Bazy danych 2024

23 kwietnia 2024

Dostęp do bazy danych z poziomu aplikacji

- ORM,
- Osadzony SQL,
- API,
- Programistyczny SQL (PL/SQL w Oracle, PL/pgSQL w Postgresie itp.) - programowanie funkcji/wyzwalaczy.

Osadzony SQL

```
int main() {
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
      int OrderID;          /* Employee ID (from user)         */
      int CustID;             /* Retrieved customer ID         */
      char SalesPerson[10]    /* Retrieved salesperson name      */
      char Status[6]          /* Retrieved order status        */
   EXEC SQL END DECLARE SECTION;

   /* Set up error processing */
   EXEC SQL WHENEVER SQLERROR GOTO query_error;
   EXEC SQL WHENEVER NOT FOUND GOTO bad_number;

   printf ("Enter order number: ");
   scanf_s("%d", &OrderID);

   EXEC SQL SELECT CustID, SalesPerson, Status
      FROM Orders
      WHERE OrderID = :OrderID
      INTO :CustID, :SalesPerson, :Status;

   printf ("Customer number:  %d\n", CustID);
   printf ("Salesperson: %s\n", SalesPerson);
   printf ("Status: %s\n", Status);

query_error:
   printf ("SQL error: %ld\n", sqlca->sqlcode);
   exit();

bad_number:
   printf ("Invalid order number.\n");
   exit();
}
```

Dostęp do bazy postgresql w pythonie

Psycopg 2 – PostgreSQL database adapter for Python: `https://www.psycopg.org/docs`
Psycopg 3 – `https://www.psycopg.org/psycopg3/`

Dostęp do bazy postgresql w pythonie

Psycopg 2 – PostgreSQL database adapter for Python: `https://www.psycopg.org/docs`
Psycopg 3 – `https://www.psycopg.org/psycopg3/`

```
import psycopg2
conn = psycopg2.connect("dbname=test user=postgres")

# Open a cursor to perform database operations
cur = conn.cursor()

cur.execute("CREATE TABLE test (id serial PRIMARY KEY, num integer, data varchar);")
```

Dostęp do bazy postgresql w pythonie

Psycopg 2 – PostgreSQL database adapter for Python: https://www.psycopg.org/docs
Psycopg 3 – https://www.psycopg.org/psycopg3/

```
import psycopg2
conn = psycopg2.connect("dbname=test user=postgres")

# Open a cursor to perform database operations
cur = conn.cursor()

cur.execute("CREATE TABLE test (id serial PRIMARY KEY, num integer, data varchar);")
# Let Psycopg perform the correct conversion (no more SQL injections!)
cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)", (100, "abc'def"))

cur.execute("SELECT * FROM test;")
print cur.fetchone()    # (1, 100, "abc'def")

conn.commit()

cur.close()
conn.close()
```

Dostęp do bazy postgresql w pythonie

Psycopg 2 – PostgreSQL database adapter for Python: `https://www.psycopg.org/docs`
Psycopg 3 – `https://www.psycopg.org/psycopg3/`

```
import psycopg2
conn = psycopg2.connect("dbname=test user=postgres")

# Open a cursor to perform database operations
cur = conn.cursor()

cur.execute("CREATE TABLE test (id serial PRIMARY KEY, num integer, data varchar);")
# Let Psycopg perform the correct conversion (no more SQL injections!)
cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)", (100, "abc'def"))

cur.execute("SELECT * FROM test;")
print cur.fetchone()    # (1, 100, "abc'def")

conn.commit()

cur.close()
conn.close()
```

Autocommit

```
conn.set_session(readonly=True, autocommit=True)
conn.autocommit=True
```

Dostęp do bazy postgresql w pythonie: iteracja

```
cur = conn.cursor() # client side cursor

cur.execute("SELECT * FROM test;")
for record in cur:
     print record
```

BTW - zapytania w pętli to zwykle nie jest dobry pomysł!

```
...
for user in user_list:
  cur.execute("SELECT * FROM test WHERE user=%s", (user))
  ...
  ...
```

