# Subconsultas, Views e o CRUD em Python

## Aula 15 — Montando as 4 operações

Aula 15 · 28/10/2026 · Indicador 5

UniSenac Centro Universitário RS
Senac
Fecomércio
IFEP
Sesc
Sindicatos Empresariais

# O mapa de hoje

- **Bloco 1** — **subconsultas**: uma consulta dentro da outra
- **Bloco 2** — **views**: guardar uma consulta com nome
- **Bloco 3** — o **CRUD** em Python: as 4 funções
- **Bloco 4** — mão na massa: construir o CRUD do projeto

> O **CRUD** é o instrumento principal de avaliação da UC2. Ele nasce hoje e é demonstrado na Aula 18.

# Retomando a Aula 14

- Qual a forma do `ON` num `INNER JOIN`?
- Quando usar `LEFT JOIN` em vez de `INNER`?
- Qual palavra em português indica `GROUP BY`?

> Abram o `crud.ipynb` do grupo. Ele nasceu na Aula 12 e hoje ele cresce de verdade.

# Bloco 1

## Subconsultas — uma consulta dentro da outra

# O problema

*"Quais jogos são do PlayStation 5?"*

Você sabe o **nome** da plataforma, mas a tabela `jogo` guarda o **número**.

- Opção A: consultar `plataforma` para descobrir o id, anotar, e usar na segunda consulta
- Opção B: `INNER JOIN` (Aula 14)
- Opção C: **subconsulta** — deixar o banco fazer as duas de uma vez

# A subconsulta

```
SELECT data_retirada
FROM   emprestimo
WHERE  id_cliente = (SELECT id_cliente
                     FROM   cliente
                     WHERE  nome = 'Ana Souza');
```

# A de dentro roda primeiro

## A DE DENTRO RODA PRIMEIRO




**GLOSSÁRIO**

| | |
|---|---|
| `subquery` | subconsulta — consulta dentro de outra |

# `IN` — quando a subconsulta devolve vários

```
SELECT titulo
FROM   jogo
WHERE  id_jogo IN (SELECT id_jogo
                   FROM   jogos_plataforma
                   WHERE  id_plataforma = 1);
```

Com `=`, a subconsulta precisa devolver **um** valor só. Se devolver dois, dá erro. Quando pode devolver vários, use `IN`.

# Subconsulta com agregação

```sql
SELECT titulo, ano_lancamento
FROM   jogo
WHERE  ano_lancamento > (SELECT AVG(ano_lancamento) FROM jogo);
```

*"Quais jogos são mais novos que a média do catálogo?"*

Essa pergunta **não** tem como ser respondida com `JOIN`. É o caso em que a subconsulta é insubstituível.

# Checagem do Bloco 1

- Qual parte roda primeiro: a de dentro ou a de fora?
- Quando usar `IN` em vez de `=`?
- Escreva: *"clientes que se cadastraram depois do cliente de id 5"*



forms.gle/MZnaevBNd39g7bD1A

# Intervalo

**10 minutos**

# Bloco 2

## Views — guardar uma consulta com nome

# O problema

Aquela consulta linda da Aula 14 — LEFT JOIN, COUNT, GROUP BY, ORDER BY — tem **seis linhas**.

- O dono quer esse relatório **toda semana**
- Ninguém vai digitar seis linhas toda semana
- E se cada pessoa digitar de um jeito, cada uma vê um número diferente

# CREATE VIEW

```sql
CREATE VIEW jogos_por_plataforma AS
SELECT p.nome                AS plataforma,
       COUNT(jp.id_jogo)     AS total_jogos
FROM   plataforma p
LEFT JOIN jogos_plataforma jp ON p.id_plataforma = jp.id_plataforma
GROUP BY p.nome;
```

