Data Base Systems

VU 184.686, WS 2020

PL/pgSQL

Anela Lolić

Institute of Logic and Computation, TU Wien

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Acknowledgements

The slides are based on the slides (in German) of Sebastian Skritek.

More information to this topic can be found here:
`https://www.postgresql.org/docs/current/static/plpgsql.html`

PL/pgSQL

1. Introduction
2. Structure of PL/pgSQL Programs
3. Variables
4. Expressions
5. Control Structures
6. Cursors

Overview

1. Introduction
   - 1.1 Why procedural DB-Languages?
   - 1.2 PostgreSQL
   - 1.3 Remarks

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures

6. Cursors

Overview

1. Introduction

1.1 Why procedural DB-Languages?

1.2 PostgreSQL

1.3 Remarks

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures

6. Cursors

SQL User Defined Functions

SQL allows for the definition of functions:

SQL User Defined Functions

SQL allows for the definition of functions:

Example

```
CREATE FUNCTION newSemester() RETURNS void
AS $$
  UPDATE students SET sem=sem+1;
  ...
$$ LANGUAGE SQL;
```

SQL User Defined Functions

SQL allows for the definition of functions:

Example

```
CREATE FUNCTION newSemester() RETURNS void
AS $$
  UPDATE students SET sem=sem+1;
  ...
$$ LANGUAGE SQL;
```

call via:

```
SELECT newSemester();
```

SQL vs. Procedural Languages

SQL:

- declarative programming languages (*what*, not *how*)

SQL vs. Procedural Languages

SQL:

- declarative programming languages (*what*, not *how*)
- advantages: better possibilities for optimization, compact, . . .

SQL vs. Procedural Languages

SQL:

- declarative programming languages (*what*, not *how*)
- advantages: better possibilities for optimization, compact, . . .
- disadvantages: restricted expressive power; “typical” constructs from programming languages would be nice to have

SQL vs. Procedural Languages

SQL:

- declarative programming languages (*what*, not *how*)
- advantages: better possibilities for optimization, compact, . . .
- disadvantages: restricted expressive power; “typical” constructs from programming languages would be nice to have

⇒ almost all DBMS offer **procedural DB programming languages**

Why procedural DB-Languages?

- performance:
  - less “back and forth” that is actually not needed between client and DB
  - (network-) traffic is saved
  - multiple parsing of one query can be avoided
- application logic in central location
- exact access control possible

Overview

1. Introduction

1.1 Why procedural DB-Languages?

1.2 PostgreSQL

1.3 Remarks

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures

6. Cursors

PostgreSQL

- PostgreSQL allows *user-defined functions* in arbitrary languages
- source code treated as text by PostgreSQL and passed on to corresponding adapter of programming language
- 4 languages supported by default:
  PL/pgSQL, PL/Tcl, PL/Perl, PL/Python
- additional languages may be installed
- in this lecture: PL/pgSQL
  - very similar to PL/SQL (Oracle)

Overview

1. Introduction
1.1 Why procedural DB-Languages?
1.2 PostgreSQL
1.3 Remarks

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures

6. Cursors

PL/pgSQL Coding Parts in the Lecture

- in most cases code is represented on the slides only in some parts (only the “essential” parts)
- as parts are missing code might not be running

Sources

- PostgreSQL online documentation: `https://www.postgresql.org/docs/current/`
- slides provide only an overview, details can be found in the online documentation

Overview

1. Introduction

2. Structure of PL/pgSQL Programs
2.1 Definition of User-Defined Functions
2.2 PL/pgSQL Blocks

3. Variables

4. Expressions

5. Control Structures

6. Cursors

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

2.1 Definition of User-Defined Functions

2.2 PL/pgSQL Blocks

3. Variables

4. Expressions

5. Control Structures

6. Cursors

User-Defined Functions (PostgreSQL)

```
CREATE [OR REPLACE] FUNCTION
  name ([ [argname] argtype [, ...] ])
   [ RETURNS rettype
    |RETURNS TABLE (colname coltype [, ...])]
AS $$
  ...    – actual source code
$$ LANGUAGE plpgsql;
```

