CS307

Procedures

This lab will cover in some detail procedures, especially in the case of PostgreSQL.

STORE FOR REUSE

Like functions, stored procedures are a way to store something complex for reuse. However, procedures aren't complex expressions on one column but, usually, a complex series of operations that change the database. Procedures are often seriously misused, and most tutorials and books use very questionable examples. Let's put the record straight.

Sequence of change operations

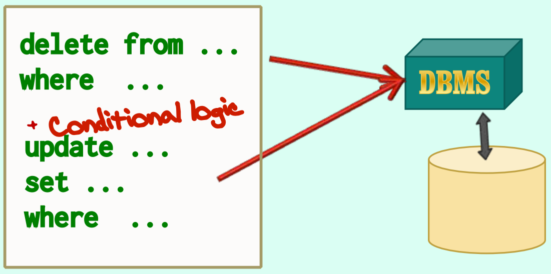
As previously said, procedures should change the database. Some people write procedures that just query, but there is another database object (called a view) for storing queries.

Single business process

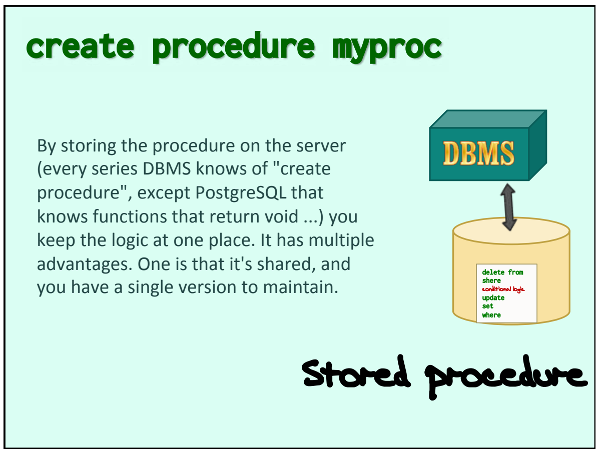
Procedures usually embody a single "business process", that may not exactly match database transactions. They may also help ensure "business constraints" that cannot easily be defined with the declarative constraints of SQL.

Avoid `begin transaction`/`commit` inside

You should think twice before adding transaction management to a procedure, because "commit" is a slow operation (it usually means writing to a file) and it may hurt if the procedure is called repeatedly.

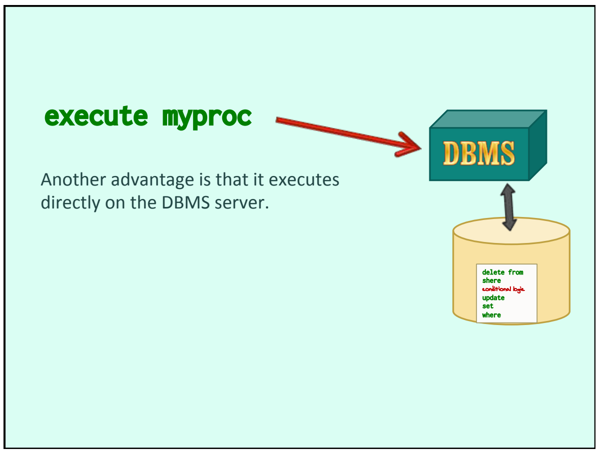

```
delete from ...
where  ...
+ conditional logic
update ...
set ...
where  ...
```

If you run a business procedure from another computer, you are going to send statements, wait for a return code/status, issue another statement and repeat. When the server is remote (in the cloud) you can spend a lot of time in the network.



create procedure myproc

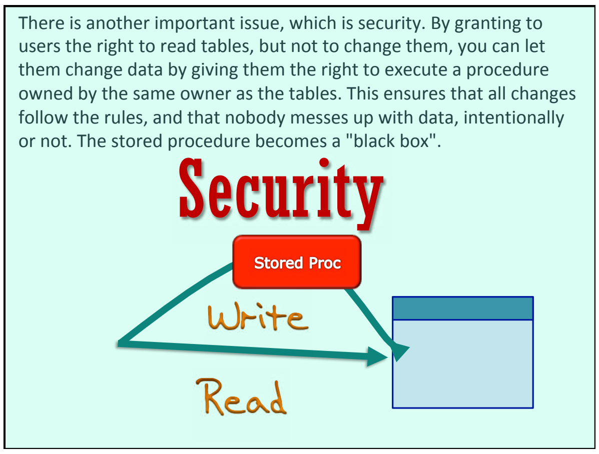
By storing the procedure on the server (every series DBMS knows of "create procedure", except PostgreSQL that knows functions that return void ...) you keep the logic at one place. It has multiple advantages. One is that it's shared, and you have a single version to maintain.



