox:/var/log/postgresql$ sudo systemctl restart  postgresql
sergi@sergi-VirtualBox:/var/log/postgresql$ tail -n 7 /var/log/postgresql/postgresql-12-main.log
2020-03-29 17:29:11.268 CEST [4338] LOG:  starting PostgreSQL 12.2 (Ubuntu 12.2-2.pgdg18.04+1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7
.4.0, 64-bit
2020-03-29 17:29:11.270 CEST [4338] LOG:  could not translate host name "kkk", service "5432" to address: Name or service not known
2020-03-29 17:29:11.270 CEST [4338] WARNING:  could not create listen socket for "kkk"
2020-03-29 17:29:11.270 CEST [4338] FATAL:  could not create any TCP/IP sockets
2020-03-29 17:29:11.270 CEST [4338] LOG:  database system is shut down
pg_ctl: could not start server
Examine the log output.
```

Very important, to understand linux commands visit this website:
https://explainshell.com/explain?cmd=tail+-n+7+%2Fvar%2Flog%2Fpostgresql%2Fpoastgresql-12-main.log

9.5. Postgresql configuration files.

The pg_hba.conf file:

```
# Database administrative login by Unix domain socket
local   all             postgres                                peer

# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     peer
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
host    all             all             192.168.56.0/32         md5
# IPv6 local connections:
host    all             all             ::1/128                 md5
# Allow replication connections from localhost, by a user with the
# replication privilege.
local   replication     all                                     peer
host    replication     all             127.0.0.1/32            md5
host    replication     all             ::1/128                 md5
```

1.- Authentication method (password encryption): ident, trust, md5 (among others).
2.- IPv6 syntax for defining localhost.
3.- IPv4 Syntax for defining a network range. The first part in this case 192.168.56.0 is the network address. The /24 is the bit mask. In this example, we are allowing anyone from network 192.168.56.0 to connect as long as they provide a valid md5 encrypted password.
4.- If you use 'hostssl' (instead of 'host'), users must connect through SSL. More information here.
5.- Defines a range of IPs allowed to replicate with this server.

9.5. Postgresql configuration files.

```
postgres@192:postgres> SELECT pg_reload_conf();
+-----------------+
| pg_reload_conf  |
|-----------------|
| True            |
+-----------------+
SELECT 1
Time: 0.018s
```

You can reload settings from Postgresql shell!!

Reloading configuration files:

- Sometimes it is enough to reload the files without restarting the server:
  - `sudo systemctl reload postgresql`
  - `sudo service postgresql reload`
- Another times you must restart the server (service will be down for a few seconds):
  - `sudo systemctl restart postgresql`
  - `sudo service postgresql restart`

9.5. Postgresql configuration files.

9.6. Extensions.

Check PostgreSQL extensions here.

In PostgreSQL, an extension is a **modular component that provides additional functionality** and features to the database system. Extensions allow you to **extend the capabilities** of PostgreSQL without modifying its core codebase.

An extension typically consists of a **set of SQL scripts, functions, data types, operators, and other database objects packaged together**. These objects are designed to enhance the functionality of PostgreSQL or provide specialized features for specific purposes.

To use an extension, you need to **install it into your PostgreSQL database**. The installation process varies depending on the extension, but it usually involves executing SQL scripts or using dedicated commands provided by the extension.

Once installed, you can enable an extension for a particular database using the **CREATE EXTENSION** command. This command makes the extension's functionality available within the specified database.

9.6. Extensions.

To see which extensions you have already installed, run the query:

```
SELECT *
FROM pg_available_extensions
WHERE comment LIKE '%string%' OR
    installed_version IS NOT NULL
ORDER BY name;
```

