Modernes PL/pgSQL

Stefan Keller

Swiss PGDay 2022 – The Swiss PostgreSQL Conference

1. Juli 2022

Inhaltsüberblick

1. Stored Procedures und Procedural Languages
2. Sprache PL/pgSQL
3. Unit-Testing mit PL/pgSQL

Diskussion ~10 min.

Besonderer Dank geht an:

- Lukas Buchli, IFS OST, und Miles Strässle, Informatik OST (pgUnit)
- Jim Mlodgenski, AWS (Sprache PL/pgSQL)
- Daniel Westermann, dbi Services (Server-side Programming)

Stored Procedures und Procedural Languages

Stored Procedures und Procedural Languages

- "Stored Prozedures" (SQL Standard): Funktionen (Prozeduren), die in der Datenbank selbst ausgeführt werden

- "Procedural Languages" (PL) sind schneller als alles andere (Domain Logik auf Anwendungsserver), da die Daten mit PL nahe an dem Ort prozessiert werden, an dem sie sind ("server-side"); der Verbindungsaufwand zur Datenbank entfällt

Stored Procedures in PostgreSQL

- PostgreSQL erlaubt das Schreiben von benutzerdefinierten Funktionen (Prozeduren) in vielen verschiedenen prozeduralen Sprachen - "mitgeliefert":
  - PL/SQL
  - PL/pgSQL (der "Default")
  - PL/Python
  - sowie PL/Perl und PL/Tkl
- Von "Dritten"
  - PL/Java - https://github.com/tada/pljava/wiki
  - PLv8 Javascript engine - https://github.com/plv8/plv8
  - PL/R (https://github.com/postgres-plr/plr
  - PL/Scheme (https://github.com/vy/plscheme
  - PL/Sh (https://github.com/petere/plsh
  - PL/Lua (https://pllua.github.io/pllua/
  - PL/Rust (https://github.com/zombodb/plrust
- Veraltet? PL/PHP und PL/Ruby

Stored Procedures in PostgreSQL

- Implementations-Prinzip in PostgreSQL
  - Der Datenbankserver hat kein eingebautes "Wissen" darüber, wie der Quelltext der Funktion zu interpretieren ist
  - Stattdessen wird die Aufgabe an einen Handler weitergegeben, der die Details der jeweiligen Sprache kennt

Stored Procedures in PostgreSQL

Prozedurale Sprachen kommen in PG als Erweiterungen:

```
postgres=# create extension plperl;
CREATE EXTENSION

postgres=# \dx
List of installed extensions
Name   | Version |   Schema   |         Description
--------+---------+------------+------------------------------
plperl  | 1.0     | pg_catalog | PL/Perl procedural language
plpgsql | 1.0     | pg_catalog | PL/pgSQL procedural languag
```

Sprache PL/pgSQL

Was ist PL/pgSQL?

- PL/pgSQL ist die prozedurale Erweiterung von SQL mit allen nötigen Merkmalen von Programmiersprachen wie Schleifen, Bedingungen und Rückgabetypen
- Familie ADA/Algol/Pascal/Modula mit Einschränkungen in I/O, kein GOTO
- Datenmanipulation und Abfrageanweisungen von SQL sind in innerhalb von prozeduralen Code-Einheiten
- Dies erlaubt mehr Freiheiten als allgemeines SQL und ist leichter als der Aufruf aus einem Client-Programm

Wie PL/pgSQL arbeitet

- PL/pgSQL ist wie jede andere prozedurale Sprache.
- Wenn eine PL-Funktion ausgeführt wird, lädt der interne "Function Manager (fmgr)" den Sprach-Handler und ruft ihn auf.
- Der Sprach-Handler interpretiert dann den Inhalt des pg_proc-Eintrags für die Funktion (proargtypes, prorettype, prosrc).

Wie PL/pgSQL arbeitet

- Beim ersten Aufruf einer Funktion in einer Sitzung wird der Call-Handler einen Funktionsanweisungsbaum "kompilieren".

- SQL-Abfragen in der Funktion werden an dieser Stelle nur als String gespeichert.

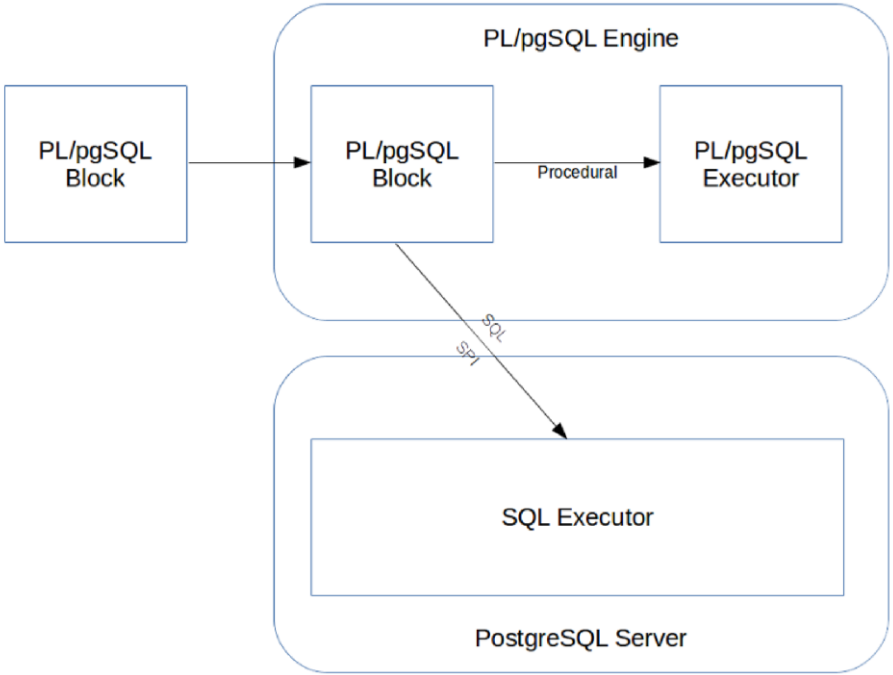
- Was für Sie wie ein Ausdruck aussehen mag, ist in Wirklichkeit eine SELECT Abfrage:

```
my_variable := some_parameter * 100;
```

Wie PL/pgSQL arbeitet

- Der PL/pgSQL-Anweisungsbaum ist dem PostgreSQL-Ausführungsbaum sehr ähnlich.
- Der Call-Handler führt dann den Anweisungsbaum aus.
- Bei der ersten Ausführung eines Anweisungsknotens, der eine SQL Abfrage enthält, wird diese Abfrage über das Server Programming Interface (SPI) vorbereitet.
- Der vorbereitete Plan wird dann bei jedem Aufruf der betreffenden Anweisung in der aktuellen Sitzung ausgeführt.

PL/pgSQL-Umgebung sowie "vom Source Code bis zur Ausführung"



Arten von PL/pgSQL-Blöcken

- Die Grundeinheit in jedem PL/pgSQL-Code ist ein BLOCK.
- Der gesamte PL/pgSQL-Code besteht aus einem einzigen Block oder aus Blöcken, die entweder sequentiell oder in einem anderen Block verschachtelt sind.
- Es gibt zwei Arten von Blöcken:
  - Anonymous blocks (DO)
    Im Allgemeinen dynamisch aufgebaut und nur einmal vom Benutzer ausgeführt. Es handelt sich um eine Art komplexe SQL-Anweisung.
  - Named blocks (Functions and Stored Procedures)
    Haben einen Namen, der mit ihnen verbunden ist, werden in der Datenbank gespeichert,
    und können wiederholbar ausgeführt werden, und sie können Parameter aufnehmen

Struktur von anonymen Blöcken

```
DO $$
    [ <<label>> ]
DECLARE
    /* Declare section (optional). */
BEGIN
    /* Executable section (required). */
EXCEPTION
    /* Exception handling section (optional). */
END [ label ]
$$
```

Kommentare

- Es gibt zwei Arten von Kommentaren in PL/pgSQL
  - -- beginnt einen Kommentar, der bis zum Ende der Zeile reicht
  - /* mehrzeilige Kommentare */

- Kommentare sind notwendig, um den Leuten mitzuteilen, was beabsichtigt ist und warum es auf diese bestimmte Weise gemacht wurde.

- Vermeiden Sie zu viele Kommentare

Variablen

- Verwenden Sie Variablen für
  - Vorübergehende Speicherung von Daten
  - Manipulation von gespeicherten Werten
  - Wiederverwendbarkeit
  - Leichte Wartung

- Deklariert im deklarativen Abschnitt innerhalb eines Block

```
v_last_name VARCHAR(15);
```

Umgang mit Variablen

- Die im Deklarationsabschnitt vor einem Block deklarierten Variablen, werden bei jedem Aufruf des Blocks auf ihre Standardwerte initialisiert, nicht nur einmal pro Funktionsaufruf.
- Variablen in einem Deklarationsabschnitt können gleichnamige Variablen in einem äusseren Block überschreiben.
- Wenn der äussere Block mit einem Label benannt ist, sind seine Variablen weiterhin verfügbar unter <label>.<varname>.