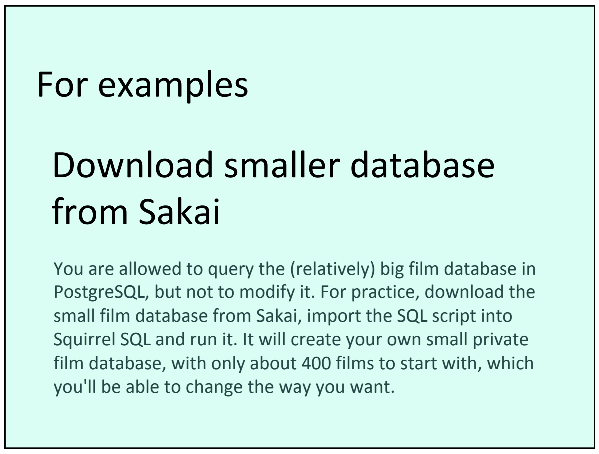
execute myproc

Another advantage is that it executes directly on the DBMS server.

With a single call and all the logic being tested on the server, you save a lot on network exchanges.



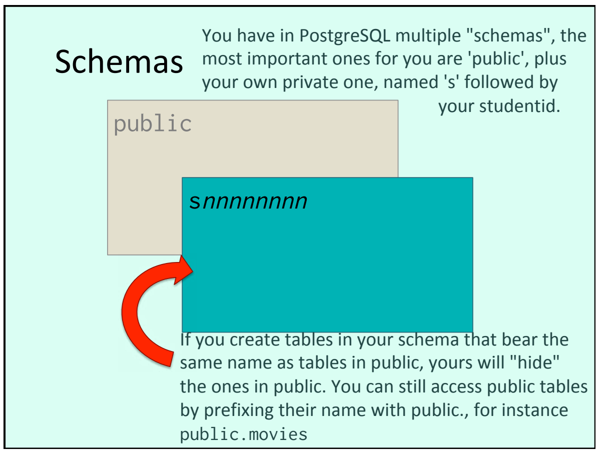
There is another important issue, which is security. By granting to users the right to read tables, but not to change them, you can let them change data by giving them the right to execute a procedure owned by the same owner as the tables. This ensures that all changes follow the rules, and that nobody messes up with data, intentionally or not. The stored procedure becomes a "black box".



For examples

Download smaller database from Sakai

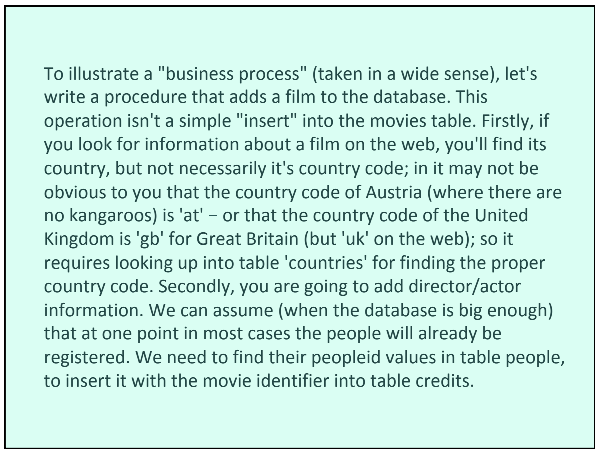
You are allowed to query the (relatively) big film database in PostgreSQL, but not to modify it. For practice, download the small film database from Sakai, import the SQL script into Squirrel SQL and run it. It will create your own small private film database, with only about 400 films to start with, which you'll be able to change the way you want.



