PostgreSQL 9.2 efektivně

Programování uložených procedur

Pavel Stěhule

http://www.postgres.cz

21. 1. 2013

1. Podpora PostgreSQL na internetu
2. Uložené procedury, co a proč
3. SQL
4. Úvod do PL/pgSQL
5. Syntaxe příkazu CREATE FUNCTION
6. Blokový diagram PL/pgSQL
7. Příkazy PL/pgSQL
8. Dynamické SQL
9. Použití dočasných tabulek v PL/pgSQL
10. Ošetření chyb
11. Triggery v PL/pgSQL
12. Rekurze
13. Tipy pro PL/pgSQL
14. Příloha, Extenze
15. Příloha, Transakce
16. Security definer funkce
17. Příloha, Doporuční pro datový návrh
18. Řešení příkladů

Pavel Stěhule
pavel.stehule@gmail.com

- vývojář PostgreSQL od roku 1999 - vývoj PL/pgSQL,
- lektor PostgreSQL a SQL od roku 2006,
- instalace, migrace databází, audit aplikací postavených nad PostgreSQL,
- vývoj zákaznických modulů do PostgreSQL - více na http://www.postgres.cz/index.php/Služby,
- inhouse školení PostgreSQL,
- veřejná školení SQL a PLpgSQL,
- konzultace ohledně problémů s výkonem (performance) PostgreSQL,
- komerční podpora PostgreSQL

Informace a podpora PostgreSQL na internetu

- http://www.postgresql.org
- http://wiki.postgresql.org
- http://archives.postgresql.org
- http://www.postgres.cz
- http://groups.google.com/group/postgresql-cz?hl=cs
- http://pgfoundry.org
- irc://irc.freenode.net/#postgresql

Uložené procedury

Popis architektury I.

Uložené procedury

- uživatelem definovaný kód prováděný na serveru,
- skripty nebo přeložený kód využívající SPI rozhraní,
- nativní PL jazyk nebo klasický prg. jazyk,
- jeden ze základních bloků SQL frameworku.

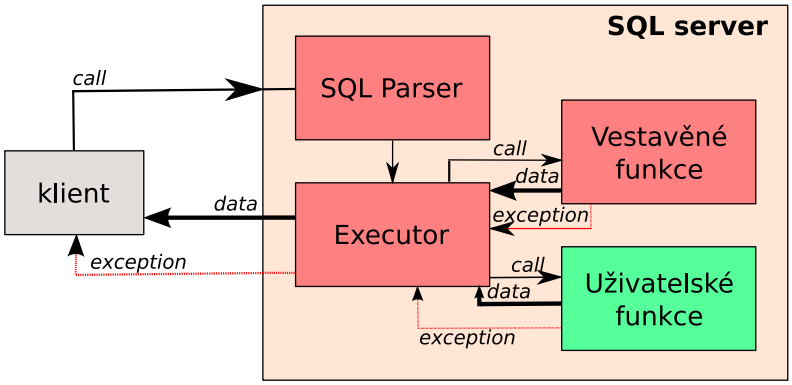


Schéma aktivace uložených procedur

Uložené procedury

Popis architektury II.

Výhody

- bezpečnost (omezení přístupu k tabulkám),
- bezpečnost a zapouzdření (triggery a uložené procedury nelze obejít),
- výkon (odstranění mezivrstev mezi kódem a serverem),
- výkon (snížení síťového provozu),
- výkon a bezpečnost (předpřipravené dotazy),
- znovupoužitelnost (každá db. aplikace sdílí data i uložené procedury),
- adaptabilita (možnost doplnění funkcionality databáze).

Nevýhody

- komplikovanější ladění (vzdálený proces),
- z důvodu bezpečnosti omezená funkcionalita (přístup k IO),
- z důvodu běhu na serveru nulový přístup k IO klienta,
- nutnost pochopit další architekturu,
- minimální přenositelnost (chybí širší podpora SQL/PSM).

Uložené procedury

Porovnání PL a klasických prg. jazyků I.

Výhody nativních PL jazyků (tzv. SQL procedury)

- čitelnost a efektivnost zápisu (integrace SQL),
- minimální režie (všechny funkce jsou sdílené s SQL serverem),
- výkon (binární kompatibilita typů s SQL serverem),
- výkon a bezpečnost (interně téměř vždy předpřipravené SQL příkazy).
- bezpečnost (chybí nebezpečné funkce).

Výhody klasických prg. jazyků (tzv. externí procedury)

- komplexní funkcionalita včetně přístupu k IO,
- přístup ke knihovnám,
- rychlé a efektivní operace nad zákl. datovými typy.

Uložené procedury

Porovnání PL a klasických prg. jazyků II.

Závěr

- pokud výkonostně a funkčně postačuje PL, upřednostnit PL,
- jinak libovolný známý podporovaný prg. jazyk (Perl, Python, Java).

Pozn. k Javě

- standard SQLJ (určitý potenciál pro přenositelnost) je relativně rozšířen a podporován,
- norma je navržena tak, aby se minimalizovali specifika SP (teoreticky kód klientské aplikace se specifickým *connection stringem*),

Uložené procedury

Vzory užití

Kdy použít uložené procedury?

- Jen když je to nezbytně nutné (výkon, bezpečnost).
- Vždy (požadavek architektury).
- Kdykoliv, když mohou být užitečné (výkon, bezpečnost, čitelnost).

Jak použít uložené procedury?

V aplikacích pro SQL Server Veškerou interakci lze zapouzdřit do uložených procedur (včetně dotazů).

V aplikacích pro Oracle Jelikož vrátit tabulku, jako výsledek PL/SQL funkce, je mnohem pracnější než v T-SQL, aplikace pro Oracle používají uložené procedury spíše pro implementaci procesů, a nativní SQL dotazy k získání výsledných tabulek.

SQL

Použití SQL jako PL jazyka

Popis

- jednoduchá syntaxe (řada parametrizovaných SQL příkazů),
- výsledkem funkce je výsledek posledního SQL příkazu,
- vždy po ruce (není nutné jej explicitně povolit),
- používá se jako MACRO jazyk (inlining),
- nebo pro návrh funkcí z vrstvy minimalizující rozdíly mezi RDBMS.

```
CREATE OR REPLACE FUNCTION right(text, int)
RETURNS text AS $$
  SELECT substring($1 FROM length($1) - $2 + 1 FOR $2)
$$ LANGUAGE sql;

CREATE OR REPLACE FUNCTION left(text, int)
RETURNS text AS $$
  SELECT substring($1 FROM 1 FOR $2)
$$ LANGUAGE sql;
```

SQL
Cyklus v SQL

Pozn.

V SQL konstrukci cyklu lze nahradit funkcí *generate_series*, *generate_subscripts* a *unnest*. Jedná se o funkce, které generují sloupec hodnot.

```
postgres=# SELECT * FROM unnest(ARRAY[4,2,3]);
 unnest
--------
      4
      2
      3
(3 rows)
postgres=# SELECT substring('abc' FROM i FOR 1)
              FROM generate_series(1,3) g(i);
 substring
-----------
 a
 b
 c
(3 rows)
```


SQL
Uložení mezivýsledků a agregace v SQL

Pozn.

V SQL nelze zapsat do proměnných. Jedinou možností je převést výslednou tabulku na pole (funkce *array_agg*) a poté pole na cílový typ (funkce *array_to_string*).

```
postgres=# SELECT array_to_string(array_agg('<step>' || i
                                            || '</step>'),'')
              FROM generate_series(1,4) g(i);
                       array_to_string
-------------------------------------------------------------
 <step>1</step><step>2</step><step>3</step><step>4</step>
(1 row)
```

SQL
Integrace SQL v SQL funkcích

Pozn.

Veškeré úlohy je nutné vyřešit v čistém SQL - zde využití schopnosti řazení, tj. klauzule ORDER BY.

```
postgres=# CREATE OR REPLACE FUNCTION sort(anyarray)
           RETURNS anyarray AS $$
             SELECT ARRAY(SELECT * FROM unnest($1) ORDER BY 1)
           $$ LANGUAGE SQL IMMUTABLE;
CREATE FUNCTION
postgres=# select sort(ARRAY[10,12,1,3]);
    sort
------------
 1,3,10,12
(1 row)
```


SQL
Variadické SQL funkce

Pozn.

Variadické funkce mají variabilní počet parametrů - variadické parametry jsou předány jako pole. Funkce avgc vrací průměr ze zadaných hodnot.

```
postgres=# CREATE OR REPLACE FUNCTION avgc(VARIADIC float[])
           RETURNS float  AS $$
             SELECT avg(v) FROM unnest($1) g(v)
           $$ LANGUAGE SQL IMMUTABLE;
CREATE FUNCTION

postgres=# select avgc(1), avgc(1,0), avgc(1,0,1);
 avgc | avgc |       avgc
------+------+------------------
    1 |  0.5 | 0.666666666666667
(1 row)
```

SQL
Cvičení

Úloha č. 1.

V jazyce SQL navrhněte funkci last_day - vrací poslední den v měsíci. Strategie: Od 1. následujícího měsíce (date_trunc) odečtěte jeden den.

Úloha č. 2.

Pouze v jazyce SQL navrhněte nerekurzivní formu funkce *rvrs*. Tato funkce zrcadlově prohodí znaky v řetězci. Nápověda: `CASE`, `generate_series()` a `ARRAY`

Pozor

```
postgres=# SELECT 5 / 2 AS CD, 5.0 / 2.0 AS FD;
 cd |         fd
----+--------------------
  2 | 2.5000000000000000
(1 row)
```


SQL
Cvičení

Úloha č. 3.

Pouze v jazyce SQL navrhněte rekurzivní formu funkce *rvrs*.

Úloha č. 4.

V SQL navrhněte funkci, které odstraní duplicity z pole.

Úloha č. 5.

V SQL navrhněte funkci, která vrátí průnik dvou polí.

Úloha č. 6.

V SQL navrhněte variadickou funkci, která je obdobou funkce *least*.

Úloha č. 7.

Definujte operátor ||| pro průnik polí.

SQL

Vlastní operátor

Pozn.

V zápisu je povinné zachování pořadí určení atributů.

```
postgres=# \h create operator
Command:     CREATE OPERATOR
Description: define a new operator
Syntax:
CREATE OPERATOR name (
    PROCEDURE = funcname
    [, LEFTARG = lefttype ] [, RIGHTARG = righttype ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
)
```


PL/Perlu

Použití perlu pro implementaci SP I.

