POSTGRESQL::

1. **FOR PRINT HELLO WORLD** :

CREATE OR REPLACE FUNCTION HELLOWORLD() RETURNS VARCHAR AS $$

BEGIN

RETURN 'Hello, World!';

END;

$$ LANGUAGE plpgsql;

**FOR RUN:**

SELECT HELLOWORLD();

2) **TAKE NAME FROM USER AND CONCATE WITH HELLO :**

CREATE OR REPLACE FUNCTION HELLO(name VARCHAR) RETURNS VARCHAR AS $$

BEGIN

RETURN ' HELLO, '|| name ||' ! ';

END;

$$ LANGUAGE plpgsql;

SELECT HELLO('Robert'); DROP TABLE students; --');

**notes:**

database.execute("INSERT INTO students (name) VALUES ('" + name + "');");

INSERT INTO students (name) VALUES ('Elaine');

INSERT INTO students (name) VALUES ('Robert'); DROP TABLE students;--');

INSERT INTO students (name) VALUES ('Robert');

DROP TABLE students;

--');

[**https://www.explainxkcd.com/wiki/index.php/Little\_Bobby\_Tables**](https://www.explainxkcd.com/wiki/index.php/Little_Bobby_Tables)

**3)** FILTERING :

format(), quote\_ident(), quote\_literal() or quote\_nullable()

<https://www.postgresql.org/docs/current/static/functions-string.html>

CREATE OR REPLACE FUNCTION hello(name VARCHAR) RETURNS VARCHAR AS $$

BEGIN

RETURN format('Hello , %s!', name);

END;

$$ LANGUAGE plpgsql;

NOTE : HELLO(), HELLO(NAME);

IF YOU NOT PASS PERAMETER THAN IT WILL DISPLAY YOU HELLO()

OTHERWISE HELLO(NAME);

---- FUNCTION OVERLOADING

➔Schemas

➔public

➔Functions

➔hello ()

➔hello (character varying)

TASK 2: MORE PARAMETERS

CREATE OR REPLACE FUNCTION swap(INT, INT, OUT INT, OUT INT) AS

'SELECT $2, $1' LANGUAGE sql;

1. **SWAPING TWO NUMBERS:**

CREATE OR REPLACE FUNCTION swap(INT, INT, OUT INT, OUT INT) AS $body$

BEGIN

$3:= $2;

$4 = $1;

END;

$body$ LANGUAGE plpgsql;

* SELECT \* FROM swap(1, 2);

1. **Task: create a new function that is able to swap these for any other datatype**

CREATE OR REPLACE FUNCTION swap(IN ANYELEMENT, input2 ANYELEMENT, OUT

output1 ANYELEMENT, OUT ANYELEMENT) AS $$

BEGIN

output1 := input2; -- could still be written as $3 = $2

$4 = $1;

END;

$$ LANGUAGE plpgsql;

SELECT \* FROM swap(1.88,3.6); //flot int

SELECT \* FROM swap('test'::TEXT, 'string'); //string data

**NOTE:** IN can be omitted as this is the default

Polymorphic functions:

ANYELEMENT must use the same type(e.g. you can’t mix integers and floating points.

1. **String literature write function as typecast needed**

CREATE FUNCTION make\_array(anyelement, anyelement) RETURNS anyarray AS $$

SELECT ARRAY[$1, $2];

$$ LANGUAGE SQL;

SELECT make\_array(1, 2) AS intarray, make\_array('a'::text, 'b') AS textarray;

intarray | textarray

----------+-----------

{1,2} | {a,b}

(1 row)

SELECT \* FROM swap('test'::TEXT, 'string');

Notes: using input and output parameters and literals for calculations is not always enough and you need local variables.

1. **write a new function twentyone() which**

**• declares two variables “seventeen” and “four”**

**• assigns them the values “17” and “4”**

**• return the sum of them (should be “21”)**

CREATE OR REPLACE FUNCTION twentyone() RETURNS INTEGER AS $$

DECLARE

seventeen INTEGER;

four CONSTANT INT = 4; -- variables can be initialized on declaration

BEGIN

seventeen = 17; -- … or later at usage

RETURN seventeen + four;

END;

$$ LANGUAGE plpgsql;

**Swapping IN and OUT Less parameter**

• there are not only IN and OUT parameter, but also INOUT parameter.

