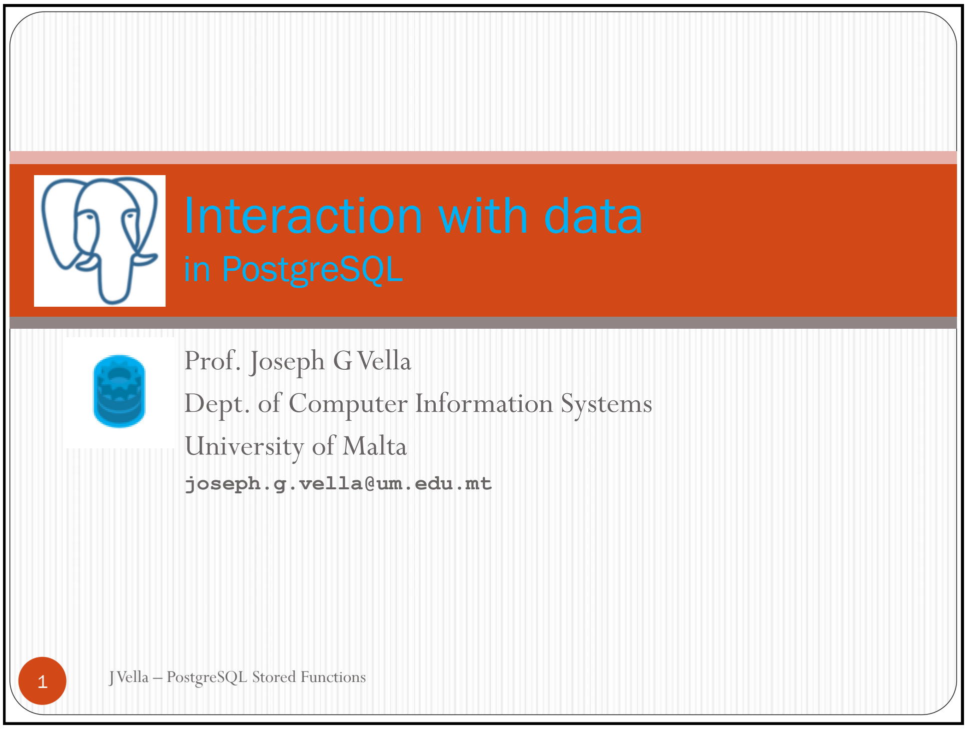
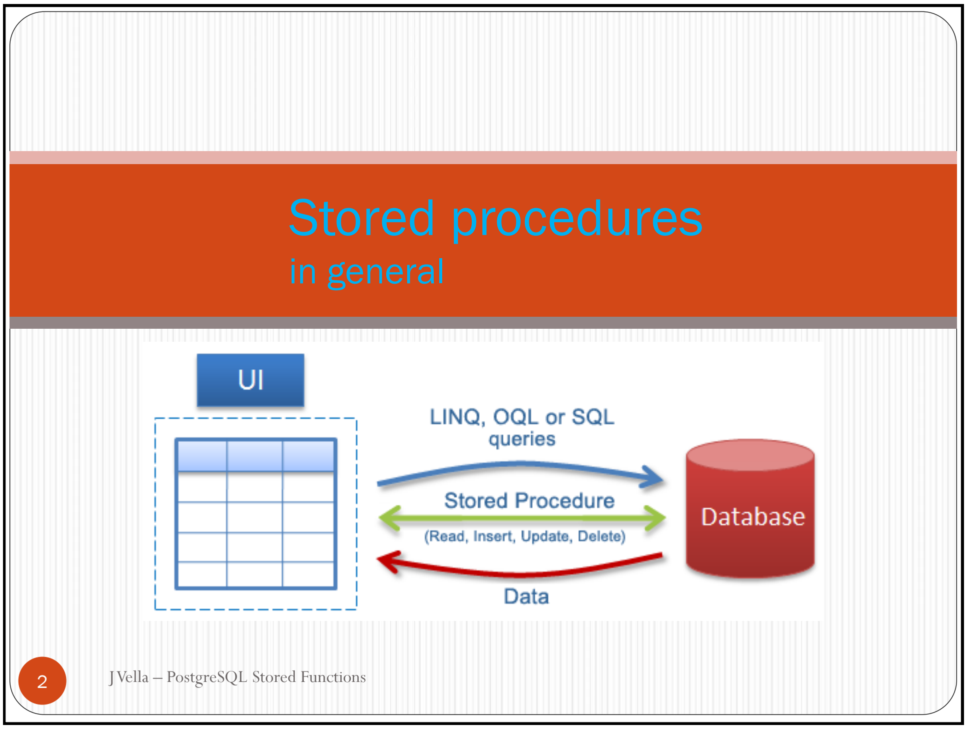


Interaction with data

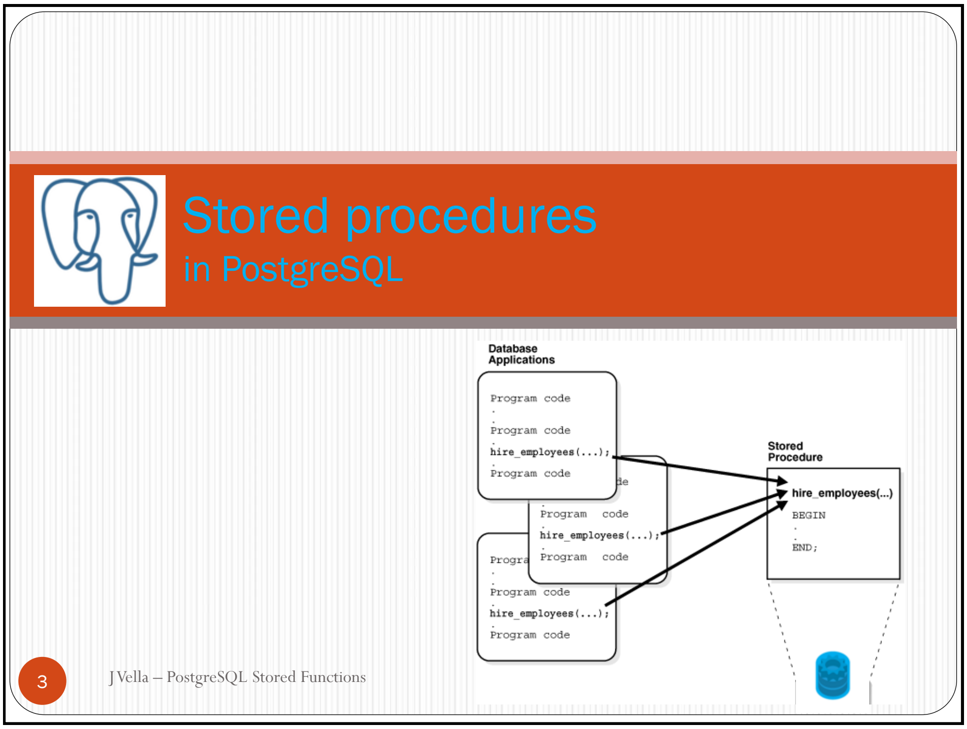
in PostgreSQL

Prof. Joseph G Vella
Dept. of Computer Information Systems
University of Malta
joseph.g.vella@um.edu.mt



Stored procedures

in general

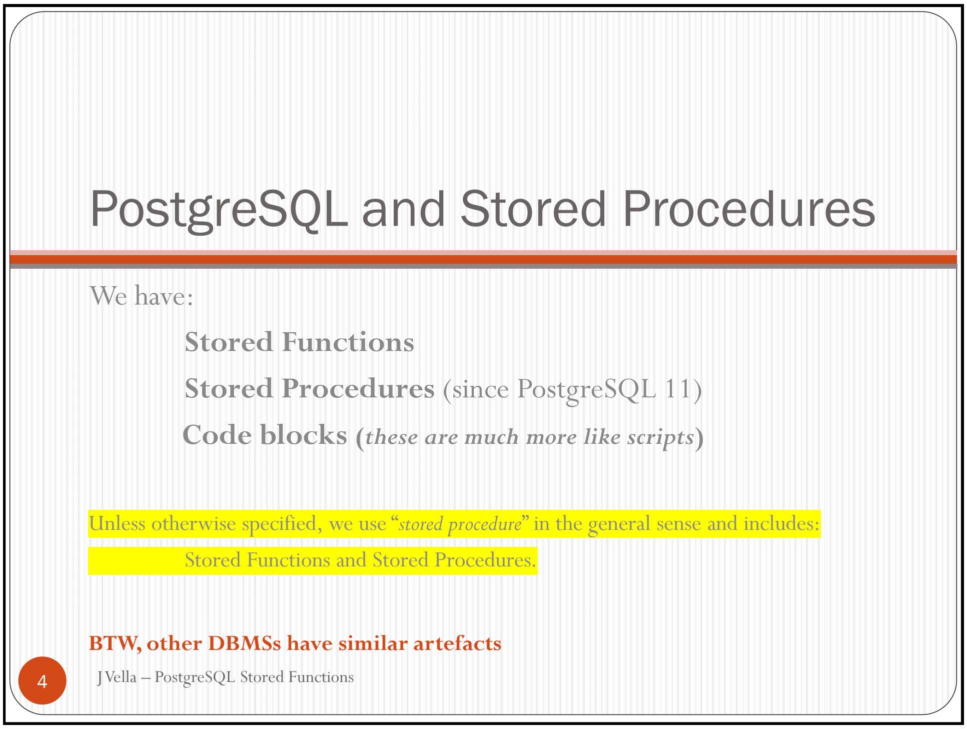


Stored procedures

in PostgreSQL

3 J Vella – PostgreSQL Stored Functions

3



PostgreSQL and Stored Procedures

We have:

Stored Functions

Stored Procedures (since PostgreSQL 11)

Code blocks (***these are much more like scripts***)

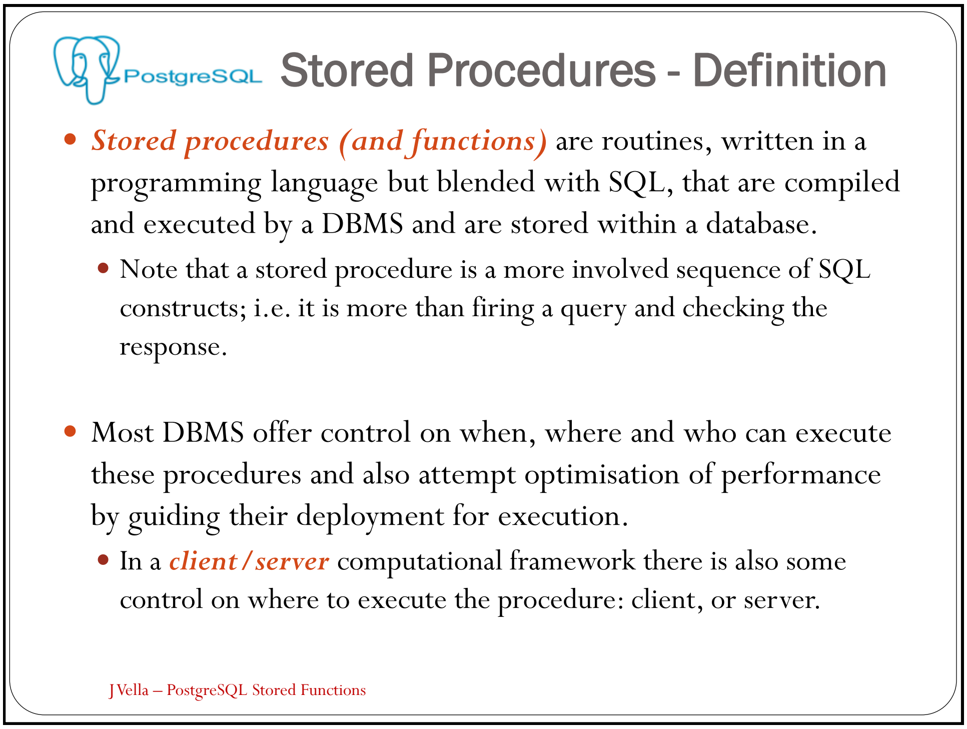
Unless otherwise specified, we use "*stored procedure*" in the general sense and includes:

Stored Functions and Stored Procedures.

BTW, other DBMSs have similar artefacts

4 J Vella – PostgreSQL Stored Functions

4

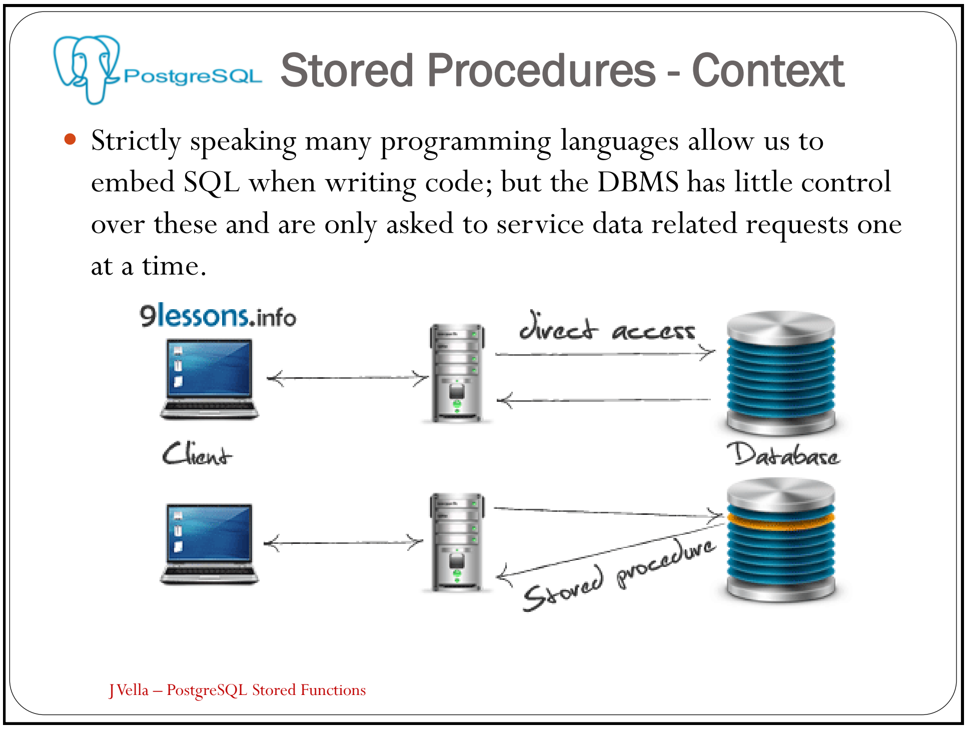


Stored Procedures - Definition

- *Stored procedures (and functions)* are routines, written in a programming language but blended with SQL, that are compiled and executed by a DBMS and are stored within a database.
  - Note that a stored procedure is a more involved sequence of SQL constructs; i.e. it is more than firing a query and checking the response.
- Most DBMS offer control on when, where and who can execute these procedures and also attempt optimisation of performance by guiding their deployment for execution.
  - In a ***client/server*** computational framework there is also some control on where to execute the procedure: client, or server.

J Vella – PostgreSQL Stored Functions

5

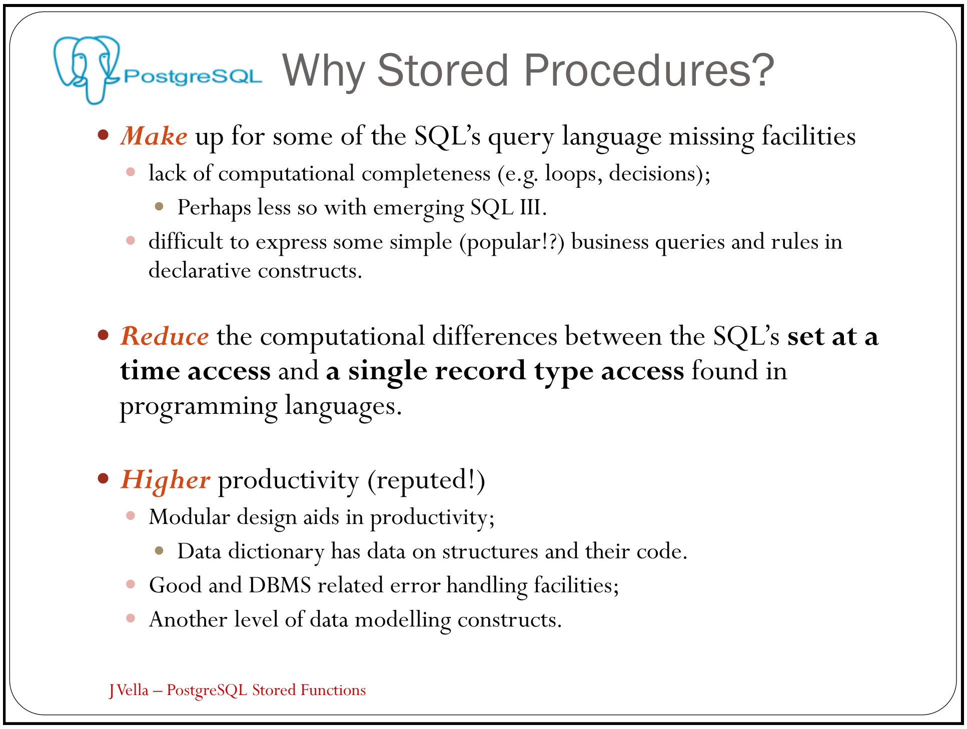


Stored Procedures - Context

- Strictly speaking many programming languages allow us to embed SQL when writing code; but the DBMS has little control over these and are only asked to service data related requests one at a time.