Popis

Perl lze používat ve dvou variantách - trusted (plperl) a untrusted (plperlu). Trusted varianta nedovolí použití ”potenciálně” rizikových příkazů a funkcí. Použití pro intenzivní matematické operace a operace s řetězci, využití archivu CPAN.

```
CREATE OR REPLACE FUNCTION check_email(varchar)
RETURNS boolean AS $$
use strict;
use Email::Valid;
my $address = $_[0];
my $checks = {-address => $address, -mxcheck => 1,
              -tldcheck => 1,        -rfc822 => 1,};
if (defined Email::Valid->address( %$checks )) {return 'true'}
elog(WARNING, "address failed $Email::Valid::Details check.");
return 'false';
$$ LANGUAGE plperlu IMMUTABLE STRICT;
```

PL/Perlu

Použití perlu pro implementaci SP II.

Použití DBI pro import dat

Rozhraní DBI je faktický standard coby DB rozhraní v systémech typu Unix. K dispozici jsou kvalitni ovladače pro všechny databáze a formáty.

```
CREATE OR REPLACE FUNCTION ext.rmysql(varchar, varchar, varchar,
  varchar) RETURNS SETOF RECORD AS $$
   use DBI;
   my $dbh = DBI->connect('dbi:mysql:'.$_[0],$_[1],$_[2],
      { RaiseError => 1, AutoCommit = > });
   $dbh->do("set character_set_results='latin2'");
   my $sth = $dbh->prepare($_[3]);
   $sth->execute(); my $myref;
   while ($dat = $sth->fetchrow_hashref) {push @$myref, $dat; }
   $sth->finish(); $dbh->disconnect();
   return $myref;
$$ LANGUAGE plperlu;
```


PL/pgSQL

Úvod

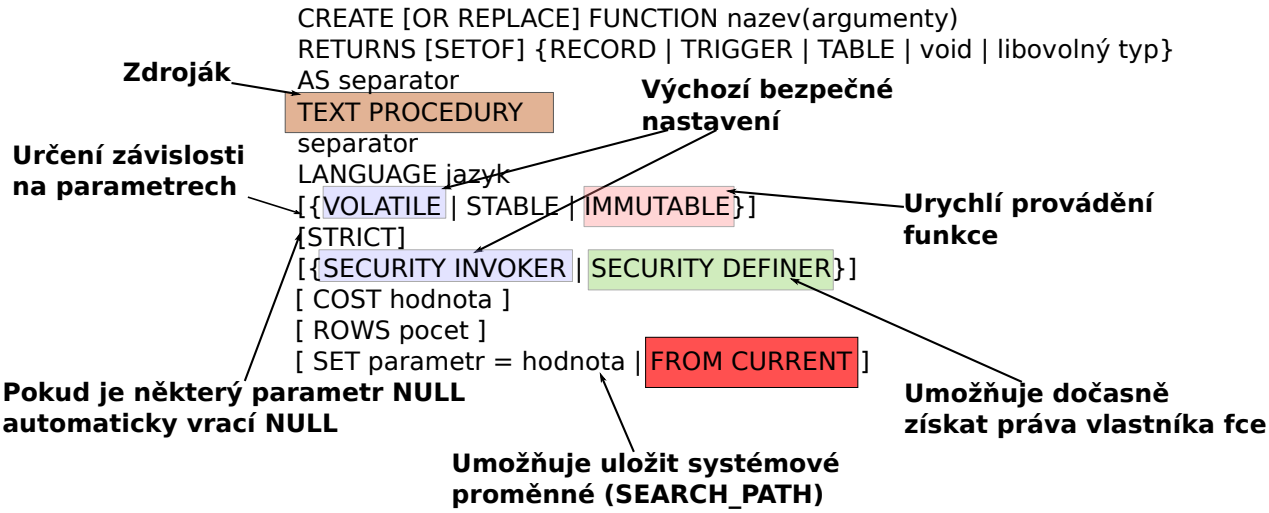
Popis

- procedurální programovací jazyk vycházející z prg. jazyka ADA,
- téměř kompatibilní se staršími verzemi PL/SQL fy. Oracle,
- minimalismus (SQL + proměnné + několik základních konstrukcí),
- podpora OUT parametrů a polymorfních funkcí,
- podpora SRF funkcí (výstupem je tabulka),
- podpora triggerů,
- chybí podpora procedur (pouze funkce).

```
CREATE OR REPLACE FUNCTION hello(text)
RETURNS text AS $$
BEGIN
  RETURN 'Hello ' || $1;
END;
$$ LANGUAGE plpgsql IMMUTABLE;
```

Příkaz *CREATE FUNCTION*

Popis



Představení jazyka PL/pgSQL

Vstupní parametry

```
CREATE OR REPLACE FUNCTION fib(prvku int)
RETURNS SETOF int AS $$
DECLARE
  a int = 0;
  b int = 0;
  c int;
BEGIN
  FOR i IN 1..prvku LOOP
    c := CASE a + b WHEN 0 THEN 1
                    ELSE a + b END;
    a := b; b := c;
    RETURN NEXT c;
  END LOOP;
  RETURN;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

Představení jazyka PL/pgSQL

Blok

```
CREATE OR REPLACE FUNCTION fib(prvku int)
RETURNS SETOF int AS $$
DECLARE
  a int = 0;
  b int = 0;
  c int;
BEGIN
  FOR i IN 1..prvku LOOP
    c := CASE a + b WHEN 0 THEN 1
                    ELSE a + b END;
    a := b; b := c;
    RETURN NEXT c;
  END LOOP;
  RETURN;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```


Představení jazyka PL/pgSQL

Deklarace lokálních proměnných bloku

```
CREATE OR REPLACE FUNCTION fib(prvku int)
RETURNS SETOF int AS $$
DECLARE
  a int = 0;
  b int = 0;
  c int;
BEGIN
  FOR i IN 1..prvku LOOP
    c := CASE a + b WHEN 0 THEN 1
                    ELSE a + b END;
    a := b; b := c;
    RETURN NEXT c;
  END LOOP;
  RETURN;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

Představení jazyka PL/pgSQL

Cyklus *FOR*

```
CREATE OR REPLACE FUNCTION fib(prvku int)
RETURNS SETOF int AS $$
DECLARE
  a int = 0;
  b int = 0;
  c int;
BEGIN
  FOR i IN 1..prvku LOOP
    c := CASE a + b WHEN 0 THEN 1
                    ELSE a + b END;
    a := b; b := c;
    RETURN NEXT c;
  END LOOP;
  RETURN;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```


Představení jazyka PL/pgSQL

Vložené SQL příkazy

```
CREATE OR REPLACE FUNCTION fib(prvku int)
RETURNS SETOF int AS $$
DECLARE
  a int = 0;
  b int = 0;
  c int;
BEGIN
  FOR i IN 1..prvku LOOP
    c := CASE a + b WHEN 0 THEN 1
                    ELSE a + b END;
    a := b; b := c;
    RETURN NEXT c;
  END LOOP;
  RETURN;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

Představení jazyka PL/pgSQL

Přiřazení

```
CREATE OR REPLACE FUNCTION fib(prvku int)
RETURNS SETOF int AS $$
DECLARE
  a int = 0;
  b int = 0;
  c int;
BEGIN
  FOR i IN 1..prvku LOOP
    c := CASE a + b WHEN 0 THEN 1
                    ELSE a + b END;
    a := b; b := c;
    RETURN NEXT c;
  END LOOP;
  RETURN;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