Dostęp do bazy postgresql w pythonie: Server Side Cursors

```
cur = conn.cursor('myNamedCursor')
cur.itersize = 100000 # how many records are fetched at time

cur.execute("SELECT * FROM test;")
for record in cur:
     print record
```

Dostęp do bazy postgresql w pythonie: fetch

```
cur.execute("SELECT * FROM test;")

cur.fetchmany(2)
# [(1, 100, "abc'def"), (2, None, 'dada')]

cur.fetchmany(2)
# [(3, 42, 'bar')]

cur.fetchmany(2)
# []

cur.fetchall()      #Fetch all (remaining) rows of a query result,
                    # returning them as a list of tuples.
```

Dostęp do bazy postgresql w pythonie

```
conn = psycopg2.connect(DSN)          # DSN = data source name

with conn:
    with conn.cursor() as curs:       # transation
        curs.execute(SQL1)            # commited or rolled back

with conn:                            # another transaction
    with conn.cursor() as curs:
        curs.execute(SQL2)

conn.close()                          # closes connection
```

Dostęp do bazy postgresql w pythonie

Warning

Never, never, NEVER use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string. Not even at gunpoint.

SQL injection

https://owasp.org/www-community/attacks/SQL_Injection

SQL injection

https://owasp.org/www-community/attacks/SQL_Injection

```
SELECT id, firstname, lastname FROM author;
```

SQL injection

https://owasp.org/www-community/attacks/SQL_Injection

```
SELECT id, firstname, lastname FROM author;
```

Firstname: **evil'ex** Lastname: **Newman**

SQL injection

https://owasp.org/www-community/attacks/SQL_Injection

```
SELECT id, firstname, lastname FROM author;
```

Firstname: **evil'ex** Lastname: **Newman**

Incorrect syntax near il' as the database tried to execute evil.

SQL injection

```
string query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ ItemName.Text + "'";
```

SQL injection

```
string query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ ItemName.Text + "'";
```

```
SELECT * FROM items
WHERE owner =
AND itemname = ;
```

SQL injection

```
string query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ ItemName.Text + "'";
```

```
SELECT * FROM items
WHERE owner =
AND itemname = ;
```

Itemname: "name' OR 'a'='a"

SQL injection

```
string query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ ItemName.Text + "'";
```

```
SELECT * FROM items
WHERE owner =
AND itemname = ;
```

Itemname: "name' OR 'a'='a"

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name' OR 'a'='a';
```

SQL injection

```
string query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ ItemName.Text + "'";
```

SQL injection

```
string query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ ItemName.Text + "'";
```

itemName: "name'; DELETE FROM items; –"

SQL injection

```
string query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ ItemName.Text + "'";
```

itemName: "name'; DELETE FROM items; –"

```
SELECT * FROM items
WHERE owner = 'hacker'
AND itemname = 'name';

DELETE FROM items;

--'
```

Funkcje i procedury SQL — prosty przykład

SQL pozwala pisać proste funkcje będące ciągiem kilku poleceń SQL. Funkcja może przyjmować argumenty, które mogą być użyte w treści w miejscu stałych.

```
CREATE FUNCTION nazwa(argumenty) [RETURNS typ_wyniku]
AS $X$ tresc_funkcji $X$ LANGUAGE sql;
```

```
CREATE FUNCTION usun_grupe(int) AS $X$
DELETE FROM wybor WHERE kod_grupy=$1;
DELETE FROM grupa WHERE kod_grupy=$1;
$X$ LANGUAGE sql;
```

Wywołanie funkcji powoduje wykonanie zawartych w niej poleceń:

```
SELECT usun_grupe(123);
SELECT usun_grupe(kod_grupy) FROM grupa WHERE kod_uz=234;
```

Funkcje i procedury SQL — zwracana wartość

Funkcja SQL może zwracać jedną wartość, rekord lub zbiór rekordów (relację). Wynikiem funkcji jest odpowiedź na ostatnie zapytanie zawarte w treści funkcji.