J Vella – PostgreSQL Stored Functions

6

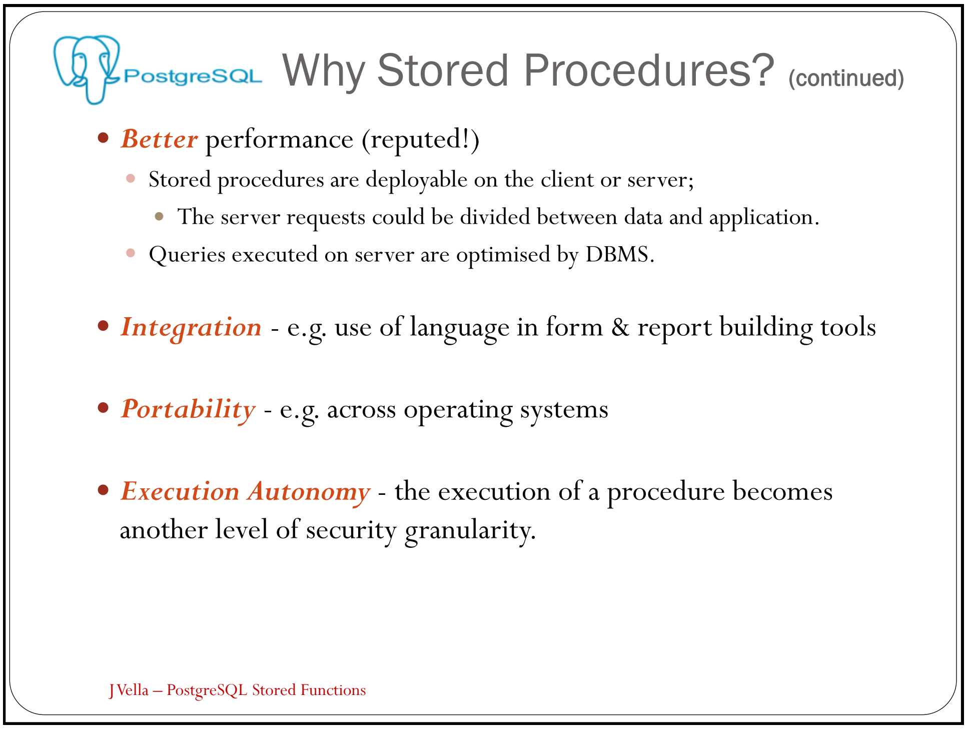


Why Stored Procedures?

- *Make* up for some of the SQL's query language missing facilities
  - lack of computational completeness (e.g. loops, decisions);
    - Perhaps less so with emerging SQL III.
  - difficult to express some simple (popular!?) business queries and rules in declarative constructs.
- *Reduce* the computational differences between the SQL's **set at a time access** and **a single record type access** found in programming languages.
- *Higher* productivity (reputed!)
  - Modular design aids in productivity;
    - Data dictionary has data on structures and their code.
  - Good and DBMS related error handling facilities;
  - Another level of data modelling constructs.

J Vella – PostgreSQL Stored Functions

7



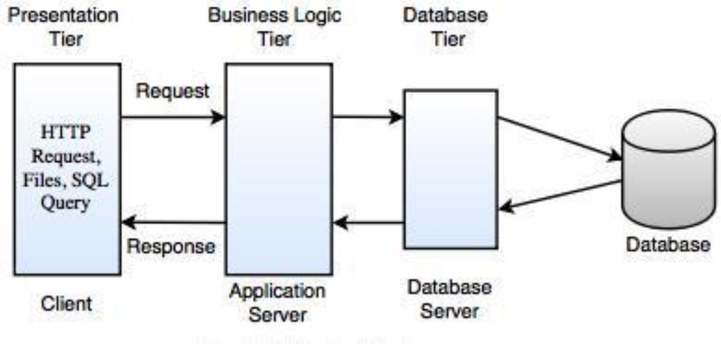
Why Stored Procedures? (continued)

- *Better* performance (reputed!)
  - Stored procedures are deployable on the client or server;
    - The server requests could be divided between data and application.
  - Queries executed on server are optimised by DBMS.
- *Integration* - e.g. use of language in form & report building tools
- *Portability* - e.g. across operating systems
- *Execution Autonomy* - the execution of a procedure becomes another level of security granularity.

J Vella – PostgreSQL Stored Functions

8

Example of Stored Proc adoption

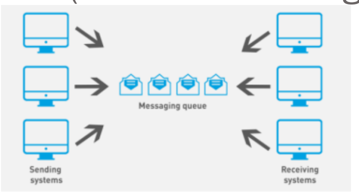


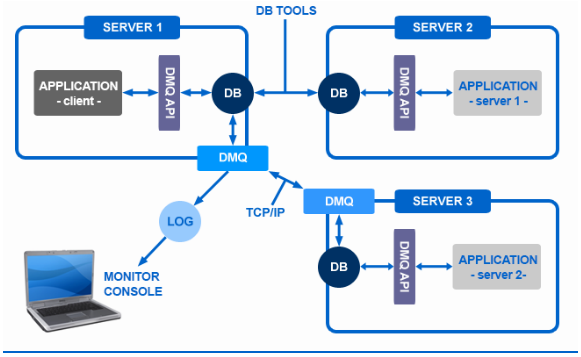
9 J Vella – PostgreSQL Stored Functions

9

Another Example of adoption

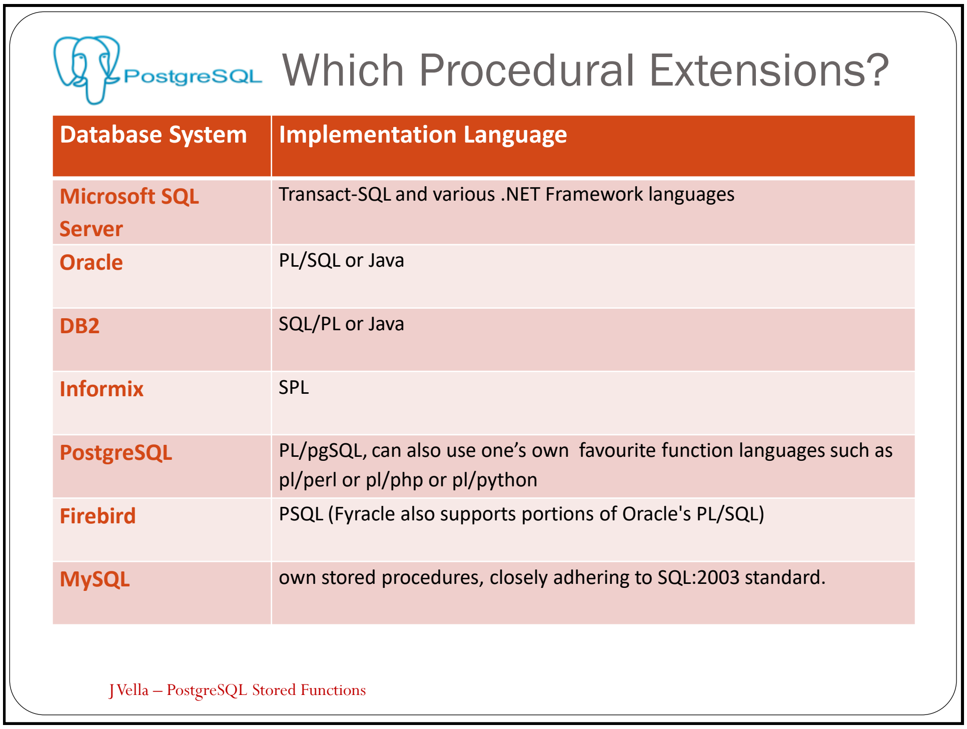
(with Data Message Queues – DMQ)





10 J Vella – PostgreSQL Stored Functions

10

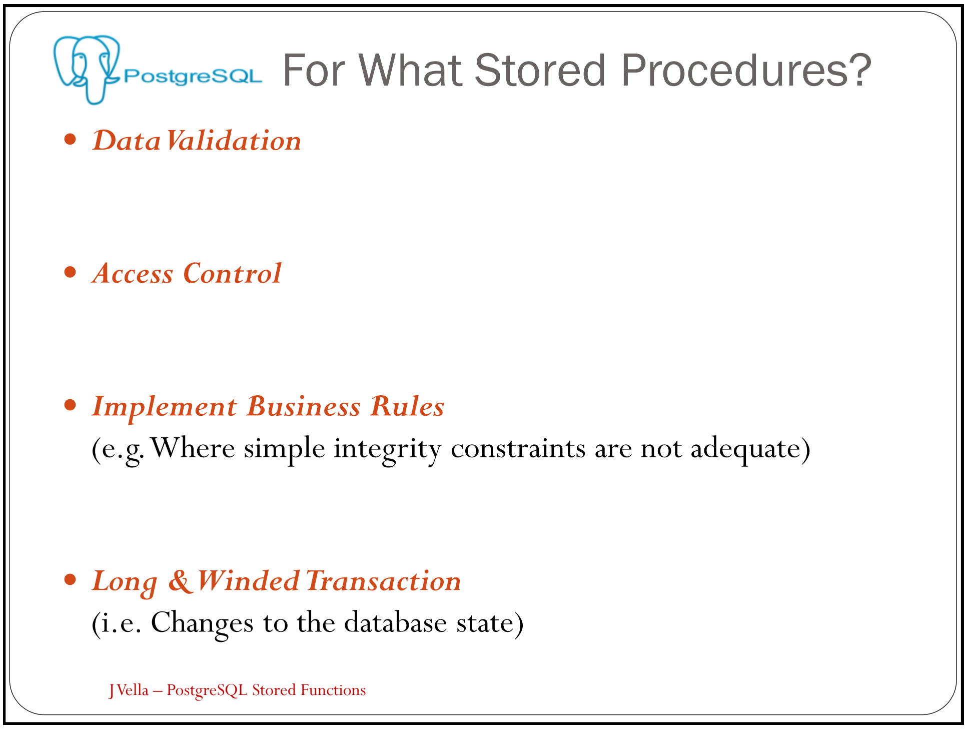


Which Procedural Extensions?

Database System	Implementation Language
Microsoft SQL Server	Transact-SQL and various .NET Framework languages
Oracle	PL/SQL or Java
DB2	SQL/PL or Java
Informix	SPL
PostgreSQL	PL/pgSQL, can also use one's own favourite function languages such as pl/perl or pl/php or pl/python
Firebird	PSQL (Fyracle also supports portions of Oracle's PL/SQL)
MySQL	own stored procedures, closely adhering to SQL:2003 standard.

J Vella – PostgreSQL Stored Functions

11

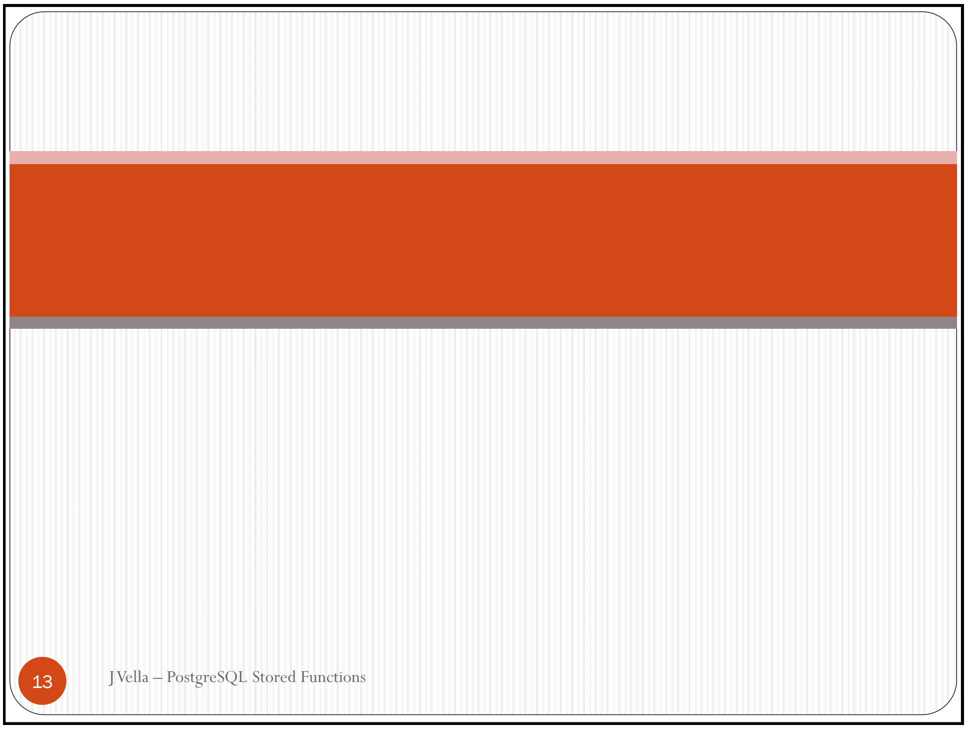


For What Stored Procedures?

- *Data Validation*
- *Access Control*
- *Implement Business Rules*
  (e.g. Where simple integrity constraints are not adequate)
- *Long & Winded Transaction*
  (i.e. Changes to the database state)

J Vella – PostgreSQL Stored Functions

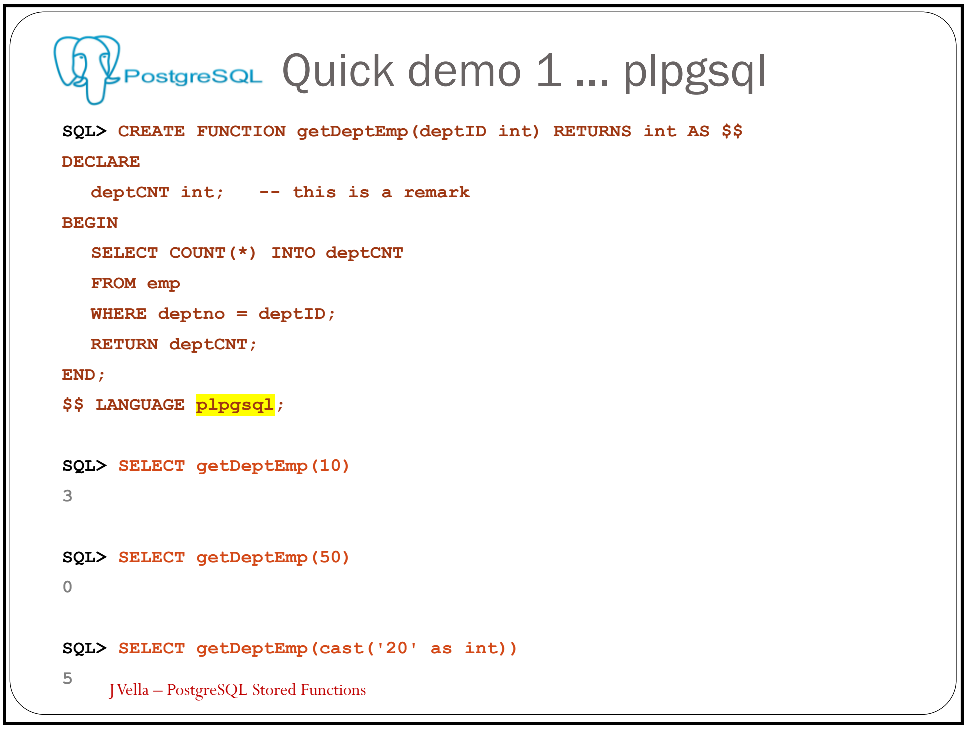
12



13

J Vella – PostgreSQL Stored Functions

13

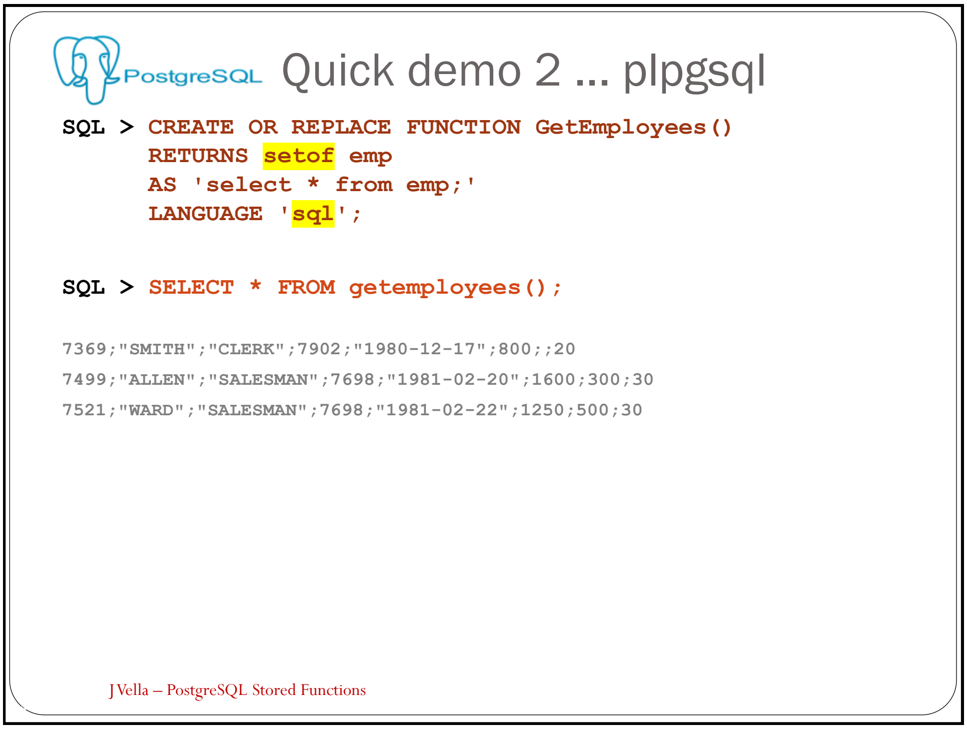


Quick demo 1 ... plpgsql

```
SQL> CREATE FUNCTION getDeptEmp(deptID int) RETURNS int AS $$
DECLARE
   deptCNT int;   -- this is a remark
BEGIN
   SELECT COUNT(*) INTO deptCNT
   FROM emp
   WHERE deptno = deptID;
   RETURN deptCNT;
END;
$$ LANGUAGE plpgsql;

SQL> SELECT getDeptEmp(10)
3

SQL> SELECT getDeptEmp(50)
0

SQL> SELECT getDeptEmp(cast('20' as int))
5
```

