Week 05 Lectures

PLpgSQL (recap)

1/65

PLpgSQL is a programming language

- containing variables, assignment, conditionals, loops, functions
- combined with database interactions (via SQL)
- functions are stored in the database and invoked from SQL

Example:

```
create or replace function
   div(x integer, y integer) returns integer
as $$
declare
   result integer;    -- variable
begin
   if (y <> 0) then   -- conditional
      result := x/y;  -- assignment
   else
      result := 0;    -- assignment
   end if;
   return result;
end;
$$ language plpgsql;
```

... PLpgSQL (recap)

2/65

PLpgSQL syntax and control structures

Functions	`create or replace function`
	`   FunctionName(Params) returns [setof] Type`
	`as $$`
	`declare Variables begin Code end;`
	`$$ language plpgsql;`

Assignment	`Variable := Expression` `select Attrs into Variables ...`

Selection	`if` *Condition*$_1$ `then` *Statements*$_1$
	`elsif` *Condition*$_2$ `then` *Statements*$_2$ `...`
	`else` *Statements*$_n$ `end if`

Iteration	`loop Statements end loop`
	`while Condition loop Statements end loop`
	`for IntVariable in Lo..Hi loop ...`
	`for RecordVariable in Query loop ...`

Exercise 1: Factorial in PLpgSQL

3/65

Write two PLpgSQL functions ...

- an iterative version of *n!*
- a recursive version of *n!*

Function definition looks like

```
create or replace function
   fac(n integer) returns integer
as $$
...
$$ language plpgsql;
```

Exercise 2: Debugging Output

Depending on how PostgreSQL is configured

- `raise notice` allows you to display info from a function
- displayed in `psql` window during the function's execution
- usage:
  `raise notice` *StringWith%s*`,`*Value*$_1$`,...`*Value*$_n$`;`

Write a PLpgSQL function ...

- that takes an integer value *n*, but returns nothing
- iterates for 1 .. *n* and prints a message for each value

... PLpgSQL (recap)

Capturing value(s) from the database, e.g.

```
create or replace function
   full_name(_beer text) returns text
as $$
declare
   _brewer text;
begin
   select manf into _brewer
   from Beers where name = _beer;
   if (not found) then
      return 'No beer called "'||_beer||'"';
   else
      return _brewer||' '||_beer;
   end if;
end;
$$ language plpgsql;
```

If query returns multiple tuples, use value(s) in first tuple only.

Exercise 3: Full Names of Beers

Use the `full_name()` function

- to get names of individual beers
- to handle unknown beers
- to get names of all beers

For the unknown beers case ...

- handle it by returning an 'unknown beer' string
- handle it by raising an `error` exception

Returning Multiple Values

PLpgSQL functions can return a `setof` values

- effectively a function returning a table
- values could be atomic ⇒ like a single column
- values could be tuples ⇒ like a full table

Atomic types, e.g.

`integer, float, numeric, date, text, varchar(`*n*`), ...`

Tuple types, e.g.

```
create type Point as (x float, y float);
create type Student as (id integer, name text,
                        degree text, wam float, ...);
```

Exercise 4: Functions returning numbers

8/65

Write three functions called `iota()` ...

- each function returns a `setof integer` values
- `iota(`*hi*`)` returns numbers in range 1..*hi*
- `iota(`*lo*`,`*hi*`)` returns numbers in range *lo*..*hi*
- `iota(`*lo*`,`*hi*`,`*inc*`)` returns
  lo, *lo+inc*, *lo+2*inc*, *lo+3*inc*, ..., max ≤ *hi*

Functions returning `setof` *Type* are used in the `from` clause.