- source code passed as text
- frequently we will skip this “frame”

Arguments and Return Values

arguments:

- 4 kinds of arguments: `IN`, `OUT`, `INOUT`, `VARIADIC`
- name is optional
    - access via `$i` notation possible
- possibility to define default values

Arguments and Return Values

arguments:

- 4 kinds of arguments: `IN`, `OUT`, `INOUT`, `VARIADIC`
- name is optional
  - access via `$i` notation possible
- possibility to define default values

return values

- type of return value has to be stated
  - explicitly via `RETURNS`
  - implicitly via `OUT`/`INOUT` parameters
- in case no values are returned: `RETURNS void`
- tables (`RETURNS TABLE`) or sets of values (`SETOF`) can be returned

Arguments and Return Values: Examples

Example

```
CREATE FUNCTION sum (integer, integer)
  RETURNS integer AS $$
    ... $1 + $2  ... $$ LANGUAGE plpgsql;
```

Arguments and Return Values: Examples

Example

```
CREATE FUNCTION sum (integer, integer)
 RETURNS integer AS $$
  ... $1 + $2  ... $$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION nsum (IN a integer,
 IN b integer, OUT c integer) AS $$
  ... c = a + b ... $$ LANGUAGE plpgsql;
```

Arguments and Return Values: Examples

Example

```
CREATE FUNCTION sum (integer, integer)
  RETURNS integer AS $$
    ... $1 + $2  ... $$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION nsum (IN a integer,
  IN b integer, OUT c integer) AS $$
   ... c = a + b ... $$ LANGUAGE plpgsql;
```

```
SELECT nsum(2,3);
```

Arguments and Return Values: Examples

Example

```
CREATE FUNCTION nsumd (IN a integer = 1,
  IN b integer DEFAULT 1, OUT c integer)
  AS $$
    ... c = a + b ...
  $$ LANGUAGE plpgsql;
```

Arguments and Return Values: Examples

Example

```
CREATE FUNCTION nsumd (IN a integer = 1,
  IN b integer DEFAULT 1, OUT c integer)
  AS $$
    ... c = a + b ...
  $$ LANGUAGE plpgsql;
```

```
SELECT nsumd();
```

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

2.1 Definition of User-Defined Functions

2.2 PL/pgSQL Blocks

3. Variables

4. Expressions

5. Control Structures

6. Cursors

PL/pgSQL Programs are Structured in Blocks

Example (minimal example for sum)

```
CREATE FUNCTION nsum (IN a integer,
  IN b integer, OUT c integer) AS $$
   BEGIN
    c = a + b;
  END; $$ LANGUAGE plpgsql;
```

Structure of a PL/pgSQL Block

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
  excpthandling ]
END [ label ];
```

Structure of a PL/pgSQL Block

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
  excpthandling ]
END [ label ];
```

`label` assigns a name to a block (optional)

Structure of a PL/pgSQL Block

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
  excpthandling ]
END [ label ];
```

`label` assigns a name to a block (optional)

`DECLARE` contains definition of local variables (optional)

Structure of a PL/pgSQL Block

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
  excpthandling ]
END [ label ];
```

`label` assigns a name to a block (optional)

`DECLARE` contains definition of local variables (optional)

`BEGIN` contains program logic

Structure of a PL/pgSQL Block

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
  excpthandling ]
END [ label ];
```

label assigns a name to a block (optional)

DECLARE contains definition of local variables (optional)

BEGIN contains program logic

EXCEPTION contains error handling (optional)

Elements of a PL/pgSQL Block

- blocks can be nested arbitrarily

Elements of a PL/pgSQL Block

- blocks can be nested arbitrarily
- smallest possible block:

```
BEGIN
END;
```

resp.

