>>

PLpgSQL (iii)

- PLpgSQL Functions (recap)
- Query results in PLpgSQL
- Dynamically Generated Queries
- Functions vs Views

^ >>

❖ PLpgSQL Functions (recap)

Defining PLpgSQL functions:

```
CREATE OR REPLACE
    funcName(param1, param2, ....)
    RETURNS rettype
AS $$
DECLARE
    variable declarations
BEGIN
    code for function
END;
$$ LANGUAGE plpgsql;
```

<< ^ >>

❖ Query results in PLpgSQL

Can evaluate a query and iterate through its results

- one tuple at a time, using a `for ... loop`

```
declare
   tup Type;
begin
   for tup in Query
   loop
      Statements;
   end loop;
end;
```

Type of `tup` variable must match type of *Query* results

If declared as `record`, will automatically match *Query* results type

<< ^ >>

❖ Query results in PLpgSQL (cont)

Example: count the number of Employees earning more than min.salary

```
create or replace function
   well_paid(_minsal integer) returns integer
as $$
declare
   nemps integer := 0;
   tuple record;  -- could also be tuple Employees;
begin
   for tuple in
      select * from Employees where salary > _minsal
   loop
      nemps := nemps + 1;
   end loop;
   return nemps;
end;
$$ language plpgsql;
```

<< ^ >>

❖ Query results in PLpgSQL (cont)

Alternative to the above (but less efficient):

```
create or replace function
   well_paid(_minsal integer) returns integer
as $$
declare
   nemps integer := 0;
   tuple record;
begin
   for tuple in
      select name,salary from Employees
   loop
      if (tuple.salary > _minsal) then
         nemps := nemps + 1;
      end if;
   end loop;
   return nemps;
end;
$$ language plpgsql;
```

<< ^ >>

❖ Query results in PLpgSQL (cont)

And the example could be done more simply (and efficiently) as:

```
create or replace function
   well_paid(_minsal integer) returns integer
as $$
declare
   nemps integer;
begin
   select count(*) into nemps
   from   Employees where salary > _minsal
   return nemps;
end;
$$ language plpgsql;
```

<< ^ >>

❖ Dynamically Generated Queries

EXECUTE takes a string and executes it as an SQL query.

Examples:

```
-- constant string
execute 'select * from Employees';
-- concatenation of constant strings
execute 'select * from '||'Employees';
-- using a name of e.g. table or attribute
execute 'select * from '||quote_ident($1);
-- using a value generated in the program
execute 'delete from Accounts' ||
        ' where holder='||quote_literal($1);
```

Can be used in any context where an SQL query is expected

This mechanism allows us to construct queries "on the fly".

<< ^ >>

❖ Dynamically Generated Queries (cont)

Example: a wrapper for updating a single text field

```
create or replace function
   set(_table text, _attr text, _val text) returns void
as $$
declare
   query text;
begin
   query := 'update ' || quote_ident(_table);
   query := query || ' SET ' || quote_ident(_attr);
   query := query || ' = ' || quote_literal(_val);
   execute query;
end; $$ language plpgsql;
```

which could be used as e.g.

```
select set('branches','assets','0.00');
```

<< ^ >>

❖ Dynamically Generated Queries (cont)

One limitation of EXECUTE:

- cannot use `select into` inside dynamic queries

Needs to be expressed instead as:

```
declare tuple R%rowtype; n int;
execute 'select * from R where id='||n into tuple;
-- or
declare x int; y int; z text;
execute 'select a,b,c from R where id='||n into x,y,z;
```

Notes:

- if query returns multiple tuples, first one is stored
- if query returns zero tuples, all nulls are stored

<< ^ >>

❖ Functions vs Views

A difference between views and functions returning a `SETOF`:

- `CREATE VIEW` produces a "virtual" table definition
- `SETOF` functions require an existing tuple type

In examples above, we used existing `Employees` tuple type.

In general, you need to define the tuple return type via

```
create type TupleType as ( attr1 type1, ... attrn typen );
```

Other major differences between `setof` functions and views ...

- functions have parameters; views don't (although `where` might help)
- functions are "run-time" objects; views are interpolated into queries

<< ^ >>

❖ Functions vs Views (cont)

Another example of function returning `setof` tuples ...

```
create type EmpInfo as (name text, pay integer);

create or replace function
   richEmps(_minsal integer) returns setof EmpInfo
as $$
declare
   emp record;   info EmpInfo;
begin
   for emp in
      select * from Employees where salary > _minsal
   loop
      info.name := emp.name;
      info.pay := emp.salary;
      return next info;
   end loop;
end; $$ language plpgsql;
```

<< ^

❖ Functions vs Views (cont)

Using the function ...

```
select * from richEmps(100000);
```

versus a view

```
create or repalce view richEmps(name,pay) as
select name, salary from Employees where salary > 100000;

select * from richEmps;  -- but no scope for different salary
```

versus an SQL function

```
create or replace function
   richEmps(_minsal integer) returns setof EmpInfo
as $$
select name, salary from Employees where salary > _minsal;
$$ language sql;
```

Produced: 27 Feb 2021