... Returning Multiple Values

9/65

Example function returning a set of tuples

```
create type MyPoint as (x integer, y integer);

create or replace function
   points(n integer, m integer) returns setof MyPoint
as $$
declare
   i integer;  j integer;
   p MyPoint;  -- tuple variable
begin
   for i in 1 .. n loop
      for j in 1 .. m loop
         p.x := i;  p.y := j;
         return next p;
      end loop;
   end loop;
end;
$$ language plpgsql;
```

Query results in PLpgSQL

10/65

Can evaluate a query and interate through its results

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

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

... Query results in PLpgSQL

11/65

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;
```

INSERT ... RETURNING

Can capture values from tuples inserted into DB:

```
insert into Table(...) values
(Val1, Val2, ... Valn)
returning ProjectionList into VarList
```

Useful for recording id values generated for `serial` PKs:

```
declare newid integer; colour text;
...
insert into T(id,a,b,c) values (default,2,3,'red')
returning id,c into newid,colour;
-- id contains the primary key value
-- for the new tuple T(?,2,3,'red')
```

Exceptions

PLpgSQL supports exception handling via

```
begin
    Statements...
exception
    when Exceptions1 then
        StatementsForHandler1
    when Exceptions2 then
        StatementsForHandler2
    ...
end;
```

Each *Exceptions$_i$* is an `OR` list of exception names, e.g.

```
division_by_zero OR floating_point_exception OR ...
```

A list of exceptions is in Appendix A of the PostgreSQL Manual.

... Exceptions

When an exception occurs:

- control is transferred to the relevant exception handling code
- all database changes so far in this transaction are undone
- all function variables retain their current values
- handler executes and then transaction aborts (and function exits)

If no handler in current scope, exception passed to next outer level.

Default exception handlers at outermost level exit and log error.

... Exceptions

Example of exception handling:

```
-- table T contains one tuple ('Tom','Jones')
declare
   x integer := 3;
begin
   update T set firstname = 'Joe'
   where lastname = 'Jones';
    -- table T now contains ('Joe','Jones')
   x := x + 1;
   y := x / 0;
exception
   when division_by_zero then
      -- update on T is rolled back to ('Tom','Jones')
      raise notice 'caught division_by_zero';
      return x;  -- return may or may not work here
      -- if it does, value returned is 4
end;
```

... Exceptions

The `raise` operator can generate server log entries, e.g.

```
raise debug 'Simple message';
raise notice 'User = %',user_id;
raise exception 'Fatal: value was %',value;
```

There are several levels of severity:

- `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, and `EXCEPTION`
- not all severities generate a message to the client

Your CSE server log is the file `/srvr/`*`YOU`*`/pgsql/Log`

Server logs can grow *very* large; delete when you shut your server down

Dynamically Generated Queries

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

Examples:

```
execute 'select * from Employees';
execute 'select * from '||'Employees';
execute 'select * from '||quote_ident($1);
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

Example: a wrapper for updating a single text field

```
create or replace function
   set(_tab text, _attr text, _val text) returns void
as $$
declare
   query text;
begin
```

```
   query := 'update ' || quote_ident(_tab);
   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','address','Beach St.');
```

... Dynamically Generated Queries

19/65

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

20/65

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 ($attr_1$ $type_1$, ... $attr_n$ $type_n$);

Other mjaor 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

21/65

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

Using the function ...

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

versus a view

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

Aggregates

Aggregates

Aggregates reduce a collection of values into a single result.

Examples: `count(`*Tuples*`)`, `sum(`*Numbers*`)`, `max(`*AnyOrderedType*`)`

The action of an aggregate function can be viewed as:

```
State = initial state
for each item V {
    # update State to include V
    State = updateState(State, V)
}
return makeFinal(State)
```

... Aggregates

Aggregates are commonly used with `GROUP BY`.

In that context, they "summarise" each group.

Example:

```
R                  select a,sum(b),count(*)
 a | b | c         from R group by a
---+---+---
 1 | 2 | x          a | sum | count
 1 | 3 | y         ---+-----+-------
 2 | 2 | z          1 |   5 |     2
 2 | 1 | a          2 |   6 |     3
 2 | 3 | b
```

User-defined Aggregates

SQL standard does not specify user-defined aggregates.

But PostgreSQL provides a mechanism for defining them.