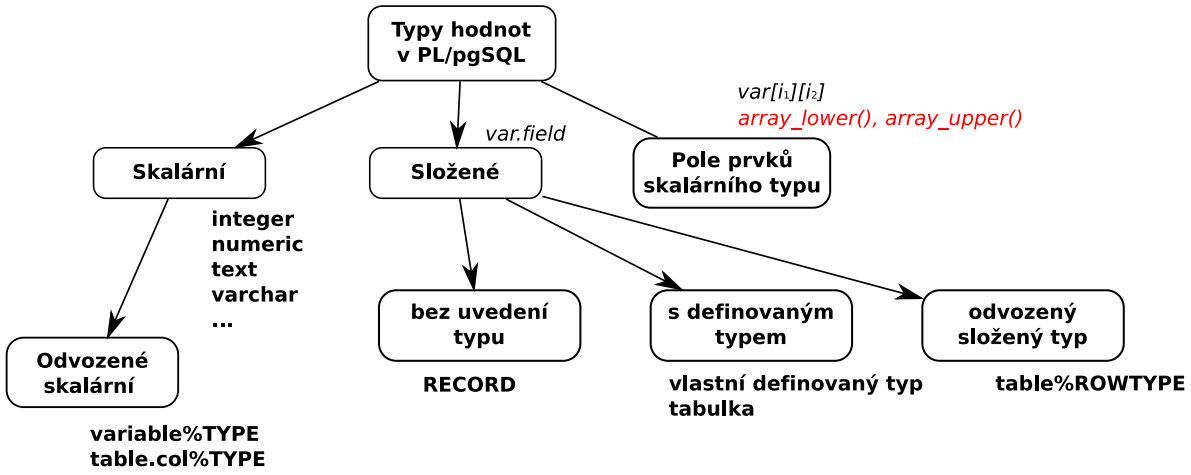

Představení jazyka PL/pgSQL

Ukončení běhu *RETURN*

```
CREATE OR REPLACE FUNCTION fib(prvku int)
RETURNS SETOF int AS $$
DECLARE
  a int = 0;
  b int = 0;
  c int;
BEGIN
  FOR i IN 1..prvku LOOP
    c := CASE a + b WHEN 0 THEN 1
                    ELSE a + b END;
    a := b; b := c;
    RETURN NEXT c;
  END LOOP;
  RETURN;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

Představení jazyka PL/pgSQL

Datové typy PL/pgSQL



Představení jazyka PL/pgSQL

Cvičení

Úloha č. 8.

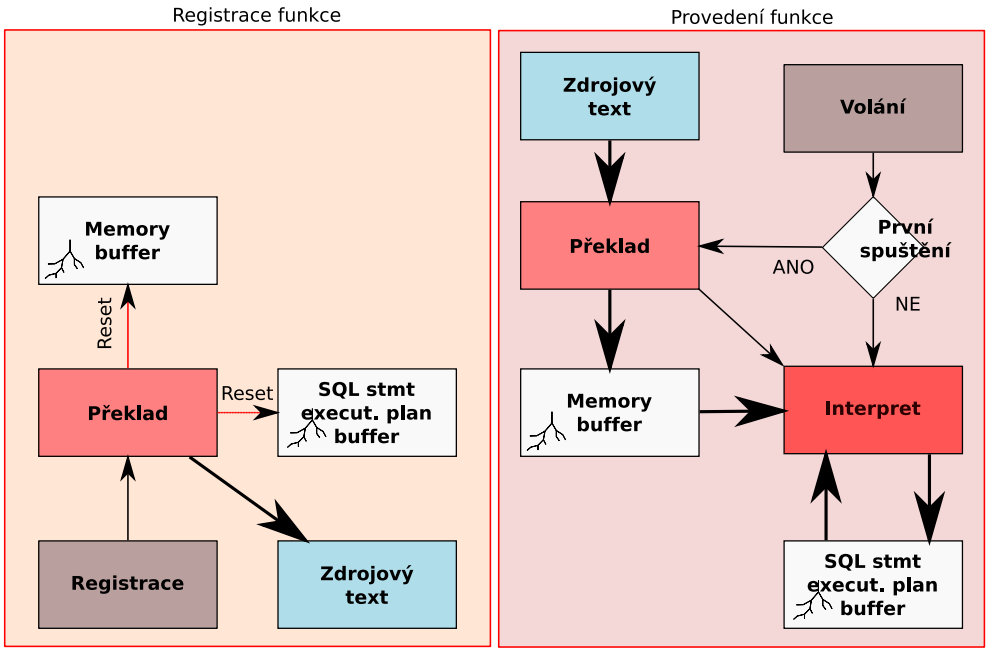
V jazyce PL/pgSQL navrhněte funkci *sort*, jejíž úkolem je setřídit zadané celočíselné pole metodou *bublesort*.

Úloha č. 9.

Pouze v jazyce PL/pgSQL navrhněte funkci *rvrs*. Tato funkce zrcadlově prohodí znaky v řetězci.

Blokový diagram PL/pgSQL

Registrace a interpretace funkce v plpgsql



Blokové diagram interpretace funkce v PL/pgSQL

Příkazy PL/pgSQL

Blok, lokální proměnné, zpracování chyb

```
[<<návěstí>>]
[DECLARE
  prom typ [NOT] NULL [= výchozí hodnota];
  [prom typ [NOT] NULL ... ]]
BEGIN
  ... -- libovolný SQL nebo PL/pgSQL příkaz
[EXCEPTION
  WHEN chybový kód THEN
    ... -- ošetření chyby
  WHEN chybové kód THEN
    ... -- ošetření chyby
  WHEN OTHERS THEN
    ... -- ošetření chyby
END;
```

Příkazy PL/pgSQL

Podmíněný provádění IF

```
IF logický výraz THEN
  ... -- libovolné příkazy
[ELSIF ... THEN]
  ...
[ELSE]
  ...
END IF;
```


Příkazy PL/pgSQL

Příkaz CASE

```
-- simple case
CASE výraz
     WHEN výraz THEN seznam příkazů
    [WHEN výraz THEN seznam příkazů]
    [ELSE seznam příkazů] END CASE;

-- search case
CASE
    WHEN logický výraz THEN seznam příkazů
   [WHEN logický výraz THEN seznam příkazů]
   [ELSE seznam příkazů] END CASE;
```

Pozor

Pokud není žádný test pravdivý a chybí část ELSE, pak je vyhozena výjimka CASE_NOT_FOUND.

Příkazy PL/pgSQL

Příkaz cyklu WHILE

```
[<<návěstí>>]
WHILE podmínka LOOP
  ...
  CONTINUE [návěstí] [WHEN podmínka];
  EXIT [návěstí] [WHEN podmínka];
  ...
END LOOP;
```


Příkazy PL/pgSQL

Příkaz cyklu FOR (celočíselná varianta)

Pozn.

Proměnná *prom* je vždy deklarovaná automaticky a viditelná pouze uvnitř cyklu. Pro případ, že by překryla vně deklarovanou proměnnou, existují tzv. *kvalifikované identifikátory* (návěstí.proměnná).

```
[<<návěstí>>]
FOR prom IN [REVERSE] výraz .. výraz [BY výraz] LOOP
  ...
  CONTINUE [návěstí] [WHEN podmínka];
  EXIT [návěstí] [WHEN podmínka];
  ...
END LOOP [návěstí];
```

Příkazy PL/pgSQL

Příkaz cyklu FOREACH (iterace pole)

Pozn.

Pomocí klauzule SLICE je možná iterace přes vícerozměrné pole.

```
[<<návěstí>>]
FOREACH prom [SLICE dim] IN ARRAY výraz LOOP
  ...
  CONTINUE [návěstí] [WHEN podmínka];
  EXIT [návěstí] [WHEN podmínka];
  ...
END LOOP [návěstí];
```


Příkazy PL/pgSQL

Příkaz cyklu FOR (iterace napříč výsledkem dotazu)

Pozn.

Nedefinuje vlastní řídící proměnné jako celočíselná varianta. Řádek výsledku lze uložit do proměnné typu RECORD nebo seznamu proměnných oddělených čárkou.

```
[<<návěstí>>]
FOR cílové proměnné IN [ dotaz | EXECUTE výraz ] LOOP
  ...
  CONTINUE [návěstí] [WHEN podmínka];
  EXIT [návěstí] [WHEN podmínka];
  ...
END LOOP [návěstí];
```

Příkazy PL/pgSQL

Plnění proměnných výsledkem SQL dotazu I.

Pozor

Jednou z nejhůře postižitelných chyb je kolize názvů PL/pgSQL proměnných a názvů atributů v tabulkách. Pokud k této situaci může dojít, používejte vždy prefix '_' v názvech lokálních proměnných. Dále proměnná se nikdy nesmí vyskytnout ve významu názvu sloupce nebo tabulky. Možné kolize identifikátorů a proměnných jsou explicitně identifikovány počínaje PostgreSQL 9.0, nicméně je praktické psát kód bez kolizí.

```
SELECT INTO p1 [,p2 ..] v1 [,v2 ..] FROM ...;
SELECT INTO r v1 [, v2 ..] FROM ...;
a := (SELECT .. FROM ...); -- dle SQL/PSM
a := v1 FROM ...; -- nedokumentovaná vlastnost
```

Pozn.

Po provedení příkazů lze číst systémovou proměnnou *FOUND*, obsahující hodnotu pravda, pokud předchozí SQL příkaz vrátil (modifikoval) alespoň jeden záznam.

Příkazy PL/pgSQL

Plnění proměnných výsledkem SQL dotazu II.

Pozor

Při kolizi názvu proměnné a sloupce většinou nedochází ke syntaktické chybě, ale ke změně prováděcího plánu - mění se chování dotazu. Obrana: prefixy pro proměnné nebo použití kvalifikátorů.

```
CREATE OR REPLACE FUNCTION test(OUT a varchar)
RETURNS SETOF varchar AS $$
DECLARE s varchar;
BEGIN
  --FOR a IN SELECT a FROM foo ORDER BY a LOOP
  FOR s IN EXPLAIN VERBOSE SELECT a FROM foo ORDER BY a LOOP
    RAISE NOTICE '%', s;
  END LOOP;
  RETURN;
END; $$ LANGUAGE plpgsql;
```

Příkazy PL/pgSQL

Plnění proměnných výsledkem SQL dotazu III.

```
postgres=# select * from test();
NOTICE:  Seq Scan on foo  (cost=0.00..23.10 rows=1310 width=0)
NOTICE:    Output: $1
 a
---
(0 rows)

-- po opravě
-- FOR s IN EXPLAIN VERBOSE SELECT foo.a FROM foo ORDER BY foo.a LOOP
postgres=# select * from test();
NOTICE:  Sort  (cost=90.93..94.20 rows=1310 width=32)
NOTICE:    Output: a
NOTICE:    Sort Key: a
NOTICE:    ->  Seq Scan on foo  (cost=0.00..23.10 rows=1310 width=32)
NOTICE:          Output: a
 a
---
(0 rows)
```


Příkazy PL/pgSQL

Test na neprázdný výsledek

```
-- špatně
SELECT INTO c count(*)
   FROM pg_stat_activity
  WHERE procpid <> pg_backend_pid;
IF c > 0 THEN ...

-- dobře
IF EXISTS(SELECT procpid
            FROM pg_stat_activity
           WHERE procpid <> pg_backend_pid)
THEN
  ...
```

Pozor

Pokud potřebujeme zjistit existenci určitých dat, pak vždy používáme EXISTS, nikdy count(*) (provádění op. EXISTS se ukončí po nalezení prvního řádku).

Příkazy PL/pgSQL

Použití SQL příkazu IV.

Pozn.

S výjimkou příkazu *SELECT* se všechny ostatní SQL příkazy zapisují standardně.

Proměnné

Proměnná PL/pgSQL se může použít pouze jako parametr, tedy nikdy ve smyslu určení (názvu) sloupce nebo tabulky.

