PL/pgSQL

Pavel Stěhule

Ukážka 1

```
CREATE OR REPLACE FUNCTION german_date(d date)
RETURNS text AS $$
  SELECT to_char($1, 'DD.MM.YYYY');
$$ LANGUAGE sql;

postgres=# SELECT german_date(CURRENT_DATE);
 german_date
-------------
 02.02.2017
(1 row)
```

Ukážka 2

```
CREATE OR REPLACE FUNCTION gs(rs int)
RETURNS SETOF int AS $$
BEGIN
  FOR i IN 1..rs LOOP
    RETURN NEXT i;
  END LOOP;
  RETURN;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM gs(100);
```

Uložené procedury

- Kód, který je spouštěn db serverem
- Výsledek je okamžitě k dispozici v db
- V kódu lze jednoduše manipulovat s daty
- Kde jsou data, tam také běží kód

Pro

- Bezpečnost - nelze obejít (vynechat), navrženo pro zajištění maximální bezpečnosti.
- Rychlost - minimalizuje přenos po síti, konverze (V Postgresu běží uvnitř SQL engine)
- Jednoduchost - minimalistický jazyk bez I/O
- Čitelnost a integrace SQL - SQL není řetězec
- Podporuje dekompozici na datové operace a prezentaci dat

Ukážka 3

```
CREATE OR REPLACE FUNCTION foo(x int)
RETURNS int int AS $$
BEGIN
  IF x < 0 THEN x := -x; END IF;
  RETURN (SELECT *
            FROM T
           WHERE T.x = foo.x);
END;
$$ LANGUAGE plpgsql;
```

Proti

- Jen minimálně přenositelné - SQL/PSM, PL/SQL, PLpgSQL, T-SQL
- Umožňuje nepoučeným začátečníkům spadnout do několika pastí
  - fascinace triggery - úkolem triggerů jsou primárně kontroly a logování. Triggery by neměly obsahovat běžnou logiku a suplovat funkce
  - fascinace funkcemi - funkce by neměly suplovat pohledy - každá funkce je pro optimalizátor izolovaná a zmenšuje prostor pro optimalizaci.

Použití - skalární fce

- Jednoduché skalární funkce - rozšiřuje možnosti databáze a umožňuje transformaci dat (díky inliningu není taková SQL funkce pro optimalizátor překážkou).

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

Použití - manipulace s daty

- Vytváření API pro datové operace
- Odstínění aplikace od způsobu uložení
- Vlastní kontroly + vlastní chybové hlášení
- Zajištění konzistence dat
  - využití transakcí - buďto kód doběhne a zaručeně modifikuje data nebo nedoběhne a data se nezmění

Použití - manipulace s daty

```
CREATE OR REPLACE FUNCTION new_usr(name text,
                                   surname text)
RETURNS int AS $$
DECLARE r int;
BEGIN
  IF EXISTS(SELECT * FROM users WHERE ..)
  THEN
    RAISE EXCEPTION 'name, surname is not ..';
  END IF;
  INSERT INTO users VALUES(name, surname)
    RETURNING id INT r;
  RETURN r;
END;
$$ LANGUAGE plpgsql;
```

Základní struktury o

- Funkce

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

SELECT hello('World');
```

Základní struktury 1

- Blok

```
DECLARE
var1 integer DEFAULT 10;
var2 integer; -- DEFAULT NULL
BEGIN
  var1 := var1 * 100;
  var2 := var1 + 100;
END;
```

Základní struktury 2

- Podmínka

```
IF var1 % 100 = 0 THEN
  RAISE NOTICE 'var1 = %', var1;
END IF;
```

Základní struktury 3

- Cyklus

```
FOR i IN 1..10 LOOP
RAISE NOTICE 'i=%', i;
END LOOP;

DECLARE r RECORD;
BEGIN
  FOR r IN SELECT * FROM pg_proc LOOP
    RAISE NOTICE 'r=%', r;
  END LOOP;
END;
```

Funkce a transakce

- Na rozdíl od jiných systémů se transakce v uložených procedurách na PostgreSQL prakticky neřeší.
- V PostgreSQL každý příkaz běží pod transakcí: implicitní zahájenou serverem nebo explicitní zahájenou uživatelem.
- Pokud dojde k chybě, tak se transakce ruší (explicitní příkazem `ROLLBACK`).
- Pokud k chybě nedojde, tak se transakce potvrdí (explicitní příkazem `COMMIT`).

Ukážka 4

```
CREATE OR REPLACE FUNCTION fx(a int)
RETURNS void AS $$
BEGIN
  INSERT INTO mytab
    VALUES(10/a);
END;
$$ LANGUAGE plpgsql;
```

Implicitní transakce