Depois, basta:

```sql
SELECT * FROM jogos_por_plataforma;
```

# O que a view guarda — e o que ela não guarda

## O QUE UMA VIEW GUARDA

**CREATE VIEW**
a consulta escrita
uma vez só, com nome

→

**SELECT * FROM**
jogos_por_plataforma

→

**Resultado**
sempre atualizado

# Para que serve uma view

- **Simplificar** — a consulta difícil fica escrita uma vez só
- **Padronizar** — todo mundo usa a mesma definição de "jogo disponível"
- **Proteger** — dar acesso à view sem dar acesso à tabela inteira

> O terceiro item é o **nível de visões** da Aula 02 virando comando. E é a ponte para a Aula 16: o estagiário recebe permissão na *view*, não na tabela `cliente`.

# DROP VIEW e o cuidado com dependências

```
DROP VIEW jogos_por_plataforma;
```

Apagar a **view** não apaga nenhum dado — ela é só uma consulta guardada. Mas apagar uma **tabela** que a view usa quebra a view.

# Checagem do Bloco 2

A dona quer ver, toda segunda, quantos jogos há por plataforma.

- Escreva o comando que guarda essa consulta como `v_jogos_por_plataforma`
- Você cadastra um jogo na terça. Na segunda seguinte o número muda sozinho?
- O estagiário pode consultar essa view **sem** poder ler a tabela `jogo`?



forms.gle/Agc2Q8HuzeuQHqEC8

# Intervalo

**10 minutos**

# Bloco 3

## O CRUD em Python

# O que é CRUD

AS QUATRO FUNÇÕES E O SQL DE CADA UMA

**criar_jogo()**
INSERT + commit

**listar_jogos()**
SELECT + fetchall

**atualizar_jogo()**
UPDATE + commit

**apagar_jogo()**
DELETE + commit

UM BANCO SÓ

**Banco locadora**
uma conexão só,
quatro comandos

# As quatro letras, e o comando de cada uma

**C · Create**
criar
`INSERT`

**R · Read**
ler
`SELECT`

**U · Update**
atualizar
`UPDATE`

**D · Delete**
apagar
`DELETE`

# C — criar

```python
def criar_jogo(titulo, genero, preco_diaria):
    cursor.execute(
        "INSERT INTO jogo (titulo, genero, preco_diaria) VALUES (%s, %s, %s);",
        (titulo, genero, preco_diaria)
    )
    conexao.commit()
    print(f"Jogo '{titulo}' cadastrado.")
```

Os `%s` são **marcadores**, não aspas. O `psycopg2` coloca as aspas certas sozinho — e isso protege contra dado malicioso.

# R — ler

```python
def listar_jogos():
    cursor.execute(
        "SELECT id_jogo, titulo, genero FROM jogo ORDER BY titulo;"
    )
    for id_jogo, titulo, genero in cursor.fetchall():
        print(f"{id_jogo:3} | {titulo:30} | {genero}")
```

**Sem** `commit()`. Ler não muda nada.

# U — atualizar

```python
def atualizar_jogo(id_jogo, novo_genero):
    cursor.execute(
        "UPDATE jogo SET genero = %s WHERE id_jogo = %s;",
        (novo_genero, id_jogo)
    )
    conexao.commit()
    print(f"{cursor.rowcount} linha(s) atualizada(s).")
```

`cursor.rowcount` diz **quantas linhas** foram afetadas. É o hábito de segurança da Aula 12, agora automático: se aparecer um número maior do que você esperava, alguma coisa está errada.

# D — apagar

```python
def apagar_jogo(id_jogo):
    cursor.execute("DELETE FROM jogo WHERE id_jogo = %s;", (id_jogo,))
    conexao.commit()
    print(f"{cursor.rowcount} linha(s) apagada(s).")
```

Repare na vírgula em `(id_jogo,)` — sem ela, o Python não entende que é uma tupla de um elemento só. É um erro chato e silencioso.