```
UPDATE tab SET c1 = 10 WHERE c2 = _a;
INSERT INTO tab VALUES(11, _a);
```


Příkazy PL/pgSQL

Ukončení funkce. příkaz RETURN a RAISE

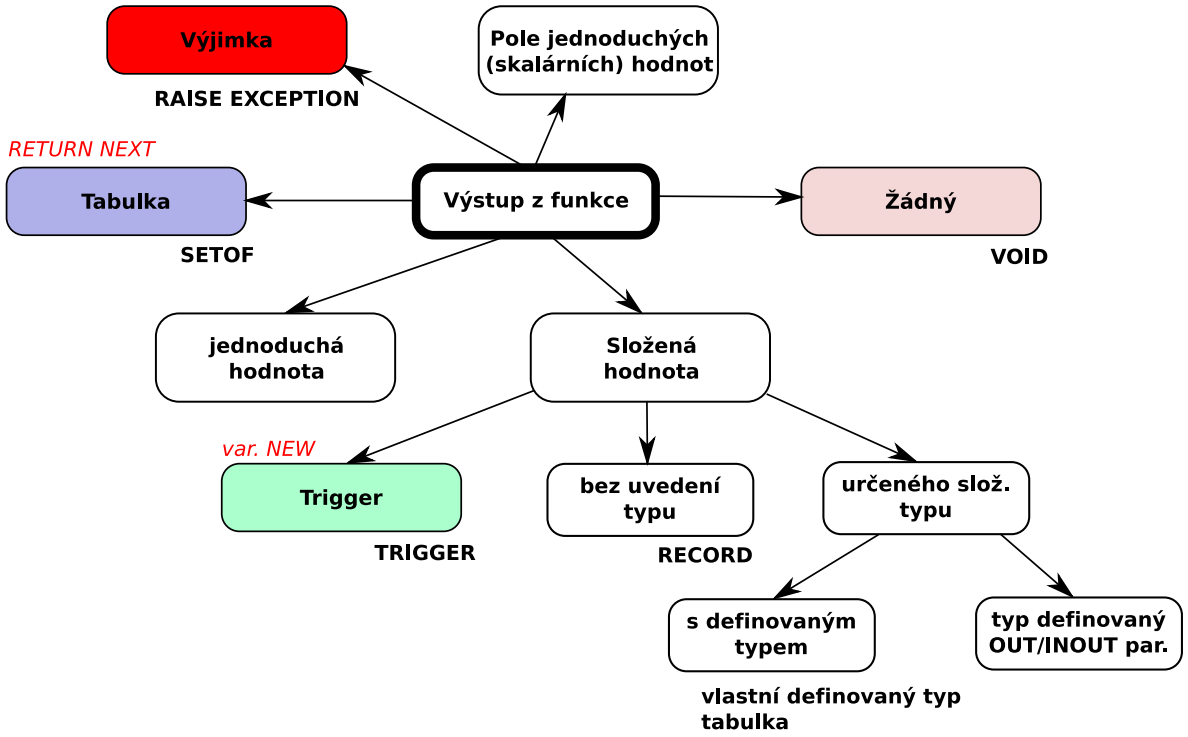
```
RETURN [NEXT] [výraz];

RETURN QUERY [EXECUTE] dotaz [USING seznam výrazů];

RAISE {NOTICE | WARNING | EXCEPTION}
   'formátovací řetězec [%] '  [, seznam výrazů];

RAISE level [SQLSTATE='xxxxx'] [USING parametr=výraz [, ...]];
RAISE;
```

Úrovně přerušení příkazu RAISE

NOTICE ladění a oznámení,
WARNING upozornění (zapíše se do logu),
EXCEPTION vyjímka (přeruší transakci).

Příkazy PL/pgSQL

Výstup z funkce, datové typy



Příkazy PL/pgSQL

Výstup z funkce, poznámky

Poznámky

- identifikace OUT a INOUT argumentů atributy *OUT* a *INOUT* před jménem parametru,
- funkce vracející tabulku musí být deklarovaná *RETURNS SETOF ...*,
- pokud funkce vrací OUT parametry, není nutné používat specifikaci typu *RETURNS ...*,
- funkce vracející tabulku OUT parametrů je deklarována jako *RETURNS SETOF RECORD* případně *RETURNS TABLE (...)*
- pozor na pouze jeden OUT parametr (výsledkem není RECORD),
- do generované tabulky přidáváme hodnoty příkazem *RETURN NEXT* nebo řádky příkazem *RETURN QUERY*.

```
CREATE OR REPLACE FUNCTION foo (OUT a integer, OUT b integer)
RETURNS SETOF RECORD AS $$
BEGIN b := 1;
  a := 10; RETURN NEXT; a := 20; RETURN NEXT;
  RETURN;
END; $$ LANGUAGE plpgsql IMMUTABLE;
```

Úloha č. 10.

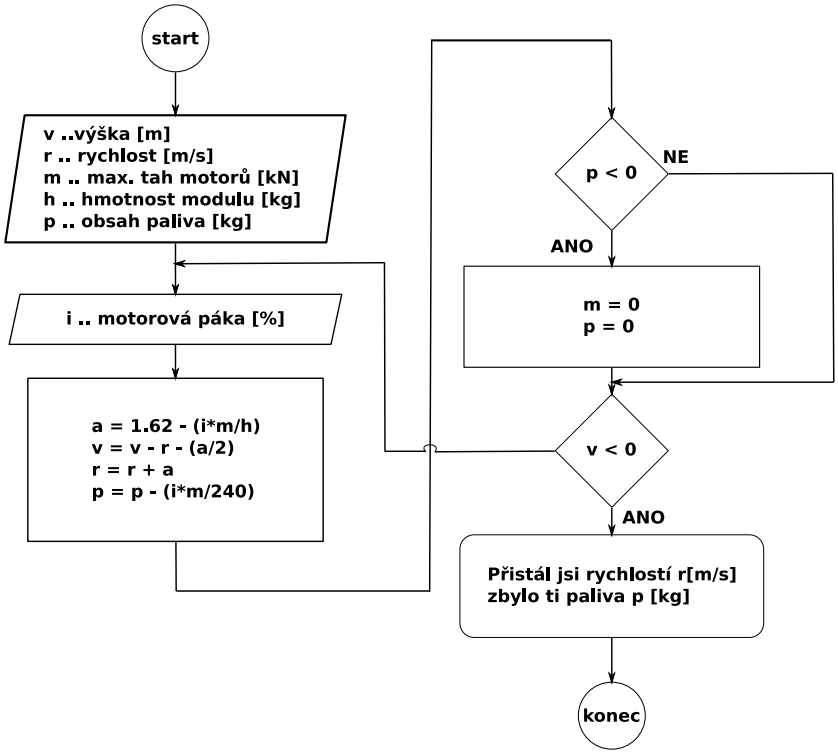
Připravenou tabulku naplňte hodnotami z rozsahu 0..2*pi s zadaným krokem.

Úloha č. 11.

Napište simulaci hry ”přistání na měsíci” v prostředí uložených procedur. Uživatel opakovaně zadává intenzitu brždění a systém zobrazuje aktuální výšku, rychlost a stav paliva. Algoritmus je popsán v následujícím vývojovém diagramu. Výsledkem funkce je aktuální výška. Další výsledky jsou zobrazeny pomocí *RAISE NOTICE*. Hodnoty stavových proměnných (statických) udržujte v tabulce.

Úloha č. 12.

Navrhněte funkci array_info vracejí minimum, maximum a medián ze zadaného pole. Pro výstup hodnot použijte OUT parametry. Medián je definován jako prostřední hodnota setříděného souboru. V případě sudého počtu je pak průměr obou prostředních hodnot.



Příkazy PL/pgSQL

Chování operátorů IS NULL a IS NOT NULL pro složené typy

Pozor

Operátor IS NULL je pravdivý, pokud jsou všechny položky NULL. Naopak, operátor IS NOT NULL je pravdivý, pokud jsou všechny položky NOT NULL. Pokud je alespoň jedna položka NULL a alespoň jedna NOT NULL, pak oba operátory nabývají hodnoty false.

```
 a  | b  | isnull | isnotnull
----+----+--------+-----------
    |    | t      | f
    | 10 | f      | f
 10 | 10 | f      | t
(3 rows)
```


Příkazy PL/pgSQL

Cvičení

Úloha č. 13.

Pro níže uvedenou tabulku připravte SRF funkci, generující kumulativní mezisoučty a koncový součet tabulky

```
  zbozi    | kategorie |  cena
-----------+-----------+--------
 spekacky  | uzeniny   | 100.00
 sekana    | uzeniny   |  60.00
 chleb     | pecivo    |  25.00
 rohliky   | pecivo    |  30.00
(4 rows)
```

```
  zbozi    | kategorie |  cena
-----------+-----------+--------
 chleb     | pecivo    |  25.00
 rohliky   | pecivo    |  30.00
           | pecivo    |  55.00
 spekacky  | uzeniny   | 100.00
 sekana    | uzeniny   |  60.00
           | uzeniny   | 160.00
           |           | 215.00
(7 rows)
```

Příkazy PL/pgSQL

Přetížení funkcí

Popis

Lze definovat množinu funkcí stejného jména lišících se seznamem IN a INOUT parametrů. Použije se ta funkce, která má stejný počet parametrů a vyžaduje nejméně konverze argumentů. V některých případech je implicitní konverze nedostatečná a je třeba explicitní přetypování.

Použití

- jako náhrada za neimplementované defaultní hodnoty parametrů,
- jako nástroj implementace specifik funkcí pro konkrétní datové typy.

Úloha č. 14.

Navrhněte vlastní implementaci funkce *generate_series*. Tato funkce generuje tabulku obsahující vzestupnou řadu celých čísel. Třetím parametrem této funkce je *krok*. který je defaultně nastaven na hodnotu 1.

Dynamické SQL

Popis

Vlastnosti

- zadaný řetězec se provede jako SQL příkaz,
- lze provést i sql příkazy, které normálně nelze provést (create function),
- lze použít v případech, kdy potřebujeme jako parametr použít název sloupce nebo tabulky,
- prováděcí plán se generuje opakovaně vždy v okamžiku provádění SQL příkazu
  - v některých případech kvalitnější prováděcí plán,
  - odpadají problémy s kešováním prováděcích plánů,
  - generování prováděcího plánu v některých případech zabere víc času než provedení dotazu.
- nezabezpečené proti SQL injection (nepoužívat pod SECURITY DEFINER s právy superusera) v případě, že se nepoužívá klauzule USING.

```
EXECUTE cmdstr [INTO target] [USING expressions];
FOR target IN EXECUTE cmdstr [USING expr] LOOP ... END LOOP;
```

Dynamické SQL
Popis

Pozor: dynamické SQL neaktualizuje proměnnou *FOUND*

Z důvodů kompatibility s PL/SQL nemá smysl testovat proměnnou *FOUND* po provedení dynamického dotazu. Místo tohoto příkazu lze získat diagnostiku - počet dotčených řídků posledním SQL příkazem (včetně dynamického).

```
EXECUTE 'UPDATE tab SET col = 1 WHERE col = $1' USING var;
-- nemá smysl, obsah proměnné FOUND není definován
IF NOT FOUND THEN RAISE EXCEPTION ...