Erklärungen

- Syntax

```
identifier [CONSTANT] datatype [NOT NULL] [:= | = | DEFAULT expr];
```

- Beispiele

```
DECLARE
    v_birthday DATE;
    v_age INT NOT NULL = 21;
    v_name VARCHAR(15) := 'Homer’;
    v_magic CONSTANT NUMERIC := 42;
    v_valid BOOLEAN DEFAULT TRUE;
```

%TYPE

- Deklarieren Sie die Variable gemäss :
    - Definition einer Datenbankspalte
    - Einer anderen zuvor deklarierten Variablen

identifier table.column_name%TYPE;

- Beispiel

```
DECLARE
    v_email users.email%TYPE;
    v_my_email v_email%TYPE := 'rds-postgres-extensions-request@amazon.com';
```

%ROWTYPE

- Deklarieren einer Variablen mit dem Typ einer ROW einer Tabelle

identifier table%ROWTYPE;

- Beispiel

```
DECLARE
    v_user users%ROWTYPE;
```

Records

- Ein RECORD ist ein Variablentyp, der dem ROWTYPE ähnlich ist, aber keine vordefinierte Struktur besitzt.

- Die eigentliche Struktur des RECORDs wird erstellt wenn die Variable erstmals einen Wert zugewiesen bekommt.

- Ein RECORD ist kein echter Datentyp, nur ein Platzhalter.

```
DECLARE
    r record;
```

Variable Scope

```
DO $$
DECLARE
        quantity integer := 30;
BEGIN
        RAISE NOTICE 'Quantity here is %', quantity; -- 30
        quantity := 50;
        -- Create a subblock
        DECLARE
                quantity integer := 80;
        BEGIN
                RAISE NOTICE 'Quantity here is %', quantity; -- 80
        END;
        RAISE NOTICE 'Quantity here is %', quantity; -- 50
END
$$;
```

Qualify an Identifier

```
DO $$
<< mainblock >>
DECLARE
          quantity integer := 30;
BEGIN
          RAISE NOTICE 'Quantity here is %', quantity; --30
          quantity := 50;
          -- Create a subblock
          DECLARE
                    quantity integer := 80;
          BEGIN
                    RAISE NOTICE 'Quantity here is %', mainblock.quantity; --50
                    RAISE NOTICE 'Quantity here is %', quantity; --80
          END;
          RAISE NOTICE 'Quantity here is %', quantity; --50
END
$$;
```

RAISE

- Meldet Nachrichten
  - Kann vom Client gesehen werden, wenn der entsprechende Level verwendet wird

RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;

Assigning Values

- Verwenden Sie den Zuweisungsoperator (:= oder =)

```
DECLARE
    v_last_name VARCHAR := 'Smith’;
    v_date DATE;
BEGIN
    v_last_name := lower(v_last_name);
    v_date := to_date('2000-01-01', 'YYYY-MM-DD')
```

SELECT in PL/pgSQL

- Abrufen von Daten aus der Datenbank mit einer SELECT-Anweisung.
- Abfragen dürfen nur eine Zeile zurückgeben.
- INTO-Klausel ist erforderlich.

```
DECLARE
    v_first_name users.first_name%TYPE;
    v_last_name users.last_name%TYPE;
BEGIN
    SELECT first_name, last_name
    INTO v_first_name, v_last_name
    FROM users
    WHERE user_id = 1;
END
```

INSERT / UPDATE / DELETE

```
DECLARE
    v_forum_name forums.name%TYPE := 'Hackers';
BEGIN
    INSERT INTO forums (name)
    VALUES (v_forum_name);
    UPDATE forums
    SET moderated = true
    WHERE name = v_forum_name;
END
```

PERFORM <Function>

- Wertet einen Ausdruck oder eine Abfrage aus, verwirft aber das Ergebnis.
- Häufig verwendet bei der Ausführung von Wartungsbefehlen

```
BEGIN
    PERFORM create_partition('moderation_log', '201606');
END
```

Struktur von benannten Blöcken

```
CREATE FUNCTION [ function_name ] ()
RETURNS [return_type] $$
[ <<label>> ]
DECLARE
    /* Declare section (optional). */

BEGIN
    /* Executable section (required). */

EXCEPTION
    /* Exception handling section (optional). */
END [ label ]
$$ LANGUAGE plpgsql;
```

Beispiel einer Funktion

```
CREATE FUNCTION get_user_count()
    RETURNS integer
AS $$
DECLARE
    v_count integer;
BEGIN
    SELECT count(*)
    INTO v_count
    FROM users;

    RETURN v_count;
END
$$ LANGUAGE plpgsql;
```

Dollar Quoting

- Der Tag $$ bezeichnet den Anfang und das Ende einer Zeichenkette
- Optional kann ein nicht-leerer Tag als Teil des Anführungszeichens verwendet werden
    - $_$
    - abc
- Kann verwendet werden, um unnötige Escape-Zeichen im ganzen String zu vermeiden

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

Funktionsparameter

- Es können ein oder mehrere Parameter verwendet werden.
- Parameternamen sind optional, werden aber dringend empfohlen

```
CREATE FUNCTION get_user_name(varchar, p_last_name varchar)
RETURNS varchar AS $$
DECLARE
    v_first_name varchar;
    v_name varchar;
BEGIN
    v_first_name := $1;
    SELECT name INTO v_name FROM users
    WHERE first_name = v_first_name AND last_name = p_last_name
    LIMIT 1;

    RETURN v_name;
END
$$ LANGUAGE plpgsql;
```

Default Parameter

- Parameter können einen default-Wert haben
- Dies macht sie zu optionalen Parametern

```
CREATE FUNCTION get_user_count(p_active boolean DEFAULT true)
    RETURNS integer AS $$
DECLARE
    v_count integer;
BEGIN
    SELECT count(*) INTO v_count
    FROM users
    WHERE active = p_active;

    RETURN v_count;
END
$$ LANGUAGE plpgsql;
```

FUNCTION vs. PROCEDURE

- PROCEDUREs
  - können nicht in einer SQL-Anweisung aufgerufen werden, da sie keinen Rückgabewert haben
  - müssen "CALL" verwenden (statt EXECUTE)
  - können während Transaktionen enthalten (FUNCTION nicht), solange der aufrufende CALL-Befehl nicht Teil eines expliziten Transaktionsblocks ist

Assertions

- Eine bequeme Abkürzung für das Einfügen von Debugging Checks
- Kann durch die Variable plpgsql.check_asserts gesteuert werden

```
CREATE FUNCTION get_user_count(p_active boolean DEFAULT true)
    RETURNS integer AS $$
DECLARE
    v_count integer;
BEGIN
    ASSERT p_active IS NOT NULL;

    SELECT count(*) INTO v_count
    FROM users
    WHERE active = p_active;

    RETURN v_count;
END
$$ LANGUAGE plpgsql;
```

PL/pgSQL Kontroll-Strukturen

Kontrolle des Programmflusses

- Der logische Ablauf von Anweisungen kann durch bedingte IF-Anweisungen und Schleifenkontrollstrukturen verändert werden
    - Bedingte Strukturen
    - Schleifen-Strukturen

IF Statements

- IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

- IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

Nested IF Statements

```
IF boolean-expression THEN
    IF boolean-expression THEN
        statements
    END IF;
ELSE
    statements
END IF;
```

ELSIF Statements

- Eine Folge von Anweisungen, die auf mehreren Bedingungen basieren

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- the only other possibility is that number is null
    result := 'NULL';
END IF;
```

CASE Statements

- Wird für komplexe Konditionale verwendet
- Ermöglicht den Test einer Variablen auf Gleichheit mit einer Liste von Werten

```
BEGIN
    CASE status
        WHEN 'Pending' THEN RAISE NOTICE 'PENDING';
        WHEN 'Accepted' THEN RAISE NOTICE 'ACCEPTED';
        WHEN 'Declined' THEN RAISE NOTICE 'DECLINED';
        WHEN 'Blocked' THEN RAISE NOTICE 'BLOCKED';
        ELSE RAISE NOTICE 'UNKNOWN';
    END CASE;
END
```

Searched CASE Statements

- Jede WHEN clause wird nacheinander ausgewertet, bis sie TRUE ist
- Nachfolgende WHEN-Ausdrücke werden nicht ausgewertet