J Vella – PostgreSQL Stored Functions

14

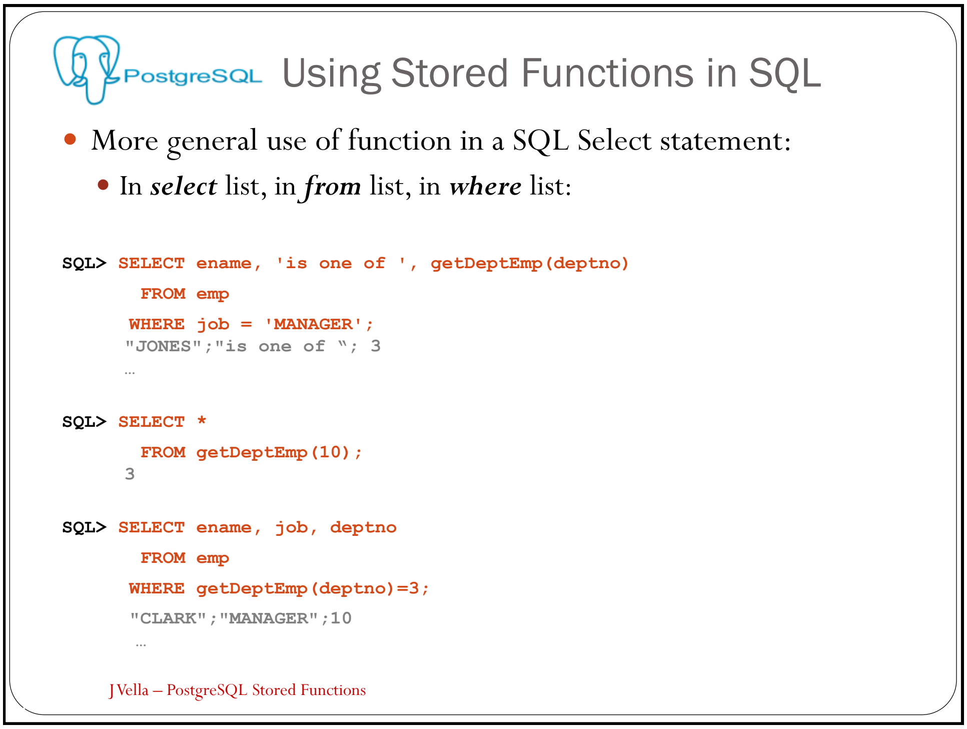


Quick demo 2 … plpgsql

```
SQL > CREATE OR REPLACE FUNCTION GetEmployees()
      RETURNS setof emp
      AS 'select * from emp;'
      LANGUAGE 'sql';

SQL > SELECT * FROM getemployees();

7369;"SMITH";"CLERK";7902;"1980-12-17";800;;20
7499;"ALLEN";"SALESMAN";7698;"1981-02-20";1600;300;30
7521;"WARD";"SALESMAN";7698;"1981-02-22";1250;500;30
```

J Vella – PostgreSQL Stored Functions

15



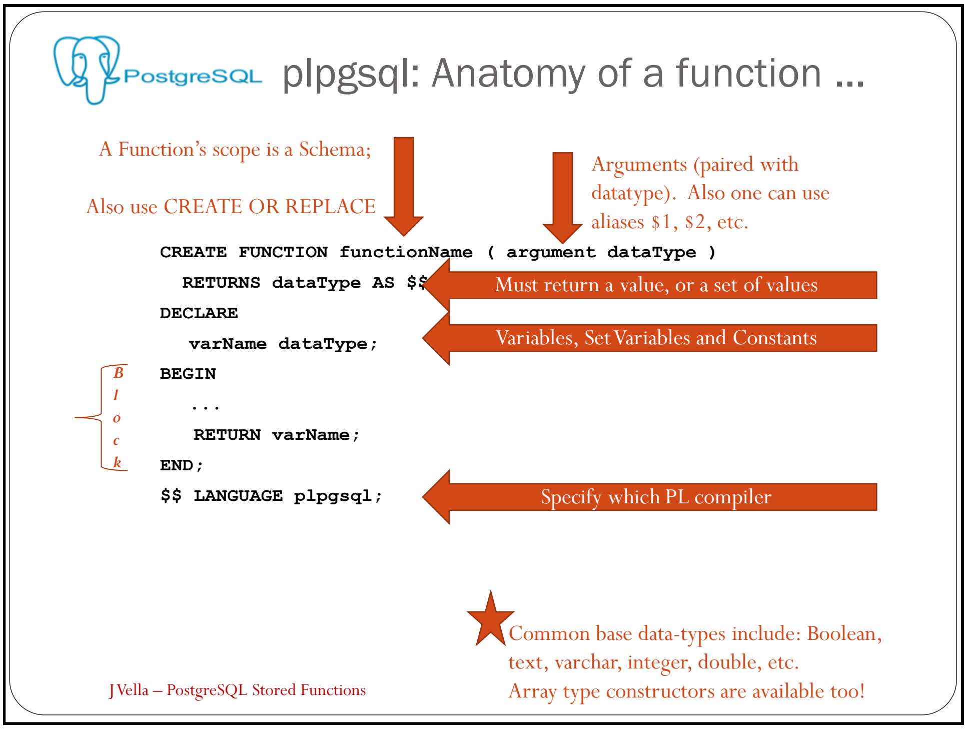
Using Stored Functions in SQL

- More general use of function in a SQL Select statement:
  - In *select* list, in *from* list, in *where* list:

```
SQL> SELECT ename, 'is one of ', getDeptEmp(deptno)
       FROM emp
      WHERE job = 'MANAGER';
     "JONES";"is one of "; 3
     …

SQL> SELECT *
       FROM getDeptEmp(10);
     3

SQL> SELECT ename, job, deptno
       FROM emp
      WHERE getDeptEmp(deptno)=3;
      "CLARK";"MANAGER";10
       …
```

J Vella – PostgreSQL Stored Functions

17



PostgreSQL plpgsql: Anatomy of a function …

A Function's scope is a Schema;

Also use CREATE OR REPLACE

Arguments (paired with datatype). Also one can use aliases $1, $2, etc.

```
CREATE FUNCTION functionName ( argument dataType )
  RETURNS dataType AS $$
DECLARE
   varName dataType;
BEGIN
   ...
   RETURN varName;
END;
$$ LANGUAGE plpgsql;
```

Must return a value, or a set of values

Variables, Set Variables and Constants

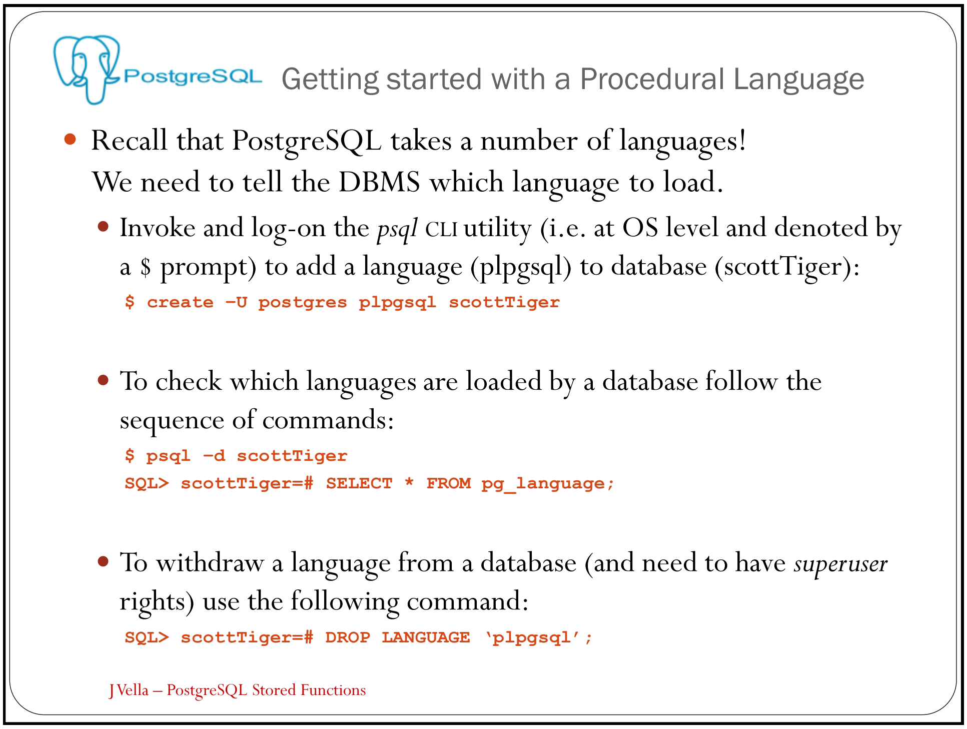
Block

Specify which PL compiler

Common base data-types include: Boolean, text, varchar, integer, double, etc.
Array type constructors are available too!

J Vella – PostgreSQL Stored Functions

18



PostgreSQL Getting started with a Procedural Language

- Recall that PostgreSQL takes a number of languages!
  We need to tell the DBMS which language to load.
  - Invoke and log-on the *psql* CLI utility (i.e. at OS level and denoted by a $ prompt) to add a language (plpgsql) to database (scottTiger):

```
$ create -U postgres plpgsql scottTiger
```

  - To check which languages are loaded by a database follow the sequence of commands:

```
$ psql -d scottTiger
SQL> scottTiger=# SELECT * FROM pg_language;
```

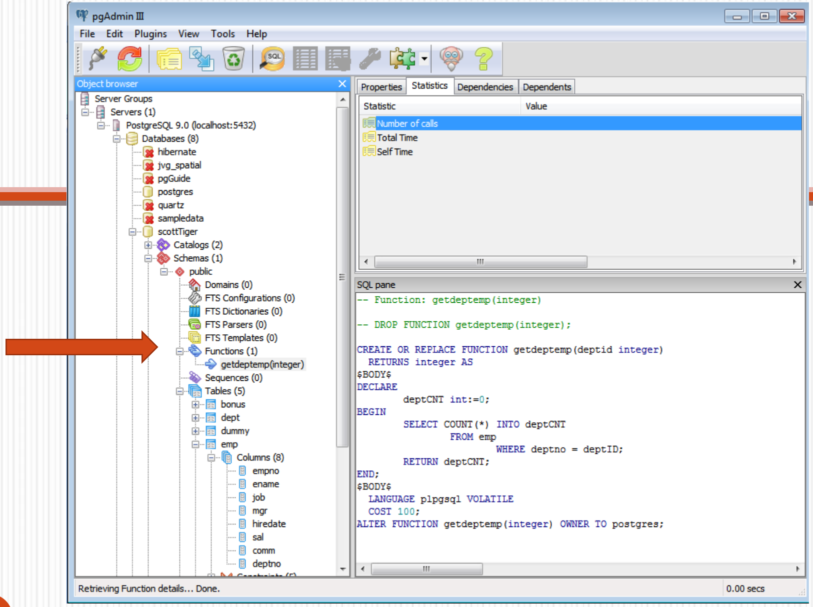
  - To withdraw a language from a database (and need to have *superuser* rights) use the following command:

```
SQL> scottTiger=# DROP LANGUAGE 'plpgsql';
```

J Vella – PostgreSQL Stored Functions

19

pgAdmin III and Functions



Statistic	Value
Number of calls	
Total Time	
Self Time	

```
-- Function: getdeptemp(integer)

-- DROP FUNCTION getdeptemp(integer);

CREATE OR REPLACE FUNCTION getdeptemp(deptid integer)
  RETURNS integer AS
$BODY$
DECLARE
        deptCNT int:=0;
BEGIN
        SELECT COUNT(*) INTO deptCNT
                FROM emp
                        WHERE deptno = deptID;
        RETURN deptCNT;
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;
ALTER FUNCTION getdeptemp(integer) OWNER TO postgres;
```

Retrieving Function details... Done.   0.00 secs

20 J Vella – PostgreSQL Stored Functions

20

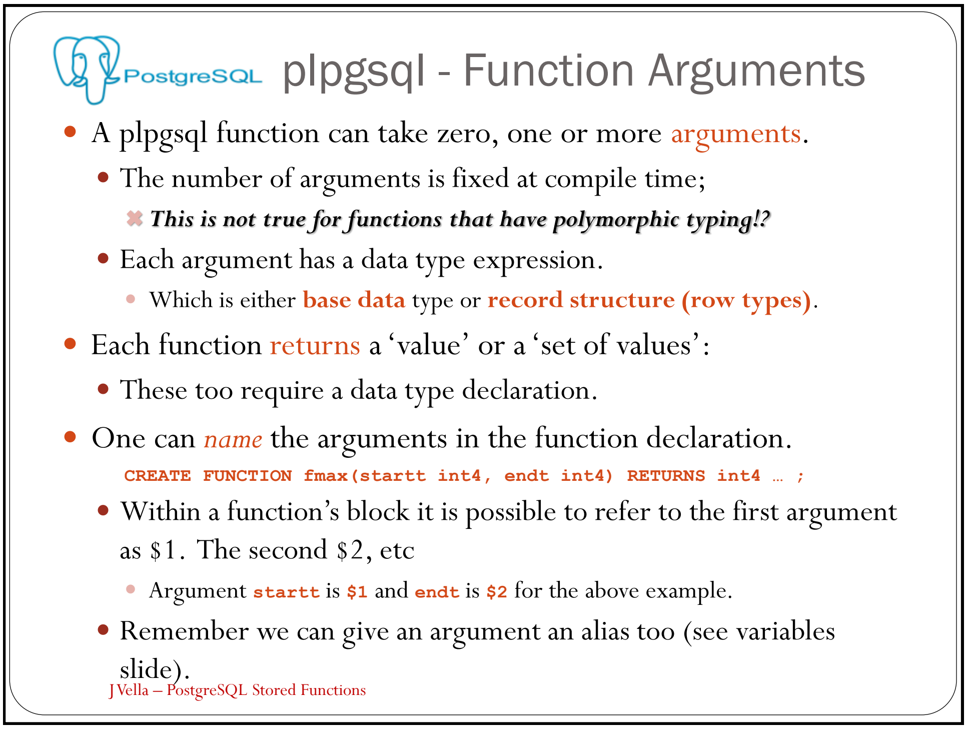
plpgsql - Function Names

- The name of a function follows variable names rule.
  - Functions can be also qualified by a schema name.
    - Schema less function definitions are stored in the current schema.
- A function name (or schema qualified function names) can be overloaded. For example, the following is allowed:

```
SQL > CREATE OR REPLACE FUNCTION GetEmployees()
      RETURNS setof emp
      AS 'select * from emp;'
      LANGUAGE 'sql';
SQL > CREATE OR REPLACE FUNCTION GetEmployees(integer)
      RETURNS setof emp
      AS 'select * from emp where deptno = $1;'
      LANGUAGE 'sql';
SQL > SELECTC GetEmployees();
      …
SQL > SELECT GetEmployees(10);
      …
```

J Vella – PostgreSQL Stored Functions

21



plpgsql - Function Arguments

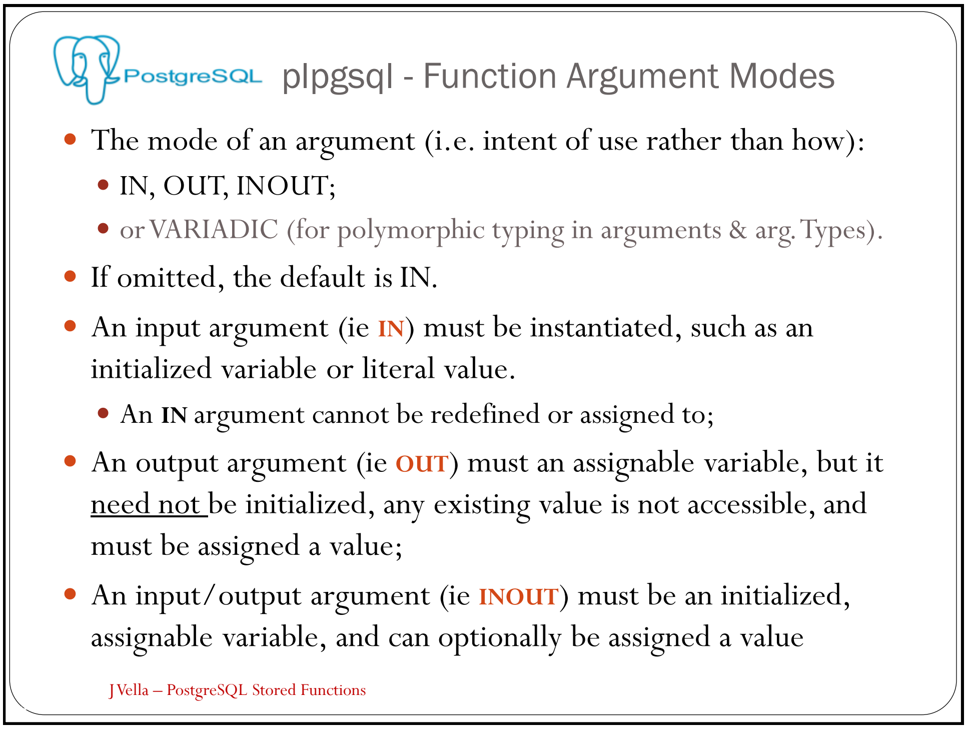
- A plpgsql function can take zero, one or more arguments.
  - The number of arguments is fixed at compile time;
    - ***This is not true for functions that have polymorphic typing!?***
  - Each argument has a data type expression.
    - Which is either **base data** type or **record structure (row types)**.
- Each function returns a 'value' or a 'set of values':
  - These too require a data type declaration.
- One can *name* the arguments in the function declaration.

```
CREATE FUNCTION fmax(startt int4, endt int4) RETURNS int4 … ;
```

  - Within a function's block it is possible to refer to the first argument as $1. The second $2, etc
    - Argument `startt` is `$1` and `endt` is `$2` for the above example.
  - Remember we can give an argument an alias too (see variables slide).