```
BEGIN
  NULL;  -- Oracle compatibility
END;
```

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

3.1 Definition

3.2 Special Variable Types

3.3 Variable Substitutions

4. Expressions

5. Control Structures

6. Cursors

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

3.1 Definition

3.2 Special Variable Types

3.3 Variable Substitutions

4. Expressions

5. Control Structures

6. Cursors

Variable Definition

- variables can have arbitrary SQL type
- all variables have to be declared in DECLARE section
- syntax:

```
name [CONSTANT] type [NOT NULL] [{DEFAULT|=} expression];
```

Variable Definition

- variables can have arbitrary SQL type
- all variables have to be declared in DECLARE section
- syntax:

```
name [CONSTANT] type [NOT NULL] [{DEFAULT|=} expression];
```

Example (variable declarations)

```
  matrNr integer;
  input1 ALIAS FOR $1;
  nameNew ALIAS FOR nameOld;
  grade integer NOT NULL = 1;
  rank CONSTANT varchar(2) = 'C4';
```

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

3.1 Definition

3.2 Special Variable Types

3.3 Variable Substitutions

4. Expressions

5. Control Structures

6. Cursors

Special Variable Types

- copying types
- row types
- record types

Copying Types

- allows copying type of another variable or of a table column

Example

```
matrNr students.matrNr%TYPE;
lec lectures.title%TYPE;
current lec%TYPE;
```

Copying Types

- allows copying type of another variable or of a table column

Example

```
matrNr students.matrNr%TYPE;
lec lectures.title%TYPE;
current lec%TYPE;
```

advantages:

- type need not be known
- when the type changes the code does not necessarily have to be changed

Row Types

- variable composed out of several fields (*composite type*)
- might contain whole row of a relation
- fixed structure
- single fields are accessed via .-notation: `rowvar.field`
- typical application:
  `name table_name%ROWTYPE`

Example

```
student students%ROWTYPE;
```

Record Types

similar to row type, but:

- structure not fixed
- structure is taken over dynamically, “reusable”
- use: `name RECORD`

Example

```
result RECORD;
SELECT * INTO result FROM students ...
SELECT * INTO result FROM professors ...
```

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables
3.1 Definition
3.2 Special Variable Types
3.3 Variable Substitutions

4. Expressions

5. Control Structures

6. Cursors

Substitutions of Variables in Expressions

- **problem:** string might denote variables or tables

Example

```
matrNr students.matrNr%TYPE;
SELECT matrNr INTO matrNr FROM students
  WHERE matrNr=matrNr;
```

Substitutions of Variables in Expressions

- **problem:** string might denote variables or tables

Example

```
matrNr students.matrNr%TYPE;
SELECT matrNr INTO matrNr FROM students
  WHERE matrNr=matrNr;
```

- no problem in case it is unique based on syntax
- if not unique: error displayed (default)

Substitutions of Variables in Expressions

- **problem:** string might denote variables or tables

Example

```
matrNr students.matrNr%TYPE;
SELECT matrNr INTO matrNr FROM students
  WHERE matrNr=matrNr;
```

- no problem in case it is unique based on syntax
- if not unique: error displayed (default)
- **solution:** avoidance
  - qualified names (`students.matrNr`)
  - corresponding name convention for variables

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

4.1 Simple Expressions

4.2 (SQL) Commands without Result (Result Set)

4.3 (SQL) Commands with Single-Lined Result

5. Control Structures

6. Cursors

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

4.1 Simple Expressions

4.2 (SQL) Commands without Result (Result Set)

4.3 (SQL) Commands with Single-Lined Result

5. Control Structures

6. Cursors

Value Assignments, Comparisons, Simple Expressions

- assignments: := or =
- operators in expressions: as in SQL
  - arithmetical operators: +, −, ∗, /
  - comparison operators: =, >, <, >=, <= not equal: != or <>
  - logical operators: AND, OR, NOT
  - string comparisons: LIKE, NOT LIKE (wildcards: %, _)
  - string concatenation: ||
  - further SQL-operations: IS NULL, IS NOT NULL
    x BETWEEN a AND b, x IN (1,2,3)
  - ...

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

4.1 Simple Expressions

4.2 (SQL) Commands without Result (Result Set)

4.3 (SQL) Commands with Single-Lined Result

5. Control Structures

6. Cursors

SQL Commands without Result

SQL commands that have no result (*result set*) can be called as usual

- for example INSERT, UPDATE, . . . (without RETURNING)
- but not SELECT

SQL Commands without Result

SQL commands that have no result (*result set*) can be called as usual

- for example INSERT, UPDATE, . . . (without RETURNING)
- but not SELECT

Example