```
BEGIN
    CASE
        WHEN x BETWEEN 0 AND 10 THEN
            RAISE NOTICE 'Value is between zero and ten';
        WHEN x BETWEEN 11 AND 20 THEN
            RAISE NOTICE 'Value is between eleven and twenty';
    END CASE;
$$;
```

FOUND

- FOUND, das vom Typ boolean ist, beginnt mit false in jedem PL/pgSQL-Funktionsaufruf. FOUND ist kein Error
- FOUND wird von jeder der folgenden Arten von Anweisungen gesetzt:
  - Eine SELECT INTO-Anweisung setzt FOUND auf true, wenn sie eine Zeile zurückgibt, false, wenn keine Zeile zurückgegeben wird
  - Eine PERFORM-Anweisung setzt FOUND true, wenn sie eine Zeile erzeugt (und verwirft), false, wenn keine Zeile erzeugt wird
  - UPDATE-, INSERT- und DELETE-Anweisungen setzen FOUND true, wenn mindestens eine Zeile betroffen ist, false, wenn keine Zeile betroffen ist
  - Eine FETCH-Anweisung setzt FOUND auf true, wenn sie eine Zeile zurückgibt, false, wenn keine Zeile zurückgegeben wird.
  - Eine FOR-Anweisung setzt FOUND true, wenn sie ein oder mehrere Male durchläuft, sonst false.

FOUND

```
DECLARE
    v_first_name users.first_name%TYPE;
    v_last_name users.last_name%TYPE;
BEGIN
    SELECT first_name, last_name
    INTO v_first_name, v_last_name
    FROM users
    WHERE user_id = 1;

    IF FOUND THEN
        RAISE NOTICE 'User Found';
    ELSE
        RAISE NOTICE 'User Not Found';
END
```

Schleifen-Strukturen

- Unconstrained Loop
- WHILE Loop
- FOR Loop
- FOREACH Loop

Unbeschränkte Schleifen

- Erlaubt die Ausführung ihrer Anweisungen mindestens einmal, auch wenn die Bedingung bereits beim Eintritt in die Schleife erfüllt war

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT; -- exit loop
    END IF;
END LOOP;
```

```
LOOP
    -- some computations
    EXIT WHEN count > 0; -- same result as previous example
END LOOP;
```

Continue

```
CONTINUE [ label ] [ WHEN expression ];
```

- Wird kein Label angegeben, wird die nächste Iteration der innersten Schleife begonnen
- Wenn WHEN angegeben ist, wird die nächste Iteration der Schleife nur begonnen wenn der Ausdruck wahr ist. Andernfalls geht die Kontrolle an die Anweisung nach CONTINUE
- CONTINUE kann mit allen Arten von Schleifen verwendet werden; es ist nicht beschränkt auf unbeschränkte Schleifen.

```
LOOP
    -- some computations
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- some computations for count IN [50 .. 100]
END LOOP;
```

WHILE Schleifen

```
WHILE condition LOOP
        statement1..;
END LOOP;
```

- Wiederholt eine Folge von Anweisungen, bis die steuernde Bedingung nicht mehr TRUE ist
- Die Bedingung wird zu Beginn jeder Iteration ausgewertet

```
WHILE NOT done LOOP
    -- some computations here
END LOOP;
```

FOR Schleifen

```
FOR <loop_counter> IN [REVERSE] <low bound>..<high bound> LOOP
    -- some computations here
END LOOP;
```

- Verwenden Sie eine FOR-Schleife, um den Test für die Anzahl der Iterationen abzukürzen.
- Deklarieren Sie den Zähler nicht; er wird implizit deklariert

```
DO $$
BEGIN
    FOR i IN 1..10 LOOP
        RAISE NOTICE 'value: %', i;
    END LOOP;
END
$$;
```

FOR Schleifen

- For-Schleifen können direkt ein Abfrageergebnis verwenden

```
DECLARE
    r record;
BEGIN
    FOR r IN SELECT email FROM users LOOP
        RAISE NOTICE 'Email: %', r.email;
    END LOOP;
END
```

Schleifen über Abfrageergebnisse

- Die letzte Zeile ist auch nach Verlassen der Schleife noch zugänglich

```
DECLARE
    r record;
BEGIN
    FOR r IN SELECT email FROM users LOOP
        RAISE NOTICE 'Email: %', r.email;
    END LOOP;
    RAISE NOTICE 'Email: %', r.email;
END
```

Schleifen über Abfrageergebnisse

- Schleife über dynamisches SQL
- Bei jeder Ausführung neu geplant

```
DECLARE
    rec RECORD;
    sql TEXT;
BEGIN
    sql := 'SELECT email FROM users';
    FOR rec IN EXECUTE sql LOOP
        RAISE NOTICE 'Email: %', rec.email;
    END LOOP;
END
```

Schleifen über Arrays

- Verwendet die FOREACH-Anweisung

```
DECLARE
    users varchar[] := ARRAY['Mickey', 'Donald', 'Minnie'];
    v_user varchar;
BEGIN
    FOREACH v_user IN ARRAY users LOOP
        RAISE NOTICE 'User: %', v_user;
    END LOOP;
END
```

Schleifen über Arrays

- Verwenden Sie die SLICE-Syntax, um über mehrere Dimensionen zu iterieren

```
DECLARE
    users varchar[];
    v_dim varchar[];
BEGIN
    users := ARRAY[ARRAY['Mickey', 'Donald'], ARRAY['Mouse', 'Duck']];
    FOREACH v_dim SLICE 1 IN ARRAY users LOOP
        RAISE NOTICE 'Dimension: %', v_dim;
    END LOOP;
END
```

Verschachtelte Schleifen

- Verschachtelung von Schleifen auf mehreren Ebenen
- Verwenden Sie Label, um zwischen Blöcken zu unterscheiden.
- Verlassen Sie die äussere Schleife mit der EXIT-Anweisung, die auf das Label verweist.

```
BEGIN
    <<outer_loop>>
    LOOP
        v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10; -- leaves both loops
        <<inner_loop>>
        LOOP
            EXIT outer_loop WHEN total_done = 'YES';
            -- leaves both loops
            EXIT WHEN inner_done = 'YES';
            -- leaves inner loop only
        END LOOP inner_loop;
    END LOOP outer_loop;
END
```

Dynamisches SQL

Dynamisches SQL

- Eine Programmiermethodik zur Erzeugung und Ausführung von SQL Anweisungen während der Laufzeit

- Nützlich für:
  - Ad-hoc-Abfragesysteme
  - DDL und Datenbankwartung

EXECUTE command-string [INTO target] [USING expression [, ...]];

Dynamisches SQL - Achtung

- Für Befehle, die über EXECUTE ausgeführt werden, gibt es kein Plan Caching
  - Der Befehl wird jedes Mal geplant, wenn er ausgeführt wird.

- Anfällig auf SQL-Injection-Angriffe
  - Alle eingehenden Parameter müssen validiert werden
  - Binden Sie die Parameter an den Befehl, anstatt an die Zeichenfolgen

Ausführung

```
CREATE FUNCTION grant_select(p_table varchar, p_role varchar)
    RETURNS void AS
$$
DECLARE
    sql varchar;
BEGIN
    sql := 'GRANT SELECT ON TABLE ' || p_table || ' TO ' || p_role;
    EXECUTE sql;
END
$$ LANGUAGE plpgsql;
```

- Hinweis: Tun Sie dies nicht in produktiven Umgebungen: Validieren Sie zuerst die Parameter!

EXECUTE INTO

```
CREATE FUNCTION get_connection_count(p_role varchar)
    RETURNS integer
AS $$
DECLARE
    v_count integer;
    sql varchar;
BEGIN
    sql := 'SELECT count(*) FROM pg_stat_activity
        WHERE usename = ''' || p_role || '''';
    EXECUTE sql INTO v_count;
    RETURN v_count;
END
$$ LANGUAGE plpgsql;
```

- Hinweis: Tun Sie dies nicht in produktiven Umgebungen: Validieren Sie zuerst die Parameter!

EXECUTE mit USING

```
CREATE FUNCTION get_connection_count(p_role varchar)
    RETURNS integer
AS $$
DECLARE
    v_count integer;
    sql varchar;
BEGIN
    sql := 'SELECT count(*) FROM pg_stat_activity
        WHERE usename = $1';
    EXECUTE sql INTO v_count USING p_role;
    RETURN v_count;
END
$$ LANGUAGE plpgsql;
```