J Vella – PostgreSQL Stored Functions

22

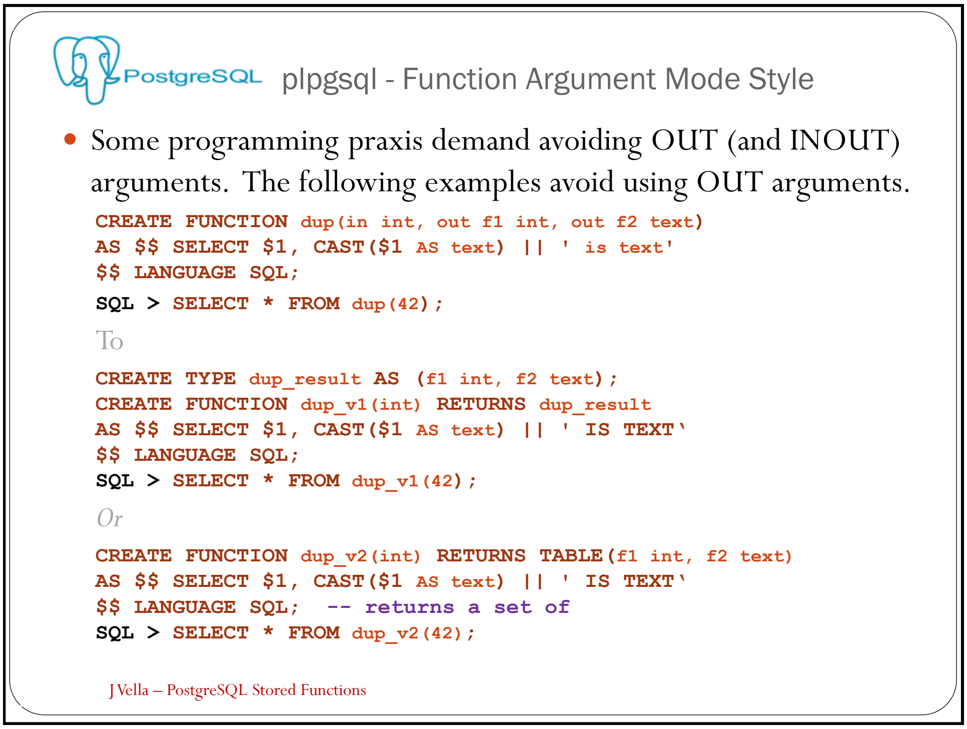


plpgsql - Function Argument Modes

- The mode of an argument (i.e. intent of use rather than how):
  - IN, OUT, INOUT;
  - or VARIADIC (for polymorphic typing in arguments & arg. Types).
- If omitted, the default is IN.
- An input argument (ie **IN**) must be instantiated, such as an initialized variable or literal value.
  - An **IN** argument cannot be redefined or assigned to;
- An output argument (ie **OUT**) must an assignable variable, but it need not be initialized, any existing value is not accessible, and must be assigned a value;
- An input/output argument (ie **INOUT**) must be an initialized, assignable variable, and can optionally be assigned a value

J Vella – PostgreSQL Stored Functions

23



plpgsql - Function Argument Mode Style

- Some programming praxis demand avoiding OUT (and INOUT) arguments. The following examples avoid using OUT arguments.

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
AS $$ SELECT $1, CAST($1 AS text) || ' is text'
$$ LANGUAGE SQL;
SQL > SELECT * FROM dup(42);
```

To

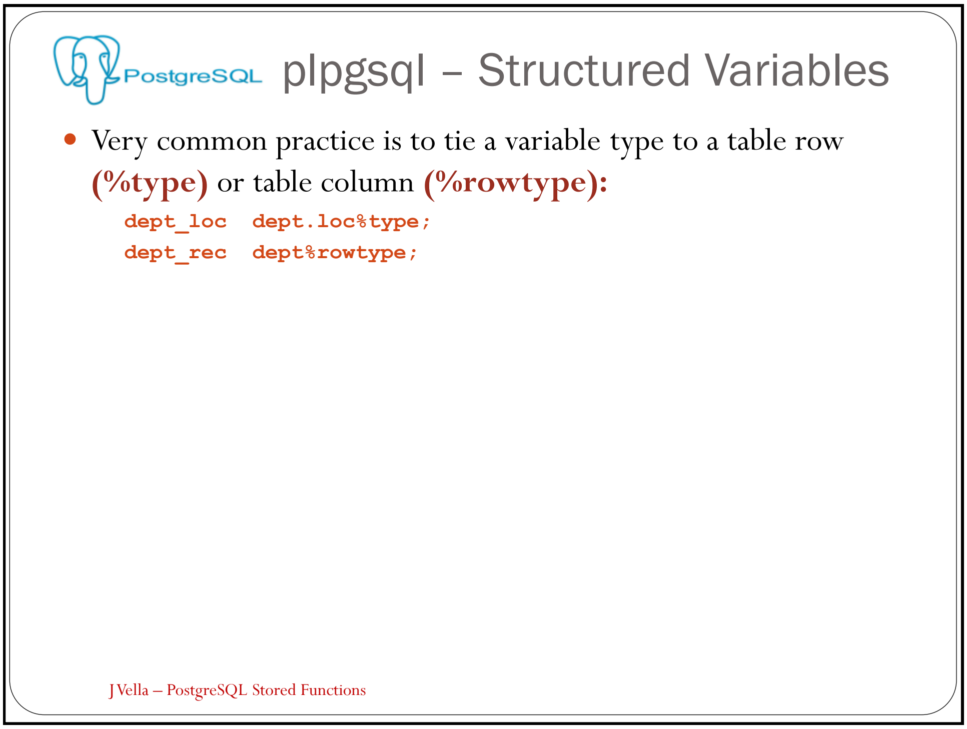
```
CREATE TYPE dup_result AS (f1 int, f2 text);
CREATE FUNCTION dup_v1(int) RETURNS dup_result
AS $$ SELECT $1, CAST($1 AS text) || ' IS TEXT`
$$ LANGUAGE SQL;
SQL > SELECT * FROM dup_v1(42);
```

Or

```
CREATE FUNCTION dup_v2(int) RETURNS TABLE(f1 int, f2 text)
AS $$ SELECT $1, CAST($1 AS text) || ' IS TEXT`
$$ LANGUAGE SQL;  -- returns a set of
SQL > SELECT * FROM dup_v2(42);
```

J Vella – PostgreSQL Stored Functions

24



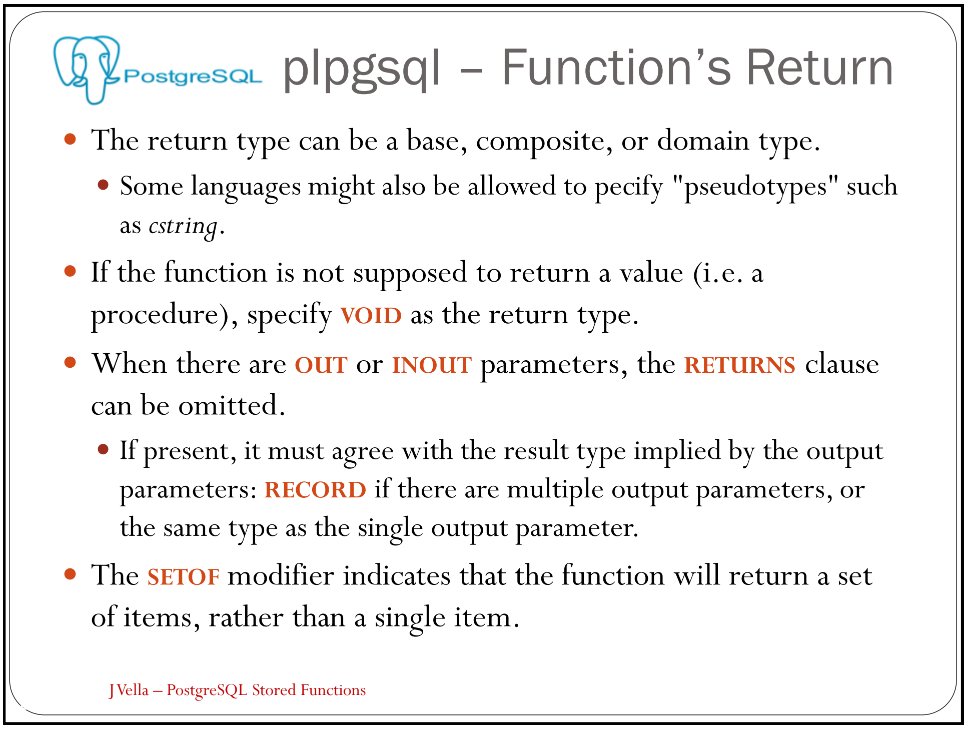
plpgsql – Structured Variables

- Very common practice is to tie a variable type to a table row **(%type)** or table column **(%rowtype):**

```
dept_loc  dept.loc%type;
dept_rec  dept%rowtype;
```

J Vella – PostgreSQL Stored Functions

25

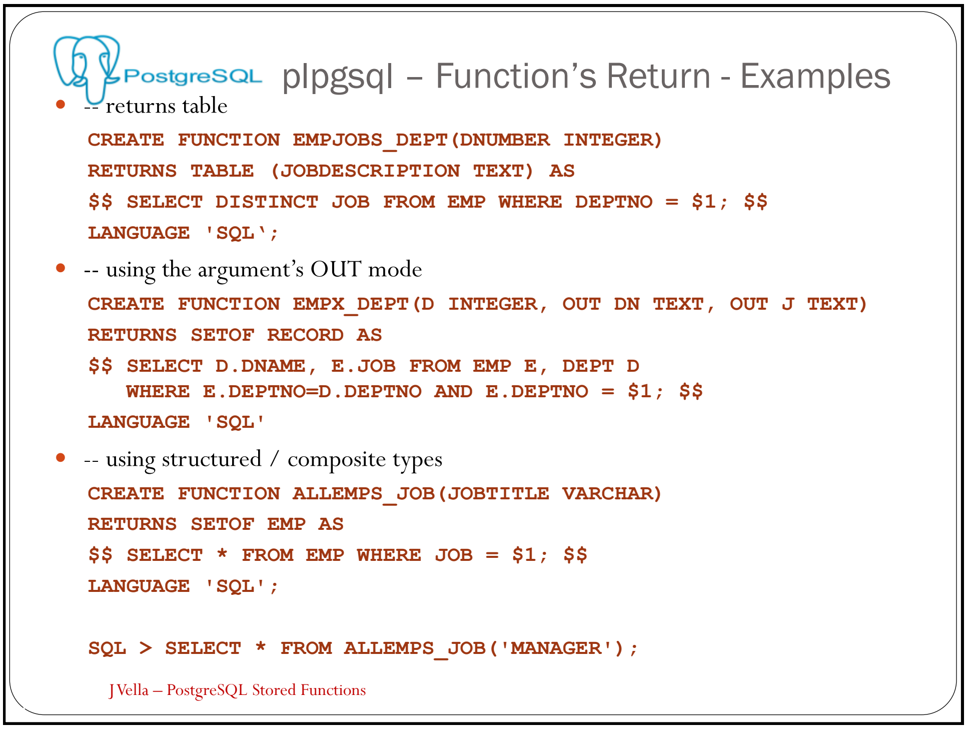


plpgsql – Function’s Return

- The return type can be a base, composite, or domain type.
  - Some languages might also be allowed to pecify "pseudotypes" such as *cstring*.
- If the function is not supposed to return a value (i.e. a procedure), specify **VOID** as the return type.
- When there are **OUT** or **INOUT** parameters, the **RETURNS** clause can be omitted.
  - If present, it must agree with the result type implied by the output parameters: **RECORD** if there are multiple output parameters, or the same type as the single output parameter.
- The **SETOF** modifier indicates that the function will return a set of items, rather than a single item.

J Vella – PostgreSQL Stored Functions

26



plpgsql – Function’s Return - Examples

- -- returns table

```
CREATE FUNCTION EMPJOBS_DEPT(DNUMBER INTEGER)
RETURNS TABLE (JOBDESCRIPTION TEXT) AS
$$ SELECT DISTINCT JOB FROM EMP WHERE DEPTNO = $1; $$
LANGUAGE 'SQL‘;
```

- -- using the argument’s OUT mode

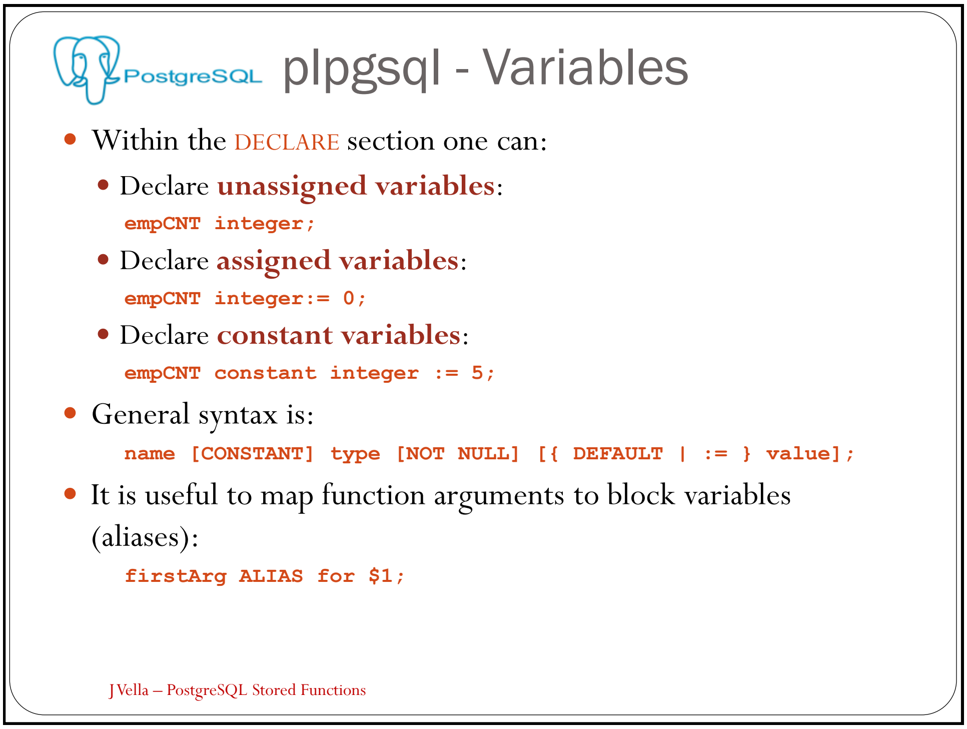
```
CREATE FUNCTION EMPX_DEPT(D INTEGER, OUT DN TEXT, OUT J TEXT)
RETURNS SETOF RECORD AS
$$ SELECT D.DNAME, E.JOB FROM EMP E, DEPT D
   WHERE E.DEPTNO=D.DEPTNO AND E.DEPTNO = $1; $$
LANGUAGE 'SQL'
```

- -- using structured / composite types

```
CREATE FUNCTION ALLEMPS_JOB(JOBTITLE VARCHAR)
RETURNS SETOF EMP AS
$$ SELECT * FROM EMP WHERE JOB = $1; $$
LANGUAGE 'SQL';

SQL > SELECT * FROM ALLEMPS_JOB('MANAGER');
```

J Vella – PostgreSQL Stored Functions

27



plpgsql - Variables

- Within the DECLARE section one can:
  - Declare **unassigned variables**:

    ```
    empCNT integer;
    ```
  - Declare **assigned variables**:

    ```
    empCNT integer:= 0;
    ```
  - Declare **constant variables**:

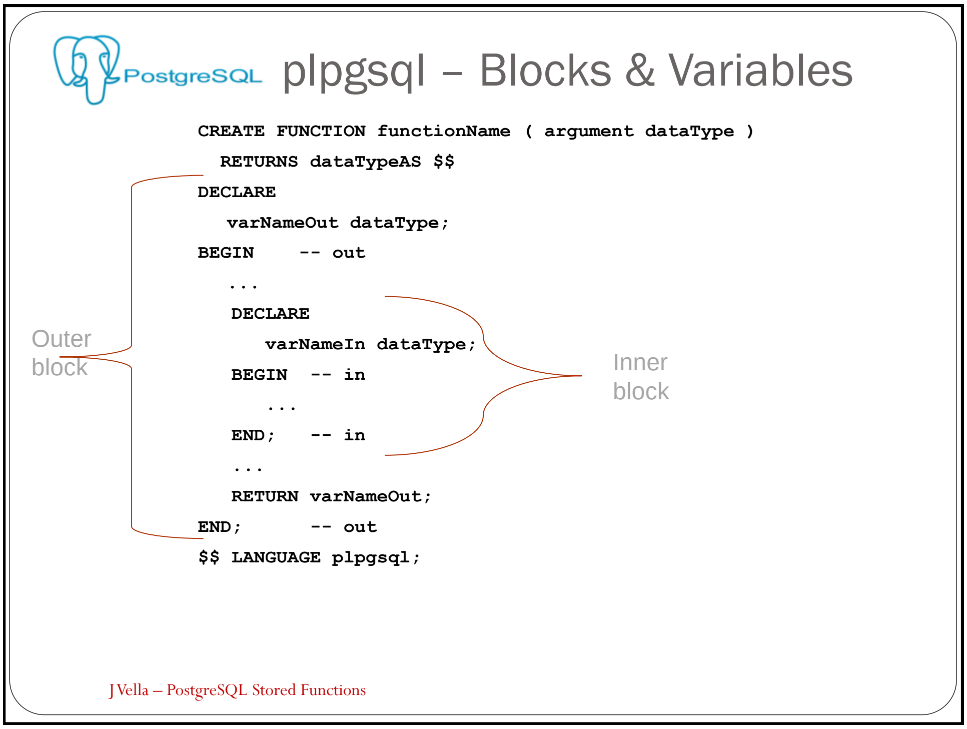
    ```
    empCNT constant integer := 5;
    ```
- General syntax is:

  ```
  name [CONSTANT] type [NOT NULL] [{ DEFAULT | := } value];
  ```
- It is useful to map function arguments to block variables (aliases):

  ```
  firstArg ALIAS for $1;
  ```

J Vella – PostgreSQL Stored Functions

28



plpgsql – Blocks & Variables

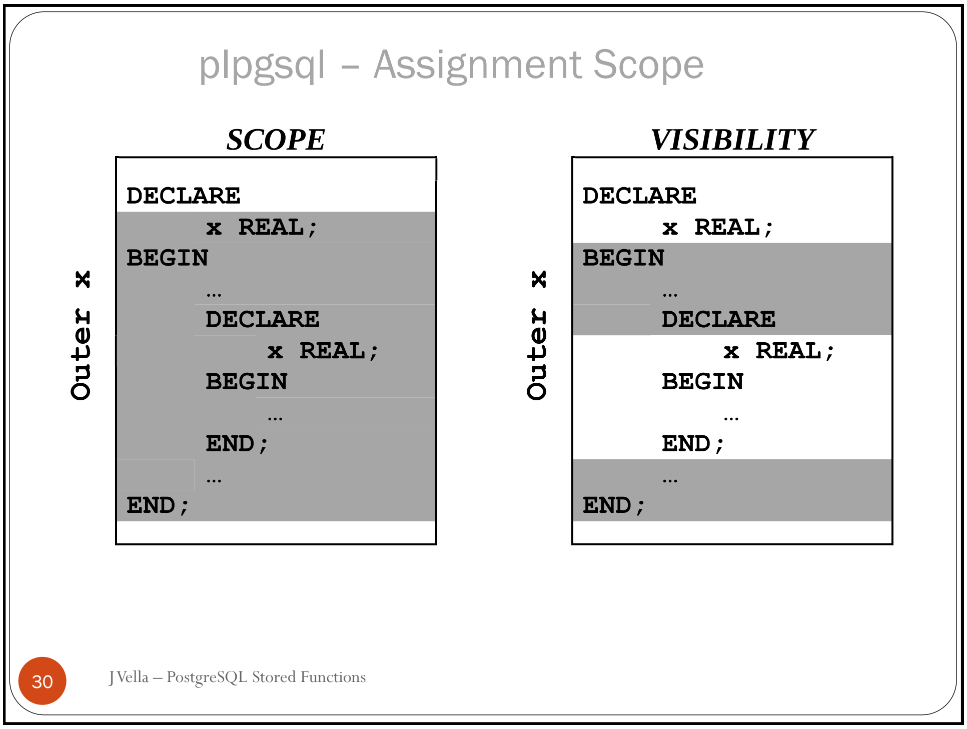
```
CREATE FUNCTION functionName ( argument dataType )
  RETURNS dataTypeAS $$
DECLARE
   varNameOut dataType;
BEGIN    -- out
   ...
   DECLARE
      varNameIn dataType;
   BEGIN  -- in
      ...
   END;   -- in
   ...
   RETURN varNameOut;
END;      -- out
$$ LANGUAGE plpgsql;
```

Outer block

Inner block

J Vella – PostgreSQL Stored Functions

29



plpgsql – Assignment Scope

SCOPE

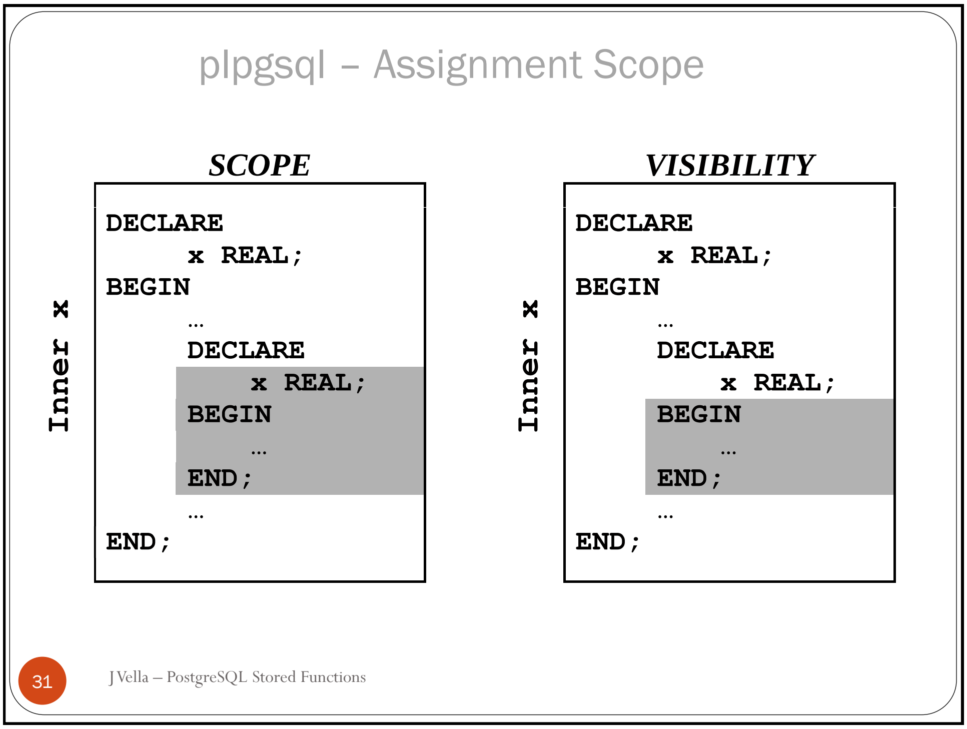
```
DECLARE
        x REAL;
BEGIN
        …
        DECLARE
              x REAL;
        BEGIN
              …
        END;
        …
END;
```

VISIBILITY

```
DECLARE
        x REAL;
BEGIN
        …
        DECLARE
              x REAL;
        BEGIN
              …
        END;
        …
END;
```

30 J Vella – PostgreSQL Stored Functions

30



plpgsql – Assignment Scope

SCOPE

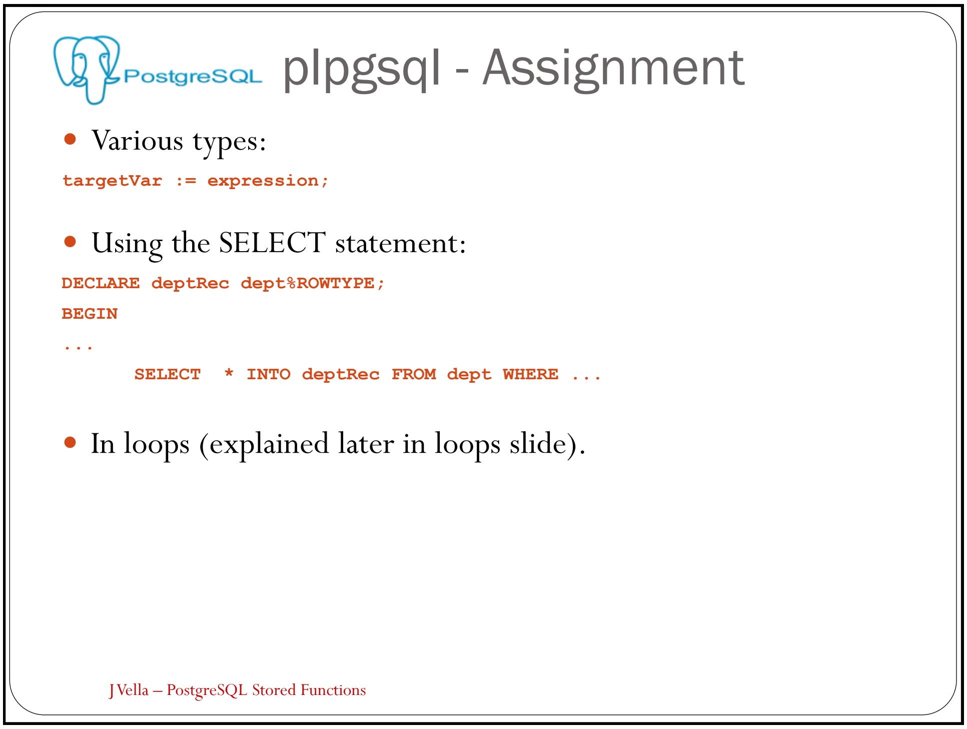
```
DECLARE
        x REAL;
BEGIN
        …
        DECLARE
              x REAL;
        BEGIN
              …
        END;
        …
END;
```

VISIBILITY

```
DECLARE
        x REAL;
BEGIN
        …
        DECLARE
              x REAL;
        BEGIN
              …
        END;
        …
END;
```

31 J Vella – PostgreSQL Stored Functions

31



plpgsql - Assignment

- Various types:

```
targetVar := expression;
```

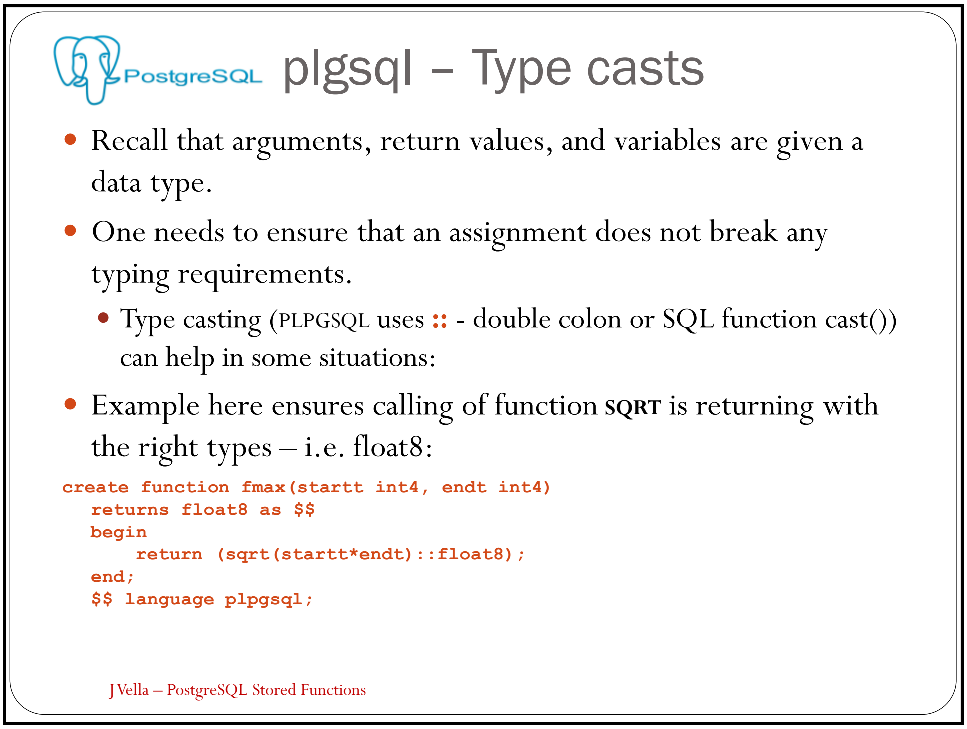
- Using the SELECT statement:

```
DECLARE deptRec dept%ROWTYPE;
BEGIN
...
      SELECT  * INTO deptRec FROM dept WHERE ...
```

- In loops (explained later in loops slide).

J Vella – PostgreSQL Stored Functions

32



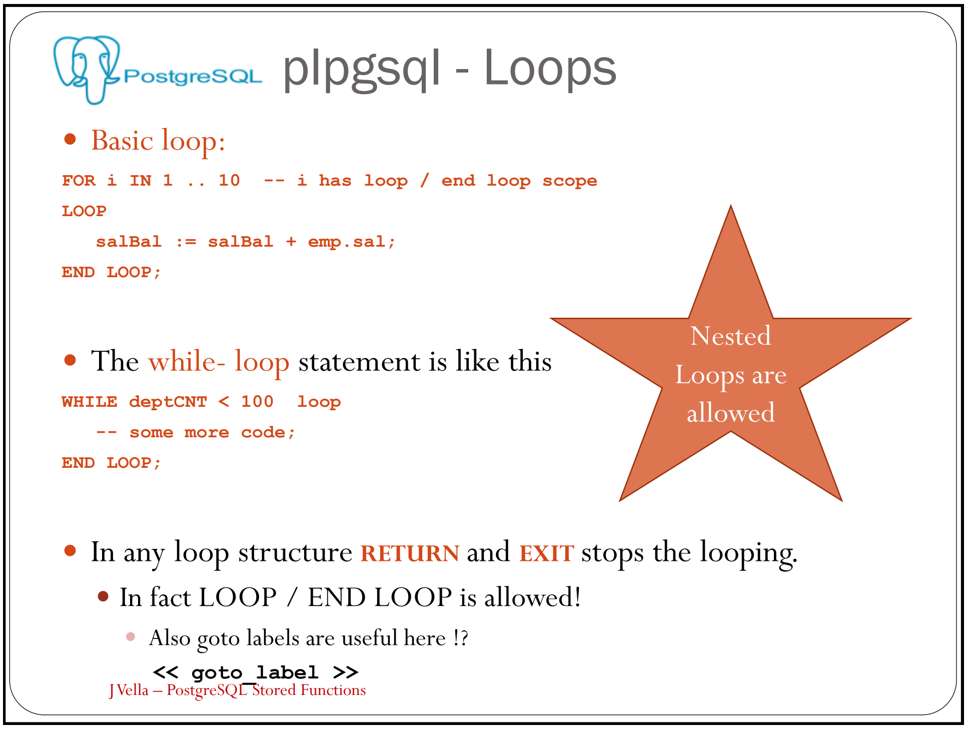
plgsql – Type casts

- Recall that arguments, return values, and variables are given a data type.
- One needs to ensure that an assignment does not break any typing requirements.
  - Type casting (PLPGSQL uses **::** - double colon or SQL function cast()) can help in some situations:
- Example here ensures calling of function **SQRT** is returning with the right types – i.e. float8:

```
create function fmax(startt int4, endt int4)
  returns float8 as $$
  begin
      return (sqrt(startt*endt)::float8);
  end;
  $$ language plpgsql;
```

J Vella – PostgreSQL Stored Functions

33



plpgsql - Loops

- Basic loop:

```
FOR i IN 1 .. 10  -- i has loop / end loop scope
LOOP
   salBal := salBal + emp.sal;
END LOOP;
```

- The while- loop statement is like this

```
WHILE deptCNT < 100  loop
   -- some more code;
END LOOP;
```

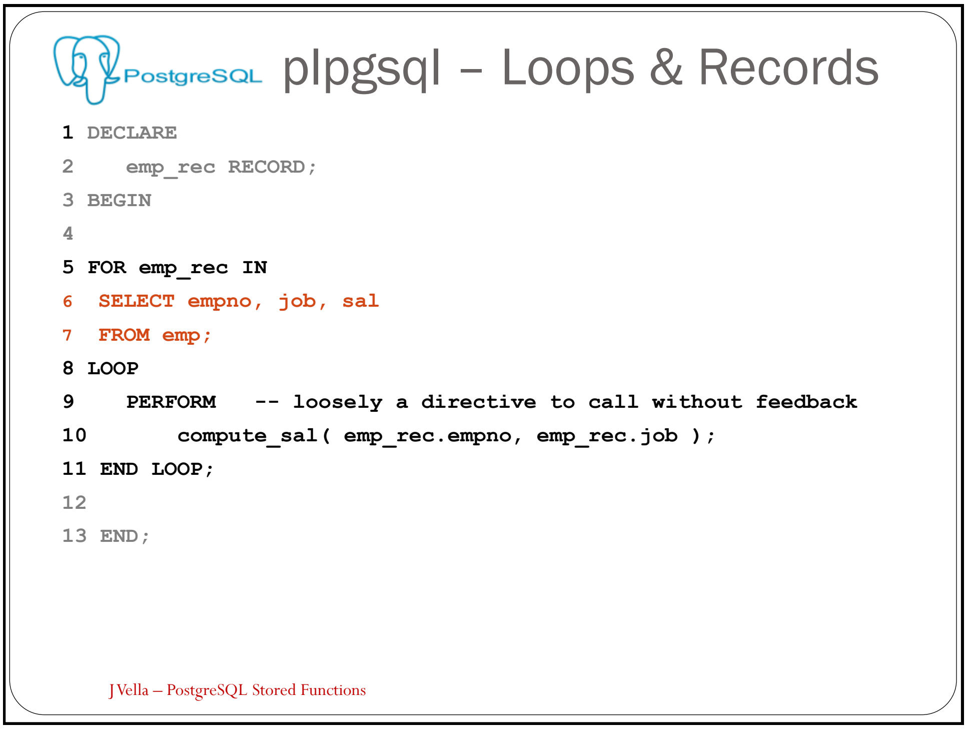
Nested Loops are allowed

- In any loop structure **RETURN** and **EXIT** stops the looping.
  - In fact LOOP / END LOOP is allowed!
    - Also goto labels are useful here !?