DECLARE updated_rows integer;
...
EXECUTE 'UPDATE ...
GET DIAGNOSTICS updated_rows = ROW_COUNT;
IF updated_rows = 0 THEN RAISE EXCEPTION ...
```


Dynamické SQL
SQL injection

Pozor: lze použít na *libovolnou* SQL databázi

Cílem útočníka je změna SQL příkazů - podsunutím části SQL příkazu. Je nutné sanitizovat *každý* vstup do dynamického SQL.

```
-- nebezpečné!
EXECUTE 'SELECT * FROM ' || tablename ||
   ' WHERE colname = \'' || value || '\'';

-- bezpečné
EXECUTE 'SELECT * FROM ' || quote_ident(tablename) ||
  ' WHERE colname = ' || quote_literal(value);

-- bezpečné - kratší zápis
EXECUTE format('SELECT * FROM %I WHERE colname = %L',
               tablename, value);

-- bezpečné a efektivnější (nedochází k serializaci parametru)
EXECUTE 'SELECT * FROM ' || quote_ident(tablename) ||
  ' WHERE colname = $1' USING value;
```

Použití dočasných tabulek v PL/pgSQL

Popis

Charakteristika dočasných tabulek

- automaticky zanikají s ukončením session,
- existují pouze v jedné session
  - odpadá problém s kolizí názvu tabulek mezi jednotlivými session,
  - odpadá problém s selekcí dat pro jednotlivé session (uživatele),
  - není nutné řešit bezpečnost (k datům má přístup pouze aktuální uživatel).

Použití v kódu PL/pgSQL

Pokud chceme používat dočasné tabulky v PL/pgSQL, tak nikdy nesmíme použitou dočasnou tabulku odstranit (to se stane automaticky v okamžiku ukončení session). Jinak pravděpodobně narazíme na chybu typu: relation with OID xxxxx does not exist (částečně odstraněno v 8.3 - v této verzi dojde při zrušení db objektu ke kontrole validity prováděcích plánů - *režie kontroly, nutnost vygenerovat nové plány*).

Použití dočasných tabulek v PL/pgSQL

Ukázka kódu, chybně

```
CREATE OR REPLACE FUNCTION init_tmp()
RETURNS VOID AS $$
BEGIN
  DROP TABLE IF EXISTS tmptab;
  CREATE TEMP TABLE tmptab(a integer);
  FOR i IN 1 .. 10 LOOP
    INSERT INTO tmptab
       VALUES(i);
  END LOOP;
END;
$$ LANGUAGE plpgsql VOLATILE;
```

Proč

- při druhém průchodu skončí chybou v příkazu INSERT (v 8.2 a starších),
- v 8.3 je nutné vygenerovat nový prováděcí plán (implicitně).

Použití dočasných tabulek v PL/pgSQL

Ukázka kódu, správně

```
CREATE OR REPLACE FUNCTION init_tmp()
RETURNS VOID AS $$
BEGIN
  BEGIN
    DELETE FROM tmptab;
  EXCEPTION
    WHEN undefined_table THEN
      CREATE TEMP TABLE tmptab(a integer);
  END;
  FOR i IN 1 .. 10 LOOP
    INSERT INTO tmptab
       VALUES(i);
  END LOOP;
END;
$$ LANGUAGE plpgsql VOLATILE;
```


Použití dočasných tabulek v PL/pgSQL

Cvičení

Úloha č. 15.

Upravte kód hry přistání na měsíci (zadání č. 5) tak, aby pracovní (stavové) proměnné byly drženy v dočasné tabulce. Výsledek, který byl dříve zobrazován pomocí RAISE NOTICE převeďte do OUT parametrů. Zobrazte varování pouze v případě, že přistaní bude ve větší rychlosti než 3m/s. Pokud dojde palivo, automaticky dokončete výpočet modelu, přičemž stále zobrazujte mezivýsledky (tj. funkce bude vracet tabulku).
Přidejte funkci, která bude inicializovat model (stavové proměnné).

Ošetření chyb v PL/pgSQL

Ukázka kódu

```
BEGIN
  ...
  EXCEPTION
    WHEN názevvýjimky THEN
       obsluha chyby
    WHEN názevvýjimky THEN
       obsluha
END
```

Poznámka

Použití zachycení výjmky způsobí použití subtransakce obalujícího kód v bloku. V případě výjimky je tato subtransakce odvolána (rollback). Popis výjimky lze získat pomocí příkazu `GET STACKED DIAGNOSTICS (verze 9.2)`.

```
GET STACKED DIAGNOSTICS
        _message = MESSAGE_TEXT,
        _detail = PG_EXCEPTION_DETAIL,
        _hint = PG_EXCEPTION_HINT;
```


Příkazy PL/pgSQL

Rekapitulace

Pozor

- na kolizi názvů proměnných a názvů objektů nebo atributů,
- na neplatný prováděcí plán (DROP object nebo EXECUTE),
- ISAM přístup (co lze provést pomocí SQL příkazem, řeště SQL příkazem),
- minimalizujte počet instrukcí (co instrukce to SQL příkaz),
- vyhýbejte se dynamickému SQL,
- zbytečně nepoužívejte dočasné tabulky.

Úloha č. 16.

V PL/pgSQL navrhněte funkci *next_day*. Tato funkce vrací první následující zadaný den v týdnu po zadaném datumu. Např. první pondělí po 22.11.2006 atd.

Příkazy PL/pgSQL

Cvičení

Úloha č. 17.

Hromadné změny dat lze urychlit dočasným odstraněním indexů postižených tabulek. Navrhněte dvě funkce, které tuto činost zautomatizují. Parametrem první tabulky bude pole s názvy tabulek u kterých chceme odstranit indexy a výstupem bude tabulka s informacemi potřebnými k znovuvytvoření odstraněných indexů.
Druhá procedura, jejíž vstupem bude název tabulky s informacemi o indexech, vytvoří potřebné indexy. Odstraňujte pouze NOT UNIQUE indexy.

Nápověda

Údaje o indexech naleznete v systémové tabulce pg_indexes.

Triggery v PL/pgSQL

Popis I.

Popis

- trigger zajišťuje provedení námi zvolených operací při změně obsahu tabulky (INSERT, UPDATE, DELETE),
- kód lze navázat na změnu řádku nebo provedení příkazu,
- pokud se procedura spouští při změně obsahu řádku, pak jsou k dispozici proměnné NEW a OLD,
- pokud procedura skončí výjimkou neprovede se žádná změna obsahu,
- pokud procedura vrací NULL, neprovede se změna aktuálního řádku (skip)
- do tabulky se zapíše obsah případně modifikované vrácené proměnné NEW,
- nová data jsou v tabulce viditelná až v AFTER triggeru.

Použití

- zajištění integrity databáze,
- složitější kontroly (které nelze provést v CHECK omezení),
- logování změn,
- dopočet hodnot (materializované pohledy).

Triggery v PL/pgSQL

Popis II.

```
CREATE TRIGGER name  BEFORE | AFTER   event [ OR ... ]
    ON table [ FOR [ EACH ]  ROW | STATEMENT  ]
    EXECUTE PROCEDURE funcname ( arguments )
```

```
CREATE OR REPLACE FUNCTION simplytrg() RETURNS TRIGGER AS $$
BEGIN
  NEW.c := NEW.a + NEW.b;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER simply BEFORE INSERT OR UPDATE
  ON foodata FOR EACH ROW
  EXECUTE PROCEDURE simplytrg();
```


Triggery v PL/pgSQL

Popis III.

Předdefinované proměnné v obslužných procedurách PL/pgSQL

NEW	proměná typu RECORD viditelná v případě INSERT, UPDATE,
OLD	proměná typu RECORD viditelná v případě UPDATE, DELETE,
TG_WHEN	textová proměnná obsahující BEFORE nebo AFTER,
TG_OP	textová proměnná obsahující INSERT, UPDATE nebo DELETE,
TG_TABLE_NAME	obsahuje název tabulky na které byl spuštěn trigger,
TG_NAME	obsahuje název triggeru.

Pozn.

Za normálních okolností triggery nemají znatelný vliv na rychlost operací INSERT, UPDATE, a DELETE (přesunutím logiky z aplikace do db serveru nesnižuji zatížení serveru). Zcela zásadně však mohou zpomalit provádění hromadných operací nad tabulkami. Příkaz *TRUNCATE*) neaktivuje trigger. Triggery dočasně deaktivovat příkazem *ALTER TABLE DISABLE TRIGGER*.

Triggery v PL/pgSQL

Cvičení

Úloha č. 18.

Do tabulky audit zapište všechny změny v tabulce data.

Úloha č. 19.

V tabulce current_rows udržujte aktuální počet řádek tabulek data_01 a data_02.

Úloha č. 20.

Napište trigger, který nedovolí uživateli modifikovat po 31.1 data z předchozího roku - hlaseni(vlozeno, komentar).

Rekurze

Generování výsledku rekurzivním voláním

Pozn.

Výsledkem běhu každé SRF funkce je nezávislá materializovaná tabulka. V případě, že SRF funkce je volána z další SRF funkce a ta tuto tabulku explicitne nepřebírá, dochází k zániku této tabulky.
Až na vyjímky se SRF funkce volají přímo. Vyjímkou je rekurze.

```
CREATE OR REPLACE FUNCTION ls(int[], int)
RETURNS SETOF rec_src AS $$
  SELECT id, parent, repeat(' ', $2) || v
     FROM rec_src WHERE parent = ANY($1)
  UNION ALL
  SELECT * FROM ls(empty2null(ARRAY(SELECT id
                                      FROM rec_src
                                     WHERE parent = ANY($1))),
                   $2 + 1);
$$ LANGUAGE SQL STRICT;
```

Úloha č. 21.

Ve výsledné tabulce odsaďte hodnoty sloupce v podle úrovně (hloubky) rekurze.