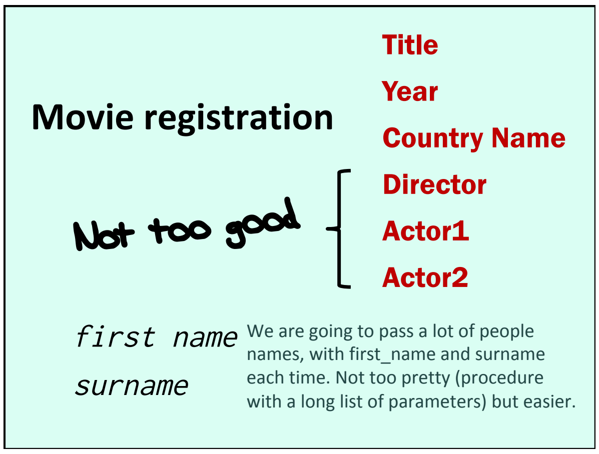
Schemas

You have in PostgreSQL multiple "schemas", the most important ones for you are 'public', plus your own private one, named 's' followed by your studentid.

If you create tables in your schema that bear the same name as tables in public, yours will "hide" the ones in public. You can still access public tables by prefixing their name with public., for instance `public.movies`



To illustrate a "business process" (taken in a wide sense), let's write a procedure that adds a film to the database. This operation isn't a simple "insert" into the movies table. Firstly, if you look for information about a film on the web, you'll find its country, but not necessarily it's country code; in it may not be obvious to you that the country code of Austria (where there are no kangaroos) is 'at' – or that the country code of the United Kingdom is 'gb' for Great Britain (but 'uk' on the web); so it requires looking up into table 'countries' for finding the proper country code. Secondly, you are going to add director/actor information. We can assume (when the database is big enough) that at one point in most cases the people will already be registered. We need to find their peopleid values in table people, to insert it with the movie identifier into table credits.



Movie registration

Title

Year

Country Name

Not too good {

Director

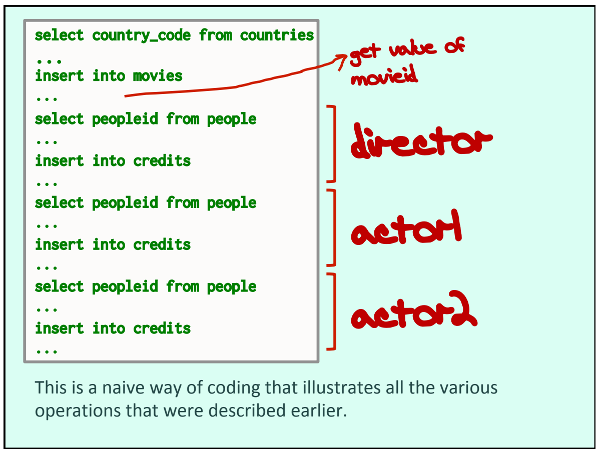
Actor1

Actor2

first name

surname

We are going to pass a lot of people names, with first_name and surname each time. Not too pretty (procedure with a long list of parameters) but easier.



```
select country_code from countries
...
insert into movies
...
select peopleid from people
...
insert into credits
...
select peopleid from people
...
insert into credits
...
select peopleid from people
...
insert into credits
...
```