```
<< goto_label >>
```

J Vella – PostgreSQL Stored Functions

34

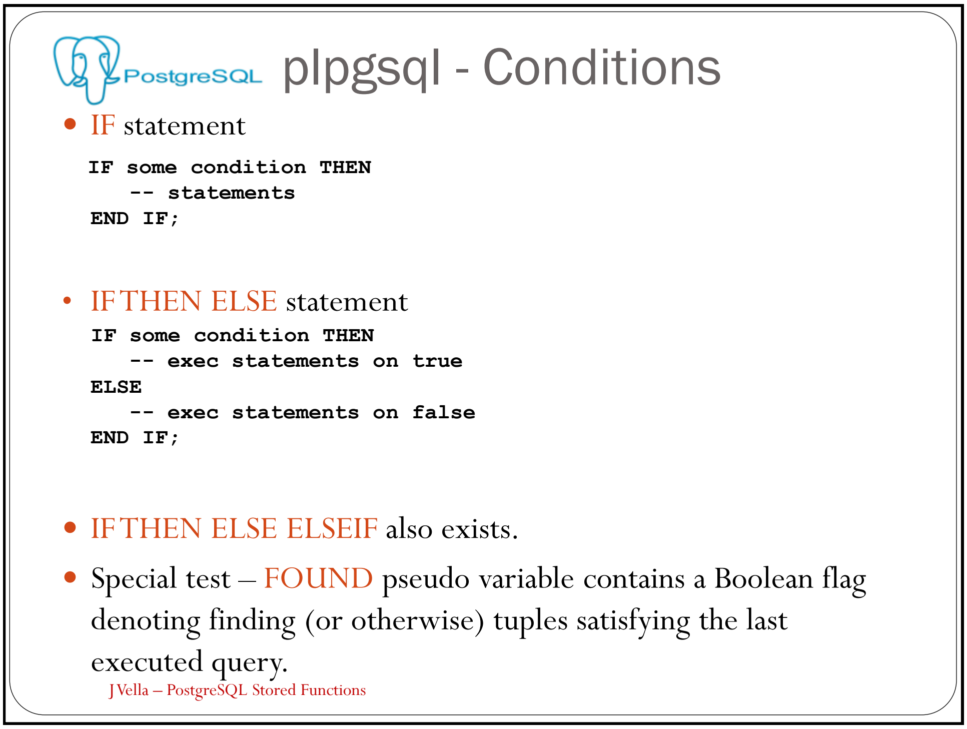


plpgsql – Loops & Records

```
DECLARE
   emp_rec RECORD;
BEGIN

FOR emp_rec IN
 SELECT empno, job, sal
 FROM emp;
LOOP
   PERFORM   -- loosely a directive to call without feedback
       compute_sal( emp_rec.empno, emp_rec.job );
 END LOOP;

 END;
```

J Vella – PostgreSQL Stored Functions

35



plpgsql - Conditions

- IF statement

```
IF some condition THEN
   -- statements
END IF;
```

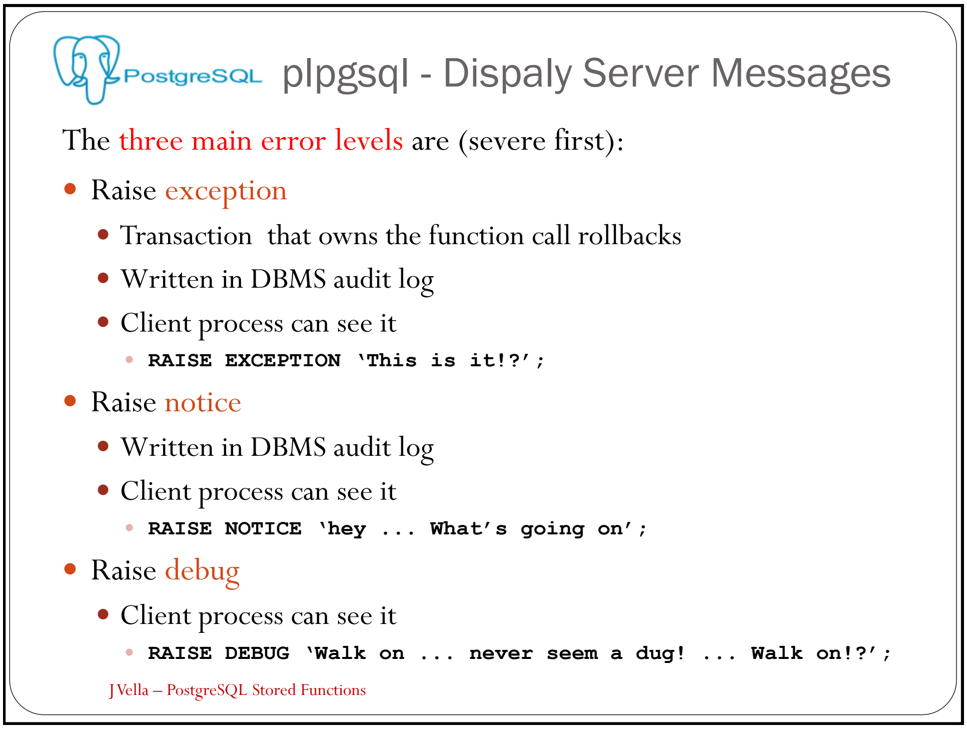
- IF THEN ELSE statement

```
IF some condition THEN
   -- exec statements on true
ELSE
   -- exec statements on false
END IF;
```

- IF THEN ELSE ELSEIF also exists.
- Special test – FOUND pseudo variable contains a Boolean flag denoting finding (or otherwise) tuples satisfying the last executed query.

J Vella – PostgreSQL Stored Functions

36



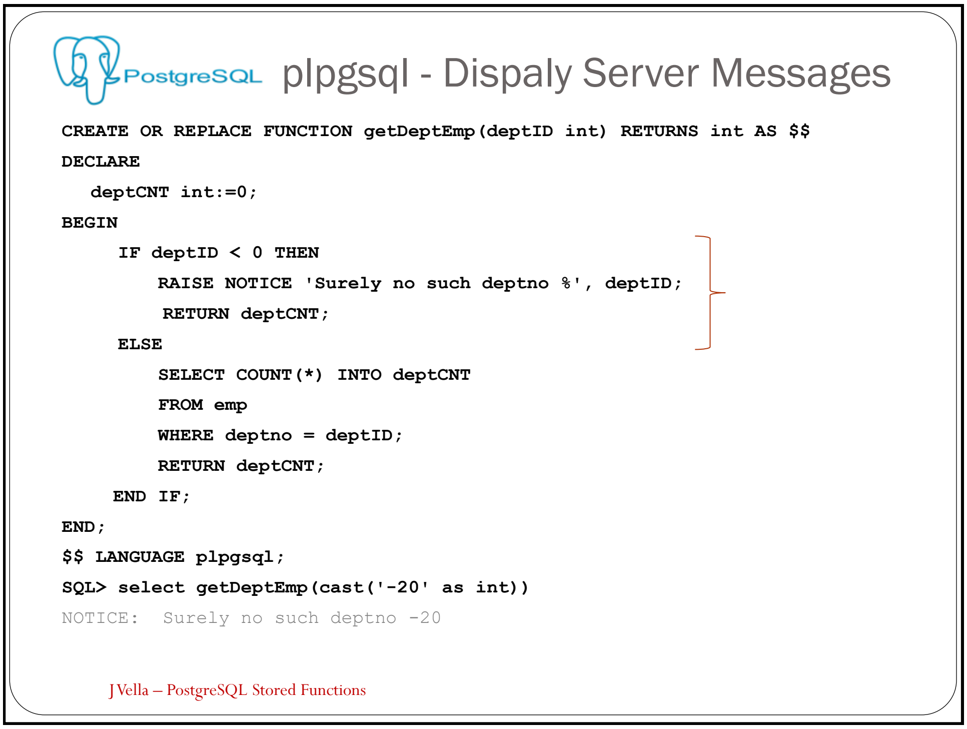
plpgsql - Dispaly Server Messages

The three main error levels are (severe first):

- Raise exception
  - Transaction that owns the function call rollbacks
  - Written in DBMS audit log
  - Client process can see it
    - `RAISE EXCEPTION 'This is it!?';`
- Raise notice
  - Written in DBMS audit log
  - Client process can see it
    - `RAISE NOTICE 'hey ... What's going on';`
- Raise debug
  - Client process can see it
    - `RAISE DEBUG 'Walk on ... never seem a dug! ... Walk on!?';`

J Vella – PostgreSQL Stored Functions

37

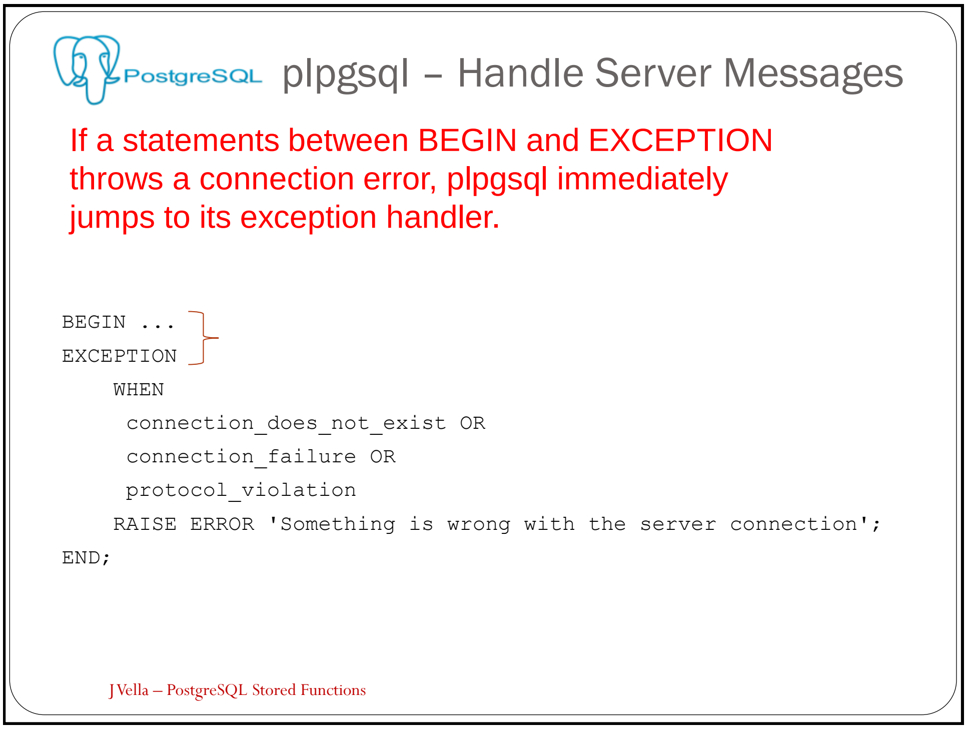


plpgsql - Dispaly Server Messages

```
CREATE OR REPLACE FUNCTION getDeptEmp(deptID int) RETURNS int AS $$
DECLARE
   deptCNT int:=0;
BEGIN
      IF deptID < 0 THEN
          RAISE NOTICE 'Surely no such deptno %', deptID;
           RETURN deptCNT;
      ELSE
          SELECT COUNT(*) INTO deptCNT
          FROM emp
          WHERE deptno = deptID;
          RETURN deptCNT;
     END IF;
END;
$$ LANGUAGE plpgsql;
SQL> select getDeptEmp(cast('-20' as int))
NOTICE:  Surely no such deptno -20
```

J Vella – PostgreSQL Stored Functions

38



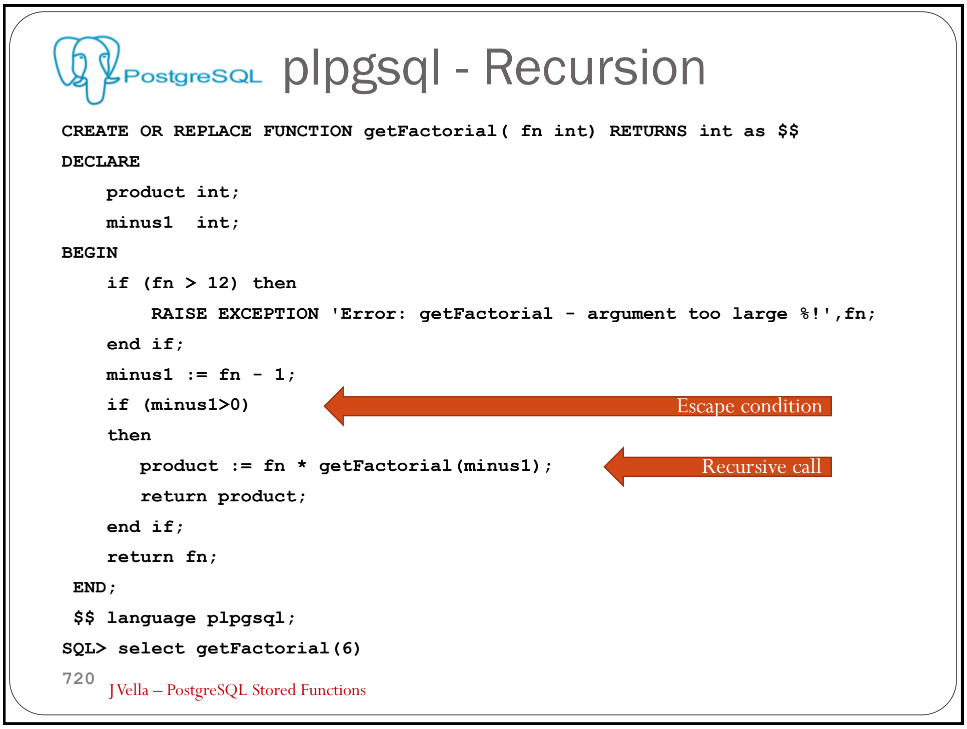
plpgsql – Handle Server Messages

If a statements between BEGIN and EXCEPTION throws a connection error, plpgsql immediately jumps to its exception handler.