1. **Task: rewrite the swap() function to use only two parameters of type INOUT.**

Syntax**:** CREATE OR REPLACE FUNCTION swap(INOUT ANYELEMENT, INOUT ANYELEMENT) AS 'SELECT $2, $1' LANGUAGE sql;

CREATE OR REPLACE FUNCTION swap(INOUT ANYELEMENT, INOUT ANYELEMENT) AS $$

DECLARE

tmp $1%type; -- its not possible to declare it as ANYELEMENT

-- so make it the same type like the first parameter

BEGIN

tmp = $1;

$1 = $2;

$2 = tmp;

END;

$$ LANGUAGE plpgsql;

1. **write a new function that**

**• takes a name as input parameter**

**• assigns the return value of the function call to twentyone() to a local variable**

**• assigns the value “42” to another variable**

**• swaps the return values of the function calls to hello() and hello(name) and returns its return value**

**• also returns the return value of a function call to swap() with the two local variables**

**• hint: it should return something like “Hello, name!”, “Hello, World!”, 42, 21**

CREATE OR REPLACE FUNCTION combination(name VARCHAR, OUT str1 VARCHAR, OUT str2 VARCHAR, OUT int1 INTEGER, OUT int2 INTEGER) AS $$

DECLARE

twenty\_one CONSTANT INTEGER = twentyone();

forty\_two CONSTANT INTEGER = 42;

BEGIN

-- one could also just declare the first variable and then assign the -- value at this point like

--SELECT \* INTO twenty\_one FROM twentyone();

SELECT \* INTO str1, str2 FROM swap (hello(), hello(name));

SELECT \* INTO int1, int2 FROM swap (twenty\_one, forty\_two);

END;

$$ LANGUAGE plpgsql;

NOTE: the hello(name) function doesn’t return anything, if the name is NULL

•if it is NULL then return the same as the hello() function without parameter does

Exercise 3 Topic 2:

Extensions of SQL

**Control statements**

**IF, THEN, ELSE:**

NOTE: the hello(name) function doesn’t return anything, if the name is NULL

1. **Task: rewrite the function and check the input parameter**

**• if it is NULL then return the same as the hello() function without parameter does**

CREATE OR REPLACE FUNCTION hello(name VARCHAR) RETURNS VARCHAR AS $$

BEGIN

IF name IS NULL THEN

RETURN hello();

**ELSEIF name = '' THEN**

**RETURN hello('Anonymous');**

ELSE

RETURN format('Hello, %s !', name);

END IF;

END;

$$ LANGUAGE plpgsql;

1. Task: rewrite your function to use “Anonymous” as name for empty strings

CREATE OR REPLACE FUNCTION hello(name VARCHAR) RETURNS VARCHAR AS $$

BEGIN

IF name IS NULL THEN

RETURN hello();

**ELSEIF name = '' THEN**

**RETURN hello('Anonymous');**

ELSE

RETURN format('Hello, %s!', name);

END IF;

END;

$$ LANGUAGE plpgsql;

1. **Task: now, also check for “Bob” and use “Robert” instead and change “Bill” to “William”, too**

CREATE OR REPLACE FUNCTION hello(name VARCHAR) RETURNS VARCHAR AS $$

BEGIN

IF name IS NULL THEN

**name = 'World';**

ELSEIF name = ' ' THEN

**name = 'Anonymous';**

**ELSEIF name = 'Bob' THEN //case sensitive**

**name = 'Robert';**

**ELSEIF name = 'Bill' THEN**

**name = 'William';**

END IF;

**RETURN format('Hello, %s!', name);**

END;

$$ LANGUAGE plpgsql;

1. **Task: rewrite it and try to use the CASE control structure instead**

CREATE OR REPLACE FUNCTION hello(name VARCHAR) RETURNS VARCHAR AS $$

BEGIN

**CASE** -- using CASE name WHEN ... WHEN ... is not possible here

**WHEN** name IS NULL THEN -- because you can’t check for NULL with this construct

name = 'World';

