DBS_CP4

Last edited by **Filip Kopecký** just now

Vytvoření view

```
CREATE VIEW currently_rented_cars AS
SELECT
 c.plate_number,
c.brand,
c.type,
o.operation_date,
cu.name AS customer_name,
r.price_per_day
FROM car c
 JOIN operation o ON c.id = o.car_id
 JOIN rental r ON o.id = r.operation_id
 JOIN customer cu ON o.customer_id = cu.id
WHERE c.is_borrowed = TRUE;
```

použití view

```
SELECT * FROM currently_rented_cars;
```

View je tedy virtuální tabulka, která je výsledkem SELECT dotazu a zobrazuje pouze data ze SELECTu.

Vytvoření triggeru

```
RETURNS TRIGGER AS
$$
DECLARE
 v_car_id INT;
v_customer_id INT;
v_op_id INT;
BEGIN
 -- 1) car
 SELECT id INTO v_car_id
FROM car
WHERE plate_number = NEW.plate_number;
IF NOT FOUND THEN
 INSERT INTO car (plate_number, brand, type, mileage, is_borrowed)
VALUES (NEW.plate_number, NEW.brand, NEW.type, 0, TRUE)
RETURNING id INTO v_car_id;

--save id into v_car_id

 ELSE
 UPDATE car
SET is_borrowed = TRUE
 WHERE id = v_car_id;
END IF;

-- 2) customer
 SELECT id INTO v_customer_id
FROM customer
WHERE name = NEW.customer_name;
IF NOT FOUND THEN
 INSERT INTO customer (name, street, city, postal_code)
VALUES (NEW.customer_name, 'Neznámá', 'Neznámo', '00000')
RETURNING id INTO v_customer_id;
END IF;

-- 3) Operation
 INSERT INTO operation (operation_date, customer_id, car_id)
VALUES (NEW.operation_date, v_customer_id, v_car_id)
RETURNING id INTO v_op_id;

-- 4) Rental
 INSERT INTO rental (operation_id, price_per_day, fine_per_day)
```

```
VALUES (v_op_id, NEW.price_per_day, 500);

RETURN NULL;
END;
$$
 LANGUAGE plpgsql;

CREATE TRIGGER insert_into_borrowed_table
INSTEAD OF INSERT ON currently_rented_cars
FOR EACH ROW
EXECUTE FUNCTION insert_into_borrowed();

INSERT INTO currently_rented_cars
(plate_number, brand, type, operation_date, customer_name, price_per_day)
VALUES
 ('AAA2221', 'Mazda', 'electric', '2025-05-10', 'Matěj Nový', 2000);

SELECT * FROM currently_rented_cars;
```

Vytvoříme nejdříve funkci, která vrací trigger jako proceduru která po insertu nových dat proloopuje všechny joinuté tabulky a přidá tam nově insertnutá data do view. Trigger se aktivuje vždy po insertu do view. Poslední je select dotaz, který zobrazí view po insertu nových dat.

Index

```
--employee table has 32k data, it makes sense to try it here
CREATE INDEX search_by_phone ON employee(phone);

EXPLAIN ANALYZE
SELECT * FROM employee WHERE phone='+420888999000';

DROP INDEX search_by_phone;
```

Index jsme použili v tabulce, která je naplněná největším množstvím dat, aby byla vidět efektivnost indexu. Explain analyze je tam pro to, aby bylo vidět kolik trvá analýza bez indexu a s indexem.

```
Seq Scan on employee  (cost=0.00..677.70 rows=1 width=39) (actual time=0.052..7.541 rows=1 loops=1)
  Filter: ((phone)::text = '+420888999000'::text)
  Rows Removed by Filter: 31975
Planning Time: 0.248 ms
Execution Time: 7.579 ms
```

```
Index Scan using search_by_phone on employee  (cost=0.29..8.30 rows=1 width=39) (actual time=0.071..0.072 rows=1 loops=1)
  Index Cond: ((phone)::text = '+420888999000'::text)
Planning Time: 0.409 ms
Execution Time: 0.099 ms
```

První je s indexem, druhý bez indexu. Je vidět znatelná prodleva mezi hledáním s indexem a bez indexu.

Transakce

```
-- first trans - dirty read simulation - update data
--1. step
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;
UPDATE car SET mileage = mileage - 500 WHERE id = 1;
SELECT * from car where id = 1;

--3.step
COMMIT;

--second trans - read uncommitted data -> mileage is still not changed
--2. step
BEGIN;
SELECT * FROM car WHERE id = 1;
COMMIT;
```

V prvním kroku updatneme data, ale necomittneme, tedy by nastal dirty read, ale díky našemu isolation level read committed povoluje db čtení jen potvrzených dat, tedy nenastane dirty read.