```
BEGIN ...
EXCEPTION
    WHEN
     connection_does_not_exist OR
     connection_failure OR
     protocol_violation
    RAISE ERROR 'Something is wrong with the server connection';
END;
```

J Vella – PostgreSQL Stored Functions

39



plpgsql - Recursion

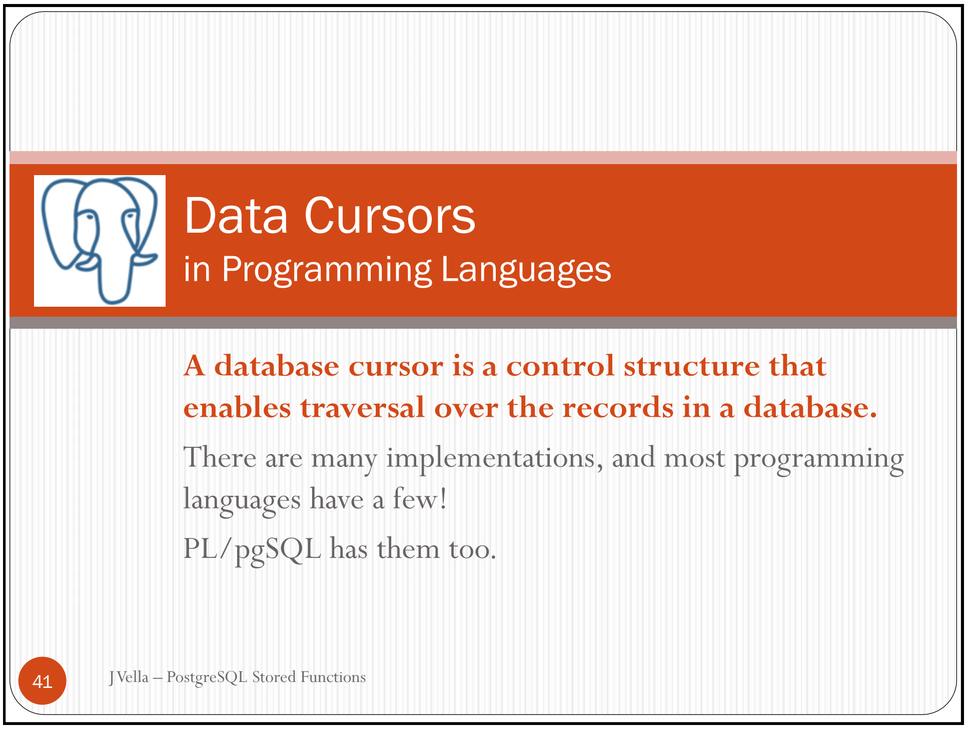
```
CREATE OR REPLACE FUNCTION getFactorial( fn int) RETURNS int as $$
DECLARE
    product int;
    minus1  int;
BEGIN
    if (fn > 12) then
        RAISE EXCEPTION 'Error: getFactorial - argument too large %!',fn;
    end if;
    minus1 := fn - 1;
    if (minus1>0)
    then
       product := fn * getFactorial(minus1);
       return product;
    end if;
    return fn;
 END;
 $$ language plpgsql;
SQL> select getFactorial(6)
720
```

Escape condition

Recursive call

J Vella – PostgreSQL Stored Functions

40



Data Cursors

in Programming Languages

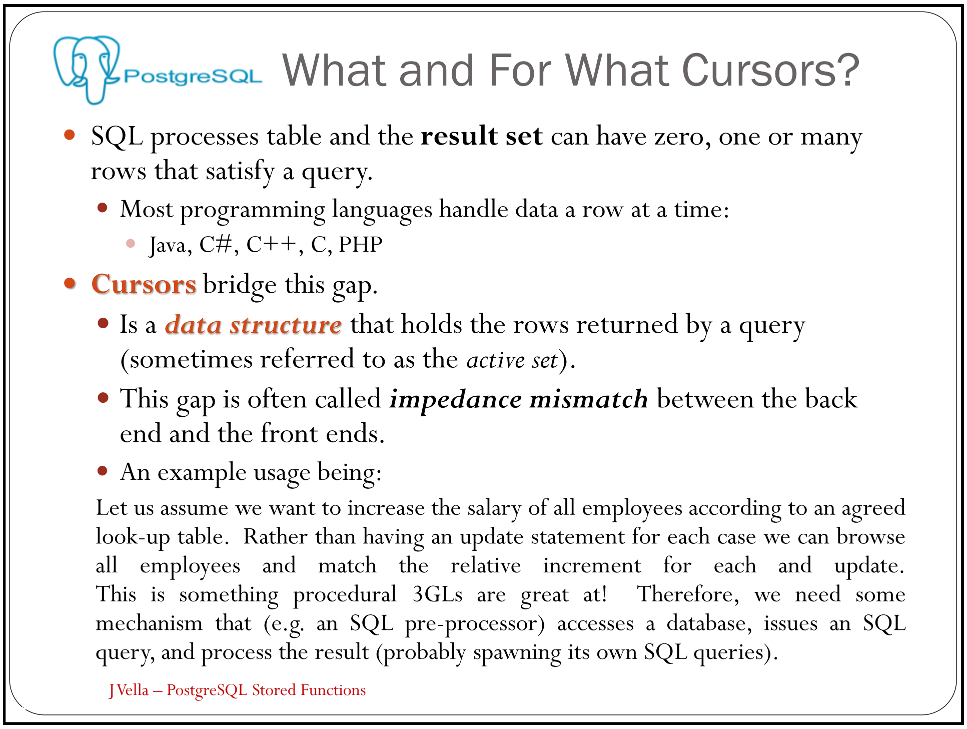
A database cursor is a control structure that enables traversal over the records in a database.

There are many implementations, and most programming languages have a few!

PL/pgSQL has them too.

41 J Vella – PostgreSQL Stored Functions

41



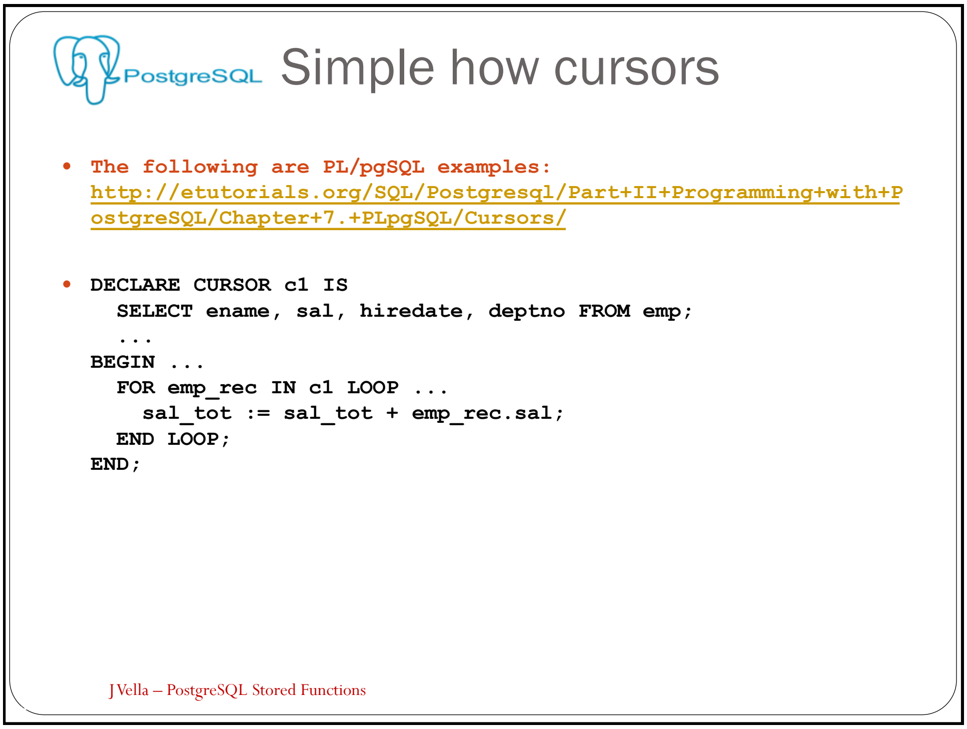
What and For What Cursors?

- SQL processes table and the **result set** can have zero, one or many rows that satisfy a query.
  - Most programming languages handle data a row at a time:
    - Java, C#, C++, C, PHP
- **Cursors** bridge this gap.
  - Is a ***data structure*** that holds the rows returned by a query (sometimes referred to as the *active set*).
  - This gap is often called ***impedance mismatch*** between the back end and the front ends.
  - An example usage being:

Let us assume we want to increase the salary of all employees according to an agreed look-up table. Rather than having an update statement for each case we can browse all employees and match the relative increment for each and update. This is something procedural 3GLs are great at! Therefore, we need some mechanism that (e.g. an SQL pre-processor) accesses a database, issues an SQL query, and process the result (probably spawning its own SQL queries).

J Vella – PostgreSQL Stored Functions

42



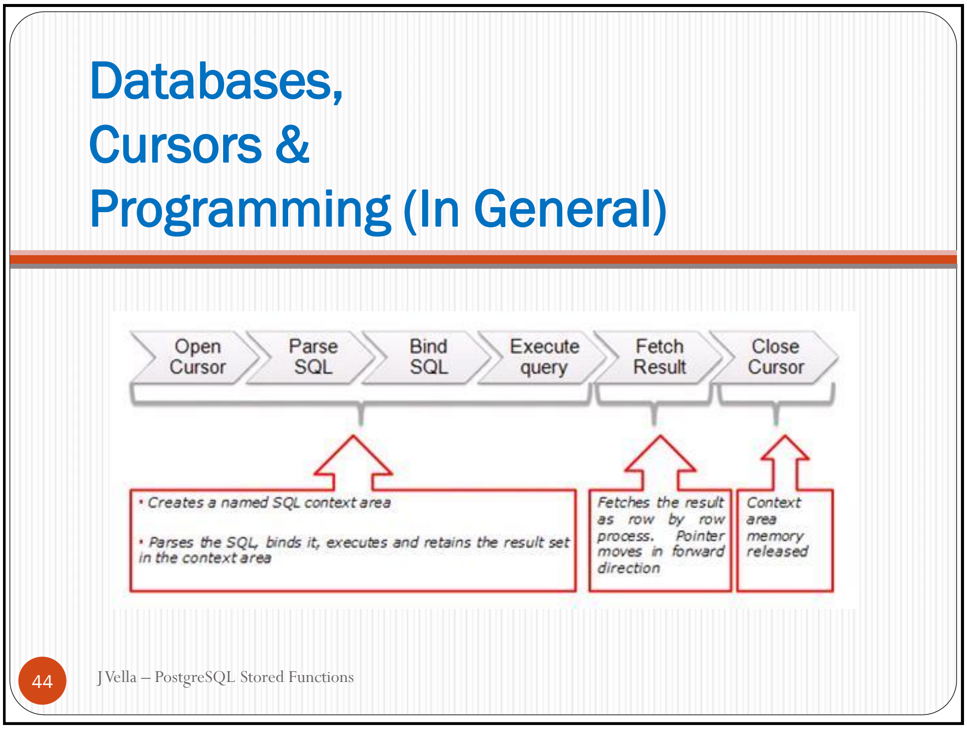
Simple how cursors

- **The following are PL/pgSQL examples: http://etutorials.org/SQL/Postgresql/Part+II+Programming+with+PostgreSQL/Chapter+7.+PLpgSQL/Cursors/**

-
```
DECLARE CURSOR c1 IS
  SELECT ename, sal, hiredate, deptno FROM emp;
  ...
BEGIN ...
  FOR emp_rec IN c1 LOOP ...
    sal_tot := sal_tot + emp_rec.sal;
  END LOOP;
END;
```

J Vella – PostgreSQL Stored Functions

43



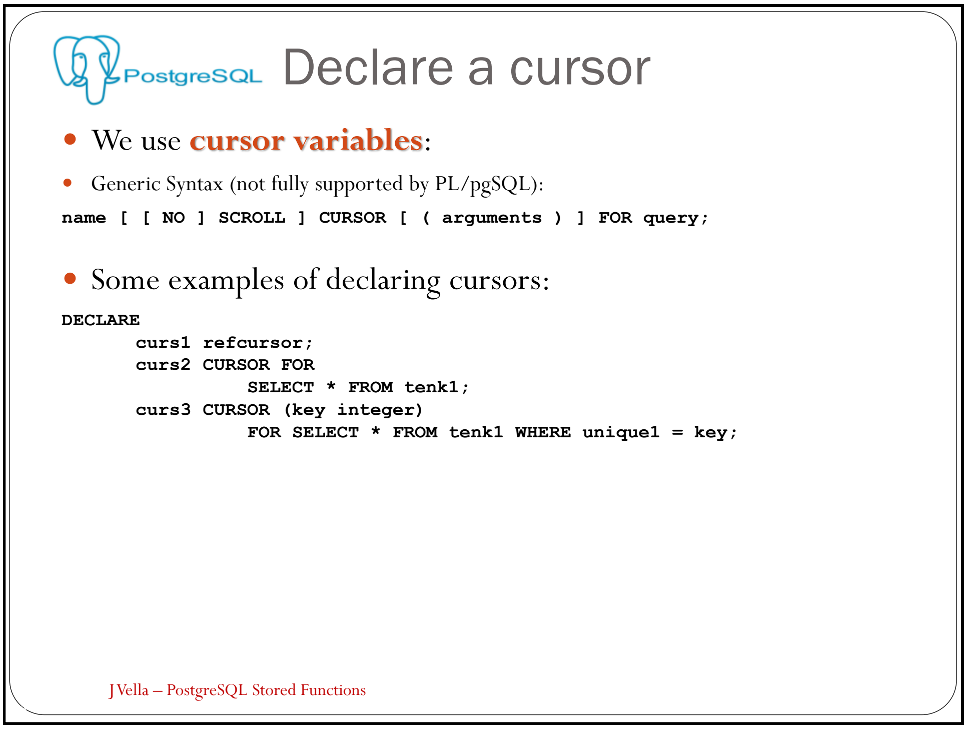
Databases, Cursors & Programming (In General)

Open Cursor → Parse SQL → Bind SQL → Execute query → Fetch Result → Close Cursor

- Creates a named SQL context area
- Parses the SQL, binds it, executes and retains the result set in the context area

Fetches the result as row by row process. Pointer moves in forward direction

Context area memory released



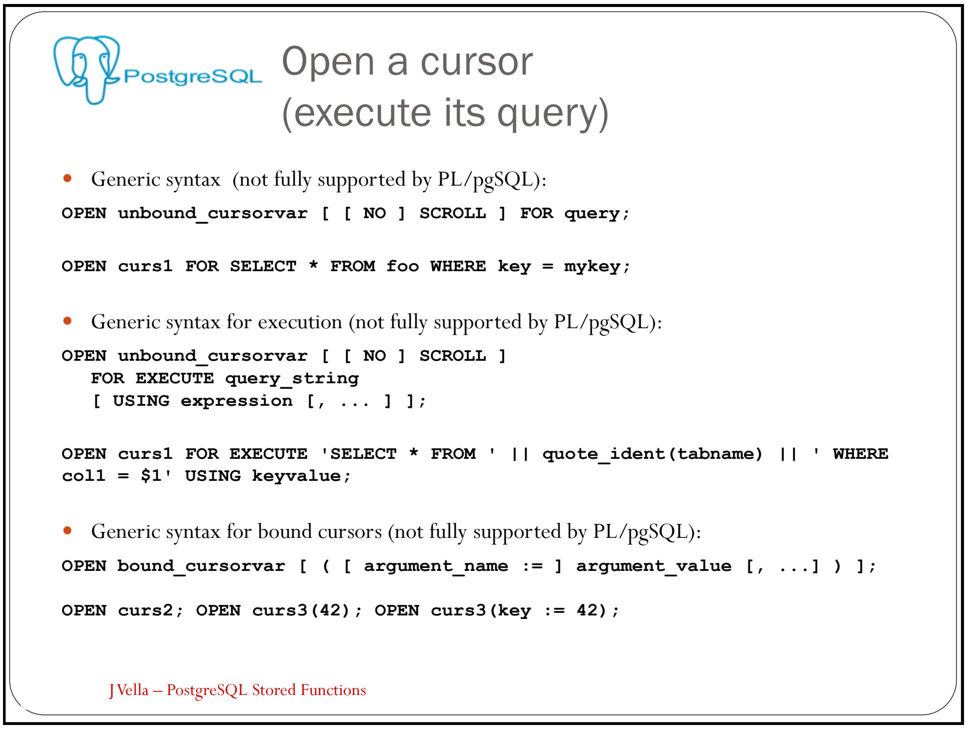
Declare a cursor

- We use **cursor variables**:
- Generic Syntax (not fully supported by PL/pgSQL):

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

- Some examples of declaring cursors:

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



Open a cursor (execute its query)

- Generic syntax (not fully supported by PL/pgSQL):

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;

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

- Generic syntax for execution (not fully supported by PL/pgSQL):

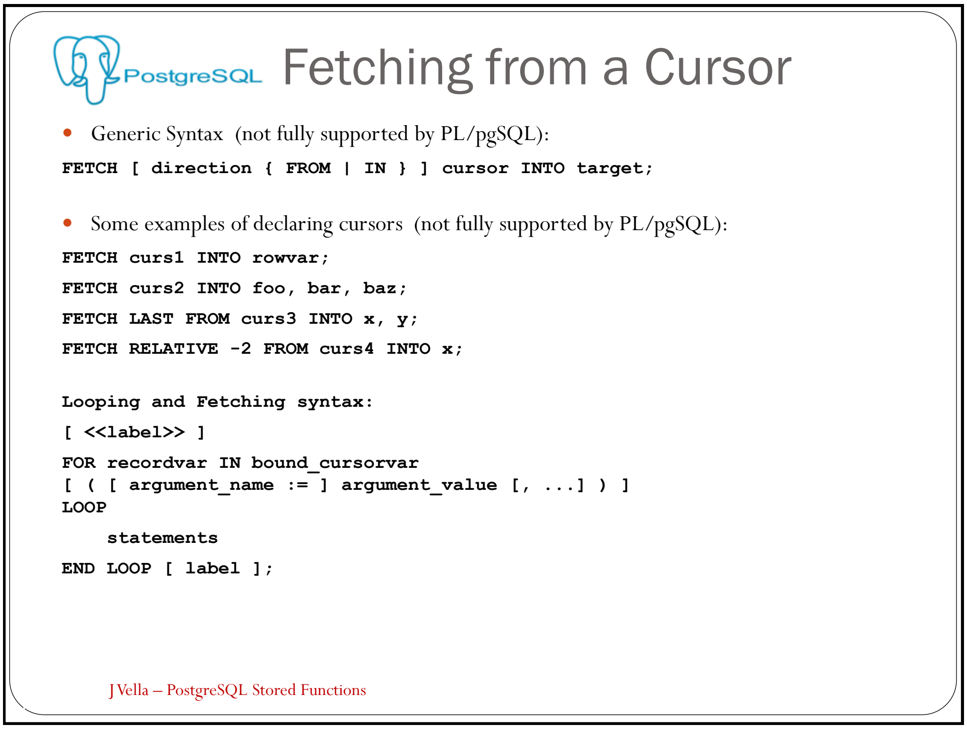
```
OPEN unbound_cursorvar [ [ NO ] SCROLL ]
   FOR EXECUTE query_string
   [ USING expression [, ... ] ];

OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident(tabname) || ' WHERE
col1 = $1' USING keyvalue;
```

- Generic syntax for bound cursors (not fully supported by PL/pgSQL):

```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];

OPEN curs2; OPEN curs3(42); OPEN curs3(key := 42);
```

J Vella – PostgreSQL Stored Functions

46



Fetching from a Cursor

- Generic Syntax (not fully supported by PL/pgSQL):

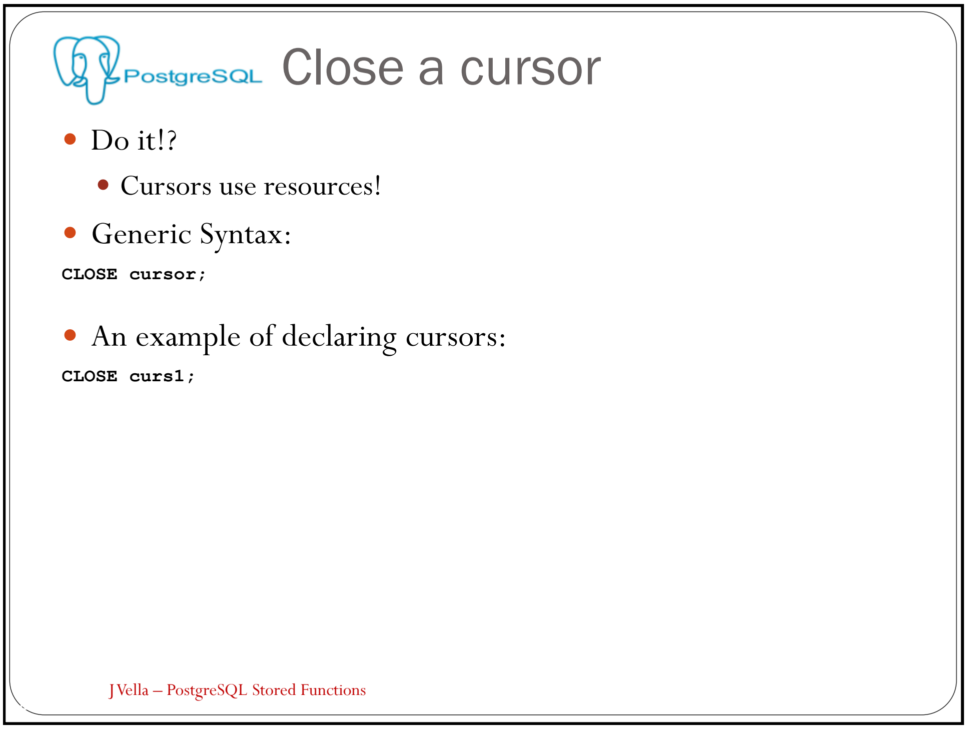
```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