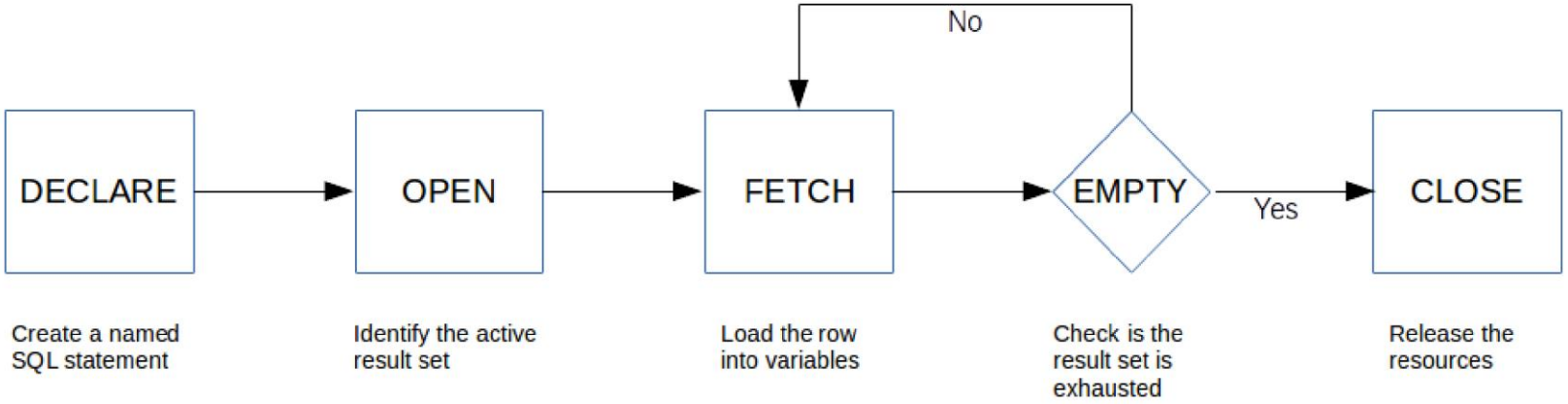
PL/pgSQL Cursors

CURSOR

- Jede von PostgreSQL ausgeführte SQL-Anweisung hat einen individuellen Cursor zugeordnet
  - Implizite Cursor: Deklariert für alle DML und PL/pgSQL SELECT Anweisungen
  - Explizite Cursor: Deklariert und benannt durch den Programmierer

- Verwenden Sie CURSOR, um jede Zeile einzeln zu verarbeiten, die eine mehrzeilige SELECT-Anweisung zurückgibt

CURSOR Datenfluss



Cursor deklarieren

- Ein Cursor muss als Variable deklariert werden.
  - Verwenden Sie das Schlüsselwort SCROLL, um sich rückwärts durch einen Cursor zu bewegen

name [[NO] SCROLL] CURSOR [(arguments)] FOR query;

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT *
                                   FROM tenk1
                                   WHERE unique1 = key;
```

Cursor öffnen

- Welche OPEN-Methode zu verwenden ist, hängt davon ab, wie sie deklariert wurde

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
OPEN cur2;

OPEN curs3(42);
OPEN curs3 (key := 42);
```

Abrufen von Daten

- FETCH gibt die nächste Zeile zurück

FETCH curs2 INTO foo, bar, baz;

- FETCH kann auch den Cursor bewegen

FETCH LAST FROM curs3 INTO x, y;

Abrufen von Daten

```
CREATE FUNCTION grant_select(p_role varchar)
    RETURNS void AS $$
DECLARE
    sql varchar;
    r record;
    tbl_cursor CURSOR FOR SELECT schemaname, relname
                          FROM pg_stat_user_tables;
BEGIN
    OPEN tbl_cursor;
    LOOP
        FETCH tbl_cursor INTO r;
        EXIT WHEN NOT FOUND;
        sql := 'GRANT SELECT ON TABLE ' || r.schemaname ||
            '.' || r.relname || ' TO ' || p_role;
        EXECUTE sql;
    END LOOP;
    CLOSE tbl_cursor;
END
$$ LANGUAGE plpgsql;
```

PL/pgSQL Returning Data

Returning Scalars

- Einfachster Rückgabetyp

```
CREATE FUNCTION get_connection_count()
    RETURNS integer AS $$
DECLARE
    v_count integer;
BEGIN
    SELECT count(*) INTO v_count
    FROM pg_stat_activity;

    RETURN v_count;
END
$$ LANGUAGE plpgsql;


SELECT get_connection_count();
get_connection_count
----------------------
                   11
(1 row)
```

Returning Nothing

- Einige Funktionen benötigen keinen Rückgabewert
  - Es handelt sich in der Regel um eine Wartungsfunktion, wie z.B. das Erstellen von Partitionen oder das Bereinigen von Daten
  - Ab PostgreSQL 11 können Stored Procedures in diesen Fällen verwendet werden
- Returning VOID

```
CREATE FUNCTION purge_log()
    RETURNS void AS
$$
BEGIN
    DELETE FROM moderation_log
    WHERE log_date < now() - '90 days'::interval;
END
$$ LANGUAGE plpgsql;
```

Returning Sets

- Funktionen können eine Ergebnismenge zurückgeben
- SETOF verwenden
- RETURN NEXT verwenden
  - RETURN NEXT kehrt nicht tatsächlich aus der Funktion zurück.
  - Aufeinanderfolgende RETURN NEXT-Befehle bilden eine Ergebnismenge.
- Ein abschliessendes RETURN verlässt die Funktion

Returning Sets

```
CREATE FUNCTION fibonacci(num integer)
    RETURNS SETOF integer AS $$
DECLARE
    a int := 0;
    b int := 1;
BEGIN
    IF (num <= 0)
        THEN RETURN;
    END IF;
    RETURN NEXT a;
    LOOP
        EXIT WHEN num <= 1;
        RETURN NEXT b;
        num = num - 1;
        SELECT b, a + b INTO a, b;
    END LOOP;
END;
$$ language plpgsql;
```

Returning Records

- Komplexere Strukturen können zurückgegeben werden

```
CREATE FUNCTION get_oldest_session()
    RETURNS record AS
$$
DECLARE
    r record;
BEGIN
    SELECT *
    INTO r
    FROM pg_stat_activity
    WHERE usename = SESSION_USER
    ORDER BY backend_start DESC
    LIMIT 1;

    RETURN r;
END
$$ LANGUAGE plpgsql;
```

Returning Records

- Die Verwendung eines generischen Satztyps erfordert die Definition der Struktur während der Laufzeit

```
SELECT * FROM get_oldest_session();
ERROR: a column definition list is required for functions ...
LINE 1: SELECT * FROM get_oldest_session();
```

```
SELECT * FROM get_oldest_session()
AS (a oid, b name, c integer, d oid, e name, f text, g inet,
    h text, i integer, j timestamptz, k timestamptz,
    l timestamptz, m timestamptz, n boolean, o text, p xid,
    q xid, r text);
```

Returning Records

- Alle Tabellen und Views haben automatisch entsprechende Typdefinitionen, so dass sie als Rückgabetypen verwendet werden können

```
CREATE FUNCTION get_oldest_session()
    RETURNS pg_stat_activity AS $$
DECLARE
    r record;
BEGIN
    SELECT *
    INTO r
    FROM pg_stat_activity
    WHERE usename = SESSION_USER
    ORDER BY backend_start DESC
    LIMIT 1;

    RETURN r;
END
$$ LANGUAGE plpgsql;
```

Returning Records

- Oft wird nur eine Teilmenge der Tabellendaten benötigt.
- Eine View kann verwendet werden, um die erforderliche Struktur zu definieren

```
CREATE VIEW running_queries AS
SELECT CURRENT_TIMESTAMP - query_start as runtime, pid,
    usename, waiting, query
    FROM pg_stat_activity
ORDER BY 1 DESC
LIMIT 10;
```

Returning Records

- RETURN QUERY kann zur Vereinfachung der Funktion verwendet werden

```
CREATE FUNCTION running_queries(p_rows int, p_len int DEFAULT 50)
    RETURNS SETOF running_queries AS
$$
BEGIN
    RETURN QUERY SELECT runtime, pid, usename, waiting,
                        substring(query,1,p_len) as query
                 FROM running_queries
                 ORDER BY 1 DESC
                 LIMIT p_rows;
END
$$ LANGUAGE plpgsql;
```

OUT Parameters

- Dient der Rückgabe strukturierter Informationen
- RETURNS ist optional, muss aber aufgezeichnet werden, wenn es verwendet wird.

```
CREATE FUNCTION active_locks(OUT p_exclusive int, OUT p_share int)
```

OUT Parameters

```
CREATE FUNCTION active_locks(OUT p_exclusive int, OUT p_share int) AS $$
DECLARE
    r record;
BEGIN
    p_exclusive := 0;
    p_share := 0;
    FOR r IN SELECT l.mode
            FROM pg_locks l, pg_stat_activity a
            WHERE a.pid = l.pid
            AND a.usename = SESSION_USER
    LOOP
        IF r.mode = 'ExclusiveLock' THEN
            p_exclusive := p_exclusive + 1;
        ELSIF r.mode = 'ShareLock' THEN
            p_share := p_share + 1;
    END IF;
END LOOP;
END
$$ LANGUAGE plpgsql;
```

OUT Parameters

- TIPP: Denken Sie beim Schreiben von Funktionen in Mengen - und nicht in Schleifen - um eine bessere Performance zu erreichen.
- HINWEIS: Verwenden Sie "OR REPLACE", wenn Sie Funktionen aktualisieren.

```
CREATE OR REPLACE FUNCTION active_locks(OUT p_exclusive int,
                                        OUT p_share int)
AS $$
BEGIN
    SELECT sum(CASE l.mode WHEN 'ExclusiveLock' THEN 1 ELSE 0 END),
        sum(CASE l.mode WHEN 'ShareLock' THEN 1 ELSE 0 END)
    INTO p_exclusive, p_share
    FROM pg_locks l, pg_stat_activity a
    WHERE a.pid = l.pid
    AND a.usename = SESSION_USER;
END
$$ LANGUAGE plpgsql;
```