```
 id | parent |   v
----+--------+--------
  1 |      0 | root
  2 |      1 | rodic
  3 |      1 | rodic
  4 |      2 | dite
  5 |      2 | dite
  6 |      3 | dite
  7 |      6 | vnouce
(7 rows)
```

```
 id | parent |    v
----+--------+----------
  1 |      0 | root
  2 |      1 |  rodic
  3 |      1 |  rodic
  4 |      2 |   dite
  5 |      2 |   dite
  6 |      3 |   dite
  7 |      6 |    vnouce
(7 rows)
```


CTE

Pokud používáte verzi 8.4 a vyšší, je vhodné použít Comṁon Table Expressions:

```
pavel=# WITH RECURSIVE x AS (
             SELECT *, 0 AS level FROM fam WHERE parent = 0
             UNION ALL SELECT fam.*, level + 1
                           FROM fam JOIN x ON x.id=fam.parent)
          SELECT id, parent, repeat(' ', level) || v FROM x ORDER BY id;
 id | parent | ?column?
----+--------+----------
  1 |      0 | root
  2 |      1 |  rodic
  3 |      1 |  rodic
  4 |      2 |   dite
  5 |      2 |   dite
  6 |      3 |   dite
  7 |      6 |    vnouce
 (7 rows)
```

Tipy

Několik rad, jak psát v PL/pgSQL

- používejte prefixy proměnných,
- používejte kvalifikované atributy ve všech SQL příkazech (tabulka.sloupec),
- používejte odvozené typy %TYPE
- snažte se používat nativní SQL všude, kde to je možné a rozumné,
- nepište redundantní kód, snažte se vyhnout redundantním SQL příkazům,
- pozor na volání funkcí obsahujících SQL uvnitř cyklu,
- pro texty vyjímek dodržujte předem dohodnutou notaci,
- výsledek dotazy ověřujte vždy prostřednictvím proměnné FOUND,
- nespoléhejte se na aut. konverze u typů date a timestamp (používejte to_char a to_date),
- v triggeru neopravujte data,
- nepiště si zbytečně vlastní funkce,
- pokud vaše funkce nemá vedlejší efekty, označte ji jako IMMUTABLE,
- chyby signalizujte výjimkami, nikoliv návratovým kódem.

Tipy

Několik rad, jak nepsat v PL/pgSQL

- nepoužívejte C, Fortran styl - používejte co nejvíc vestavěného aparátu - COALESCE, ORDER BY,
- při přetypování na string používejte funkce umožňující explicitní určení formátu,
- dynamické SQL vždy zabezpečte - quote_literal, quote_ident, klauzule USING,
- PLpgSQL používejte tam, kde má smysl - pro jednořádkové funkce preferujte jazyk SQL,
- nepoužívejte SELECT INTO tam, kde můžete použít přiřazovací příkaz.

Tipy

Nepoužívat C style

```
-- špatně
IF x1 IS NOT NULL THEN
  s := s || x1 || ',';
ELSE
  s := s || 'NULL,';
END IF;
IF x2 IS NOT NULL THEN
  s := s || x2 || ','
ELSE
  s := s || 'NULL,';
END IF;
...

-- správně
s := coalesce(x1 || ',', 'NULL,') ||
  coalesce(x2 || ',','NULL,') || ...
```


Tipy

Vyhýbejte se implicitnímu přetypování datových položek

```
--špatně
m = substring(current_date::text FROM 6 FOR 2)::int;

-- lépe
m = substring(to_char(current_date, 'YYYY-MM-DD')
              FROM 6 FOR 2):: int;

--správně
m = EXTRACT(month FROM current_date);
```

Extenze

Popis

Účel

- umožnit snadnou instalaci a aktualizaci doplňků,
- umožnit evidenci verzí.

Implementace

- instalační skript - *foo.sql* - obsahuje vlastní deklarace,
- migrační skript - *foo–1.0–1.1.sql* - obsahuje deklarace změn z verze n na verzi m,
- migrační skript unpacked - *foo–unpacked–1.0.sql* - sdružuje instalované zatím nezávislé funkce do doplňků,
- řídící soubor - *foo.control* - metadata (default_version, requires, schema, ...).

Pozn.

Výše uvedené soubory se zkopírují do adrsáře *pgbin/share/extension*.

Transakce

Popis

Účel

- zajistit konzistenci dat,
- umužnit efektivní výceuživatelský přístup (serializace transakcí).

Prostředky

- snímky (podporováno multigenerační architekturou),
- jsou definována a aplikována pravidla pro řešení kolizí.

Transakce

Režimy izolace

Vytváření snímků

SERIALIZABLE snímek se vytvoří při zahájení prvního dotazu v transakci,
READ COMMITED snímek se vytváří při zahájení každého dotazu v transakci.

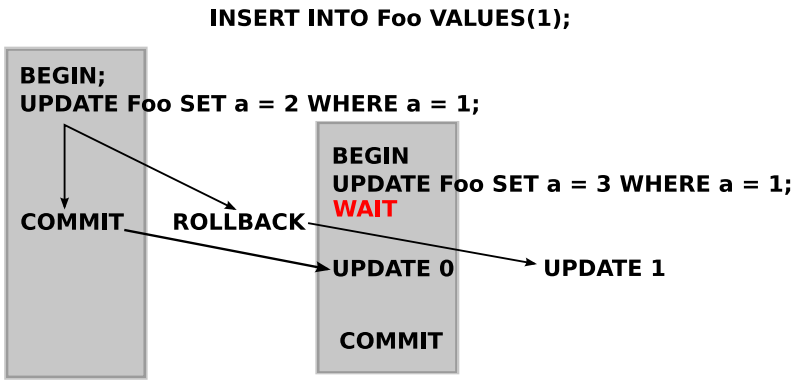
Vždy platí, že nepotvrzená data jsou viditelná pouze v transakci, která je za ně zodpovědná. V případě SERIALIZABLE není šance detekovat změny v databázi způsobené ostatními uživateli. V READ COMMITED vidíme postupně potvrzovaná data. Pokud nedojde je kolizi zápisů, nedochází k čekání.

Pozor na déle trvající transakce

Dokud je záznam viditelný z transakce, považuje se za živý a tudíž je příkaz VACUUM neodstraní. Doba zpracování příkazů ROLLBACK a COMMIT je vždy konstantní.

Transakce

Schéma transakce typu READ COMMITED



Transakce
Cvičení

Úloha č. 22.

V jakém režimu je bezpečné provedení analýzy rozvahy?

```
BEGIN;
SELECT SUM(castka) FROM Aktiva;
SELECT SUM(castka) FROM Pasiva;
COMMIT;
```


Transakce
Pravidla pro řešení kolizí

V případě kolize

Pokud se transakce pokouší modifikovat záznam, který je již modifikován ale nepotvrzen jinou transakcí:

1. čeká se na commit nebo rollback první transakce,
2. pokud první transakce skončila rollbackem, normálně se pokračuje,
3. v případě, že první transakce data potvrdila:
   - v úrovni SERIALISABLE se ukončí chybou ”can’t serialize error”,
   - v úrovni READ COMMITED se otestuje, zda potvrzená hodnota vyhovuje kritériím příkazu. Pokud ano, normálně se pokračuje.

Transakce

Pozor na dvojici SELECT, UPDATE

```
-- spatne
BEGIN;
SELECT hits FROM webpages WHERE url = '...';
$newval = $hits + 1;
UPDATE webpages SET hits = $newval WHERE ursl = '...';

-- dobre
BEGIN;
SELECT hits FROM webpages WHERE url = '...' FOR UPDATE;
$newval = $hits + 1;
UPDATE webpages SET hits = $newval WHERE ursl = '...';
```


Transakce

Vzor pro klienta v SERIALIZABLE režimu

```
loop
  BEGIN
  SELECT hits FROM webpages WHERE url = '...';
  $newval = $hits + 1;
  UPDATE webpages SET hits = $newval WHERE url = '...';
  if (no error)
    break loop;
  else
    ROLLBACK;
 end loop;
 COMMIT;
```

Vlastní operaci je třeba uzavřít do cyklu, jelikož je tu možnost chyby ”serialization error”, a případně znovu opakovat.

Security definer funkce

Statická změna identity uživatele

Atribut SECURITY DEFINER

Označení funkce atributem *SECURITY DEFINER* mění identitu uživatele během běhu funkce na identitu vlastníka funkce.

Pozor na zabezpečení

Obyčejně se toto chování používá na dočasné povýšení práv. Proto je nutné, aby funkce, která má atribut SECURITY DEFINER byla maximálně bezpečná. Základem je nastavit sys, proměnnou *search_path* a nepoužívat dynamcké SQL.

Úloha č. 23.

Vytvořte proceduru, která ověří identitu uživatele. K tabulce md5 hashů má přístup pouze privilegovaný uživatel.

Doporuční pro datový návrh I.

Několik tipů jak správně navrhovat datové schéma

- v názvu tabulek a sloupců nepoužívejte klíčová slova,
- snažte se, aby položky tvořící primární klíč, byly deklarovány na začátku tabulky,
- pokud nemáte k dispozici data, použijte hodnotu NULL. Nepoužívejte 0 nebo prázdný řetězec,
- vyhněte se širokým tabulkám, zpomalují sekvenční čtení a mají výrazně vyšší paměťové nároky (dop. limit 12. sloupců),
- nepoužívejte prefixy určující objekt (tabulka, sekvence, pohled) např. t_processes,
- nepoužívejte velbloudí (Camel) způsob zápisu identifikátorů (např. isoCode),
- názvy sloupečků by měli být výstižné, rozumně dlouhé, zapsané malými písmeny,
- tabulky nazývejte v množném čísle a zapisujte s prvním velkým a ostatními malými písmeny,
- klíčova slova zapisujte vždy malými nebo vždy velkými písmeny,

Doporuční pro datový návrh II.

Několik tipů jak správně navrhovat datové schéma

- používejte předem dohodnutou sadu sufixů (_id, _code, _date, _nbr, _name, _size, _tot, _cat, _class, _type).
- nepoužívejte pref(su)fixy PK, FK,
- název tabulky neopakujte v názvu sloupce,
- nepoužívejte speciální znaky (vyjímka znak '_'), diakritiku v identifikátorech tabulek a sloupců.
- za čárkou nebo středníkem vždy použijte mezeru nebo nový řádek, nezačínejte řádek čárkou,
- při zápisu SQL příkazu použijte odsazení,
- hledejte přirozené kódy,
- pokud to lze, používejte standardizované konstrukce a funkce.

Doporuční pro datový návrh III.

Několik tipů jak správně navrhovat datové schéma

- držte pohromadě atributy, které spolu věcně, chronologicky souvisí,
- je chybou rozdělit data do tabulek: FemalePersonel, MalePersonel, prodej2001, prodej2002, atd,
- je chybou když datum máte ve třech sloupcích (den, měsíc, rok), tj. používejte odpovídající datové typy,
- je chybou, když data rozdělíte zbytečně do dvou řádků, např. zahájení události, konec události,
- kódy držte v databázi v speciálních tabulkách. Je chybou používat jednu univerzální tabulku pro všechny třídy kódů,
- udržujte datové schéma přehledné a čitelné. Nepoužívané tabulky okamžitě odstraňte, nepřidávejte zbytečné tabulky.

Posouzení vlivu SQL a PL/pgSQL příkazů na rychlost

Kde a jak optimalizovat

Rychlost jednotlivých příkazů

Operace	Rychlost
10K prázdných iterací cyklu FOR	1.7ms
10K náhrada cyklu FOR cyklem WHILE	22ms
10K jednoduchých operací (eval simple)	23ms
10K netriviálních operací	150ms
10K čtení relace (z cache)	260ms
10K itarací nad prázdným blokem s EXCEPTION	100ms

Posouzení vlivu SQL a PL/pgSQL příkazů na rychlost

Pozor na jednoduché operace a netriviální operace