```
CREATE FUNCTION f1(int) RETURNS text AS $X$
SELECT nazwisko::text FROM uzytkownik WHERE kod_uz=$1;
$X$ LANGUAGE sql;
```

Funkcje i procedury SQL — zwracana wartość

Funkcja SQL może zwracać jedną wartość, rekord lub zbiór rekordów (relację). Wynikiem funkcji jest odpowiedź na ostatnie zapytanie zawarte w treści funkcji.

```
CREATE FUNCTION f1(int) RETURNS text AS $X$
SELECT nazwisko::text FROM uzytkownik WHERE kod_uz=$1;
$X$ LANGUAGE sql;
```

```
CREATE FUNCTION f2(uzytkownik.kod_uz%TYPE) RETURNS uzytkownik AS
'SELECT * FROM uzytkownik WHERE kod_uz=$1;'
LANGUAGE sql;
```

Funkcje i procedury SQL — zwracana wartość

Funkcja SQL może zwracać jedną wartość, rekord lub zbiór rekordów (relację). Wynikiem funkcji jest odpowiedź na ostatnie zapytanie zawarte w treści funkcji.

```
CREATE FUNCTION f1(int) RETURNS text AS $X$
SELECT nazwisko::text FROM uzytkownik WHERE kod_uz=$1;
$X$ LANGUAGE sql;
```

```
CREATE FUNCTION f2(uzytkownik.kod_uz%TYPE) RETURNS uzytkownik AS
'SELECT * FROM uzytkownik WHERE kod_uz=$1;'
LANGUAGE sql;
```

```
CREATE FUNCTION f3() RETURNS SETOF uzytkownik AS $X$
SELECT * FROM uzytkownik WHERE kod_uz IN (SELECT kod_uz FROM grupa);
$X$ LANGUAGE sql;
```

Oznaczenie `RETURNS SETOF uzytkownik` jest poprawne i oznacza, że funkcja zwraca zbiór krotek typu `uzytkownik%rowtype`.

Funkcje i procedury SQL — zwracana wartość

Funkcja SQL może zwracać jedną wartość, rekord lub zbiór rekordów (relację). Wynikiem funkcji jest odpowiedź na ostatnie zapytanie zawarte w treści funkcji.

```
CREATE FUNCTION f1(int) RETURNS text AS $X$
SELECT nazwisko::text FROM uzytkownik WHERE kod_uz=$1;
$X$ LANGUAGE sql;
```

```
CREATE FUNCTION f2(uzytkownik.kod_uz%TYPE) RETURNS uzytkownik AS
'SELECT * FROM uzytkownik WHERE kod_uz=$1;'
LANGUAGE sql;
```

```
CREATE FUNCTION f3() RETURNS SETOF uzytkownik AS $X$
SELECT * FROM uzytkownik WHERE kod_uz IN (SELECT kod_uz FROM grupa);
$X$ LANGUAGE sql;
```

Oznaczenie `RETURNS SETOF uzytkownik` jest poprawne i oznacza, że funkcja zwraca zbiór krotek typu `uzytkownik%rowtype`. Funkcji zwracających relacje używamy w zapytaniach SELECT tak jak *zwykłych* relacji bazy danych:

```
SELECT nazwisko FROM f3() ORDER BY 1;
```

Funkcje i procedury SQL — usuwanie funkcji

Funkcję usuwamy (kasujemy) poleceniem DROP. W poleceniu trzeba podać nazwę funkcji i listę typów jej parametrów, bo dopiero to w pełni pozwala zidentyfikowac funkcję.

```
DROP FUNCTION f3();
DROP FUNCTION f1(INT);
```

Funkcje i procedury SQL — zwracana wartość nowego typu

Wartość zwracana przez funkcję może być rekordem nowego typu, niezgodnego z żadną z istniejących w bazie relacji.

```
CREATE TYPE grupa_z_liczba AS
(kod_grupy INT, osob INT, max_osob INT, kod_uz INT);
```

Funkcje i procedury SQL — zwracana wartość nowego typu

Wartość zwracana przez funkcję może być rekordem nowego typu, niezgodnego z żadną z istniejących w bazie relacji.