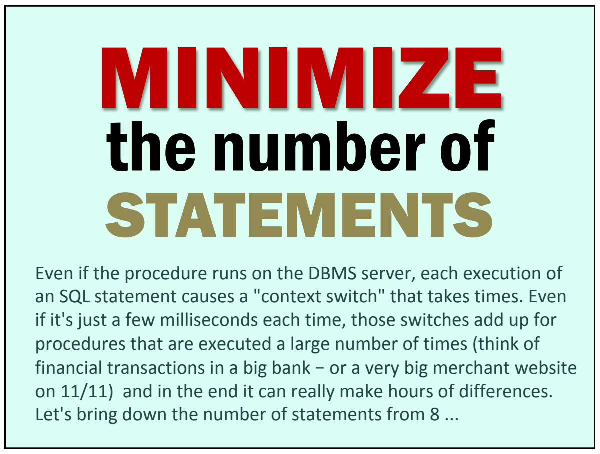
get value of movieid

director

actor1

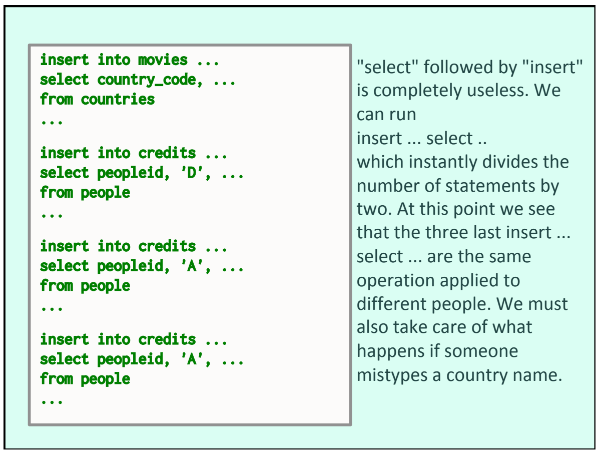
actor2

This is a naive way of coding that illustrates all the various operations that were described earlier.



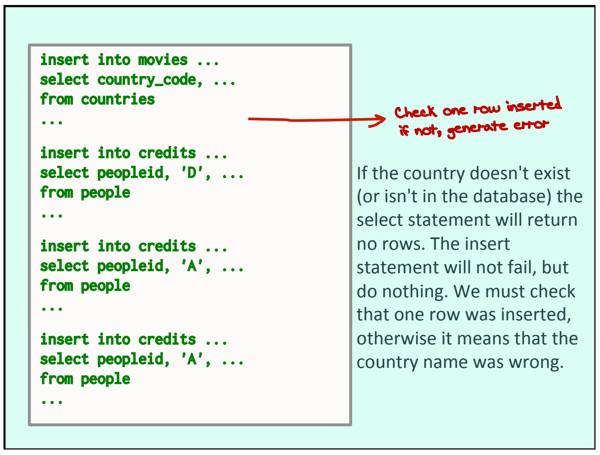
MINIMIZE the number of STATEMENTS

Even if the procedure runs on the DBMS server, each execution of an SQL statement causes a "context switch" that takes times. Even if it's just a few milliseconds each time, those switches add up for procedures that are executed a large number of times (think of financial transactions in a big bank – or a very big merchant website on 11/11) and in the end it can really make hours of differences. Let's bring down the number of statements from 8 ...



```
insert into movies ...
select country_code, ...
from countries
...
insert into credits ...
select peopleid, 'D', ...
from people
...
insert into credits ...
select peopleid, 'A', ...
from people
...
insert into credits ...
select peopleid, 'A', ...
from people
...
```

"select" followed by "insert" is completely useless. We can run insert ... select .. which instantly divides the number of statements by two. At this point we see that the three last insert ... select ... are the same operation applied to different people. We must also take care of what happens if someone mistypes a country name.



```
insert into movies ...
select country_code, ...
from countries
...
insert into credits ...
select peopleid, 'D', ...
from people
...
insert into credits ...
select peopleid, 'A', ...
from people
...
insert into credits ...
select peopleid, 'A', ...
from people
...
```

Check one row inserted
if not, generate error

If the country doesn't exist (or isn't in the database) the select statement will return no rows. The insert statement will not fail, but do nothing. We must check that one row was inserted, otherwise it means that the country name was wrong.

```
insert into movies ...
select country_code, ...
from countries
...

insert into credits ...
select peopleid, ...
from (select director, 'D', ...
      union all
      select actor1, 'A', ...
      union all
      select actor2, 'A', ...) a
     inner join people
...
```

From 8 statements down to 2

Creative SQL allows to combine the three insert ... select for people into one. A UNION ALL statement that joins the three passed parameters into a kind of virtual can be joined to people to retrieve the corresponding identifiers.

Check rows inserted

We can check if we have inserted as many people as we wanted (otherwise some are missing from the database)

Function that returns void

(PostgreSQL only)

The logic given previously works with any database system that supports stored procedures. The way you check how many rows were inserted isn't the same every where, the way you generate errors may also vary , but these are relatively small details (just keep the docs at hand). Once again, PostgreSQL is the only product for which, probably under the influence of languages such as C or Java, procedures are just functions that return nothing.

```
create function movie_registration
          (p_title        varchar,
           p_country_name varchar,
           p_year         int,
           p_director_fn  varchar,
           p_director_sn  varchar,
           p_actor1_fn    varchar,
           p_actor1_sn    varchar,
           p_actor2_fn    varchar,
           p_actor2_sn    varchar)
returns void
as $$
declare
   n_rowcount int;
   n_movieid  int;
   n_people   int;
begin
  insert into movies(title, country, year_released)
  select p_title, country_code, p_year
  from countries
  where country_name = p_country_name;
```

Ugly list of parameters.