To define a new aggregate, first need to supply:

- *BaseType* ... type of input values
- *StateType* ... type of intermediate states
- state mapping function: *sfunc(state,value) → newState*
- [optionally] an initial state value (defaults to null)
- [optionally] final function: *ffunc(state) → result*

... User-defined Aggregates

New aggregates defined using `CREATE AGGREGATE` statement:

```
CREATE AGGREGATE AggName(BaseType) (
    sfunc     = UpdateStateFunction,
    stype     = StateType,
    initcond  = InitialValue,
    finalfunc = MakeFinalFunction,
    sortop    = OrderingOperator
);
```

- `initcond` (type *StateType*) is optional; defaults to `NULL`
- `finalfunc` is optional; defaults to identity function
- `sortop` is optional; needed for min/max-type aggregates

... User-defined Aggregates

Example: defining the `count` aggregate (roughly)

```
create aggregate myCount(anyelement) (
    stype    = int,     -- the accumulator type
    initcond = 0,       -- initial accumulator value
    sfunc    = oneMore -- increment function
);

create function
    oneMore(sum int, x anyelement) returns int
as $$
begin return sum + 1; end;
$$ language plpgsql;
```

... User-defined Aggregates

Example: `sum2` sums two columns of integers

```
create type IntPair as (x int, y int);

create function
    AddPair(sum int, p IntPair) returns int
as $$
begin return p.x + p.y + sum; end;
$$ language plpgsql;

create aggregate sum2(IntPair) (
    stype     = int,
    initcond  = 0,
    sfunc     = AddPair
);
```

Exercise 5: Product Aggregate

PostgreSQL has many aggregates (e.g. `sum`, `count`, ...)

But it doesn't have a product aggregate.

Implement a `prod` aggregate that

- computes the product of values in a column of integer data

Usage:

```
select prod(*) from iota(5);
 prod
------
  120
```

But we need: `select prod(iota::integer) from iota(5)`

Exercise 6: String Concatenation Aggregate

Define a `concat` aggregate that

- takes a column of string values
- returns a comma-separated string of values

Example:

```
select count(*), concat(name) from Employee;
-- returns e.g.
  count |          concat
 -------+----------------------
      4 | John,Jane,David,Phil
```

Use it to get a list of beers liked by each drinker.

Constraints

Constraints

So far, we have considered several kinds of constraints:

- attribute (column) constraints
- relation (table) constraints
- referential integrity constraints

Examples:

```
create table Employee (
   id      integer primary key,
   name    varchar(40),
   salary  real,
   age     integer check (age > 15),
   worksIn integer
              references Department(id),
   constraint PayOk check (salary > age*1000)
);
```

... Constraints

Column and table constraints ensure validity of one table.

Ref. integrity constraints ensure connections between tables are valid.

However, specifying validity of entire database often requires constraints involving multiple tables.

Simple example (from banking domain):

```
for all Branches b
   b.assets == (select sum(acct.balance)
                from   Accounts acct
                where  acct.branch = b.location)
```

i.e. assets of a branch is sum of balances of accounts held at that branch

---

Assertions

Assertions are schema-level constraints

- typically involving multiple tables
- expressing a condition that must hold at all times
- need to be checked on each change to relevant tables
- if change would cause check to fail, reject change

SQL syntax for assertions:

```
CREATE ASSERTION name CHECK (condition)
```

The *condition* is expressed as "there are no violations in the database"

Implementation: ask a query to find all the violations; check for empty result

---

... Assertions

Example: #students in any UNSW course must be < 10000

```
create assertion ClassSizeConstraint check (
   not exists (
      select c.id from Courses c, CourseEnrolments e
      where  c.id = e.course
      group  by c.id having count(e.student) > 9999
   )
);
```

Needs to be checked after *every* change to either `Course` or `CourseEnrolment`

... Assertions

Example: assets of branch = sum of its account balances

```
create assertion AssetsCheck check (
   not exists (
      select branchName from Branches b
      where  b.assets <>
             (select sum(a.balance) from Accounts a
              where a.branch = b.location)
   )
);
```

Needs to be checked after *every* change to either `Branch` or `Account`

... Assertions

On each update, it is expensive

- to determine which assertions need to be checked
- to run the queries which check the assertions

A database with many assertions would be way too slow.