Strukturierte Datensätze

- OUT-Parameter und SETOF-Datensatz verwenden

```
CREATE FUNCTION all_active_locks(OUT p_lock_mode varchar,
                                 OUT p_count int)
    RETURNS SETOF record AS $$
DECLARE
    r record;
BEGIN
    FOR r IN SELECT l.mode, count(*) as k
                FROM pg_locks l, pg_stat_activity a
                WHERE a.pid = l.pid
                AND a.usename = SESSION_USER
                GROUP BY 1
    LOOP
        p_lock_mode := r.mode;
        p_count := r.k;
        RETURN NEXT;
    END LOOP;
    RETURN;
...
```

Strukturierte Datensätze

- kann eine TABLE zurückgeben

```
CREATE FUNCTION all_active_locks()
    RETURNS TABLE (p_lock_mode varchar, p_count int) AS $$
DECLARE
    r record;
BEGIN
    FOR r IN SELECT l.mode, count(*) as k
                FROM pg_locks l, pg_stat_activity a
                WHERE a.pid = l.pid
                AND a.usename = SESSION_USER
                GROUP BY 1
    LOOP
        p_lock_mode := r.mode;
        p_count := r.k;
        RETURN NEXT;
    END LOOP;
    RETURN;
END
$$ LANGUAGE plpgsql;
```

Refcursors

- Bei grossen Ergebnismengen kann ein Cursor zurückgegeben werden
- Die einzige Möglichkeit, mehrere Ergebnissätze aus einer Funktion zurückzugeben

```
CREATE FUNCTION active_info(OUT p_queries refcursor,
                            OUT p_locks refcursor)
```

Refcursors

```
CREATE FUNCTION active_info(OUT p_queries refcursor,
                            OUT p_locks refcursor)
AS $$
BEGIN
    OPEN p_queries FOR SELECT runtime, pid, usename, waiting,
                       substring(query,1,50) as query
                       FROM running_queries
                       ORDER BY 1 DESC;
    OPEN p_locks FOR SELECT l.mode, count(*) as k
                       FROM pg_locks l, pg_stat_activity a
                       WHERE a.pid = l.pid
                       AND a.usename = SESSION_USER
                       GROUP BY 1;
END
$$ LANGUAGE plpgsql;
```

Handhabung von Metainformationen und Ausnahmen

Meta-Informationen

- Informationen über den zuletzt ausgeführten Befehl innerhalb einer Funktion
- Verfügbare Variablen
  - ROW_COUNT
  - RESULT_OID
  - PG_CONTEXT

GET DIAGNOSTICS variable { = | := } item [, ...];

Meta-Informationen

```
CREATE OR REPLACE FUNCTION purge_log()
    RETURNS void AS
$$
DECLARE
    l_rows int;
BEGIN
    DELETE FROM moderation_log
    WHERE log_date < now() - '90 days'::interval;

    GET DIAGNOSTICS l_rows = ROW_COUNT;
    RAISE NOTICE 'Deleted % rows from the log', l_rows;
END
$$ LANGUAGE plpgsql;
```

Ausnahmen (EXCEPTIONs)

- Eine Ausnahme ist eine Bezeichnung in PL/pgSQL, die während der Ausführung
- Sie wird ausgelöst, wenn ein Fehler auftritt oder explizit durch die Funktion
- Sie wird entweder im EXCEPTION-Block behandelt oder an die aufrufende Umgebung übergeben

```
[DECLARE]
BEGIN
    Exception/Error is Raised
EXCEPTION
    Error is Trapped
END
```

Ausnahmen (EXCEPTIONs)

- Verwenden Sie den WHEN-Block innerhalb des EXCEPTION-Blocks, um bestimmten Fälle zu fangen
- Sie können den Fehlernamen oder den Fehlercode im EXCEPTION-Block verwenden.

WHEN division_by_zero THEN ...

WHEN SQLSTATE '22012' THEN …

- Verwenden Sie die Sonderbedingungen OTHERS als Auffangtabelle

WHEN OTHERS THEN ...

Ausnahmen (EXCEPTIONs): Beispiel Error Codes

Code	Name
22000	data_exception
22012	division_by_zero
2200B	escape_character_conflict
22007	invalid_datetime_format
22023	invalid_parameter_value
2200M	invalid_xml_document
2200S	invalid_xml_comment
23P01	exclusion_violation

Ausnahmen (EXCEPTIONs)

```
CREATE OR REPLACE FUNCTION get_connection_count()
    RETURNS integer AS $$
DECLARE
    v_count integer;
BEGIN
    SELECT count(*)
    INTO STRICT v_count
    FROM pg_stat_activity;

    RETURN v_count;
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        RAISE NOTICE 'More than 1 row returned';
        RETURN -1;
    WHEN OTHERS THEN
        RAISE NOTICE 'Unknown Error';
        RETURN -1;
END
$$ LANGUAGE plpgsql;
```

Ausnahmen (EXCEPTIONs)

- SQLSTATE Gibt den numerischen Wert für den Fehlercode zurück.
- SQLERRM Gibt die mit dem Fehler verbundenen Error Nummer zurück

```
DECLARE
    v_count integer;
    err_num integer;
    err_msg varchar;
    BEGIN
...
EXCEPTION
    WHEN OTHERS THEN
        err_num := SQLSTATE;
        err_msg := SUBSTR(SQLERRM,1,100);
        RAISE NOTICE 'Trapped Error: %', err_msg;
        RETURN -1;
END
```

Informationen zu Ausnahmen (EXCEPTIONs)

- Die Details eines Fehlers werden in der Regel benötigt, wenn es darum geht, mit den Fehler umzugehen
- Verwenden Sie GET STACKED DIAGNOSTICS, um Details zurückzugeben

GET STACKED DIAGNOSTICS variable { = | := } item [, ...];

Informationen zu Ausnahmen (EXCEPTIONs)

Diagnostic Item
RETURNED_SQLSTATE
COLUMN_NAME
CONSTRAINT_NAME
PG_DATATYPE_NAME
MESSAGE_TEXT
TABLE_NAME
SCHEMA_NAME
PG_EXCEPTION_DETAIL
PG_EXCEPTION_HINT
PG_EXCEPTION_CONTEXT

Weitergabe von Ausnahmen (EXCEPTIONs)

- Ausnahmen können explizit durch die Funktion ausgelöst werden

```
CREATE OR REPLACE FUNCTION grant_select(p_role varchar)
    RETURNS void AS
$$
DECLARE
sql varchar;
    r record;
    tbl_cursor CURSOR FOR SELECT schemaname, relname
    FROM pg_stat_user_tables;
BEGIN
    IF NOT EXISTS (SELECT 1 FROM pg_roles
                    WHERE rolname = p_role) THEN
        RAISE EXCEPTION 'Invalid Role: %', p_role;
    END IF;
...
```

Ausnahmen (EXCEPTIONs)

- TIPP: Verwenden Sie Ausnahmen nur, wenn es notwendig ist, denn sie haben einen grosse Auswirkung auf die Performance
- Zur Behandlung der Ausnahmen werden Untertransaktionen erstellt.

```
CREATE FUNCTION t1()
RETURNS void AS $$
DECLARE
    i integer;
BEGIN
    i := 1;
END
$$ LANGUAGE plpgsql;
```

```
Avg Time: 0.0017ms
```

```
CREATE FUNCTION t2()
RETURNS void AS $$
DECLARE
    i integer;
BEGIN
    i := 1;
EXCEPTION
    WHEN OTHERS THEN
    RETURN;
END
$$ LANGUAGE plpgsql;
```

```
Avg Time: 0.0032ms
```

Nicht behandelt

- Triggers
- …

Unit-Testing mit PL/pgSQL

Dank an Miles Strässle, Lukas Buchli

Warum überhaupt?

- Unit Testing via ORM ist umständlich
- Triggers und andere komplexere Funktionen sind aber trotzdem fehleranfällig

Anwendungsfälle

- Einzelne Funktionen testen
- Triggers testen

Was gibt es für Möglichkeiten?

1. assert
2. pgTAP
3. PGUnit (original)
4. PGUnit (neue/Neue Version)
5. (pg_regress)

Was ist pgTAP?