```
\set AUTOCOMMIT on -- default
SELECT fx(10);
```

Explicitní transakce

```
postgres=# begin;
BEGIN
postgres=# SELECT fx(0);
ERROR:  division by zero
CONTEXT:  SQL statement "INSERT INTO mytab
    VALUES(10/a)"
PL/pgSQL function fx(integer) line 3
         at SQL statement
postgres=# SELECT 1;
ERROR:  current transaction is aborted,
commands ignored until end of transaction block
postgres=# ROLLBACK;
ROLLBACK
```

Hlavní rozdíly vůči konkurenci

- Nepoužívá se explicitní `COMMIT`, `ROLLBACK`
  - kód doběhne nebo se vyhodí výjimka
  - Postgres nemá problémy s dlouhými transakcemi (limitováno pouze místem na disku)
- Neošetřená chyba/výjimka automaticky skončí chybou - chyby nejsou ignorovány
- Ve výchozím nastavení funkce používá práva aktivního uživatele - v ostatních db funkce běží pod vlastníkem funkce (`SECURITY INVOKER`, `SECURITY DEFINER`)

Další rozdíly

- V Postgresu jsou pouze funkce - neexistují procedury
- Ve funkcích je umožněn zápis do db - tzv volatile funkce
- PostgreSQL neumožňuje předávat parametry odkazem - vnější parametry jsou vždy neměnné (immutable).

Implementace triggerů

- V PostgreSQL vždy pomocí funkcí
- V PostgreSQL se používají primárně řádkové triggery
- Existují event triggery - volané při změně systémového katalogu (`CREATE`, `ALTER`, `DROP`).

Ukážka 5

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

CREATE TRIGGER dop_trg BEFORE INSERT OR UPDATE
  ON test_tab
  FOR EACH ROW
  EXECUTE PROCEDURE dop_trg_func();
```

Ukázka 6

```
CREATE OR REPLACE FUNCTION kontr_trg_func()
RETURNS trigger AS $$
BEGIN
  IF EXISTS(SELECT *
              FROM faktury
             WHERE zaplaceno IS NULL
               AND stornovano IS NULL)
  THEN
    RAISE EXCEPTION 'nelze zrusit zakaznika'
      USING DETAIL 'nezaplacene faktury';
  END IF;
  RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

Jazyk SQL

- Pro one line funkce

```
CREATE OR REPLACE FUNCTION sort(anyarray)
RETURNS anyarray AS $$
  SELECT ARRAY(SELECT unnest($1)
               ORDER BY 1)
$$ LANGUAGE sql;
```

Jazyk PLpgSQL

- Pro manipulaci s daty a multiline funkce

```
CREATE OR REPLACE FUNCTION next_day(d date,
                                    day text)
RETURNS date AS $$
DECLARE ddow int DEFAULT EXTRACT(DOW FROM d);
  daydow int;
BEGIN
  daydow := array_position('{sun,mon,tue,wed,thu,fri,sat}',
                           left(lower(day),3)) - 1;
  IF daydow IS NULL THEN
    RAISE EXCEPTION 'wrong day identifier %', day;
  END IF;
  RETURN CASE WHEN daydow <= dow THEN d + daydow - dow + 7
              ELSE d + daydow - dow END;
END;
$$ LANGUAGE plpgsql;
```

Ostatní jazyky

- Trusted - např. parsování XML, JSON
  - PLPerl
- Untrusted - z hlediska bezpečnosti a stability
  - PLPerlu, PLPythonu
  - funkce v untr. jazycích může vytvářet pouze superuser
- Pozor na untrusted j. - jejich nevhodné použití může mít za následek havárii serveru (nucený restart). Pokud to není nezbytně nutné, tak nepoužívat.

Co v uložených proc. nedělat

- Suplování komunikačního serveru
  - smtp, ftp, http komunikace nepatří do SP
  - lze používat Foreign Data Wrappers
- Čekání na vnější událost
  - Uvnitř SP čekám na zámky a na dokončení DB operací
  - Čekání na dokončení externích událostí (volání externích událostí) nepatří do SP (možná v případě kritické nouze, v krátkodobém řešení - rizikové)

LISTEN/NOTIFY

- Databáze může signalizovat zaregistrovaným klientům výskyt určité události
- Na straně klienta se použije příkaz LISTEN
- Na straně serveru (v triggeru) příkaz NOTIFY
- Nesnažím se přesunout nepatřičný kód z klienta do triggeru - místo toho notifikuji klienta.
- Případně pokud by notifikovaná činnost byla také transakční - místo triggeru použiji notifikaci a 2PC

Implementace OUT parametrů

```
CREATE OR REPLACE FUNCTION foo(a int,
                               OUT b int,
                               OUT c int);

DO $$
DECLARE bt int; ct int;
BEGIN
  -- spatne - nemuze fungovat
  PERFORM foo(10, a, b);