**WHEN** name = ' ' THEN

name = 'Anonymous';

**WHEN** name = 'Bob' THEN

name = 'Robert';

**WHEN** name = 'Bill' THEN

name = 'William';

**ELSE** -- without this ELSE only the mentioned cases would work and all other cases would

result in an error

**NULL;**

**END CASE;**

RETURN format('Hello, %s!', name);

END;

$$ LANGUAGE plpgsql;

Notes :: use CASE just in case you want to test something for different values and don’t want to check for

NULL

* else it is more complicated than IF

1. **Task: write a new rating function that**

* **takes an integer as input parameter**
* **and returns a rating string**
* **“poor” for 1, 2 and 3**
* **“average” for 4, 5 and 6**
* **“good” for 7, 8 and 9**
* **“excellent” for 10**

**“out of range” for anything else** and uses the **CASE construct**

CREATE OR REPLACE FUNCTION rating(score INTEGER) RETURNS VARCHAR AS $$

BEGIN

CASE score

WHEN 1, 2, 3 THEN

return 'poor';

WHEN 4, 5, 6 THEN

return 'average';

WHEN 7, 8, 9 THEN

return 'good';

WHEN 10 THEN

return 'excellent';

ELSE

return 'out of range';

END CASE;

END;

$$ LANGUAGE plpgsql;

1. **Task: write a new function that executes this rating function with input values in range from 0 to 20 and use the FOR control construct.**

**CREATE OR REPLACE FUNCTION rating\_test() RETURNS SETOF VARCHAR AS $$**

**BEGIN**

**FOR score IN 0..20 LOOP** -- score is defined as local INTEGER

**RETURN NEXT rating(score);** -- NEXT can return multiple rows

**END LOOP;**

**RETURN;** -- this indicates that there are no more rows

**END;** -- but as we have already reached the END this is not needed

**$$ LANGUAGE plpgsql;**

**12)** Task: now write another function that returns random INTEGER values in a range of 1 to 100 and

exits when the number 42 is returned

•

each number should be returned at its own row

•

hint: the built-in function random() returns DOUBLE PRECISION values in a range of [0.0 .. 1.0)

CREATE OR REPLACE FUNCTION random42() RETURNS SETOF INTEGER AS $$

DECLARE

rnd INTEGER;

BEGIN

LOOP

rnd = ceil(random() \* 100)::INTEGER; -- INTEGER casts are rounding

RETURN NEXT rnd;

EXIT WHEN rnd = 42; -- the same as: IF rnd = 42 THEN EXIT; END IF;

END LOOP; -- also possible: IF rnd = 42 THEN RETURN; END IF;

END;

$$ LANGUAGE plpgsql**;**

**13)** **Task: let’s write another function, that**

**•counts the number of random numbers (number of rows) returned by our last function and**

**•if this count doesn’t equal to 42, increases another counter that indicates how many tries it took to get 42 random numbers**

**•returns this counter**

CREATE OR REPLACE FUNCTION count42() RETURNS INTEGER AS $$

DECLARE

c INTEGER = 1;

BEGIN

WHILE (SELECT COUNT(\*) FROM random42()) != 42 LOOP

c = c + 1; -- there is no c++ or ++c

END LOOP;

RETURN c;

END;

$$ LANGUAGE plpgsql;

**Exercise 8 Topic 2: Extensions of SQL:**

**Triggers**

* Objective: there are two tables for numbers and when something is inserted into the first table, it is added to the second table as well
* create two new tables for INTEGERs named numbers and numbers\_log

CREATE TABLE IF NOT EXISTS numbers (number INTEGER NOT NULL)

CREATE TABLE IF NOT EXISTS numbers\_log (number INTEGER NOT NULL);

(1)Task: write a new trigger function that is copying the value inserted into the table numbers to the table numbers\_log.

**CREATE OR REPLACE FUNCTION numbers\_copy() RETURNS TRIGGER AS $$**

**BEGIN**

**INSERT INTO numbers\_log VALUES (NEW.number);**

**RETURN NULL;** -- result is ignored since this will be an AFTER trigger

**END;**

**$$ LANGUAGE plpgsql;**

(2)Task: add a trigger to the table numbers that fires this trigger function after a new row was

Added.