- Produziert TAP (Test Anything Protocol)
- Hat eine Menge Testing-Funktionen (über 200!)
- Hat aber auch einige Abhängigkeiten (Perl und IPC::Run)
- Erster Commit 2008
- Immer noch aktiv

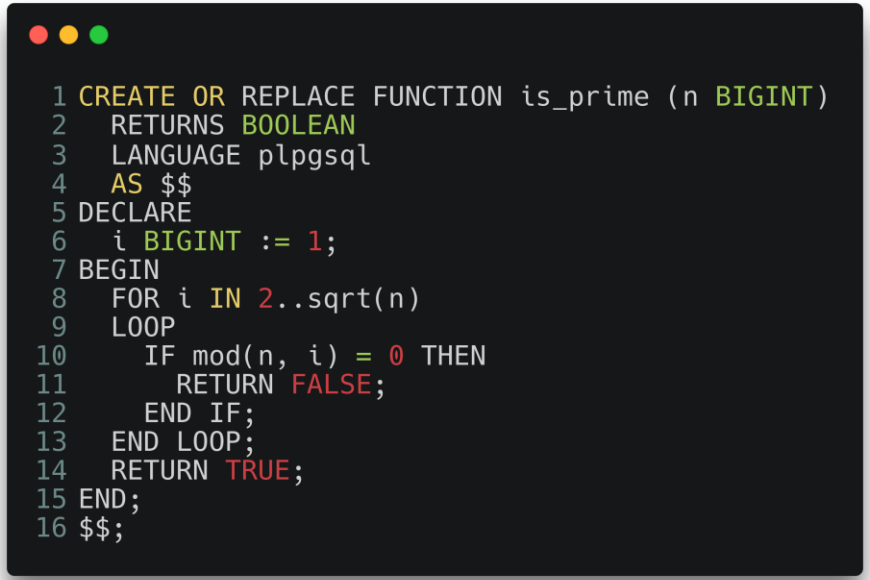
Was ist PGUnit?

- Minimalistisch
  - insgesamt 16 definierte Funktionen
- Ein einzelnes SQL File
  - doch Abhängigkeit von "Additional Supplied Module" dblink
- Erster Commit 2016 von Adrian Andrei
  - Wenig aktiv (letzter Commit Mai 2020)
- Weniger bekannt als pgTAP

Neue Version: Was ist anders?

- Neue PGUnit-Version von Lukas Buchli und Milan Strässle
- Fork im Dezember 2021
- Ziele:
  - Einfachheit
  - Keine Abhängigkeit von dblink
  - "Echte" Extension
  - Momentan 17 definierte Funktionen
  - PGXN https://pgxn.org/search?q=pgunit&in=docs

Ein Anwendungsfall



```
 1 CREATE OR REPLACE FUNCTION is_prime (n BIGINT)
 2   RETURNS BOOLEAN
 3   LANGUAGE plpgsql
 4   AS $$
 5 DECLARE
 6   i BIGINT := 1;
 7 BEGIN
 8   FOR i IN 2..sqrt(n)
 9   LOOP
10     IF mod(n, i) = 0 THEN
11       RETURN FALSE;
12     END IF;
13   END LOOP;
14   RETURN TRUE;
15 END;
16 $$;
```

Ein Anwendungsfall

Der Wikipedia-Artikel sagt:

Eine **Primzahl** (...) ist eine natürliche Zahl, die grösser als 1 und ausschliesslich durch sich selbst und durch 1 teilbar ist.

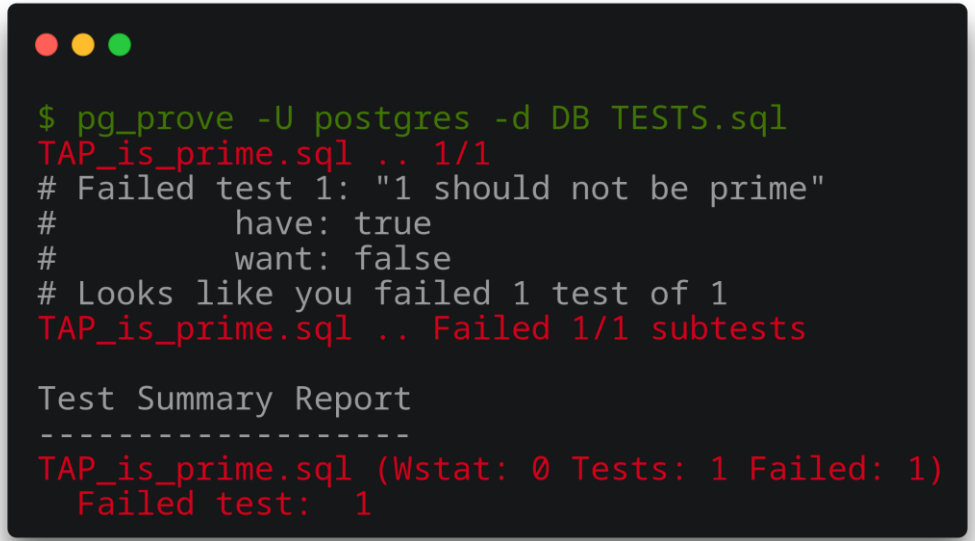
Der Test mit pgTAP

```sql
BEGIN;
SELECT plan(1);
SELECT is(public.is_prime(1)), FALSE, '1 should not be prime');
SELECT * FROM finish();
ROLLBACK;
```

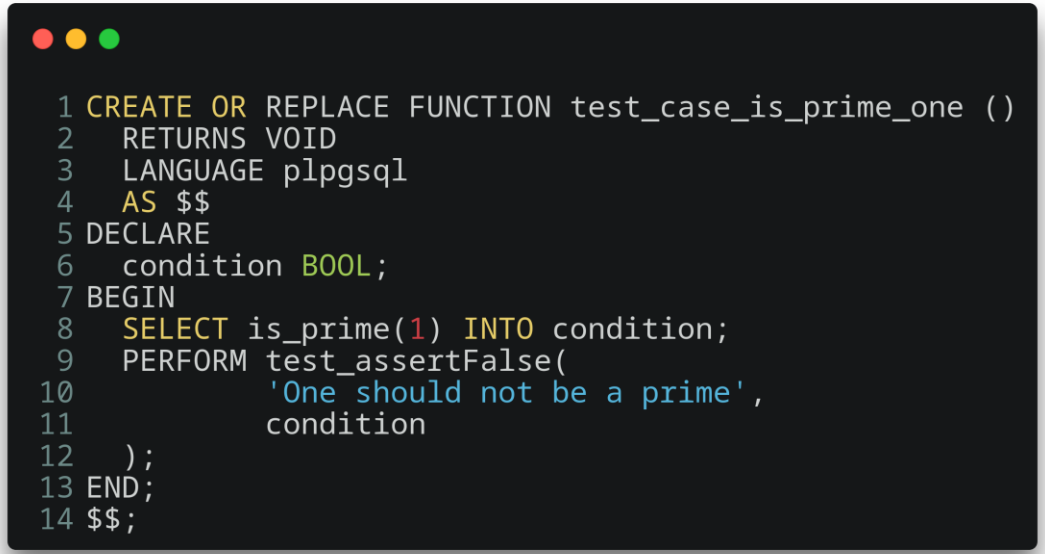
Der Test mit pgTAP

```sql
CREATE OR REPLACE FUNCTION test_is_prime(
) RETURNS SETOF TEXT AS $$
  SELECT is(public.is_prime(1)), FALSE, '1 should not be prime');
END;
$$ LANGUAGE sql;
```

Der Test mit pgTAP

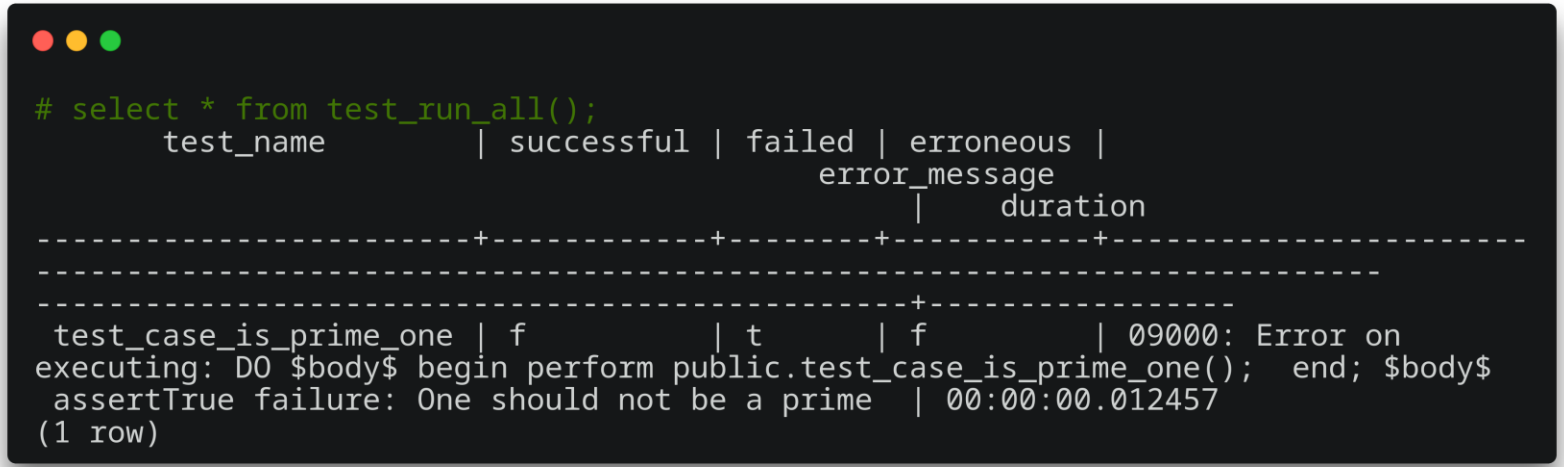


Der Test mit PGUnit (original)



```
CREATE OR REPLACE FUNCTION test_case_is_prime_one ()
  RETURNS VOID
  LANGUAGE plpgsql
  AS $$
DECLARE
  condition BOOL;
BEGIN
  SELECT is_prime(1) INTO condition;
  PERFORM test_assertFalse(
          'One should not be a prime',
          condition
  );
END;
$$;
```