# ⚠ Erro comum — esquecer o `WHERE` dentro da função

> Uma função de CRUD sem `WHERE` no `UPDATE` ou no `DELETE` é uma bomba: **toda chamada** destrói a tabela inteira, e o erro fica escondido dentro de uma função com nome inocente.

Confira, em cada função que altera dados:

- Existe `WHERE`?
- O `WHERE` usa a **chave primária**?
- A função imprime `rowcount` para você conferir?

# Checagem do Bloco 3

Esta função "funciona" e apaga o catálogo inteiro na primeira chamada:

```
def apagar(cursor, id_jogo):
    cursor.execute("DELETE FROM jogo")
```

- Qual é o defeito, e o que ele custa?
- Escreva a versão certa, usando `%s`
- Que linha você acrescenta para conferir que apagou **uma** linha só?

forms.gle/e3Zh8oDBs5yMhzZFA

# Intervalo

**10 minutos**

MÃO NA MASSA

# Construir o CRUD do projeto

1. No `crud.ipynb`, escrevam as **4 funções** para a entidade principal do projeto
2. Testem **cada uma** numa célula, conferindo o resultado no pgAdmin
3. Confiram as **3 perguntas de segurança** em cada função que altera dados
4. Criem **uma view** com a melhor consulta que vocês escreveram na Aula 14
5. Escrevam uma função `relatorio()` que consulta essa view
6. Se der tempo, repitam as 4 funções para uma **segunda** entidade

MÃO NA MASSA

# Lembrete de entrega

GLOSSÁRIO

`grupoNN-aulaNN-artefato.<ext>`

`grupo03-aula15-crud.ipynb`

`grupo03-aula15-views.sql`

O notebook precisa rodar **do começo ao fim** sem erro, na ordem das células. É assim que ele vai ser demonstrado na Aula 18.

MÃO NA MASSA

# Enquanto vocês constroem

O que eu vou conferir:

- As 4 funções existem e foram **testadas**?
- `commit()` nas três que alteram, e **não** na de ler?
- Todo `UPDATE` / `DELETE` tem `WHERE` com **chave primária**?
- Usaram `%s` em vez de montar o SQL com `+`?
- O notebook roda **do começo ao fim** com "Restart and Run All"?
- A view existe e a função `relatorio()` consulta ela?

Terminou? Escrevam uma função `buscar_jogo(parte_do_titulo)` usando `ILIKE`. É a busca que todo sistema tem.

MÃO NA MASSA

# Devolutiva — passo de mesa em mesa

Eu passo em cada grupo. Demonstrem **uma** função ao vivo, para mim:

- Chamem a função numa célula
- Mostrem o efeito no pgAdmin
- Apontem onde está o `WHERE` e o `commit()`

> É o ensaio da Aula 18 — a demonstração final vai ser exatamente assim, só que com a função completa.

PARA CASA · OPCIONAL

# Antes da próxima aula

**Testar em casa:** rode o notebook com "Restart Kernel and Run All" e confirme que ele vai do começo ao fim sem erro.

**Tarefa opcional:** implemente as 4 funções para uma segunda entidade do projeto.

**Para treinar:** `locadora-exercicios-15-subconsultas-views.sql` — nove situações, e o `NOT IN` com nulo. Como fica a entrega do grupo: `locadora-exemplo-15-views.sql`.

**Para ver acontecer:** demonstrações da Aula 15 — resolver a subconsulta de dentro para fora, e mexer na tabela para ver o que a view faz — e oito perguntas com resposta na hora. No site da UC, na página da aula.

> A partir da Aula 16 o foco muda para **gestão** — usuários, permissões e backup. O CRUD continua evoluindo em paralelo, até a entrega na Aula 18.

# Fim do bloco de Consultas

O sistema funciona. Mas **quem** pode usar? E o que acontece se o banco for perdido amanhã?



UniSenac Centro Universitário RS
Senac Fecomércio IFEP Sesc Sindicatos Empresariais