```
tests=# SELECT *
FROM pg_available_extensions
WHERE comment LIKE '%string%' OR
        installed_version IS NOT NULL
ORDER BY name;
     name      | default_version | installed_version |                        comment
---------------+-----------------+-------------------+-----------------------------------------------------
 citext        | 1.6             |                   | data type for case-insensitive character strings
 fuzzystrmatch | 1.1             |                   | determine similarities and distance between strings
 plpgsql       | 1.0             | 1.0               | PL/pgSQL procedural language
(3 rows)
```

9.6. Extensions.

Popular extensions

postgis	For OGC GIS data, demographic statistics data, or geocoding
fuzzystrmatch	Lightweight extension with functions like soundex, levenshtein, and metaphone for fuzzy string matching.
hstore	Adds key-value pair storage and index support well-suited for storing pseudo-normalized data
pg_trgm	Another fuzzy string search library.
dblink	To query other PostgreSQL databases. This is currently the only supported mechanism of cross-database interaction for Post- greSQL.
pgrypto	Provides various encryption tools including the popular PGP.

9.6. Extensions.

https://www.postgresql.fastware.com/blog/further-protect-your-data-with-pgcrypto
https://www.postgresql.org/docs/current/pgcrypto.html

PGCrypto:

```
CREATE TABLE users (
    id serial NOT NULL PRIMARY KEY,
    password text NOT NULL
);
CREATE EXTENSION pgcrypto;

SELECT crypt('alualualu', gen_salt('bf', 8));

INSERT INTO users VALUES (default, crypt('alualualu', gen_salt('bf', 8)));
SELECT * FROM users WHERE id = 1 AND password = crypt('alualualu', password);
SELECT * FROM users WHERE id = 1 AND password = crypt('atuatuatu', password);
```

Encryption of the password "alualualu" with 4 rounds of encryption algorithm blowfish.

```
 id |                           password
----+------------------------------------------------------------
  1 | $2a$08$nefV81yd/M.3qb8L1oYhhuvmG5rtRB8K.GEsfnb0ulVf2kkm9L2oG
(1 row)
```