  -- dobre - parametry jsou immutable
  SELECT foo(10) INTO a, b;
```

Tabulkové funkce

- Jednoduše lze psát funkce, které vrací tabulky
- Neměly by nahrazovat pohledy (blokují SQL optimalizátor)

```
CREATE TYPE tp AS (a int, b int);

CREATE FUNCTION fx() RETUTURNS SETOF tp ...
CREATE FUNCTION fx(OUT a int, OUT b int)
   RETURNS SETOF RECORD ...
CREATE FUNCTIONS fx()
   RETURNS TABLE (OUT a int, OUT b int) ...
```

Návrat z funkce

- `RETURN`
- `RETURN NEXT`
- `RETURN QUERY`

Ladící výpisy a ukončení

- funkce doběhne nebo vyhodí výjimku.
- `RAISE NOTICE`
- `RAISE WARNING`
- `RAISE EXCEPTION`

```
RAISE EXCEPTION 'nelze zrusit zaznam zakaznika'
  USING
     DETAIL='nezaplacene nestornovane faktury';
```

Vložené statické SQL

- Smysl SQL příkazu se v čase nemění
- Lze opakovaně použít prováděcí plán dotazu
- Imunní vůči SQL injection
- Čitelný zápis
- Lze staticky kontrolovat - plpgsql_check
- Pozor ale na generické plány - plán optimalizovaný pro nejpravděpodobnější hodnoty parametrů.

Dynamické SQL

- SQL se vytvoří za běhu - nelze provést statickou kontrolu
- Prováděcí plán se po provedení dotazu zahazuje (režie s opakovaným plánováním)
- Každý plán je ale zákaznický (optimalizovaný pro použité hodnoty parametrů)
- Hrozí riziko SQL injection - nutné ošetřit parametry dotazu

Dynamické SQL - ukážka

```
FOR i IN 1..10 LOOP
EXECUTE format('CREATE TABLE %I(a int)',
                'mytab' || i);
END LOOP;

FOR r IN
   EXECUTE
          format('SELECT * FROM %I WHERE %I=$1',
                 tablevar, columnvar)
      USING filtervar;
LOOP
...
END LOOP
```

Organizace kódu - schémata

- Pro logickou organizaci db objektů se používají schémata - a to i pro uložené procedury (analogie “packages” Oracle a “modules” DB2)
- Nastavení přístupových práv - `GRANT/REVOKE`
- Nastavení viditelnosti - `SEARCH_PATH` - nastavujte vždy na začátku session a dále neměňte.
- Pokud možno neduplikujte kód v jedné db
- Není problém mít desítky/stovky tisíc db objektů v databázi - nelze používat `pgdump/pgrestore` a potažmo `pg_upgrade` -> lépe se pracuje s větším množstvím menších db než s menším množstvím větších db.

Informace pro optimalizátor

- Myslí se tím SQL optimalizátor
- `VOLATILE` - default
- `STABLE` - v rámci jednoho dotazu vrací stejný výsledek pro stejné parametry a nemá vedlejší efekty (nezapisuje do db) - používá snapshot pro čtení dat v db
- `IMMUTABLE` - stále vrací stejný výsledek pro stejné parametry a nezapisuje do db - lze použít ve funkcionálních indexech - používá snapshot (změny v db po zahájení dotazu nejsou viditelné).

Extenze

- Způsob jak jednotně (od) instalovat skupinu databázových objektů - příkaz `DROP/CREATE EXTENSION`
- U nainstalovaných extenzí se sledují verze
- Extenze obsahuje
  - generující sql skript (obsahuje příkazy `CREATE ...`)
  - řídící soubor (komentář, název, implicitní verze, cesta ke knihovnám)
  - alter sql skripty sloužící k upgrade/downgrade extenzí - `ALTER EXTENSION UPDATE`
- Extenze může být zabalena v rpm - doručí soubory do správných adresářů
- Extenze by měly být samostatně testovatelné

Testování - regresní testy

- regresní testy - postavené nad aplikací `pg_regress` a `pg_config`
- v `Makefile` proměnná `REGRESS`
- test v adresáři `sql`
- očekávaný výsledek v adresáři `expected`
- make installcheck

Testování - pgtap testy

```
use TestLib;
use Test::More tests => 1;
use PostgresNode;

my $node = get_new_node();
$node->init;
$node_master->append_conf('postgresql.conf', qq(
max_prepared_foreign_transactions = 10
));
$node->start;
```

Testování - pgtap testy

```
$master->safe_psql('postgres', "
BEGIN;
UPDATE t1 SET c = 5 WHERE c = 5;
UPDATE t2 SET c = 6 WHERE c = 6;
PREPARE TRANSACTION 'gxid1';
BEGIN;
");
```