```
-- 2x jednoduché operace 10K iterací 22ms
a := 10; b := 10;

-- 1x netriviální operace 10K iterací 150ms
SELECT INTO a, b 10, 10;
```

Řešení příkladů

Řešení úlohy č. 1.

Funkce *last_day*

```
CREATE OR REPLACE FUNCTION last_day(date)
RETURNS date AS $$
  SELECT date_trunc('month', $1 + interval '1month')::date - 1;
$$ LANGUAGE sql;

postgres=# select last_day('2008-06-29');
  last_day
------------
 2008-06-30
(1 row)
```

Řešení úlohy č. 2.

Funkce *rvrs*

```
CREATE OR REPLACE FUNCTION rvrs(text)
RETURNS text AS $$
  SELECT array_to_string(
            ARRAY
                 (SELECT substring($$1 FROM idx.i FOR 1)
                    FROM generate_series(length($1),1,-1) idx(i)
                 ),
            '')
$$ LANGUAGE sql;
CREATE FUNCTION
```


Řešení úlohy č. 3.

Funkce *rvrs*

```
CREATE OR REPLACE FUNCTION rvrsr(text)
RETURNS text AS $$
  SELECT CASE length($1)
              WHEN 0 THEN ''
              WHEN 1 THEN $1
              WHEN 2 THEN right($1,1) || left($1,1)
              ELSE rvrsr(right($1, (trunc(length($1)/2.0))::int))
                     || rvrsr(left($1, (round(length($1)/2.0))::int)) END
$$ LANGUAGE sql;
```

Řešení úlohy č. 4. a úlohy č. 5.

Funkce *distinct_array* a *intersect_array*

```
CREATE OR REPLACE FUNCTION distinct_array(anyarray)
RETURNS anyarray AS $$
  SELECT ARRAY(
                SELECT DISTINCT *
                   FROM unnest($1))
$$ LANGUAGE sql;

CREATE OR REPLACE FUNCTION intersect_array(anyarray, anyarray)
RETURNS anyarray AS $$
  SELECT ARRAY(
                SELECT *
                   FROM unnest($1)
                INTERSECT
                SELECT *
                   FROM unnest($2))
$$ LANGUAGE sql;
```


Řešení úlohy č. 6.

Funkce *myleast*

```
CREATE FUNCTION myleast(VARIADIC anyarray)
RETURNS anyelement AS $$
  SELECT min(v)
     FROM unnest($1) g(v)
$$ LANGUAGE sql;
```

Řešení úlohy č. 7.

Vlastní operátor *Operator* ||| *- průnik polí*

```
CREATE OPERATOR ||| (
  PROCEDURE = intersect_array,
  LEFTARG=anyarray,
  RIGHTARG=anyarray
);
```


Řešení úlohy č. 8.

Funkce *sort*

```
CREATE OR REPLACE FUNCTION sort(vals integer[])
RETURNS integer[] AS $$
DECLARE
  aux integer[] = vals;
  sorted boolean = false;
  a integer;
BEGIN
  WHILE NOT sorted LOOP
    sorted := true;
    FOR i IN array_lower(aux, 1) .. array_upper(aux,1) - 1 LOOP
      IF aux[i] > aux[i+1] THEN
        a := aux[i];
        aux[i] := aux[i+1]; aux[i+1] := a;
        sorted := false;
      END IF;
    END LOOP;
  END LOOP;
  RETURN aux;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

Řešení úlohy č. 9.

Funkce *rvrs*

```
CREATE OR REPLACE FUNCTION rvrsp(str text)
RETURNS text AS $$
DECLARE accu text = '';
BEGIN
  FOR i IN REVERSE length(str) .. 1 LOOP
    accu := accu || substring(str FROM i FOR 1);
  END LOOP;
  RETURN accu;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```


Řešení úlohy č. 10.

Plnění již existující tabulky

```
CREATE TABLE sintab(v float PRIMARY KEY, sinv float);

CREATE OR REPLACE FUNCTION set_sintab()
RETURNS void AS $$
DECLARE _i float = 0;
  WHILE _i <= 2*3.14 LOOP
    INSERT INTO sintab(v, sinv) VALUES(_i, sin(_i));
  END LOOP;
$$ LANGUAGE plpgsql VOLATILE;
```

Řešení úlohy č. 11.

Přistání na měsíci

```
CREATE TABLE data(id serial PRIMARY KEY,v float, r float, m float, h float, p float);

CREATE OR REPLACE FUNCTION pristani(_id integer, _i float)
RETURNS float AS $$
DECLARE
  _d data%ROWTYPE; a float;
BEGIN
  SELECT INTO _d * FROM data WHERE id = _id;
  a := 1.62 - (_i * _d.m / _d.h);
  _d.v := _d.v - _d.r - (a / 2.0);
  _d.r := _d.r + a;
  _d.p := _d.p - (_i * _d.m / 240.0);
  IF _d.p < 0 THEN _d.m := 0; _d.p := 0; END IF;
  IF _d.v < 0 THEN
    RAISE NOTICE 'Pristal jsi, rychlost %[m/s], palivo %[kg]', _d.r, _d.p;
  ELSE
    RAISE NOTICE 'Rychlost %[m/s], Palivo %[kg]', _d.r, _d.p;
  END IF;
  UPDATE data SET v = _d.v, r = _d.r, m = _d.m, h = _d.h, p = _d.p WHERE id = _id;
  RETURN _d.v;
END;
$$ LANGUAGE plpgsql VOLATILE;
```


Řešení úlohy č. 12.

Funkce array_info

```
CREATE OR REPLACE FUNCTION array_info(
  IN v integer[],
  OUT min integer, OUT max integer, OUT med double precision)
AS $$
DECLARE
  sorted CONSTANT integer[] = sort(v);
  l CONSTANT integer = array_lower(v,1);
  u CONSTANT integer = array_upper(v,1);
  h CONSTANT integer = u / 2;
BEGIN
  min := sorted[l];  max := sorted[u];
  med := CASE u % 2 WHEN 0 THEN (sorted[h] + sorted[u-h+1])/2.0
                    WHEN 1 THEN sorted[h+1] END;
  RETURN;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

SELECT * FROM array_info(ARRAY[6,5,4,3,2,1]);
SELECT array_info(ARRAY[6,5,4,3,2,1]);
```

Řešení úlohy č. 13.

Generování reporu s mezisoučty

```
CREATE OR REPLACE FUNCTION report()
RETURNS SETOF pokladna AS $$
DECLARE
  ra pokladna; raa pokladna; r pokladna;
BEGIN
  raa := ROW(NULL, NULL, 0);
  FOR r IN SELECT * FROM pokladna ORDER BY kategorie
  LOOP
    IF ra.kategorie = r.kategorie THEN
      ra.cena := ra.cena + r.cena;
    ELSE
      IF NOT ra IS NULL THEN RETURN NEXT ra;
        raa.cena := raa.cena + ra.cena;
      END IF;
      ra := r; ra.zbozi := NULL;
    END IF;
    RETURN NEXT r;
  END LOOP;
  IF NOT ra IS NULL THEN
    raa.cena := raa.cena + ra.cena;
    RETURN NEXT ra; RETURN NEXT raa;
  END IF;
  RETURN;
END; $$ LANGUAGE plpgsql;
```


Řešení úlohy č. 14.

Funkce series

```
CREATE OR REPLACE FUNCTION series(min integer, max integer, step integer)
RETURNS SETOF integer AS $$
BEGIN
  FOR i IN min .. max BY step LOOP
    RETURN NEXT i;
  END LOOP;
  RETURN;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION series(min integer, max integer)
RETURNS SETOF integer AS $$
  SELECT * FROM series($1,$2,1)
$$ LANGUAGE SQL IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION series(max integer)
RETURNS SETOF integer AS $$
  SELECT * FROM series(1,$1,1)
$$ LANGUAGE SQL IMMUTABLE STRICT;

SELECT * FROM series(5);
```

Řešení úlohy č. 15.

Simulace přistání na Měsíci. inicializace modelu

```
CREATE OR REPLACE FUNCTION reset_model(
                              IN vyska float,
                              IN rychlost float,
                              IN max_tah float,
                              IN hmotnost float,
                              IN palivo float)
RETURNS VOID AS $$
BEGIN
  BEGIN
    DELETE FROM data;
  EXCEPTION
    WHEN undefined_table THEN
      CREATE TEMP TABLE data(v float, r float, m float, h float, p float);
  END;
  INSERT INTO data(v, r, m, h, p)
    VALUES(vyska, rychlost, max_tah, hmotnost, palivo);
END;
$$ LANGUAGE plpgsql VOLATILE;
```


Řešení úlohy č. 15.

Simulace přistání na Měsíci. simulace 1. část

```
CREATE OR REPLACE FUNCTION pristani(
                              IN _i float,
                              OUT vyska float, OUT rychlost float, OUT palivo float)
RETURNS SETOF RECORD AS $$
DECLARE
  _m float; _h float; _a float;
BEGIN
    SELECT INTO vyska, rychlost, _m, _h, palivo
                v,  r,  m,  h,  p
      FROM data LIMIT 1;
  WHILE vyska > 0.0 LOOP
    _a := 1.62 - (_i * _m / _h);
    vyska := vyska - rychlost - (_a / 2.0);
    rychlost := rychlost + _a;
    palivo := palivo - (_i * _m / 240.0);
    IF palivo < 0.0 THEN _m := 0; palivo := 0; END IF;
    RETURN NEXT;
    EXIT WHEN palivo > 0.0;
  END LOOP;
```

Řešení úlohy č. 15.

Simulace přistání na Měsíci. simulace 2. část