We need very few variables: one to store the count of rows just inserted, one to store the system-generated movieid, one to count the number of people supplied.

Very long lists of variables are often a bad sign.

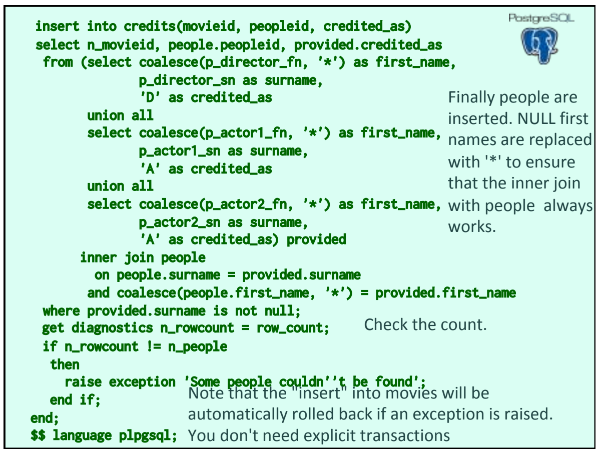
```
insert into movies(title, country, year_released)
select p_title, country_code, p_year
from countries
where country_name = p_country_name;
get diagnostics n_rowcount = row_count;
if n_rowcount = 0
then
  raise exception 'country not found in table COUNTRIES';
end if;
n_movieid := lastval();
select count(surname)
into n_people
from (select p_director_sn as surname
      union all
      select p_actor1_sn as surname
      union all
      select p_actor2_sn as surname) specified_people
where surname is not null;
```

"get diagnostics" is how PostgreSQL, DB2 and MySQL (out of memory) retrieve a how many rows were just inserted, updated or deleted.

With some products (SQL Server?) "raise" must be followed by "return". "raise exception" is the same as "throw" in Java.

If it worked, a function can always retrieve the last identifier generated by the system (lastval() with PostgreSQL)

The last query doesn't query the database at all, it just uses SQL to count how many of the passed names are not null.

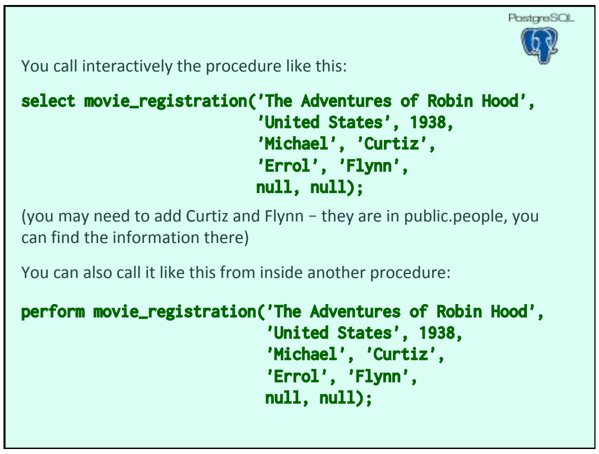


```
insert into credits(movieid, peopleid, credited_as)
select n_movieid, people.peopleid, provided.credited_as
 from (select coalesce(p_director_fn, '*') as first_name,
              p_director_sn as surname,
              'D' as credited_as
       union all
       select coalesce(p_actor1_fn, '*') as first_name,
              p_actor1_sn as surname,
              'A' as credited_as
       union all
       select coalesce(p_actor2_fn, '*') as first_name,
              p_actor2_sn as surname,
              'A' as credited_as) provided
      inner join people
        on people.surname = provided.surname
       and coalesce(people.first_name, '*') = provided.first_name
 where provided.surname is not null;
 get diagnostics n_rowcount = row_count;
 if n_rowcount != n_people
  then
    raise exception 'Some people couldn''t be found';
  end if;
end;
$$ language plpgsql;
```

Finally people are inserted. NULL first names are replaced with '*' to ensure that the inner join with people always works.

Check the count.