```
 id | password
----+----------
(0 rows)
```

9.6. Extensions.

P09_extensions_solved

Publicat el dia 14:54

9.7. Migration of data between management systems.

Data mapping:	Identifying the structure and format of data in the source system and mapping it to the corresponding structure in the target system.
Data extraction:	Extracting data from the source system. This typically involves querying the source database, exporting data in a suitable format (such as CSV or JSON), or using specialized tools or APIs provided by the source system.
Data transformation:	Transformations to align with the structure and requirements of the target system. This can include data cleaning, formatting, standardization, and validation to ensure data integrity and consistency.
Data loading:	Loading the data to the target system. May involve bulk insertions, API calls, or using specialized migration tools provided by the target system.
Data validation:	Validation checks to ensure the accuracy and completeness of the migrated data. This involves comparing the migrated data with the source data, running tests, etc.
Data Integrity and Security:	During data migration, it is crucial to maintain data integrity and security (preserving data relationships, maintaining referential integrity, handling data constraints, and ensuring security measures to protect sensitive data).
Testing (post-migration validation):	After completing the migration, thorough testing and validation should be conducted to ensure that the new system functions correctly and that the migrated data is accurate and usable.
Data backup and Rollback:	Data migration involves some level of risk, and it is essential to have a backup plan in case of any issues or unforeseen problems.
Document all the process!	Always!

9.7. Migration of data between management systems.

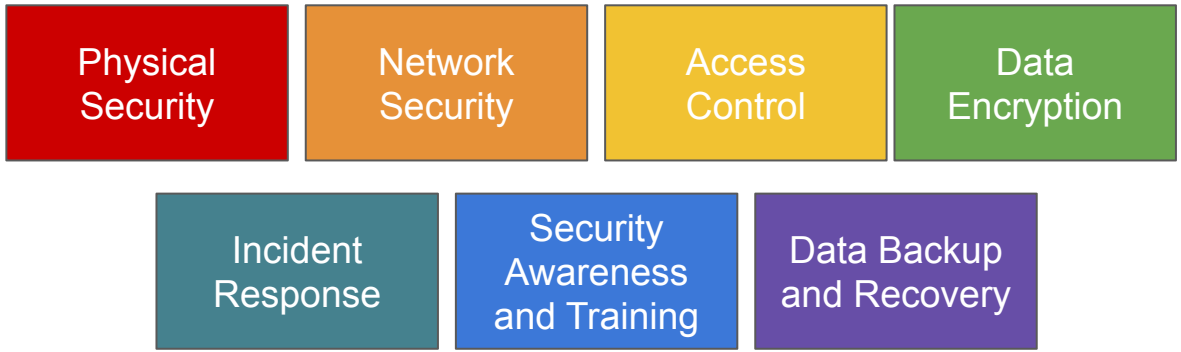
Good exercises (that I have not prepared):

- Migrating from Postgresql to MongoDB.
- Migrating from Postgresql to an Azure database.

9.8. Documentation of security measures and policies.

Security measures and policies are essential for protecting sensitive information and maintaining the integrity of systems.

We will explore the different areas of security measures and policies and their importance in safeguarding data.



9.8. Documentation of security measures and policies.

Physical Security:

- Physical security measures focus on protecting the physical assets of an organization.
- Examples include securing data centers, restricting access to server rooms, implementing surveillance systems, and employing biometric authentication.

Network Security:

- Network security involves protecting computer networks from unauthorized access, attacks, and data breaches.
- Measures include firewalls, intrusion detection systems, secure network protocols, virtual private networks (VPNs), and regular network monitoring.

Access Control:

- Access control measures ensure that only authorized individuals can access sensitive data or systems.
- This includes user authentication mechanisms (e.g., passwords, two-factor authentication), user account management, role-based access control (RBAC), and least privilege principle.

9.8. Documentation of security measures and policies.

Data Encryption:

- Data encryption protects sensitive information by encoding it in a way that can only be decrypted with the appropriate encryption key.
- Encryption methods include symmetric encryption, asymmetric encryption (public-key encryption), and secure transport protocols (e.g., SSL/TLS).

Incident Response:

- Incident response policies and procedures define how an organization responds to and manages security incidents or breaches.
- This includes incident detection, containment, investigation, communication, and recovery processes.

Security Awareness and Training:

- Security awareness and training programs educate employees about security best practices, policies, and procedures.
- Training may cover topics such as password hygiene, phishing awareness, social engineering, and safe internet browsing.

9.8. Documentation of security measures and policies.