Der Test mit PGUnit (original)

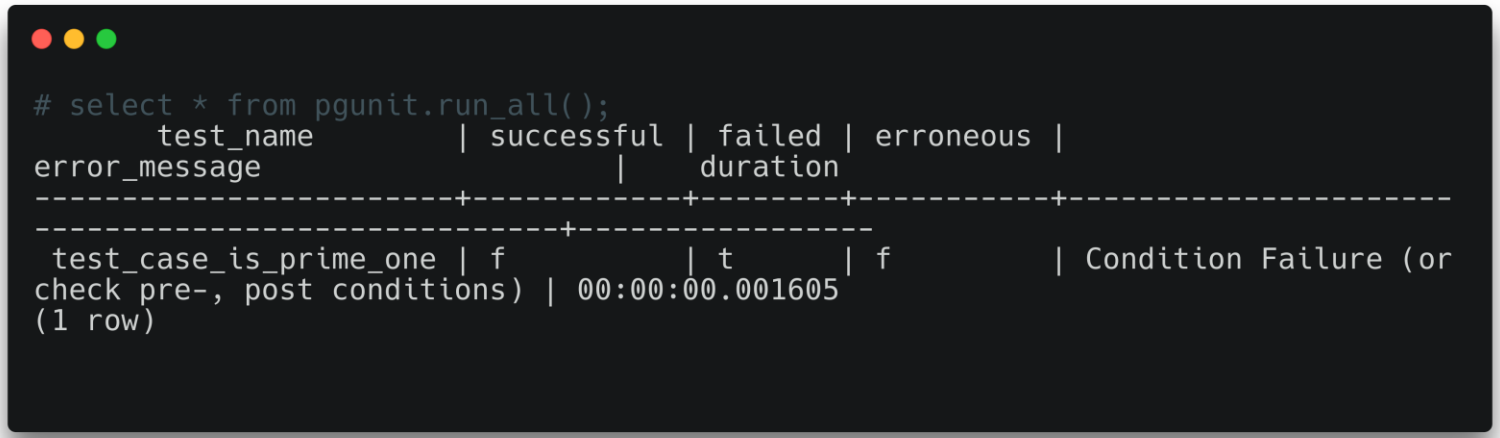


```
# select * from test_run_all();
       test_name        | successful | failed | erroneous |
                                        error_message
                                             |    duration
------------------------+------------+--------+-----------+----------------------
-------------------------------------------------------------------------------
-----------------------------------------------+-----------------
 test_case_is_prime_one | f          | t      | f         | 09000: Error on
executing: DO $body$ begin perform public.test_case_is_prime_one();  end; $body$
 assertTrue failure: One should not be a prime  | 00:00:00.012457
(1 row)
```

Der Test mit PGUnit (Neue Version)

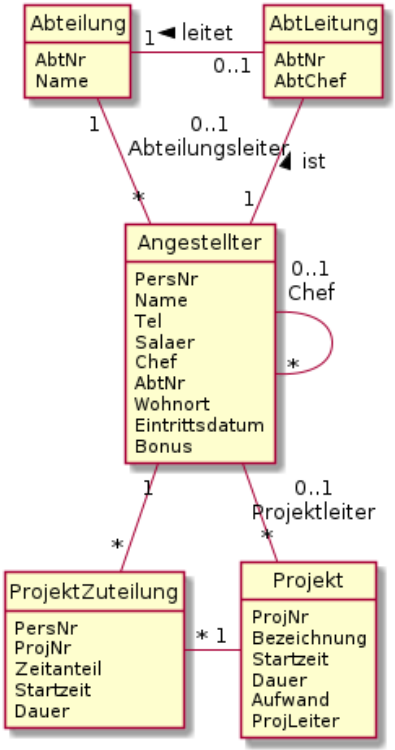
```
 1 CREATE OR REPLACE FUNCTION test_case_is_prime_one ()
 2   RETURNS VOID
 3   LANGUAGE plpgsql
 4   AS $$
 5 DECLARE
 6   condition BOOL;
 7 BEGIN
 8   SELECT is_prime(1) INTO condition;
 9   PERFORM pgunit.assertFalse(
10           'One should not be a prime',
11           condition
12   );
13 END;
14 $$;
```

Der Test mit PGUnit (Neue Version)



```
# select * from pgunit.run_all();
       test_name        | successful | failed | erroneous |
error_message                      |    duration
------------------------+------------+--------+-----------+----------------------
------------------------------+-----------------
 test_case_is_prime_one | f          | t      | f         | Condition Failure (or
check pre-, post conditions) | 00:00:00.001605
(1 row)
```

Ein realistischerer Anwendungsfall



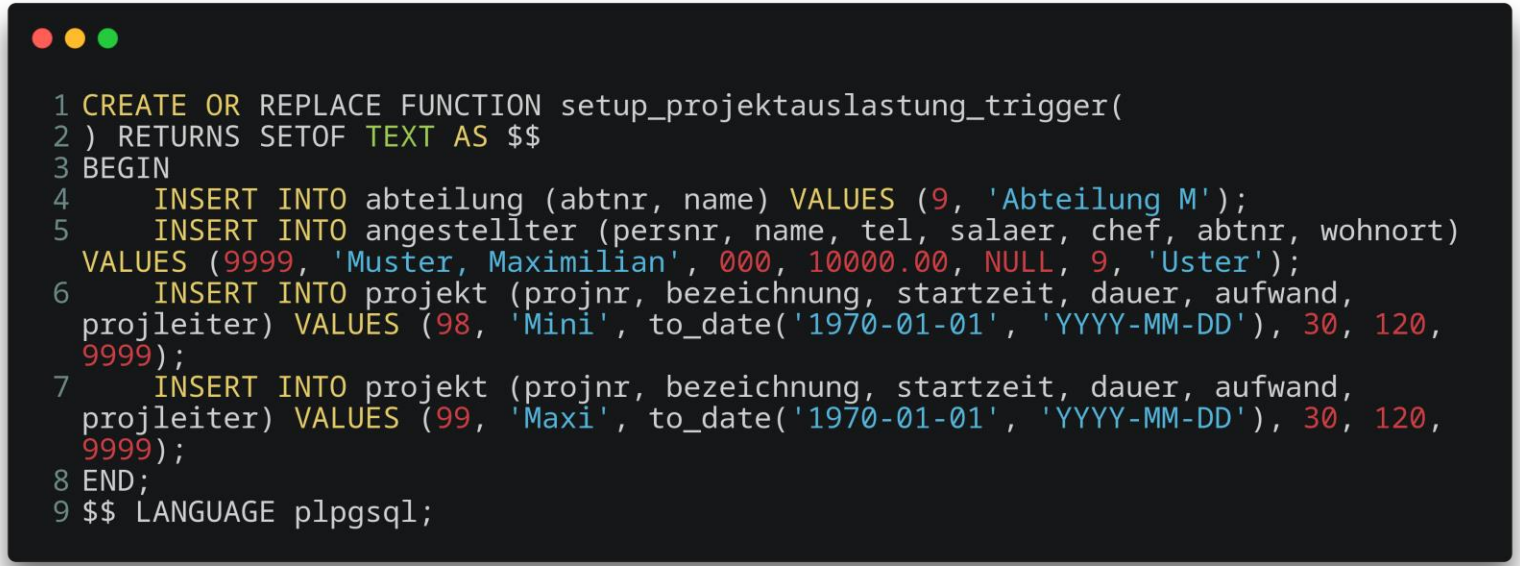
Ein realistischerer Anwendungsfall

```
1 CREATE TRIGGER check_workload
2   BEFORE INSERT OR UPDATE ON projektzuteilung
3   FOR EACH ROW
4   EXECUTE PROCEDURE check_workload();
```