```
... FUNCTION login(integer, integer)...
...
BEGIN
 INSERT INTO attend VALUES ($1, $2);
END; ...
```

PERFORM – Discard Results

PERFORM allows execution of commands/queries and the immediate discard of the result set

```
PERFORM statement
```

PERFORM – Discard Results

PERFORM allows execution of commands/queries and the immediate discard of the result set

```
PERFORM statement
```

- for example execution of queries or functions with side effects
- SELECT queries: replace SELECT by PERFORM

Example

```
PERFORM newSemester();
PERFORM pg_sleep(2);
```

PERFORM – Discard Results

PERFORM allows execution of commands/queries and the immediate discard of the result set

```
PERFORM statement
```

- for example execution of queries or functions with side effects
- SELECT queries: replace SELECT by PERFORM

Example

```
PERFORM newSemester();
PERFORM pg_sleep(2);
```

```
PERFORM * FROM students;
```

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

4.1 Simple Expressions

4.2 (SQL) Commands without Result (Result Set)

4.3 (SQL) Commands with Single-Lined Result

5. Control Structures

6. Cursors

INTO

INTO loads results of a SQL command/query into a RECORD or ROWTYPE variable

```
SELECT expr INTO [STRICT] target FROM ...;
INSERT...RETURNING expr INTO [STRICT] target;
```

INTO

INTO loads results of a SQL command/query into a RECORD or ROWTYPE variable

```
SELECT expr INTO [STRICT] target FROM ...;
INSERT...RETURNING expr INTO [STRICT] target;
```

- with STRICT: query has to return exactly one row
- without STRICT: also results with more or less than one row are permitted
  - no line: (values in) target are set to NULL
  - more than one row: “first” row is returned

Execute Dynamic SQL Commands

EXECUTE executes a (dynamically constructed) SQL command; single-lined results can be written into a variable via INTO

```
EXECUTE cmd [INTO [STRICT] var] [USING expr];
```

Execute Dynamic SQL Commands

EXECUTE executes a (dynamically constructed) SQL command; single-lined results can be written into a variable via INTO

```
EXECUTE cmd [INTO [STRICT] var] [USING expr];
```

- no caching of query plan
- parameters can be used only for data values

Execute Dynamic SQL Commands

EXECUTE executes a (dynamically constructed) SQL command; single-lined results can be written into a variable via INTO

```
EXECUTE cmd [INTO [STRICT] var] [USING expr];
```

- no caching of query plan
- parameters can be used only for data values

Example

```
  EXECUTE format('SELECT count(*) FROM %I '
   'WHERE sem > $1', tabname)
   INTO c USING v_sem;
```

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures

5.1 Return Values of a Function

5.2 Conditionals

5.3 Loops

5.4 Error Handling

6. Cursors

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures

5.1 Return Values of a Function

5.2 Conditionals

5.3 Loops

5.4 Error Handling

6. Cursors

Return Values of a Function

- `OUT` and `INOUT` variables:
  - return values that are present at function end

Return Values of a Function

- `OUT` and `INOUT` variables:
  - return values that are present at function end
- `RETURN` *expr* ;
  - ends function, returns value of *expr*
  - only `RETURN;` for type `void` or `OUT`/`INOUT` variables

Return Values of a Function

- `OUT` and `INOUT` variables:
  - return values that are present at function end
- `RETURN` *expr* ;
  - ends function, returns value of *expr*
  - only `RETURN;` for type `void` or `OUT`/`INOUT` variables
- `RETURN NEXT` *expr* ;
  `RETURN QUERY` *query* ;
  - extends result by corresponding value
  - does not end the function
  - variant with `EXECUTE` exists for `RETURN QUERY`

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures

5.1 Return Values of a Function

5.2 Conditionals

5.3 Loops

5.4 Error Handling

6. Cursors

IF-THEN-ELSE