CREATE TRIGGER numbers\_insert AFTER INSERT ON numbers

FOR EACH ROW

EXECUTE PROCEDURE numbers\_copy();

INSERT INTO numbers VALUES(1);

SELECT \* FROM numbers;

SELECT \* FROM numbers\_log;

insert some more values into the table numbers

INSERT INTO numbers VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9);

and check again if they are there

now delete some values from numbers

DELETE FROM numbers WHERE number < 5;

**Task: now create another trigger function and corresponding trigger, that fires before a new row is added to the table numbers and prevents the insertion when there are already 50 rows.**

CREATE OR REPLACE FUNCTION numbers\_stop() RETURNS TRIGGER AS $$

BEGIN

IF (SELECT COUNT(\*) FROM numbers) >= 50 THEN

RETURN NULL; --do nothing and stop all further trigger from firing

END IF;

RETURN NEW; --return it as this is what is inserted

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER numbers\_insert\_before BEFORE INSERT ON numbers

FOR EACH ROW

EXECUTE PROCEDURE numbers\_stop();

**Task: write one more trigger function and trigger, that fires right after a new row is added and inserts another number to numbers, that is one bigger than the last one.**

CREATE OR REPLACE FUNCTION numbers\_onebig\_number() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO numbers VALUES (NEW.number +1);

RETURN NULL; --do nothing and stop all further trigger from firing.

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER numbers\_insert\_onebig\_number\_after AFTER INSERT ON numbers

FOR EACH ROW

EXECUTE PROCEDURE numbers\_onebig\_number();

your second trigger should prevent your third trigger to insert more than 50 rows

• truncate your table to be able to insert and test more.

TRUNCATE TABLE numbers;

INSERT INTO numbers VALUES (1), (100);

• to see, that the execution of the function for the first value just fires a new trigger and so on, so that the

function is never executed for the second value

• you can also see the sequence of insertion in the second table numbers\_log

**Triggers Manipulation:**

**Task**:

**create another trigger function and corresponding trigger, that fires before a new row is added to the table numbers and manipulate the input data, by just doubling its value.**

CREATE OR REPLACE FUNCTION numbers\_double() RETURNS TRIGGER AS $$

BEGIN

NEW.number=NEW.number \* 2; -- MANIPULATE DATA

RETURN NEW; -- RETURNING VALUE

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER numbers\_insert\_before\_double BEFORE INSERT ON numbers

FOR EACH ROW

EXECUTE PROCEDURE numbers\_double();

**Triggers Recursion:**

* to test whether your function is working, you can just drop the self firing trigger.

unfortunately 2^50 is out of INTEGER range that only goes up to 2^32 and the DBMS stops the

* execution with an error

**DROP TRIGGER IF EXISTS numbers\_insert\_after ON numbers;**

the main problem is, that the increment trigger is firing itself and we run into this recursive trigger.

**•Task: re-enable the trigger, but rewrite it to avoid recursion.**

CREATE OR REPLACE FUNCTION numbers\_more() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO numbers VALUES (NEW.number + 1);

RETURN NULL;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER numbers\_insert\_after AFTER INSERT ON numbers

FOR EACH ROW

WHEN (pg\_trigger\_depth() = 0)

EXECUTE PROCEDURE numbers\_more();

**Triggers More logging**

create a new table named numbers\_log\_query

**CREATE TABLE IF NOT EXISTS numbers\_log\_query (query TEXT NOT NULL);**

**Task: write a new trigger function and corresponding trigger, that is logging the query, which was**

**performed on the table numbers, to the table numbers\_log\_query.**

CREATE OR REPLACE FUNCTION numbers\_query() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO numbers\_log\_query VALUES (current\_query());

RETURN NULL;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER numbers\_alter AFTER INSERT OR UPDATE OR DELETE OR TRUNCATE ON numbers

FOR EACH STATEMENT -- could be omitted, as this is the default

EXECUTE PROCEDURE numbers\_query();

* by playing around with the table numbers you can see, that current\_query() is not

updated for the queries fired by your other triggers and therefore you get a copy of the query for

each value you insert, as this triggers another INSERT in numbers\_more().