```
CREATE TYPE grupa_z_liczba AS
(kod_grupy INT, osob INT, max_osob INT, kod_uz INT);

CREATE FUNCTION zapelnienie_grup_w_semestrze(semestr.id_semestr%TYPE)
RETURNS SETOF grupa_z_liczba AS $X$
  SELECT grupa.kod_grupy,count(*) AS "osob",max_osoby, grupa.kod_uz
  FROM grupa JOIN wybor USING(kod_grupy) JOIN
       przedmiot_semestr USING(kod_przed_sem)
  WHERE semestr_id=$1
  GROUP BY grupa.kod_grupy, max_osoby;
$X$ LANGUAGE sql;
```

Funkcje i procedury SQL — zwracana wartość nowego typu

Wartość zwracana przez funkcję może być rekordem nowego typu, niezgodnego z żadną z istniejących w bazie relacji.

```
CREATE TYPE grupa_z_liczba AS
(kod_grupy INT, osob INT, max_osob INT, kod_uz INT);

CREATE FUNCTION zapelnienie_grup_w_semestrze(semestr.id_semestr%TYPE)
RETURNS SETOF grupa_z_liczba AS $X$
  SELECT grupa.kod_grupy,count(*) AS "osob",max_osoby, grupa.kod_uz
  FROM grupa JOIN wybor USING(kod_grupy) JOIN
       przedmiot_semestr USING(kod_przed_sem)
  WHERE semestr_id=$1
  GROUP BY grupa.kod_grupy, max_osoby;
$X$ LANGUAGE sql;

SELECT nazwisko,kod_grupy
FROM zapelnienie_grup_w_semestrze(29) NATURAL JOIN uzytkownik
WHERE osob>max_osoby;
```

Funkcje i procedury SQL — typ tworzony automatycznie

```
CREATE FUNCTION fun(semestr.id_semestr%TYPE, OUT kod_grupy int, OUT osob int)
   RETURNS SETOF record AS ...;
-- parametry: IN (default), OUT, INOUT

-- alternatywnie
CREATE FUNCTION fun(semestr.id_semestr%TYPE)
   RETURNS TABLE(kod_grupy int, osob int) AS
```

Jeszcze jedna funkcja SQL

```
CREATE OR REPLACE FUNCTION moi_studenci(int)
RETURNS SETOF student
AS $X$
   SELECT DISTINCT student.*
   FROM student JOIN wybor USING(kod_uz)
        JOIN grupa USING(kod_grupy)
   WHERE grupa.kod_uz=$1;
$X$
LANGUAGE SQL;
```

Jeszcze jedna funkcja SQL

```
CREATE OR REPLACE FUNCTION moi_studenci(int)
RETURNS SETOF student
AS $X$
   SELECT DISTINCT student.*
   FROM student JOIN wybor USING(kod_uz)
        JOIN grupa USING(kod_grupy)
   WHERE grupa.kod_uz=$1;
$X$
LANGUAGE SQL;

SELECT imie,nazwisko FROM moi_studenci(187) ORDER BY 2,1;
SELECT imie,nazwisko FROM pracownik
   WHERE 100 <= (SELECT COUNT(*) FROM moi_studenci(kod_uz));
```

Polimorfizm

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;
```

Polimorfizm

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f
```

Polimorfizm

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f

CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;
```

Polimorfizm

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f

CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}
```

Polimorfizm

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f

CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}

CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;
```

Polimorfizm

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f

CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}

CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |   f3
----+---------
 22 | {22,22}
```

More on ARRAYS: https://www.postgresql.org/docs/current/arrays.html

PL/pgSQL

Every SQL statement must be executed individually by the database server.
With PL/pgSQL you can group a block of computation and a series of queries **inside** the database server.

- Extra round trips between client and server are eliminated
- Intermediate results that the client does not need do not have to be marshaled or transferred between server and client
- Multiple rounds of query parsing can be avoided

Struktura funkcji PL/pgSQL

```
CREATE FUNCTION <nazwa>([<arg>] <typ_argumentu>,...)
  [RETURNS [SETOF] <typ_wyniku>]
  AS <ogranicznik> <tresc_funkcji> <ogranicznik>
  LANGUAGE plpgsql;
```

Treść funkcji ma strukturę blokową — bloki mogą być w sobie zagnieżdżane i każdy może być poprzedzony etykietą i/lub sekcją deklaracji:

```
[<<etykieta>>]
  [DECLARE <zmienna> <typ_zmiennej>;...]
  BEGIN <instrukcja>;... END
```

Deklaracje

W sekcji deklaracji umieszczamy zmienne używane w bloku i w blokach w nim zagnieżdżonych. Jako typ zmiennej można podać typ bazodanowy, typ zdefiniowany (`CREATE TYPE`), typ zgodny z pewnym atrybutem lub krotką pewnej tabeli (`uzytkownik.kod_uz%TYPE`, `uzytkownik%TYPE`, `uzytkownik`). Możemy także nadać wartości początkowe i domyślne zmiennym i zastrzec, czy mogą przyjmować wartosci puste. W sekcji deklaracji można także nadać (zmienić) nazwy atrybutom. Przykłady:

```
DECLARE liczba int NOT NULL DEFAULT 100;
        moja_strona text:='http://www.ii.uni.wroc.pl';
        osoba uzytkownik%rowtype; -- typ zgodny z tablicą bazy danych
        arg1 ALIAS FOR $1;        -- przemianowanie argumentu
        wiersz RECORD;            -- zmienna może wskazywać na dowolny rekord
```

Podstawowe instrukcje

Przypisanie:
```
<zmienna> := <wartosc_lub_wyrazenie>;
```

Instr. war.:
```
IF <wyrazenie> THEN <instrukcje>
[ELSEIF <wyrazenie> THEN <instrukcje>]
[ ELSE <instrukcje>] END IF;
```

Przełącznik:
```
CASE [<wyrazenie>]
[WHEN <wartosci> THEN <instrukcje>]
END CASE;
```

W powyższym <`instrukcje`> to ciąg instrukcji oddzielonych średnikami a `<wartosci>` to ciąg stałych i/lub wyrażeń oddzielonych przecinkami.

Pętle

- `LOOP <instrukcje> END LOOP;`
- `WHILE <wyr_log> LOOP <instrukcje> END LOOP;`
- `FOR <zmienna> IN [REVERSE] <wyr1>..<wyr2> [BY <wyrazenie>]`
  `LOOP <instrukcje> END LOOP;`

Każdą pętlę można poprzedzić etykietą: `<<etykieta>> LOOP ... END LOOP;`
Z pętli można wyjść instrukcjami

- `EXIT [ etykieta ] [ WHEN <wyrazenie> ];`
- `CONTINUE [ etykieta ] [ WHEN <wyrazenie> ];.`

Pętle cd.

Pętla po wynikach zapytania:

```
DECLARE osoba RECORD;
BEGIN
  FOR osoba IN SELECT uzytkownik.*
     FROM uzytkownik NATURAL JOIN grupa
     WHERE rodzaj='w' ORDER BY nazwisko
  LOOP
     UPDATE uzytkownik SET tytul='prof' WHERE kod_uz=osoba.kod_uz;
  END LOOP;
END;
```

Pętla FOREACH dla ARRAY

```
CREATE FUNCTION sum(int[]) RETURNS bigint AS $$
DECLARE
  s bigint := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Pętla FOREACH dla ARRAY

```
CREATE FUNCTION sum(int[]) RETURNS bigint AS $$
DECLARE
  s bigint := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

```
piotrek=# select sum('{1,2,3}'::int[]);