So, most RDBMSs do not implement general assertions.

Typically, *triggers* are provided as

- a lightweight mechanism for dealing with assertions
- a general event-based programming tool for databases

Triggers

Triggers

Triggers are

- procedures stored in the database
- activated in response to database events (e.g. updates)

Examples of uses for triggers:

- maintaining summary data
- checking schema-level constraints (assertions) on update
- performing multi-table updates (to maintain assertions)

... Triggers

Triggers provide event-condition-action (ECA) programming:

- an *event* activates the trigger
- on activation, the trigger checks a *condition*
- if the condition holds, a procedure is executed (the *action*)

Some typical variations on this:

- execute the action before, after or instead of the triggering event
- can refer to both old and new values of updated tuples
- can limit updates to a particular set of attributes
- perform action: for each modified tuple, once for all modified tuples

---

... Triggers

SQL "standard" syntax for defining triggers:

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE}  Event1 [ OR Event2 ... ]
[ FOR EACH ROW ]
ON TableName
[ WHEN ( Condition ) ]
Block of Procedural/SQL Code ;
```

Possible *Events* are `INSERT`, `DELETE`, `UPDATE`.

`FOR EACH ROW` clause ...

- if present, code is executed on each modified tuple
- if not present, code is executed once after all tuples are modified, just before changes are finally `COMMIT`ed

---

Trigger Semantics

Triggers can be activated `BEFORE` or `AFTER` the event.

If activated `BEFORE`, can affect the change that occurs:

- `NEW` contains "proposed" value of changed tuple
- modifying `NEW` causes a different value to be placed in DB

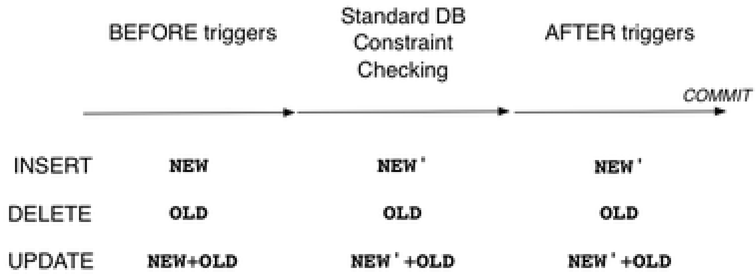
If activated `AFTER`, the effects of the event are visible:

- `NEW` contains the current value of the changed tuple
- `OLD` contains the previous value of the changed tuple
- constraint-checking has been done for `NEW`

Note: `OLD` does not exist for insertion; `NEW` does not exist for deletion.

... Trigger Semantics

Sequence of activities during database update:



Reminder: `BEFORE` trigger can modify value of new tuple

... Trigger Semantics

45/65

Consider two triggers and an INSERT statement

```
create trigger X before insert on T Code1;
create trigger Y after insert on T Code2;
insert into T values (a,b,c,...);
```

Sequence of events:

- execute `Code1` for trigger `X`
- code has access to `(a,b,c,...)` via `NEW`
- code typically checks the values of `a,b,c,..`
- code can modify values of `a,b,c,..` in `NEW`
- DBMS does constraint checking as if `NEW` is inserted
- if fails any checking, abort insertion and rollback
- execute `Code2` for trigger `Y`
- code has access to final version of tuple via `NEW`
- code typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Reminder: there is no `OLD` tuple for an `INSERT` trigger.

... Trigger Semantics

46/65

Consider two triggers and an UPDATE statement

```
create trigger X before update on T Code1;
create trigger Y after update on T Code2;
update T set b=j,c=k where a=m;
```

Sequence of events:

- execute `Code1` for trigger `X`
- code has access to current version of tuple via `OLD`
- code has access to updated version of tuple via `NEW`
- code typically checks new values of `b,c,..`
- code can modify values of `a,b,c,..` in `NEW`
- do constraint checking as if `NEW` has replaced `OLD`
- if fails any checking, abort update and rollback
- execute `Code2` for trigger `Y`
- code has access to final version of tuple via `NEW`
- code typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Reminder: both `OLD` and `NEW` exist in UPDATE triggers.

... Trigger Semantics

47/65

Consider two triggers and an DELETE statement