Data Backup and Recovery:

- Data backup and recovery strategies ensure that data can be restored in case of accidental loss, system failures, or security incidents.
- This includes regular backups, off-site storage, testing data recovery processes, and having a disaster recovery plan.

ISO 27002:

- https://www.iso.org/obp/ui/#iso:std:iso-iec:27002:ed-3:v2:en
- https://normaiso27001.es/a9-control-de-acceso/

9.8. Documentation of security measures and policies.

Security measures and policies are crucial for safeguarding sensitive data in databases.

Documentation plays a vital role in ensuring effective implementation and enforcement of security measures.

Why Document Security Measures and Policies?

What to Document?

Parts

Maintaining and Updating Documentation

9.8. Documentation of security measures and policies.

Why Document Security Measures and Policies?

1. Clear Communication:
   - Documentation provides a clear and concise means to communicate security measures and policies to all stakeholders.
   - It ensures that everyone understands their roles and responsibilities in maintaining database security.
2. Consistency and Standardization:
   - Documentation helps establish consistent security practices throughout the organization.
   - It ensures that security measures are uniformly implemented and followed across different teams and departments.
3. Compliance and Auditing:
   - Documented security measures serve as evidence of compliance with regulatory requirements and industry standards.
   - Documentation facilitates audits and inspections, demonstrating adherence to security policies and regulations.

9.8. Documentation of security measures and policies.

What to Document?

1. Security Policies:
   - Clearly define the organization's security policies, including access control, data encryption, password policies, etc.
   - Document policies for incident response, data backup and recovery, and disaster recovery.
2. Procedures and Guidelines:
   - Document step-by-step procedures and guidelines for implementing security measures.
   - Include instructions for configuring firewalls, setting up user access controls, and handling security incidents.
3. Roles and Responsibilities:
   - Clearly define the roles and responsibilities of individuals or teams involved in database security.
   - Document access levels, privileges, and responsibilities of administrators, developers, and end-users.

9.8. Documentation of security measures and policies.

Parts:

1. Security Plan:
   - Develop a comprehensive security plan that outlines the specific measures to be implemented.
   - Include details on authentication mechanisms, encryption methods, network security, and physical security.
2. Security Controls:
   - Document the implementation of security controls, such as firewalls, intrusion detection systems, and antivirus software.
   - Provide information on how these controls are configured, monitored, and updated.
3. Incident Response Plan:
   - Document the procedures to be followed in case of a security breach or incident.
   - Include steps for identifying and containing security threats, notifying relevant parties, and initiating recovery processes.

9.8. Documentation of security measures and policies.

Maintaining and Updating Documentation:

1. Regular Reviews:
   - Conduct periodic reviews of the security documentation to ensure its accuracy and relevance.
   - Update the documentation as needed to reflect changes in security practices or regulatory requirements.
2. Training and Awareness:
   - Educate employees about the importance of security documentation and their role in adhering to documented policies.
   - Provide training on using security tools and following documented procedures.

9.9. SQLi.

A SQL injection (SQLi) is a technique that attackers use to gain unauthorized access to a web application database by adding a string of malicious code to a database query.

By exploiting vulnerabilities in poorly designed or insecure web applications, attackers can modify, retrieve, or delete sensitive data from databases.

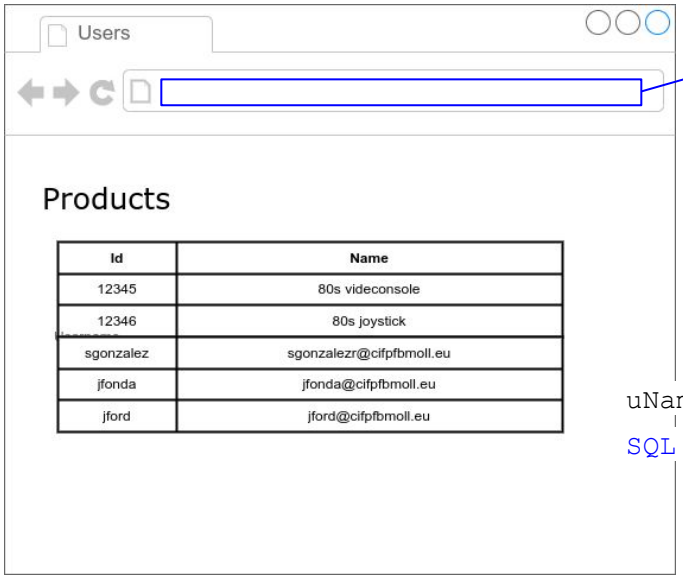