```
IF boolean-expr THEN
    statements
[ ELSIF boolean-expr THEN
    statements
[ ELSIF boolean-expr THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

CASE

"Simple CASE"

```
CASE search-expr
 WHEN expr [, expr...] THEN
     statements
 [WHEN expr [, expr...] THEN
     statements
     ... ]
 [ELSE
     statements ]
END CASE;
```

"Searched CASE"

```
CASE
 WHEN boolean-expr THEN
   statements
 [WHEN boolean-expr THEN
   statements
    ... ]
 [ELSE
   statements ]
END CASE;
```

CASE

"Simple CASE"

```
CASE search-expr
 WHEN expr[,expr...] THEN
     statements
 [WHEN expr[,expr...] THEN
     statements
     ... ]
 [ELSE
     statements ]
END CASE;
```

"Searched CASE"

```
CASE
 WHEN boolean-expr THEN
    statements
 [WHEN boolean-expr THEN
    statements
     ... ]
 [ELSE
    statements ]
END CASE;
```

- first appropriate WHEN is executed
- other WHEN are skipped
- if a buggy ELSE is reached an error occurs

Example: Simple vs. Searched CASE

Example (simple CASE – searched CASE)

```
CASE grade
 WHEN 1 THEN
  txt = ’Tutor?’;
 WHEN 2,3,4 THEN
  txt = ’Positive’;
 ELSE
  txt = ’Negative’;
END CASE;
```

```
CASE
 WHEN grade = 1 THEN
  txt = ’Tutor?’;
 WHEN grade BETWEEN
        2 AND 4 THEN
  txt = ’Positive’;
 ELSE
  txt = ’Negative’;
END CASE;
```

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures
5.1 Return Values of a Function
5.2 Conditionals
5.3 Loops
5.4 Error Handling

6. Cursors

Infinite Loop LOOP ...END LOOP;

```
[ <<label>> ]
LOOP
  statements
END LOOP;
```

- infinite loop; explicit exit via RETURN or EXIT

More Loops

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

More Loops

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

```
[ <<label>> ]
FOR name IN [REVERSE]expr...expr[BY expr]LOOP
  statements
END LOOP [ label ];
```

Control Commands for Loops: EXIT and CONTINUE

EXIT process of loop is cancelled; control flow continues after corresponding END LOOP;

CONTINUE current iteration in loop is cancelled; next iteration of corresponding loop starts

syntax:

```
EXIT/CONTINUE [label] [WHEN boolean-expr];
```

Iterating Query Results

```
[ <<label>> ]
FOR target IN query LOOP
  statements
END LOOP [ label ];
```

- internally realized as cursor

Iterating Query Results

```
[ <<label>> ]
FOR target IN query LOOP
  statements
END LOOP [ label ];
```

- internally realized as cursor

Example

```
FOR s IN SELECT * FROM students LOOP
 INSERT INTO attend VALUES (s.matrNr,184686);
END LOOP;
```

Example

Example

```
CREATE OR REPLACE FUNCTION
  searchfor(matrNr numeric(10)) RETURNS void AS $$
DECLARE
name    varchar(30);
semester numeric(2);
BEGIN
  SELECT s.name, s.semester INTO name, semester
    FROM students s WHERE s.matrNr = matrNr;
  IF (name IS NULL) THEN
    RAISE NOTICE 'nothing found';
  ELSE
    RAISE NOTICE 'Name: %, Semester: % ',
      name, semester;
  END IF;
END; $$ LANGUAGE plpgsql;
```