```
create trigger X before delete on T Code1;
create trigger Y after delete on T Code2;
delete from T where a=m;
```

Sequence of events:

* execute `Code1` for trigger `X`
* code has access to `(a,b,c,...)` via `OLD`
* code typically checks the values of `a,b,c,..`
* DBMS does constraint checking as if `OLD` is removed
* if fails any checking, abort deletion (restore `OLD`)
* execute `Code2` for trigger `Y`
* code has access to about-to-be-deleted tuple via `OLD`
* code typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Reminder: tuple `NEW` does not exist in DELETE triggers.

Triggers in PostgreSQL

PostgreSQL triggers provide a mechanism for

* `INSERT`, `DELETE` or `UPDATE` events
* to automatically activate PLpgSQL functions

Syntax for PostgreSQL trigger definition:

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE}  Event1 [OR Event2 ...]
ON TableName
[ WHEN ( Condition ) ]
FOR EACH {ROW|STATEMENT}
EXECUTE PROCEDURE FunctionName(args...);
```

... Triggers in PostgreSQL

There is no restriction on what code can go in the function.

However a `BEFORE` function must contain one of:

```
RETURN old;    or    RETURN new;
```

depending on which version of the tuple is to be used.

If `BEFORE` trigger returns `old`, no change occurs.

If exception is raised in trigger function, no change occurs.

Trigger Example #1

Consider a database of people in the USA:

```
create table Person (
   id      integer primary key,
   ssn     varchar(11) unique,
   ... e.g. family, given, street, town ...
   state   char(2), ...
);
create table States (
   id      integer primary key,
   code    char(2) unique,
   ... e.g. name, area, population, flag ...
);
```

Constraint: `Person.state ∈ (select code from States)`, or
`exists (select id from States where code=Person.state)`

... Trigger Example #1

Example: ensure that only valid state codes are used:

```
create trigger checkState before insert or update
```

```
on Person for each row execute procedure checkState();

create function checkState() returns trigger as $$
begin
   -- normalise the user-supplied value
   new.state = upper(trim(new.state));
   if (new.state !~ '^[A-Z][A-Z]$') then
      raise exception 'Code must be two alpha chars';
   end if;
   -- implement referential integrity check
   select * from States where code=new.state;
   if (not found) then
      raise exception 'Invalid code %',new.state;
   end if;
   return new;
end;
$$ language plpgsql;
```

... Trigger Example #1

52/65

Examples of how this trigger would behave:

```
insert into Person
   values('John',...,'Calif.',...);
-- fails with 'Statecode must be two alpha chars'

insert into Person
   values('Jane',...,'NY',...);
-- insert succeeds; Jane lives in New York

update Person
   set town='Sunnyvale',state='CA'
         where name='Dave';
-- update succeeds; Dave moves to California

update Person
   set state='OZ' where name='Pete';
-- fails with 'Invalid state code OZ'
```

Trigger Example #2

53/65

Example: department salary totals

Scenario:

```
Employee(id, name, address, dept, salary, ...)
Department(id, name, manager, totSal, ...)
```

An assertion that we wish to maintain:

```
create assertion TotalSalary check (
   not exists (
       select d.id from Department d
       where  d.totSal <>
              (select sum(e.salary)
               from Employee e
               where e.dept = d.id)
    )
)
```

... Trigger Example #2

54/65

Events that might affect the validity of the database

- a new employee starts work in some department
- an employee gets a rise in salary
- an employee changes from one department to another
- an employee leaves the company

A single assertion could check for this after each change.

With triggers, we have to program each case separately.

Each program implements updates to *ensure* assertion holds.