Ein realistischerer Anwendungsfall

```
1 DROP FUNCTION IF EXISTS check_workload () CASCADE;
2 CREATE OR REPLACE FUNCTION check_workload ()
3   RETURNS TRIGGER
4   LANGUAGE plpgsql
5   AS $check_workload$
6 DECLARE
7   totprozant INTEGER;
8   newprozant INTEGER;
9 BEGIN
10  SELECT
11    coalesce(sum(zeitanteil), 0) INTO totprozant
12  FROM
13    projektzuteilung pzt
14  WHERE
15    pzt.persnr = NEW.persnr;
16  IF (TG_OP = 'INSERT') THEN
17    newprozant := NEW.zeitanteil + totprozant;
18  ELSIF (TG_OP = 'UPDATE') THEN
19    newprozant := totprozant - OLD.zeitanteil + NEW.zeitanteil;
20  END IF;
21  RAISE NOTICE 'Projektauslastung neu: %', newprozant;
22  IF (newprozant > 90) THEN
23    RAISE EXCEPTION 'Projektauslastung muss <= 90 Prozent sein.';
24  END IF;
25  RETURN NEW;
26 END;
27 $check_workload$;
```

Der Test mit pgTAP



```sql
CREATE OR REPLACE FUNCTION setup_projektauslastung_trigger(
) RETURNS SETOF TEXT AS $$
BEGIN
    INSERT INTO abteilung (abtnr, name) VALUES (9, 'Abteilung M');
    INSERT INTO angestellter (persnr, name, tel, salaer, chef, abtnr, wohnort) VALUES (9999, 'Muster, Maximilian', 000, 10000.00, NULL, 9, 'Uster');
    INSERT INTO projekt (projnr, bezeichnung, startzeit, dauer, aufwand, projleiter) VALUES (98, 'Mini', to_date('1970-01-01', 'YYYY-MM-DD'), 30, 120, 9999);
    INSERT INTO projekt (projnr, bezeichnung, startzeit, dauer, aufwand, projleiter) VALUES (99, 'Maxi', to_date('1970-01-01', 'YYYY-MM-DD'), 30, 120, 9999);
END;
$$ LANGUAGE plpgsql;
```

Der Test mit pgTAP

```sql
CREATE OR REPLACE FUNCTION test_projektauslastung_trigger(
) RETURNS SETOF TEXT AS $$
    INSERT INTO projektzuteilung (persnr, projnr, zeitanteil) VALUES (9999, 98, 55);
    PREPARE erroneous AS INSERT INTO projektzuteilung (persnr, projnr, zeitanteil) VALUES (9999, 99, 40);
    SELECT throws_ok('erroneous', 'Projektauslastung muss <= 90 Prozent sein.', 'Projektauslastung Mitarbeiter über 90% sollte verhindert werden.');
$$ LANGUAGE sql;
```

Der Test mit pgTAP

```
$ pg_prove -d angproj --runtests
runtests(); .. NOTICE:  Projektauslastung neu: 55
NOTICE:  Projektauslastung neu: 95
runtests(); .. ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.01 usr +  0.00 sys =  0.01 CPU)
Result: PASS
```

Der Test mit PGUnit (original)

```sql
CREATE OR REPLACE FUNCTION test_setup_projektauslastung_trigger(
) RETURNS VOID AS $$
BEGIN
    INSERT INTO abteilung (abtnr, name) VALUES (9, 'Abteilung M');
    INSERT INTO angestellter (persnr, name, tel, salaer, chef, abtnr, wohnort)
VALUES (9999, 'Muster, Maximilian', 000, 10000.00, NULL, 9, 'Uster');
    INSERT INTO projekt (projnr, bezeichnung, startzeit, dauer, aufwand,
projleiter) VALUES (98, 'Mini', to_date('1970-01-01', 'YYYY-MM-DD'), 30, 120,
9999);
    INSERT INTO projekt (projnr, bezeichnung, startzeit, dauer, aufwand,
projleiter) VALUES (99, 'Maxi', to_date('1970-01-01', 'YYYY-MM-DD'), 30, 120,
9999);
END;
$$ LANGUAGE plpgsql;
```

Der Test mit PGUnit (original)

```sql
CREATE OR REPLACE FUNCTION test_case_projektauslastung_trigger(
) RETURNS VOID AS $$
BEGIN
    INSERT INTO projektzuteilung (persnr, projnr, zeitanteil) VALUES (9999, 98, 55);
    BEGIN
        INSERT INTO projektzuteilung (persnr, projnr, zeitanteil) VALUES (9999, 99, 40);
    EXCEPTION
        WHEN raise_exception THEN
            RETURN;
    END;
    PERFORM test_fail('Projektauslastung Mitarbeiter über 90% sollte verhindert werden.');
END;
$$ LANGUAGE plpgsql;
```

Der Test mit PGUnit (Neue Version)

```
$ SELECT * FROM test_run_all();
             test_name              | successful | failed | erroneous |
error_message |    duration
-----------------------------------+------------+--------+-----------+----------
-----+-----------------
 test_case_projektauslastung_trigger | t          | f      | f         | OK
     | 00:00:00.004039
(1 rows)
```

Der Test mit PGUnit (Neue Version)

```sql
CREATE OR REPLACE FUNCTION test_setup_projektauslastung_trigger(
) RETURNS VOID AS $$
BEGIN
    INSERT INTO abteilung (abtnr, name) VALUES (9, 'Abteilung M');
    INSERT INTO angestellter (persnr, name, tel, salaer, chef, abtnr, wohnort)
 VALUES (9999, 'Muster, Maximilian', 000, 10000.00, NULL, 9, 'Uster');
    INSERT INTO projekt (projnr, bezeichnung, startzeit, dauer, aufwand,
 projleiter) VALUES (98, 'Mini', to_date('1970-01-01', 'YYYY-MM-DD'), 30, 120,
 9999);
    INSERT INTO projekt (projnr, bezeichnung, startzeit, dauer, aufwand,
 projleiter) VALUES (99, 'Maxi', to_date('1970-01-01', 'YYYY-MM-DD'), 30, 120,
 9999);
END;
$$ LANGUAGE plpgsql;
```

Der Test mit PGUnit (Neue Version)

```
CREATE OR REPLACE FUNCTION test_case_projektauslastung_trigger(
) RETURNS VOID AS $$
BEGIN
    INSERT INTO projektzuteilung (persnr, projnr, zeitanteil) VALUES (9999, 98, 55);
    BEGIN
        INSERT INTO projektzuteilung (persnr, projnr, zeitanteil) VALUES (9999, 99, 40);
    EXCEPTION
        WHEN raise_exception THEN
            RETURN;
    END;
    PERFORM pgunit.fail('Projektauslastung Mitarbeiter über 90% sollte verhindert werden.');
END;
$$ LANGUAGE plpgsql;
```

Der Test mit PGUnit (Neue Version)

```
$ SELECT * FROM pgunit.run_all();
             test_name              | successful | failed | erroneous |
error_message |    duration
------------------------------------+------------+--------+-----------+----------
-----+-----------------
 test_case_projektauslastung_trigger | t          | f      | f         | OK
     | 00:00:00.000287
(1 row)
```

Wie funktioniert pgTAP?

1. Tests definiert als Funktion oder als SQL Script
   - Ergibt TAP
2. Interpretation Output durch pg_prove

Wie funktioniert PGUnit (original)?

- test_case_…
- Setup, Precondition, Test, Postcondition, Teardown

```
1 SELECT dblink_exec('test_auto', statement)
```

Wie funktioniert PGUnit (Neue Version)?

- test_case_…
- Setup, Precondition, Test, Postcondition, Teardown

```
1 EXECUTE 'DO $body$ BEGIN PERFORM test_case_…(); END; $body$;';
```

Vergleich

pgTAP

- Gigantisch
- Braucht TAP-Consumer
- Viele Funktionen

PGUnit

- Minimalistisch
- Pures PL/pgSQL
- 'Direkter'

Vergleich: Anwendung

pgTAP

- Bei gigantischen Projekten
- Bei Projekten mit speziellen Testbedürfnissen

PGUnit

- Kleinere oder normal grosse Projekte

PL/pgSQL Best Practices

Programmier-Praktiken

- Wenn…
  - man es in SQL machen kann, verwende SQL.
  - Wenn nicht, verwende serverseitige, prozedurale Sprache.
  - Wenn auch das nicht geht, mache es in der Domänenlogik
- Prozedurale Sprachen
  - Code durchgängig einrücken und grosszügig kommentieren
  - Praktiken der Wiederverwendung/Modularität von Code sind anders als in anderen Programmiersprachen
  - Tiefe Aufrufstapel in PL/pgSQL können leistungsintensiv sein

Namenskonventionen

- Erstellen und befolgen Sie eine einheitliche Namenskonvention für Objekte
- PostgreSQL unterscheidet nicht zwischen Gross- und Kleinschreibung, daher funktioniert init cap nicht, verwenden Sie "_", um Wörter im Namen zu trennen
- Stellen Sie allen Parameternamen etwas wie "p_" voran
- Stellen Sie allen Variablennamen etwas wie "v_" voran.

Kontakt

Kontakt

Prof. Stefan Keller
Institut für Software
OST Campus Rapperswil
Oberseestrasse 10
CH-8640 Rapperswil
stefan.keller@ost.ch
www.ost.ch/ifs