piotrek=# select sum(ARRAY [1,2,3]);
   sum
---------
     6
```

RETURN

Instrukcja `RETURN` służy do generowania wyniku i/lub do wyjścia z funkcji:

- `RETURN NEXT <wyrazenie>` oraz `RETURN QUERY <zapytanie>` dodaj do wyniku funkcji krotkę lub kilka krotek;
- `RETURN <wartosc>` powoduje wyjście i zwrócenie podanej wartości jako wartości funkcji;
- `RETURN` powoduje wyjście z funkcji i zwrócenie wyniku obliczonego przez kolejne instrukcje `RETURN NEXT` i `RETURN QUERY` lub wyniku `NULL`.

SQL

W treści funkcji możemy używać bezporednio zapytań SQL. Ograniczenia:

- instrukcje `COMMIT` i `ROLLBACK` sterują wykonaniem transakcji — instrukcji tych nie można używać wewnątrz funkcji (ale można wewnątrz **procedur**);
- zapytania zwracające jedną krotkę zadajemy w postaci `SELECT...INTO...` np. `SELECT name INTO textvar FROM users WHERE id = 123;`
- zapytania zwracające wiele krotek wykonujemy łącznie z pętlą `FOR` przebiegającą wyniki zapytania, w powiązaniu z kursorem (`DECLARE ...CURSOR FOR...` lub poleceniem `RETURN QUERY`;
- zapytania nie zwracające wyniku: `INSERT/UPDATE/DELETE` (bez `RETURNING`) ale `PERFORM query;` zamiast `SELECT`,

Wyzwalacze w PostgreSQL

Powodują określoną reakcję na modyfikację tabeli (`FOR EACH STATEMENT`) lub jej krotek (`FOR EACH ROW`). Wyzwalacz może być wywołany przed (`BEFORE`) lub po (`AFTER`) takiej modyfikacji.

Wyzwalacze w PostgreSQL

Powodują określoną reakcję na modyfikację tabeli (`FOR EACH STATEMENT`) lub jej krotek (`FOR EACH ROW`). Wyzwalacz może być wywołany przed (`BEFORE`) lub po (`AFTER`) takiej modyfikacji.

```
CREATE TRIGGER odnotuj_wypis AFTER DELETE ON wybor FOR EACH ROW
EXECUTE PROCEDURE odnotuj_wyp();
```

Wyzwalacze w PostgreSQL

Powodują określoną reakcję na modyfikację tabeli (`FOR EACH STATEMENT`) lub jej krotek (`FOR EACH ROW`). Wyzwalacz może być wywołany przed (`BEFORE`) lub po (`AFTER`) takiej modyfikacji.

```
CREATE TRIGGER odnotuj_wypis AFTER DELETE ON wybor FOR EACH ROW
EXECUTE PROCEDURE odnotuj_wyp();
```

Procedura wyzwalacza musi zwracać typ `TRIGGER` i może wewnątrz odwoływać się do zmiennych `OLD` i `NEW` oznaczających krotkę sprzed zmian (dla `DELETE` i `UPDATE`) oraz po zmianach (dla `INSERT` i `UPDATE`).

```
CREATE FUNCTION odnotuj_wyp() RETURNS TRIGGER AS $X$
BEGIN
INSERT INTO wypisy
       VALUES(OLD.kod_uz,OLD.kod_grupy,OLD.czas_zapisu,now());
RETURN NULL;
END
$X$ LANGUAGE plpgsql;
```

Zwracanie wartości przez wyzwalacze

- Wyzwalacze typu `per-statement` - powinny zwracać NULL
- Wyzwalacze typu `per-row` mogą zwrócić pewną krotkę lub wartość NULL. Dla wyzwalaczy BEFORE:
  - zwrócenie wartości NULL powoduje pominięcie operacji, która wywołała wyzwalacz (INSERT/UPDATE/DELETE);
  - w przypadku operacji INSERT/UPDATE zwrócenie krotki powoduje, że ta operacja będzie dotyczyła właśnie tej krotki (zamiast krotki NEW).

  Jeśli wyzwalacz BEFORE nie ma modyfikować przeprowadzanej operacji to musi zwrócić krotkę: NEW dla INSERT/UPDATE lub krotkę OLD dla DELETE.
- Zwracana wartość jest ignorowana w przypadku wyzwalaczy AFTER.

Wyzwalacze w PostgreSQL: zadanie

Napisz wyzwalacz, który przy próbie zapisu studenta na zajęcia do pełnej grupy (tzn. przy dodawaniu nowej krotki do tabeli wybor) utworzy nową grupę do danego przedmiotu prowadzoną przez użytkownika 593, bez przydzielonego terminu i sali i zapisze tego studenta tam. Ustaw kod nowej grupy na 1 + maksymalna wartość kodu grupy w tabeli grupa. Grupę uznajemy za pełną jeśli liczba osób zapisanych do niej wczesniej jest równa lub większa niż wartość `max_osoby` dla tej grupy.