• **for example**

**INSERT INTO numbers VALUES (1), (100);**

• will be logged three times, as it is logged once for the main query and two times more, as there are inserted another two values

• obviously this is another recursive trigger, that we could easily avoid

**Task: rewrite the trigger to avoid recursion.**

CREATE OR REPLACE FUNCTION numbers\_query() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO numbers\_log\_query VALUES (current\_query());

RETURN NULL;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER numbers\_alter AFTER INSERT OR UPDATE OR DELETE OR TRUNCATE ON numbers

FOR EACH STATEMENT

WHEN (pg\_trigger\_depth() = 0)

EXECUTE PROCEDURE numbers\_query();

DROP TRIGGER IF EXISTS numbers\_alter ON numbers;

**Task: use triggers to prevent this, so that one can only insert something to numbers\_log and**

**numbers\_log\_query**

CREATE OR REPLACE FUNCTION numbers\_copy() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO numbers\_log VALUES (NEW.number);

RETURN NULL; -- result is ignored since this will be an AFTER trigger

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER numbers\_insert AFTER INSERT ON numbers

FOR EACH ROW

EXECUTE PROCEDURE numbers\_copy();

**Triggers Exceptional:**

Pending……

**Exercise 11:**

**Object-Relational Database Systems ::**

**Row types:**

**1 Task: create a new table persons with the attributes name, address, email, telephone and date of birth**

**• the name consists of forename and surname**

**• the address consists of street, city and state**

**• the street consists of streetname and housenumber**

**• the city consists of cityname and postalcode**

**• use date-type for dateofbirth and some string-type for all other attributes or sub-attributes (there might be basic conventions on several attributes, but they differ from country to country and even house-numbers can contain letters)**

• in PostgreSQL you can’t declare tables with anonymous composite types as presented in the lecture,

so you have to create structured user-defined types at first.

CREATE TYPE nametype AS (forename VARCHAR, surname VARCHAR);

CREATE TYPE streettype AS (streetname VARCHAR, housenumber VARCHAR);

CREATE TYPE citytype AS (cityname VARCHAR, postalcode VARCHAR);

CREATE TYPE addresstype AS (street streettype, city citytype, state VARCHAR);

CREATE TABLE persons (name nametype, address addresstype, email VARCHAR, telephone VARCHAR, dateofbirth DATE);

**2. Task: create an own type for email and telephone, too and change the table definition to use the new types.**

**Domains** are useful for abstracting common constraints on fields into a single location for maintenance. For example, several tables might contain email address columns, all requiring the same CHECK constraint to verify the address syntax. Define a domain rather than setting up each table's constraint individually.

CREATE DOMAIN emailtype AS VARCHAR;

CREATE DOMAIN telephonetype AS VARCHAR;

ALTER TABLE persons

ALTER email TYPE emailtype,

ALTER telephone TYPE telephonetype;

the syntax for creating distinct types is a little bit different than creating structured types, but basically DOMAINs are types, too (with the option to define constraints for this special alias type).

**3.Task: let’s use this possibility and change the definition of emailtype to check for one “@”-sign, which is not at the beginning or the end.**

Distinct Types Constraints

**ALTER DOMAIN emailtype**

**ADD CHECK (value ~ '^[^@]+@[^@]+$');**

**4.Task: insert the data of the following people into the table persons.**

Max Mustermann, Musterstraße 42, 12345 Musterstadt, Germany, +49 1234 56789, Max@Mustermann.de, 1965 04 03 • John Doe, 21 Main Street, Anytown 12345-6789, United States of America, +1-800-555-1234, John@Doe.com, 1976 05 04

INSERT INTO persons VALUES

(ROW('Max', 'Mustermann'),

ROW(ROW('Musterstraße', '42'), ROW('Musterstadt', '12345'), 'Germany'), 'Max@Mustermann.de', '+49 1234 56789', '1965-04-03'),

(('John', 'Doe'), (('Main Street', '21'), ('Anytown', '12345-6789'), 'United States of America'), 'John@Doe.com', '+1-800-555-1234', '1976-05-04');