... Trigger Example #2

Implement the Employee update triggers from above in PostgreSQL:

Case 1: new employees arrive

```
create trigger TotalSalary1
after insert on Employees
for each row execute procedure totalSalary1();

create function totalSalary1() returns trigger
as $$
begin
    if (new.dept is not null) then
        update Department
        set    totSal = totSal + new.salary
        where  Department.id = new.dept;
    end if;
    return new;
end;
$$ language plpgsql;
```

... Trigger Example #2

Case 2: employees change departments/salaries

```
create trigger TotalSalary2
after update on Employee
for each row execute procedure totalSalary2();

create function totalSalary2() returns trigger
as $$
begin
    update Department
    set    totSal = totSal + new.salary
    where  Department.id = new.dept;
    update Department
    set    totSal = totSal - old.salary
    where  Department.id = old.dept;
    return new;
end;
$$ language plpgsql;
```

... Trigger Example #2

Case 3: employees leave

```
create trigger TotalSalary3
after delete on Employee
for each row execute procedure totalSalary3();

create function totalSalary3() returns trigger
as $$
begin
    if (old.dept is not null) then
        update Department
        set    totSal = totSal - old.salary
        where  Department.id = old.dept;
    end if;
    return old;
```

```
end;
$$ language plpgsql;
```

Exercise 7: Triggers (1)

Requirement: maintain assets in bank branches

- each branch has assets based on the accounts held there
- whenever an account changes, the assets of the corresponding branch should be updated to reflect this change

Some possible changes:

- a new account is opened
- the amount of money in an account changes
- an account moves from one branch to another
- an account is closed

Implement triggers to maintain `Branch.assets`

Exercise 8: Triggers (2)

Consider a simple airline flights/bookings database:

```
Airports(id, code, name, city)
Planes(id, craft, nseats)
Flights(id, fltNum, plane, source, dest
        departs, arrives, price, seatsAvail)
Passengers(id, name, address, phone)
Bookings(pax, flight, paid)
```

Write triggers to ensure that `Flights.seatsAvail` is consistent with number of `Bookings` on that flight.

Assume that we never `UPDATE` a booking (only insert/delete)

Catalogs

DBMS Catalog

DBMSs store ...

- data (tuples, organised into tables)
- stored procedures (e.g. PLpgSQL functions)
- indexes (to provide efficient access to data)
- meta-data (information giving the structure of the data)

The latter is stored in the *system catalog*

A standard `information_schema` exists for describing meta-data

But was developed long after DBMSs had implemented their own catalogs

PostgreSQL has both: PG catalog Ch.52, Inforomation Schema Ch.37

PostgreSQL Catalog

Catalog = meta-data = tables describing DB objects

PostgreSQL catalog is accessible via **pg_*XXX*** tables/views:

```
pg_roles(oid, rolname, rolsuper, ...)

pg_namespace(oid, nspname, nspowner, nspacl)
```

```
pg_database(oid, datname, datdba, ..., datacl)

pg_class(oid, relname, relnamespace, reltype, ...)

pg_attribute(oid, attrrelid, attname, atttypid, ...)

pg_type(oid, typname, typnamespace, typowner, ...)
```

Catalog tables use `oid` field for PKs/FKs

Standard-format catalog also available, via `information_schema`

Exercise 9: List of Databases

Use the `pg_catalog` to implement simplified `psql -l`

```
$ psql -l | cut -c1-23

     Name       | Owner
----------------+-------
 a1             | jas
 a2             | jas
 acad           | jas
 accreditation  | jas
 aims           | jas
 assess         | jas
 bank           | jas
 beer           | jas
 cmap           | jas
 company        | jas
 cse            | jas
 ...
```

Exercise 10: Size of Database Tables

Implement a `dbPop()` function that ...

- lists all of the tables in the public schema
- counts the number of tuples in each table

The function is defined as

```
function dbPop() returns setof PopulationRecord
```

where each return tuple has the type

```
PopulationRecord(table_name,n_records)
```

Exercise 11: Simple Schema Dump

Implement a function `dbSchema()` that gives a list of tables in the public schema and their attributes.

The function is defined as

```
function dbSchema()
         returns setof SchemaRecord
```

where each return tuple has the type

```
SchemaRecord(table_name, attributes)
```

Attributes = comma-separated list of names, in order.