- Some examples of declaring cursors (not fully supported by PL/pgSQL):

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;

Looping and Fetching syntax:
[ <<label>> ]
FOR recordvar IN bound_cursorvar
[ ( [ argument_name := ] argument_value [, ...] ) ]
LOOP
    statements
END LOOP [ label ];
```

J Vella – PostgreSQL Stored Functions

47



Close a cursor

- Do it!?
  - Cursors use resources!
- Generic Syntax:

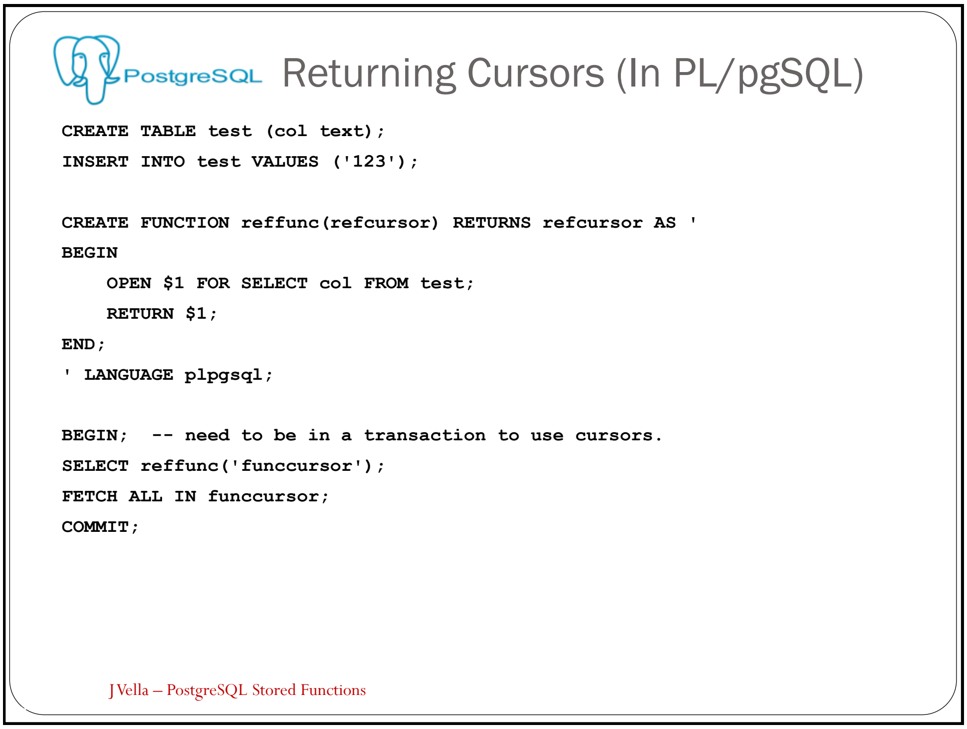
```
CLOSE cursor;
```

- An example of declaring cursors:

```
CLOSE curs1;
```

J Vella – PostgreSQL Stored Functions

48

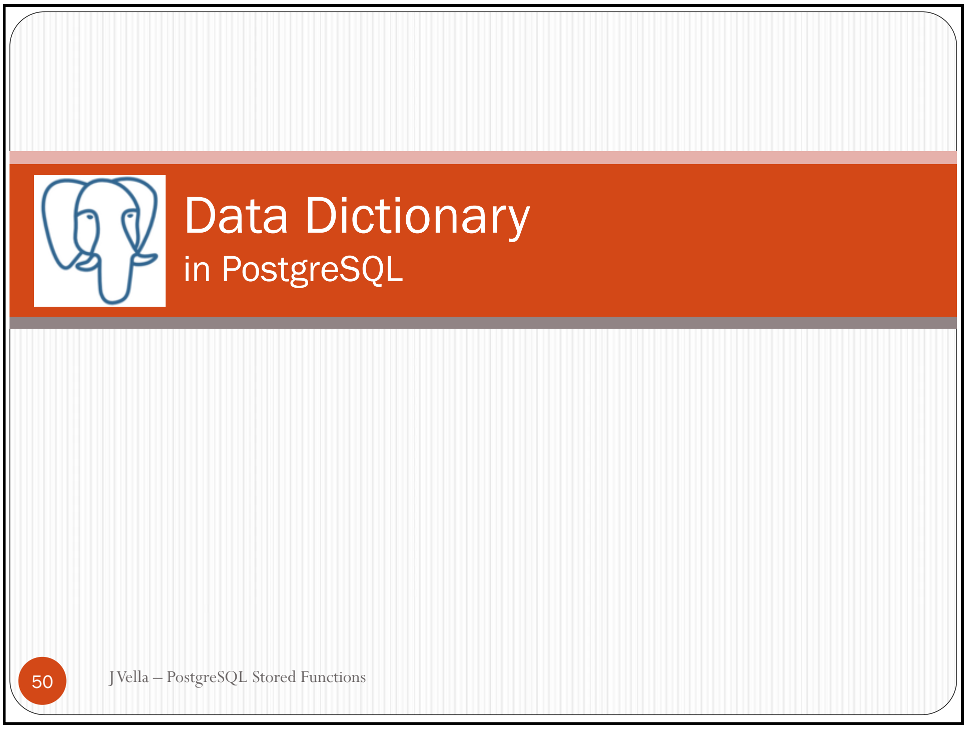


Returning Cursors (In PL/pgSQL)

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;  -- need to be in a transaction to use cursors.
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

J Vella – PostgreSQL Stored Functions

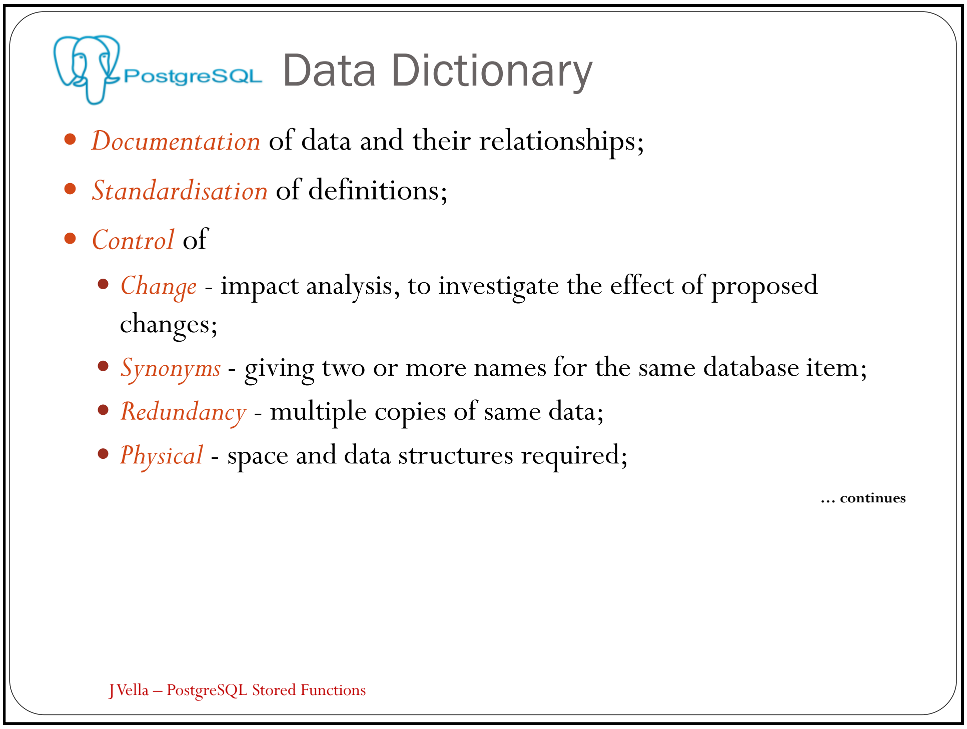
49



Data Dictionary
in PostgreSQL

50 J Vella – PostgreSQL Stored Functions

50



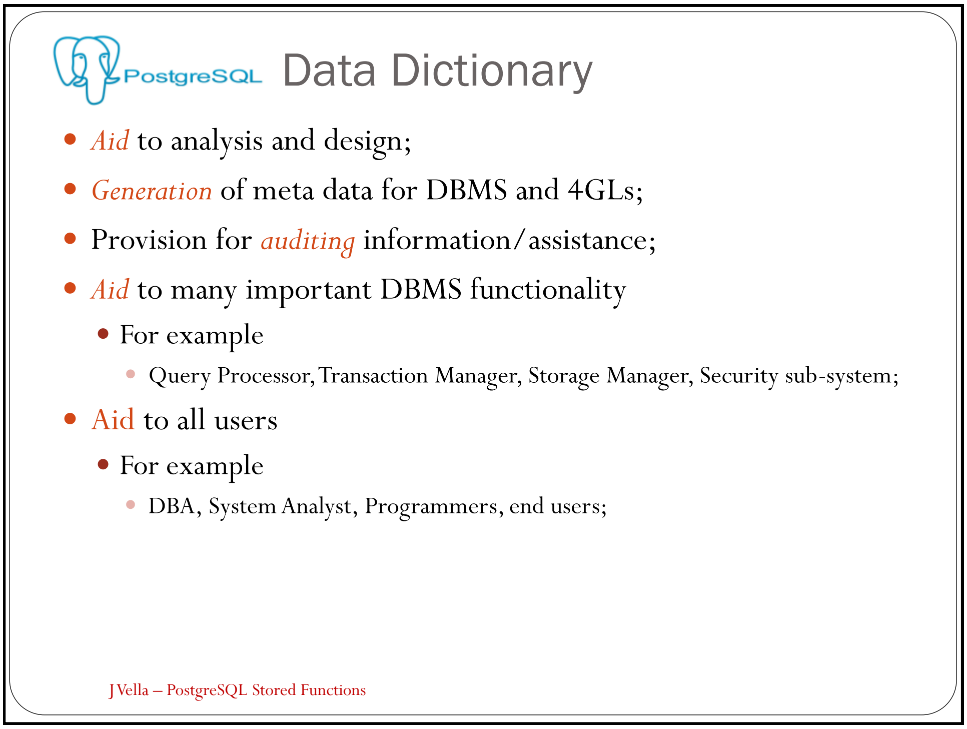
Data Dictionary

- *Documentation* of data and their relationships;
- *Standardisation* of definitions;
- *Control* of
  - *Change* - impact analysis, to investigate the effect of proposed changes;
  - *Synonyms* - giving two or more names for the same database item;
  - *Redundancy* - multiple copies of same data;
  - *Physical* - space and data structures required;

… continues

J Vella – PostgreSQL Stored Functions

51

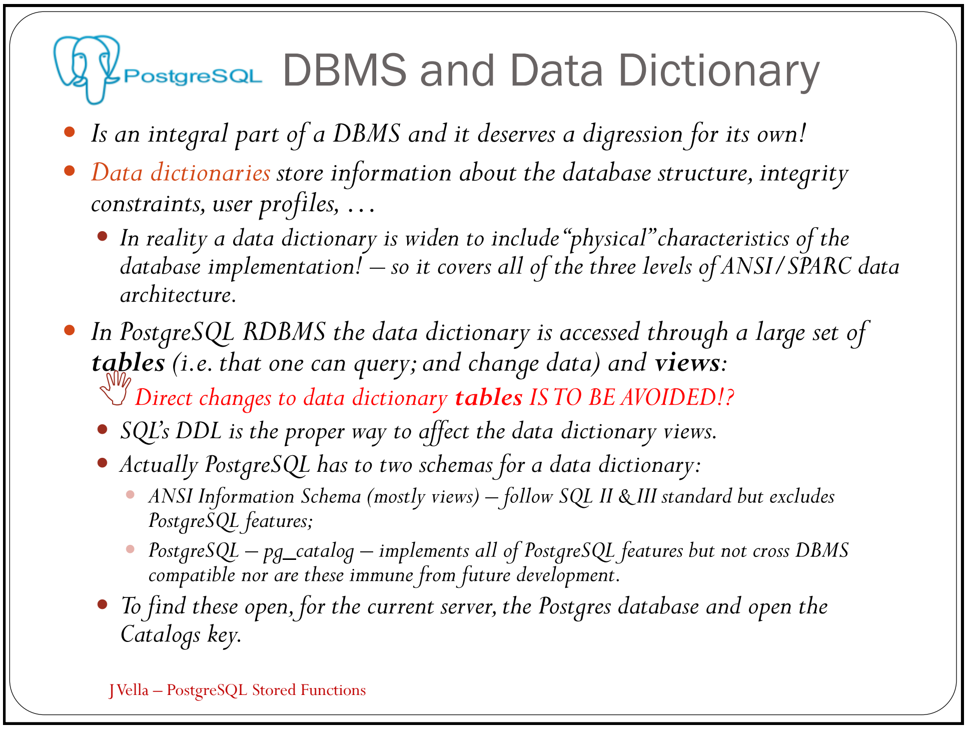


Data Dictionary

- *Aid* to analysis and design;
- *Generation* of meta data for DBMS and 4GLs;
- Provision for *auditing* information/assistance;
- *Aid* to many important DBMS functionality
  - For example
    - Query Processor, Transaction Manager, Storage Manager, Security sub-system;
- Aid to all users
  - For example
    - DBA, System Analyst, Programmers, end users;

J Vella – PostgreSQL Stored Functions

52



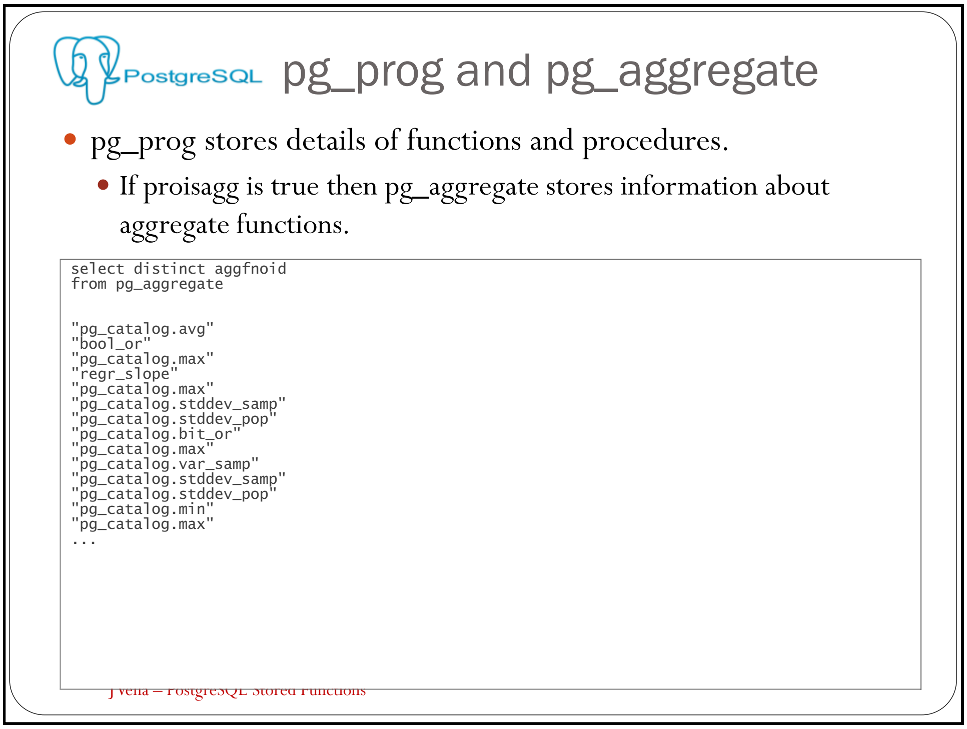
DBMS and Data Dictionary

- *Is an integral part of a DBMS and it deserves a digression for its own!*
- *Data dictionaries store information about the database structure, integrity constraints, user profiles, …*
  - *In reality a data dictionary is widen to include "physical" characteristics of the database implementation! – so it covers all of the three levels of ANSI/SPARC data architecture.*
- *In PostgreSQL RDBMS the data dictionary is accessed through a large set of* ***tables*** *(i.e. that one can query; and change data) and* ***views****:*

  Direct changes to data dictionary ***tables*** *IS TO BE AVOIDED!?*
  - *SQL's DDL is the proper way to affect the data dictionary views.*
  - *Actually PostgreSQL has to two schemas for a data dictionary:*
    - *ANSI Information Schema (mostly views) – follow SQL II & III standard but excludes PostgreSQL features;*
    - *PostgreSQL – pg_catalog – implements all of PostgreSQL features but not cross DBMS compatible nor are these immune from future development.*
  - *To find these open, for the current server, the Postgres database and open the Catalogs key.*

J Vella – PostgreSQL Stored Functions

53



pg_prog and pg_aggregate

- pg_prog stores details of functions and procedures.
  - If proisagg is true then pg_aggregate stores information about aggregate functions.

```
select distinct aggfnoid
from pg_aggregate


"pg_catalog.avg"
"bool_or"
"pg_catalog.max"
"regr_slope"
"pg_catalog.max"
"pg_catalog.stddev_samp"
"pg_catalog.stddev_pop"
"pg_catalog.bit_or"
"pg_catalog.max"
"pg_catalog.var_samp"
"pg_catalog.stddev_samp"
"pg_catalog.stddev_pop"
"pg_catalog.min"
"pg_catalog.max"
...
```

J Vella – PostgreSQL Stored Functions

54