Note that the "insert" into movies will be automatically rolled back if an exception is raised. You don't need explicit transactions



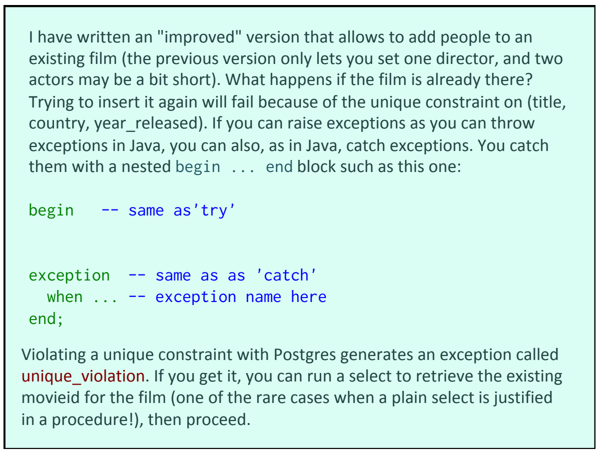
You call interactively the procedure like this:

```
select movie_registration('The Adventures of Robin Hood',
                          'United States', 1938,
                          'Michael', 'Curtiz',
                          'Errol', 'Flynn',
                          null, null);
```

(you may need to add Curtiz and Flynn – they are in public.people, you can find the information there)

You can also call it like this from inside another procedure:

```
perform movie_registration('The Adventures of Robin Hood',
                           'United States', 1938,
                           'Michael', 'Curtiz',
                           'Errol', 'Flynn',
                           null, null);
```

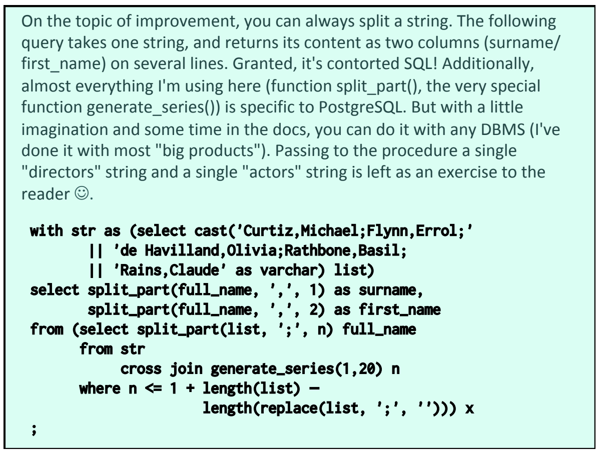


I have written an "improved" version that allows to add people to an existing film (the previous version only lets you set one director, and two actors may be a bit short). What happens if the film is already there? Trying to insert it again will fail because of the unique constraint on (title, country, year_released). If you can raise exceptions as you can throw exceptions in Java, you can also, as in Java, catch exceptions. You catch them with a nested `begin ... end` block such as this one:

```
begin   -- same as'try'


exception  -- same as as 'catch'
  when ... -- exception name here
end;
```

Violating a unique constraint with Postgres generates an exception called unique_violation. If you get it, you can run a select to retrieve the existing movieid for the film (one of the rare cases when a plain select is justified in a procedure!), then proceed.



On the topic of improvement, you can always split a string. The following query takes one string, and returns its content as two columns (surname/ first_name) on several lines. Granted, it's contorted SQL! Additionally, almost everything I'm using here (function split_part(), the very special function generate_series()) is specific to PostgreSQL. But with a little imagination and some time in the docs, you can do it with any DBMS (I've done it with most "big products"). Passing to the procedure a single "directors" string and a single "actors" string is left as an exercise to the reader ☺.

```
with str as (select cast('Curtiz,Michael;Flynn,Errol;'
       || 'de Havilland,Olivia;Rathbone,Basil;
       || 'Rains,Claude' as varchar) list)
select split_part(full_name, ',', 1) as surname,
       split_part(full_name, ',', 2) as first_name
from (select split_part(list, ';', n) full_name
      from str
           cross join generate_series(1,20) n
      where n <= 1 + length(list) –
                     length(replace(list, ';', ''))) x
;
```

To retrieve the identifier of a film already in the database, I can execute a select ... into. This, however, only works because I should get a single row. Getting none or several would raise exceptions. Although many products allow to declare some kind of "collection variables" (arrays, in memory tables) and select into them, the "official" way to deal with a select returning several rows is to use a cursor.

```
select col1, col2, ...
into local_var1, local_var2, ...
from ...
```

+ CURSORS

I hate cursors. People using cursors use tables the way they would use files.

```
declare
    c cursor for select ...;
begin
   for row_var in c
   loop
     ...
   end loop;
   ...
end;
```

This is the simplest syntax to use cursors. A cursor is a special variable associated with a query (which may contain parameters). The for loop automatically defines the "row_var" (a name you choose) variable that represents the current row. Inside the loop, you can refer to `row_var.col_name` (give an alias to expressions in the query!)