```
CREATE TABLE wybor (
    kod_grupy integer NOT NULL,
    kod_uz integer NOT NULL,
    data timestamp with time zone DEFAULT now() NOT NULL
);
CREATE TABLE grupa (
    kod_grupy integer NOT NULL,
    kod_przed_sem integer NOT NULL,
    kod_uz integer NOT NULL,
    max_osoby smallint NOT NULL,
    rodzaj_zajec character(1) NOT NULL,
    termin character(13),
    sala character varying(3)
);
```

Wyzwalacze w PostgreSQL

```
CREATE TRIGGER pg_trig AFTER INSERT ON wybor FOR EACH ROW EXECUTE PROCEDURE pg();

CREATE FUNCTION pg() RETURNS trigger AS $$
  DECLARE
    flag boolean;
    newgroup integer;
  BEGIN
  SELECT (COUNT(*) > max_osoby) INTO flag
    FROM grupa JOIN wybor USING (kod_grupy)
    WHERE kod_grupy = NEW.kod_grupy
    GROUP BY kod_grupy;

  IF (flag = true) THEN
    SELECT (max(kod_grupy)+1) INTO newgroup FROM grupa;
    INSERT INTO grupa (kod_grupy, kod_przed_sem, kod_uz, max_osoby, rodzaj_zajec)
      SELECT newgroup, kod_przed_sem, 593, max_osoby, rodzaj_zajec
      FROM grupa
      WHERE kod_grupy = NEW.kod_grupy;
    UPDATE wybor SET kod_grupy = newgroup
      WHERE kod_grupy = NEW.kod_grupy AND kod_uz = NEW.kod_uz;
  END IF;
  RETURN NULL;
  END; $$ LANGUAGE plpgsql;
```

Zadanie z SQL4 (2018)

Schemat:

```
users(screen_name, mentioned, troll) -- id, ile razy wzmiankowany, czy troll
tweets(tweet_id, screeen_name, text)  -- id tweeta, id autora tweeta, tekst tweeta
mentions(tweet_id, screen_name)  -- id tweeta, id wzmiankowanego usera
```

1. (1 pkt.) Napisz funkcję, która dla osoby podanej jako parametr (`screen_name`) zwróci ile razy ta osoba została wspomniana w tekstach tweetów.
2. (8 pkt.) Napisz wyzwalacz, który po operacji dodania nowego tweeta do tabeli `tweets` zaktualizuje tabelę `mentions` tak aby zawierała informacje o wszystkich nowych wzmiankach zawartych w tekście dodanego tweeta. W przypadku wzmianki o użytkowniku nieobecnym w tabeli `users` dodaj nowego użytkownika. Załóż, że taki użytkownik nie jest trollem. W pozostałych polach wpisz wartość NULL. Policz dla każdego tweeta tylko jedną wzmiankę - nawet jeśli tekst wzmianki np. `@jotop` jest powtarzany.

Zadanie z SQL4 (2018)—rozwiązanie

```
-- Zadanie 1

CREATE OR REPLACE FUNCTION z3(sn text) RETURNS integer AS
$X$
        SELECT mentioned
        FROM users
        WHERE screen_name = sn
$X$ LANGUAGE SQL;
```

Zadanie z SQL4 (2018)—regexp

Wskazówka: wzmianki w tekstach tweetów mają postać `@screen_name`, wzmianek można szukać używając poniższej funkcji i wyrażenia regularnego:

```
piotrek=#
 select distinct
     regexp_matches('@pwi bleble@ blabl pwi@uwr.edu @jotop @mabi @@jupi @jotop',
                    '(?:\A|\s)@(\w+)','g');
 regexp_matches
----------
 {pwi}
 {mabi}
 {jotop}
(3 rows)
```

Zadanie z SQL4 (2018)—regexp

Wskazówka: wzmianki w tekstach tweetów mają postać `@screen_name`, wzmianek można szukać używając poniższej funkcji i wyrażenia regularnego:

```
piotrek=#
 select distinct
     regexp_matches('@pwi bleble@ blabl pwi@uwr.edu @jotop @mabi @@jupi @jotop',
                    '(?:\A|\s)@(\w+)','g');
 regexp_matches
----------
 {pwi}
 {mabi}
 {jotop}
(3 rows)
```

ARRAYS: `text[], {a,b,c}`

Zadanie z SQL4 (2018)—ARRAYS

```
CREATE OR REPLACE FUNCTION mentions(input text) RETURNS SETOF text AS $X$
BEGIN
  RETURN QUERY
    SELECT match[1] FROM
      (SELECT DISTINCT regexp_matches(input, '(?:\A|\s)@(\w+)','g') as match) A;
RETURN;
END $X$ LANGUAGE plpgsql;
```

Zadanie z SQL4 (2018)—rozwiązanie

```
-- Zadanie 2
CREATE OR REPLACE FUNCTION z4() RETURNS TRIGGER AS $X$
DECLARE
  t text;
BEGIN
  FOR t IN
        SELECT DISTINCT mentions(NEW.text)
     LOOP
       IF t NOT IN (SELECT screen_name FROM users) THEN
             INSERT INTO users(troll, screen_name)
             VALUES (false, t);
             END IF;
     -- here update also the 'mentioned' value for the user 'screen_name'
     END LOOP;
     RETURN NEW;
END $X$ LANGUAGE plpgsql;

CREATE TRIGGER on_insert_to_tweets AFTER INSERT ON tweets
FOR EACH ROW EXECUTE PROCEDURE z4();
```

Implementacja FKs w Postgresql

```
CREATE TABLE parent(id int PRIMARY KEY, name text);
CREATE TABLE child(id int PRIMARY KEY, parentid int REFERENCES parent(id));

- populate with some random data (~2GB)

EXPLAIN ANALYZE DELETE FROM parent WHERE id = 50 ;

QUERY PLAN
------------------------------------------------------------------------------
Delete on parent  (cost=0.43..8.45 rows=0 width=0) (actual time=0.047..0.048 rows=0
 ->  Index Scan using parent_pkey on parent  (cost=0.43..8.45 rows=1 width=6) (actua
       Index Cond: (id = 50)
Planning Time: 0.198 ms
Trigger for constraint child_parentid_fkey: time=551.721 calls=1
Execution Time: 551.803 ms
```

Implementacja FKs w Postgresql

```
test=# ALTER TABLE LINEITEM
        ADD CONSTRAINT LINEITEM_FK1
        FOREIGN KEY (L_ORDERKEY)  REFERENCES ORDERS;

test=# SELECT tgname AS trigger_name
  FROM pg_trigger
 WHERE tgname !~ '^pg_';
         trigger_name
------------------------------
 RI_ConstraintTrigger_a_16419
 RI_ConstraintTrigger_a_16420
 RI_ConstraintTrigger_c_16421
 RI_ConstraintTrigger_c_16422
```

Implementacja FKs w Postgresql

```
CREATE CONSTRAINT TRIGGER "RI_ConstraintTrigger_c_16686"
    AFTER UPDATE ON public.tc FROM ta
      NOT DEFERRABLE INITIALLY IMMEDIATE FOR EACH ROW
                                    EXECUTE FUNCTION "RI_FKey_check_upd"()

CREATE CONSTRAINT TRIGGER "RI_ConstraintTrigger_c_16685"
    AFTER INSERT ON public.tc FROM ta
      NOT DEFERRABLE INITIALLY IMMEDIATE FOR EACH ROW EXECUTE FUNCTION "RI_FKey_chec

CREATE CONSTRAINT TRIGGER "RI_ConstraintTrigger_a_16684"
    AFTER UPDATE ON public.ta FROM tc
      NOT DEFERRABLE INITIALLY IMMEDIATE FOR EACH ROW EXECUTE FUNCTION "RI_FKey_noact

CREATE CONSTRAINT TRIGGER "RI_ConstraintTrigger_a_16683"
    AFTER DELETE ON public.ta FROM tc
      NOT DEFERRABLE INITIALLY IMMEDIATE FOR EACH ROW EXECUTE FUNCTION "RI_FKey_noact
```