

## CP-4 Pokročilé databázové technologie

/\* View\*/

/\* Using the View allows me to simplify complex queries. For example, to see a developer's id, name, and level, I had to join three tables. Now querying all developers-juniors will look much easier.\*/

```
CREATE VIEW developer_employee_name AS
SELECT employee.id, name.first, name.last, developer.level
FROM developer
JOIN name USING (employee)
JOIN employee ON (employee.id = name.employee);
```

```
SELECT *
FROM developer_employee_name
WHERE level = 'junior';
```

Query Editor   Query History   Scratch Pad

```
1  CREATE VIEW developer_employee_name AS
2  SELECT employee.id, name.first, name.last, developer.level
3  FROM developer
4  JOIN name USING (employee)
5  JOIN employee ON (employee.id = name.employee);
6
7  SELECT *
8  FROM developer_employee_name
9  WHERE level = 'junior';
```

Data Output   Notifications   Explain   Messages

	id integer	first character varying (50)	last character varying (50)	level character varying (50)
1	15930	Cecil	Spehr	junior
2	70008	Correy	Flacknell	junior
3	72932	Pauly	Gegr	junior
4	92768	Gustave	Dewar	junior
5	65739	Priscilla	Gallie	junior
6	31202	Zorah	Pencott	junior
7	75191	Garv	Ferriere	junior

/\* This will help increase data security by giving access only to part of the table.\*/

```
CREATE VIEW junior_developer AS
SELECT employee.id, name.first, name.last, developer.level
FROM developer
JOIN name USING (employee)
JOIN employee ON (employee.id = name.employee)
WHERE(developer.level = 'junior');
```

```
CREATE VIEW middle_developer AS
SELECT employee.id, name.first, name.last, developer.level
FROM developer
JOIN name USING (employee)
JOIN employee ON (employee.id = name.employee)
WHERE(developer.level = 'middle');
```

```
CREATE VIEW senior_developer AS
SELECT employee.id, name.first, name.last, developer.level
FROM developer
JOIN name USING (employee)
JOIN employee ON (employee.id = name.employee)
WHERE(developer.level = 'senior');
```

---


```
1  CREATE VIEW junior_developers AS
2  SELECT employee.id, name.first, name.last, developer.level
3  FROM developer
4  JOIN name USING (employee)
5  JOIN employee ON (employee.id = name.employee)
6  WHERE (developer.level = 'junior');
7
8  CREATE VIEW middle_developers AS
9  SELECT employee.id, name.first, name.last, developer.level
10 FROM developer
11 JOIN name USING (employee)
12 JOIN employee ON (employee.id = name.employee)
13 WHERE (developer.level = 'middle');
14
15 CREATE VIEW senior_developers AS
16 SELECT employee.id, name.first, name.last, developer.level
17 FROM developer
18 JOIN name USING (employee)
19 JOIN employee ON (employee.id = name.employee)
20 WHERE (developer.level = 'senior');
```

/\* Index\*/

/\* The index is used to optimize the search. I used index of attribute "genre" of table "games" because attributes "name" and "creation\_date" are primary key and they use index scan by default. The "games" table has the most records and you can see the speed change better on it. The EXPLAIN ANALYZE clause shows that the bitmap index scan is faster than the sequential scan.\*/

```
1 EXPLAIN ANALYZE
2 SELECT *
3 FROM game
4 WHERE (genre = 'Drama');
5
```


Data Output Notifications Explain Messages

QUERY PLAN		
	text	
1	Seq Scan on game (cost=0.00..20.50 rows=180 width=26) (actual time=0.063..0.527 rows=180 loops=1)	
2	[...] Filter: ((genre)::text = 'Drama'::text)	
3	[...] Rows Removed by Filter: 820	
4	Total runtime: 0.670 ms	

Query Editor Query History Scratch Pad

```
1 CREATE INDEX idx_game_genre ON game (genre);
2
3 EXPLAIN ANALYZE
4 SELECT *
5 FROM game
6 WHERE (genre = 'Drama');
```

Data Output Notifications Explain Messages

QUERY PLAN		
	text	
1	Bitmap Heap Scan on game (cost=9.67..19.92 rows=180 width=26) (actual time=0.063..0.130 rows=180 loops=1)	
2	[...] Recheck Cond: ((genre)::text = 'Drama'::text)	
3	[...] -> Bitmap Index Scan on idx_game_genre (cost=0.00..9.62 rows=180 width=0) (actual time=0.047..0.047 rows=180 loops=1)	
4	[...] Index Cond: ((genre)::text = 'Drama'::text)	
5	Total runtime: 0.163 ms	

/\* Transaction\*/

/\* The transaction uses the "Serializable" isolation level to lock the "work" table when a new worker is added, so that two users do not add different workers to the same workplace. This is because only the last commit will be kept, and the first worker will be left without a workplace.\*/

```
CREATE OR REPLACE FUNCTION add_employee_on_workplace(workplace_is_free BOOLEAN, em_id INT,
work_num INT, studio VARCHAR)
RETURNS void
AS $$
BEGIN
IF (workplace_is_free) THEN
    INSERT INTO work VALUES (em_id, work_num, studio);
ELSE
    RAISE EXCEPTION 'worplace is taken';
END IF;
END;
$$
language plpgsql;
```

```
CREATE OR REPLACE FUNCTION work_place_is_free(work_num INT, studio VARCHAR)
RETURNS BOOLEAN
AS $$
DECLARE
worker INT;
BEGIN
worker := (SELECT employee FROM work WHERE (workplace_num = work_num AND workplace = studio));
IF (worker IS NULL) THEN
    RETURN true;
ELSE
    RETURN false;
END IF;
END;
$$
language plpgsql;
```