```
  UPDATE data
    SET v = vyska, r = rychlost, m = _m, h = _h, p = palivo;

  IF vyska  < 0.0 THEN
    IF rychlost > 3.0 THEN
      RAISE WARNING 'Havaroval jsi v rychlosti %', rychlost;
    ELSE
      RAISE NOTICE 'Pristal jsi. Zbylo ti % kg paliva', palivo;
    END IF;
  END IF;
  RETURN;
END;
$$ LANGUAGE plpgsql VOLATILE STRICT;
```


Řešení úlohy č. 16.

Funkce next_day

```
CREATE OR REPLACE FUNCTION error(msg text) RETURNS int AS $$
BEGIN
  RAISE EXCEPTION '%', msg;
END; $$ LANGUAGE plpgsql IMMUTABLE;

CREATE OR REPLACE FUNCTION next_day(day varchar, d date) RETURNS date AS $$
DECLARE off integer;
BEGIN
off := CASE SUBSTRING(lower(day) FROM 1 FOR 3)
         WHEN 'sun' THEN 0
         WHEN 'mon' THEN 1
         WHEN 'tue' THEN 2
         WHEN 'wed' THEN 3
         WHEN 'thu' THEN 4
         WHEN 'fri' THEN 5
         WHEN 'sat' THEN 6
         ELSE error('Wrong identifier for day '||day)
              END - EXTRACT(dow FROM d);
  RETURN CASE
              WHEN off > 0 THEN $2 + off
              ELSE $2 + off + 7 END;
END;
$$ LANGUAGE plpgsql;
```

Řešení úlohy č. 17.

Funkce drop_indexes, prvotní návrh

```
CREATE OR REPLACE FUNCTION drop_indexes(_tables varchar[])
RETURNS SETOF varchar AS $$
DECLARE
  _r RECORD; _tablename VARCHAR;
BEGIN
  FOR i IN 1 .. array_upper(_tables, 1) LOOP
    IF position('.' in _tables[i]) = 0 THEN
      _tablename := 'public.' || _tables[i];
    ELSE
      _tablename := _tables[i];
    END IF;
    FOR _r IN SELECT * FROM pg_indexes p
              WHERE p.schemaname || '.' || p.tablename = _tablename
    LOOP
      IF _r.indexdef LIKE 'CREATE INDEX %' THEN
        RETURN NEXT _r.indexdef;
        EXECUTE 'DROP INDEX ' || _r.indexname;
      END IF;
    END LOOP;
  END LOOP;
  RETURN;
END;
$$ LANGUAGE plpgsql;
```


Řešení úlohy č. 17.

Funkce drop_indexes, volání funkce

```
DROP TABLE IF EXISTS back_index ;

SELECT * INTO back_index
  FROM drop_indexes(ARRAY['prg']);

SELECT * FROM back_index;
```

Řešení úlohy č. 17.

Funkce drop_indexes, optimalizovaná varianta

```
CREATE OR REPLACE FUNCTION drop_indexes(_tables varchar[])
RETURNS SETOF varchar AS $$
DECLARE
  _r RECORD;
  _tablenames VARCHAR[] = _tables;
BEGIN
  FOR i IN 1 .. array_upper(_tables,1)
  LOOP
    IF position('.' in _tables[i]) = 0 THEN
      _tablenames[i] = 'public.' || _tables[i];
    END IF;
  END LOOP;
  FOR _r IN SELECT * FROM pg_indexes p
              WHERE p.schemaname || '.' || p.tablename = ANY(_tablenames)
                    AND p.indexdef LIKE 'CREATE INDEX %'
  LOOP
    RETURN NEXT _r.indexdef;
    EXECUTE 'DROP INDEX ' || _r.indexname;
  END LOOP;
  RETURN;
END;
$$ LANGUAGE plpgsql;
```


Řešení úlohy č. 17.

Funkce create_indexes, obnovení indexů

```
CREATE OR REPLACE FUNCTION create_indexes(_tablename varchar)
RETURNS void AS $$
DECLARE _src varchar;
BEGIN
  FOR _src IN
    EXECUTE 'SELECT drop_indexes FROM ' || _tablename
  LOOP
    EXECUTE  _src;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Řešení úlohy č. 18.

Audit tabulky 1. část

```
CREATE TABLE audit(
  inserted timestamp,
  username varchar(32),
  op varchar(10),
  new_val varchar,
  old_val varchar
);

CREATE TABLE data(
  id SERIAL PRIMARY KEY,
  val varchar
);

-- žádný další trigger nezmění obsah
CREATE TRIGGER audit_trg_data AFTER INSERT OR UPDATE OR DELETE
  ON data FOR EACH ROW EXECUTE PROCEDURE audit_trg_hdl();
```


Řešení úlohy č. 18.

Audit tabulky 2. část

```
CREATE OR REPLACE FUNCTION audit_trg_hdl()
RETURNS TRIGGER AS $$
BEGIN
  IF TG_OP = 'INSERT' THEN
    INSERT INTO audit
      VALUES(CURRENT_TIMESTAMP,
             SESSION_USER, TG_OP, NEW.val, NULL);
  ELSEIF TG_OP = 'UPDATE' THEN
    INSERT INTO audit
      VALUES(CURRENT_TIMESTAMP,
             SESSION_USER, TG_OP, NEW.val, OLD.val);
  ELSEIF TG_OP = 'DELETE' THEN
    INSERT INTO audit
      VALUES(CURRENT_TIMESTAMP,
             SESSION_USER, TG_OP, NULL, OLD.val);
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Řešení úlohy č. 19.

Udržba obsahu tabulky s aktuálním počtem řádků, 1. část

```
CREATE TABLE data_01(a varchar);
CREATE TABLE data_02(a varchar);

CREATE TABLE current_rows(
  tblname varchar(32) PRIMARY KEY,
  rc integer
);

CREATE TRIGGER cr_trg_data_01 AFTER INSERT OR DELETE
  ON data_01 FOR EACH ROW EXECUTE PROCEDURE cr_trg_hdl();

CREATE TRIGGER cr_trg_data_02 AFTER INSERT OR DELETE
  ON data_02 FOR EACH ROW EXECUTE PROCEDURE cr_trg_hdl();
```


Řešení úlohy č. 19.

Udržba obsahu tabulky s aktuálním počtem řádků, 2. část

```
CREATE OR REPLACE FUNCTION cr_trg_hdl() RETURNS TRIGGER AS $$
DECLARE _rc integer;
BEGIN
  SELECT INTO _rc rc FROM current_rows
   WHERE tblname = TG_TABLE_NAME FOR UPDATE;
  IF NOT FOUND THEN
    LOCK TABLE current_rows IN SHARE MODE;
    SELECT INTO _rc rc FROM current_rows
     WHERE tblname = TG_TABLE_NAME FOR UPDATE;
    IF NOT FOUND THEN
      EXECUTE 'SELECT COUNT(*) FROM '|| TG_TABLE_NAME INTO _rc;
      INSERT INTO current_rows VALUES(TG_TABLE_NAME, _rc);
      RETURN NULL;
    END IF;
  END IF;
  _rc := _rc + CASE TG_OP WHEN 'DELETE' THEN -1
                          WHEN 'INSERT' THEN  1 END;
  UPDATE current_rows SET rc = _rc
   WHERE tblname = TG_TABLE_NAME;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Řešení úlohy č. 20.

Zabezpečení tabulky proti úpravám po uzávěrce

```
CREATE TABLE hlaseni(
  id serial PRIMARY KEY,
  vlozeno date,
  komentar text);

CREATE OR REPLACE FUNCTION cd_trg_hdl() RETURNS TRIGGER AS $$
DECLARE _rc integer;
BEGIN
  IF EXTRACT(year FROM old.vlozeno) < EXTRACT(year FROM current_date) THEN
    /* pokud je leden, tak lze editovat rok stare zapisy */
    IF NOT EXTRACT(month FROM current_date) = 1 OR
           EXTRACT_YEAR(year FROM old.vlozeno) + 1 <> EXTRACT(year FROM current_date)
    THEN
      /* uzivateli postgres povolime editaci, ostatnim nikoliv */
      IF session_user <> 'postgres' THEN
        RAISE EXCEPTION 'Nelze modifovat zapis po uzaverce';
      END IF;
    END IF;
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```


Řešení úlohy č. 21.

Rekurze

```
CREATE OR REPLACE FUNCTION lspl(root integer, level integer)
RETURNS SETOF rec_src AS $$
DECLARE r rec_src; rr rec_src;
BEGIN
  FOR r IN SELECT *
              FROM rec_src
             WHERE parent = root
  LOOP
    r.v := repeat(' ', level) || r.v;
    RETURN NEXT r;
    -- rekurzivni volani potomku
    FOR rr IN SELECT *
                FROM lspl(r.id, level + 1)
    LOOP
      -- vysledek musim pripojit k akt. vystupu
      RETURN NEXT rr;
    END LOOP;
  END LOOP;
  RETURN;
END;
```

Řešení úlohy č. 21.

Rekurze - použití RETURN QUERY

```
CREATE OR REPLACE FUNCTION lspl(root integer, level integer)
RETURNS SETOF rec_src AS $$
DECLARE r rec_src; rr rec_src;
BEGIN
  FOR r IN SELECT *
              FROM rec_src
             WHERE parent = root
  LOOP
    r.v := repeat(' ', level) || r.v;
    RETURN NEXT r;
    RETURN QUERY SELECT * FROM lspl(r.id, level + 1);
  END LOOP;
  RETURN;
END;
```


Transakce

Řešení úlohy č. 22.

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Řešení úlohy č. 23.

SECURITY DEFINER funkce

```
CREATE OR REPLACE FUNCTION valid_user(_username varchar,
                                      _passwd varchar)
RETURNS bool AS
BEGIN
  RETURN EXISTS(SELECT *
                  FROM users u
                 WHERE u.username = _username
                   AND u.passwd = md5(_passwd));
END;
$$ LANGUAGE plpgsql
SECURITY DEFINER
SET search_path = '';
```


PostgreSQL 9.2 efektivně

Programování uložených procedur

Pavel Stěhule

http://www.postgres.cz

21. 1. 2013