**5.Task: Max Mustermann has moved, so his address has to be changed to Mustergasse 42a, 01234 Musterdorf.**

Row Types Updated:

UPDATE persons

SET address = (('Mustergasse', '42a'), ('Musterdorf', '01234'), 'Germany')

WHERE name = ROW('Max', 'Mustermann')::nametype;

**6. Task: Max and John have wives, which share their husbands address and telephone-number, but have their own email-addresses, that follow the same pattern as their husbands, so add them to the table, too, but try not to retype the addresses and telephone-numbers in your SQL-query**

**• Erika Mustermann, 1966 02 01**

**• Jane Doe, 1977 08 09**

Row Types Copied

INSERT INTO persons

VALUES (ROW('Erika', 'Mustermann'),

(SELECT address FROM persons WHERE (name).forename = 'Max'),

'Erika@Mustermann.de', (SELECT telephone FROM persons WHERE (name).forename = 'Max'), '1966-02-01'),

(('Jane', 'Doe'), (SELECT address FROM persons WHERE (name).surname = 'Doe'),

'Jane@Doe.com', (SELECT telephone FROM persons WHERE (name).surname = 'Doe'), '1977-08-09');

Exercise 12:

**1. Task: having the date of birth is nice, but often you need the age, so add a method that returns the age in years as INTEGER**

CREATE OR REPLACE FUNCTION age(person persons) RETURNS INTEGER AS $$

BEGIN

RETURN EXTRACT(YEAR from AGE(person.dateofbirth));

END;

$$ LANGUAGE plpgsql;

**Note:: 1.**functionname(objectname); **2.(**objectname).functionname;

**2. Task: write a function that uses this syntax to return the age of a person, that is given by its name as parameter.**

CREATE OR REPLACE FUNCTION getAge(name nametype) RETURNS INTEGER AS $$ DECLARE person persons; BEGIN SELECT \* INTO person FROM persons WHERE persons.name = getAge.name; RETURN (person).age; END; $$ LANGUAGE plpgsql;

**SELECT (('Max', 'Mustermann')).getAge;**

**3. Task: the Doe’s got a new job abroad and therefore are moving, too, so change their address to 1 Rue Vincent d’Indy, 59650 Villeneuve-d’Ascq, France and their telephone number to +33 3 33 33 33 33**

Inheritance

UPDATE persons

SET address = (('Rue Vincent d''Indy', '1'), ('Villeneuve-d''Ascq', '59650'), 'France'), telephone = '+33 3 33 33 33 33'

WHERE (name).surname = 'Doe';

**They lecture at Université de Lille**

**• Jane is associate professor and earns 4000 EUR per month**

**• John is assistant professor and earns 3000 EUR per month**

**4. Task: create a new table for professors that inherits everything from persons and in addition stores**

**their university, rank and salary**

**• finally, add the Doe’s information to the new table**

inheritance Moved

**CREATE TABLE professors (university VARCHAR, rank VARCHAR, salary MONEY) INHERITS (persons);**

**WITH moved AS (DELETE FROM persons WHERE name = ('Jane', 'Doe')::nametype RETURNING \*) INSERT INTO professors SELECT name, address, email, telephone, dateofbirth, 'Université de Lille', 'associate professor', 4000 FROM moved;**

**WITH moved AS (DELETE FROM persons WHERE name = ('John', 'Doe')::nametype RETURNING \*) INSERT INTO professors SELECT name, address, email, telephone, dateofbirth, 'Université de Lille', 'assistant professor', 3000 FROM moved;**

Inheritance Removed

**DELETE FROM persons WHERE name = ('John', 'Doe')::nametype;**

as you can see, John is also removed from professors ,the same could be achieved by

**DELETE FROM professors WHERE name = ('John', 'Doe')::nametype;**

to keep Jane in persons and remove her ONLY from professors you have to write something like

**DELETE FROM ONLY professors WHERE name = ('Jane', 'Doe')::nametype;**

Inheritance Typed

PostgreSQL doesn’t support type inheritance but only table inheritance • therefore you have to create a table for the type first.

CREATE TABLE nametable OF nametype;