```
uName = getRequestString("username");
uPass = getRequestString("userpassword");
SQL = "SELECT * FROM Users WHERE Name ='" + uName + "' AND Pass ='" + uPass + "'"


uName = "' or ''='";
uPass = "' or ''='";
SQL = "SELECT * FROM Users WHERE Name ='' or ''='' AND Pass ='' or ''=''"
```

The WHERE condition is always TRUE!!

9.9. SQLi.

```
uName = getRequestString("catid");
SQL = "SELECT product_id, product_name FROM products WHERE catid = " + catId
```



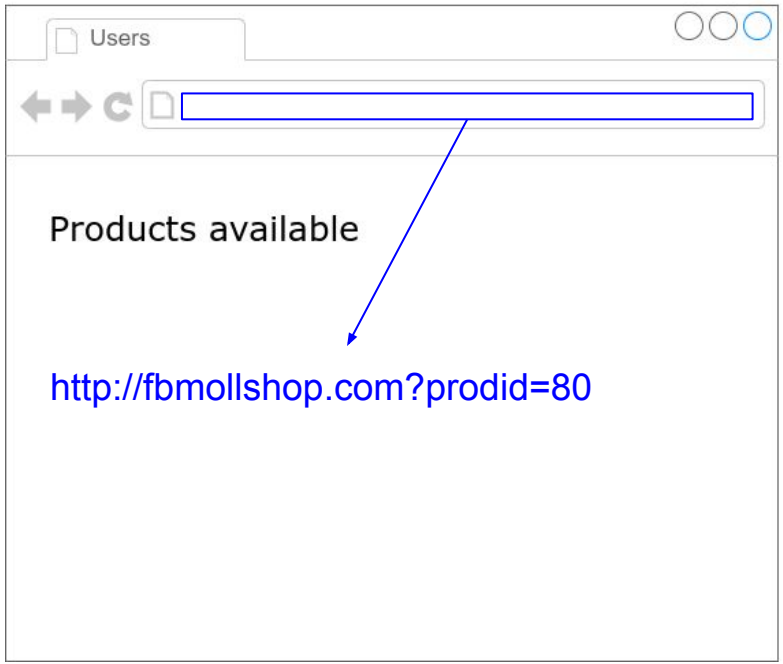
http://fbmollshop.com?catid=80
http://fbmollshop.com?catid=80 UNION select user_id as product_name, email as product_name from users

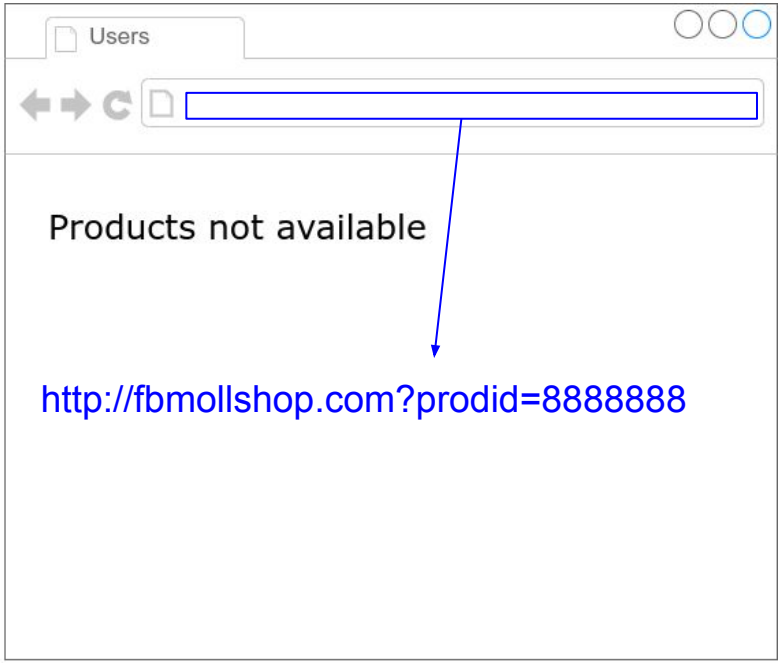
```
uName = "80";
SQL = "SELECT product_id, product_name FROM products WHERE catid = 80"
```

```
uName = "80 UNION select user_id as product_name, email as product_name from users";
SQL = "SELECT product_id, product_name FROM products WHERE catid = 80 UNION select user_id as product_name,
email as product_name from users"
```

9.9. SQLi.

Blind SQLi is based on inferring the result of the sent query through changes that occur in the application, even if they are not directly the response to that query.





9.9. SQLi.

https://owasp.org/www-project-mutillidae-ii/
https://youtu.be/9RH4l8ff-yg
https://github.com/webpwnized/mutillidae-docker

Sources.

- https://www.postgresql.org/docs/current/backup.html
- https://gist.github.com/linuxkathirvel/90771e9d658195fa59e0f0b921f7e22e
- Obe, R.; Hsu, L. PostgreSQL Up and Running. O'Reilly: 1st Edition, 2012.
- https://youtu.be/aUfPf-clLLs
- https://www.PostgreSQLtutorial.com/PostgreSQL-sequences/
- http://www.neilconway.org/docs/sequences/
- https://www.PostgreSQL.org/docs/current/manage-ag-tablespaces.html
- https://www.PostgreSQLtutorial.com/PostgreSQL-create-tablespace/
- https://wiki.postgresql.org/wiki/PostgreSQL_for_Oracle_DBAs