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM employee;
SELECT * FROM workplace;
SELECT add_employee_on_workplace(work_place_is_free(734, 'Auer, Nolan and Schoen'), 66076, 734,
'Auer, Nolan and Schoen');
SELECT * FROM work;
COMMIT TRANSACTION;
```

```
1 CREATE OR REPLACE FUNCTION add_employee_on_workplace(workplace_is_free BOOLEAN, em_id INT, work_num INT, studio VARCHAR)
2 RETURNS void
3 AS $$
4 BEGIN
5 IF (workplace_is_free) THEN
6     INSERT INTO work VALUES (em_id, work_num, studio);
7 ELSE
8     RAISE EXCEPTION 'worplace is taken';
9 END IF;
10 END;
11 $$
```

```
14 CREATE OR REPLACE FUNCTION work_place_is_free(work_num INT, studio VARCHAR)
15 RETURNS BOOLEAN
16 AS $$
17 DECLARE
18 worker INT;
19 BEGIN
20 worker := (SELECT employee FROM work WHERE (workplace_num = work_num AND workplace = studio));
21 IF (worker IS NULL) THEN
22     RETURN true;
23 ELSE
24     RETURN false;
25 END IF;
26 END;
27 $$
28 language plpgsql;

31 BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
32 SELECT * FROM employee;
33 SELECT * FROM workplace;
34 SELECT add_employee_on_workplace(work_place_is_free(734, 'Auer, Nolan and Schoen'), 66076, 734, 'Auer, Nolan and Schoen');
35 SELECT * FROM work;
36 COMMIT| TRANSACTION;
```

---

/\*Triggers\*/

/\*Creating a trigger and a function to check for email addresses that returns a trigger allows us to use this function when updating the "emails" table or inserting new values.\*/

```
CREATE FUNCTION validate_employee_email()
RETURNS TRIGGER
AS $$
BEGIN
    IF ((NEW.email IS NULL) OR (NEW.email NOT LIKE '%@%._%')) THEN
        RAISE EXCEPTION 'Invalid email format';
    END IF;
    RETURN NEW;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER validate_email
BEFORE UPDATE OR INSERT ON email_employee
FOR EACH ROW
EXECUTE PROCEDURE validate_employee_email();
```

```
INSERT INTO email_employee
VALUES (13441, 'vvv');
```

```
1 CREATE FUNCTION validate_employee_email()
2 RETURNS TRIGGER
3 AS $$
4 BEGIN
5     IF ((NEW.email IS NULL) OR (NEW.email NOT LIKE '%@%._%')) THEN
6         RAISE EXCEPTION 'Invalid email format';
7     END IF;
8     RETURN NEW;
9 END;
10 $$
11 LANGUAGE plpgsql;
12
13 CREATE TRIGGER validate_email
14 BEFORE UPDATE OR INSERT ON email_employee
15 FOR EACH ROW
16 EXECUTE PROCEDURE validate_employee_email();
17
18
19 INSERT INTO email_employee|
20 VALUES (13441, 'vvv');
```

Data Output   Notifications   Explain   Messages

ERROR: Invalid email format  
SQL state: P0001

/\*Function\*/

/\*This feature allows you to raise the level of the developer\*/

```
CREATE OR REPLACE FUNCTION enhance_developers(idj INT)
RETURNS void
AS $$
DECLARE
lev CHARACTER VARYING;
BEGIN
lev := (SELECT level FROM developer WHERE (employee = idj));
IF (lev = 'junior') THEN lev:= 'middle';
ELSIF (lev = 'middle') THEN lev:= 'senior'; END IF;
UPDATE developer SET level = lev WHERE (employee = idj);
END;
$$
language plpgsql;
```



The screenshot shows a SQL Query Editor window with three tabs: "Query Editor", "Query History", and "Scratch Pad". The "Query Editor" tab is active and displays a SQL script. The script consists of 18 lines of code. Lines 1-3 are a simple SELECT statement. Lines 4-16 are the definition of the PL/SQL function "enhance\_developers". Line 17 is a blank line. Line 18 is the execution of the function with the argument 97923. The code is syntax-highlighted, with keywords in purple, identifiers in black, and string literals in blue. The line numbers are displayed in a light blue margin on the left.

```
1  SELECT *
2  FROM developer;
3
4  CREATE OR REPLACE FUNCTION enhance_developers(idj INT)
5  RETURNS void
6  AS $$
7  DECLARE
8  lev CHARACTER VARYING;
9  BEGIN
10 lev := (SELECT level FROM developer WHERE (employee = idj));
11 IF (lev = 'junior') THEN lev:= 'middle';
12 ELSIF (lev = 'middle') THEN lev:= 'senior'; END IF;
13 UPDATE developer SET level = lev WHERE (employee = idj);
14 END;
15 $$
16 language plpgsql;
17
18 SELECT enhance_developers(97923);|
```