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures

5.1 Return Values of a Function

5.2 Conditionals

5.3 Loops

5.4 Error Handling

6. Cursors

Error Handling via EXCEPTIONS

```
BEGIN
  statements
EXCEPTION
  WHEN cond [OR Cond ...] THEN
    handler_statements
  [ WHEN cond [OR Cond ...] THEN
    handler_statements
    ... ]
END;
```

- section of a block; similar to try . . . catch in Java
- error not caught: pass “to the outside”, eventually abort; ROLLBACK of modifications to data base
- error caught: values of local variables are kept, ROLLBACK of modifications to the data base within the block

Error Handling – Example

Example (Modification/Creation of a Grade)

```
LOOP
 UPDATE examine SET grade=1 WHERE matrNr=mn
    AND lecNr=vn AND persNr=pn;
 RETURN WHEN FOUND;
 BEGIN
   INSERT INTO examine VALUES (mn,vn,pn,1);
   RETURN;
 EXCEPTION WHEN unique_violation THEN
   -- noting, try update again
 END;
END LOOP;
```

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures

6. Cursors

6.1 Definition

6.2 Application

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures

6. Cursors

6.1 Definition

6.2 Application

Reading a Result Line by Line – Cursor

Cursor . . .

- allows for a gradual iteration of results (avoids loading the whole result into the main memory)
- can be used as another variable (also as parameter/return values of functions)

Declaration:

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

Cursor – Example

Example (cursor – declarations)

```
curs1 refcursor;
curs2 CURSOR FOR SELECT * FROM students;
curs3 CURSOR (mn integer) FOR
    SELECT *
    FROM students
    WHERE matrNr = mn;
```

Overview

1. Introduction

2. Structure of PL/pgSQL Programs

3. Variables

4. Expressions

5. Control Structures

6. Cursors

6.1 Definition

6.2 Application

Cursor: Application and Access

`OPEN` cursor has to be opened before usage; syntax depending on the cursor being bound or not

`FETCH` reads the next row of the result; if there is no next entry `NULL` is read

`FOUND` variable indicating whether `FETCH` has read a value

`MOVE` allows moving the cursor without reading

`CLOSE` closes cursor and sets resources that were hold back free

Usage of a Cursor – OPEN

Example (opening a cursors)

```
OPEN curs1 FOR SELECT * FROM professors;
OPEN curs2;
OPEN curs3(42);
```

- [NO] SCROLL if cursor is not bound
- FOR *query* only if cursor is not bound

Usage of a Cursors – FETCH

```
FETCH [direction {FROM|IN}] cursor INTO target;
```

Example

```
FETCH curs1 INTO rowvar;
FETCH LAST FROM curs2 INTO stud;
FETCH PRIOR FROM curs2 INTO stud;
FETCH RELATIVE 2 FROM curs2 INTO stud;
FETCH curs3 INTO mn, name, sem;
```

Write Access

- cursors allow modifying current data set (UPDATE/DELETE)
- query has to be simple (for instance no aggregation)
- recommended: declare cursor as FOR UPDATE
  - locks data set
  - checks whether query modification is allowed

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

Cursor – Example

Example

```
lectures(lecNr, SWS):
INSERT INTO lectures VALUES (26120, 3), (27550,4);

CREATE FUNCTION reassign() RETURNS void AS $$
DECLARE
  c CURSOR FOR SELECT * FROM lectures FOR UPDATE;
  c2 refcursor;
  count integer = 0;
  row RECORD;
BEGIN
  -- next slide
END; $$ LANGUAGE plpgsql;
```

Cursor – Example (contd.)

Example

```
FOR r IN c LOOP
  RAISE NOTICE ’(%, %)’, r.lecNr, r.SWS;
  UPDATE lectures SET lecNr=count WHERE CURRENT OF c;
  count = count + 1;
END LOOP;

OPEN c2 FOR SELECT * FROM lectures;
LOOP
  FETCH c2 INTO row;
  RAISE NOTICE ’(%, %)’, row.lecNr, row.SWS;
  EXIT WHEN NOT FOUND;
END LOOP;
CLOSE c2;
```

result: (26120, 3), (27550, 4), (0, 3), (1, 4)