Did I mention that I hate cursors? The problem is that almost every tutorial or book giving cursor examples gives stupid examples, where you SHOULD NOT use a cursor. Take a look there:

https://docs.oracle.com/cd/B10500_01/appdev.920/a96624/06_ora.htm#1449

Each one of their examples can be replaced by a much faster insert ... select statement with a case statement when conditional logic is required.
I don't want to single-out Oracle, a search on the web will convince you that it's everywhere the same story.

Because of these examples, many people believe in good faith that cursors are good. They aren't. They are evil. People who use cursors a lot are usually people who struggle with SQL.

Flickr: Jeff Sandquist

Cultural mismatch

row-by-row

set processing

RELATIONAL is all about sets.

As always, there are always good reasons for using even a questionable feature. Unfortunately, these reasons aren't the ones put forward in tutorials.
Although executing data definition languages in stored procedures isn't standard, this is something that you sometimes want to do. Database administrators are the most likely people to want to do such a thing, but as a developer you may sometimes want (for instance during a migration) to copy constraints between tables or such fancy operations.

However, DDL operations are usually unsupported by stored procedures.

```
begin
   ...
   create table T ...;
   ...
end;
```

A CREATE TABLE (or ALTER TABLE) in a procedure will fail.

But you can cheat. You can in a procedure execute an SQL command stored into a string, and then the procedure will not look at what you are doing. If it's a valid SQL command, it will work.

```
begin
   ...
   cmd := 'create table T ...';
   execute cmd;
   ...
end;
```

Example

create a daily copy of tables.

Let's say that every day you want to take a copy of your tables. This isn't the best of examples, but it will show you a few interesting things.
First interesting thing: a schema called INFORMATION_SCHEMA contains "System Views" that describe the database. You can find everything about tables, columns and constraints there.
Second interesting thing: CREATE TABLE ... AS SELECT (also known as CTAS but not supported by every DBMS) let's you copy structure and data. However, the copy is imperfect:
 - all columns are created nullable
 - constraints and other features we'll see later are "forgotten"

```
create function save_tables()
returns void
as $$
declare
   v_suffix      varchar(50);
   v_create_cmd  varchar(100);
   c  cursor for select table_name
                 from information_schema.tables
                 where table_schema = current_schema()
                   and table_name not like '%' || v_suffix;
begin
  select '_save_' || to_char(current_date, 'YYMMDD')
  into v_suffix;
  for fetched_row in c
  loop
    v_create_cmd := 'create table ' || fetched_row.table_name
                   || v_suffix || ' as select * from '
                   || fetched_row.table_name;
    execute v_create_cmd;
  end loop;
end;
$$ language plpgsql;
```

The select ignores the copies of the day (in fact, it should ignore every copy, but it's just to show a parameter in use). It should also ignore tables for which a copy of the day exists.

to_char() to format a date also exists in Oracle and DB2

This would be a more correct query for the previous cursor (but it was a bit too long to fit on the page)

```
c cursor for
    select replace(table_name, v_suffix, '') as table_name
                 from information_schema.tables
                 where table_schema = current_schema()
                 group by replace(table_name, v_suffix, '')
                 having count(*) = 1  -- No copy for today
                      and replace(table_name, v_suffix, '')
                                not like '%_save_%';
```

In the previous example, there is no way to achieve the result we want without using a cursor. Here a cursor is fully justified.

TODAY'S ASSIGNMENT

Write a function called
`merge_people(pid1 int, pid2 int)`
that changes the database so as to replace people pid2 by pid1 before removing pid2

(start from lab6 – solution posted when lab6 is closed)

Raise an exception if pid1 or pid2 doesn't exist.
Raise an exception if pid1 = pid2

Lewis Jerry = Jerry Lewis ...