Lab 1
Spatial Databases

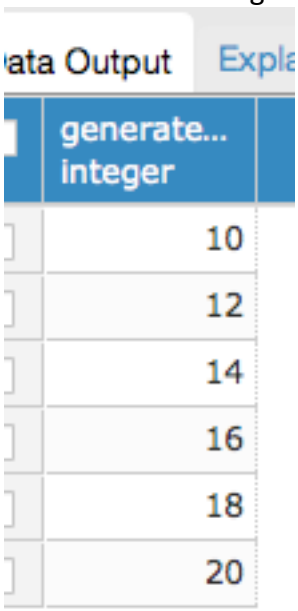
Learning objectives:

- ✓ Revise most important SQL concepts
- ✓ Get to know postgreSQL/postGIS's SQL flavour
- ✓ Gain confidence with postgreSQL

This lab is a direct result of the survey from lecture 1.

Exercise 1

1. Connect to your postgreSQL/postGIS database, revise lab 0 for a reminder on how to do this.
2. Get to know your postGIS version
   select postgis_lib_version();
3. Learn how to generate a number series
   https://www.postgresql.org/docs/9.1/static/functions-srf.html
   create the following series:



4. Use the avg() function to create the average of a number series
5. Make two series of odd and even numbers. Note the different outputs from these queries.

Lab 1
Spatial Databases

Data Output | Explain | Message

even integer	odd integer
0	1
0	3
0	5
2	1
2	3
2	5
4	1
4	3
4	5
6	1

6. Using the series you generated in (5), use the group by statement to achieve the following output, where one series gets the alias x and the other the alias y. Group by x.

Data Output | Explain | Messag

x integer	sum bigint
1	6
3	6
2	6

7. Consider the following SQL query on a series:

```
SELECT x, sum(y)
FROM generate_series(1,3) AS x, generate_series(1,3) AS y
GROUP BY x;
```

Now re-write this code to add a having clause. The output should not change. Hint: >1 works.

Lab 1
Spatial Databases

The point of this exercise is to remember about the difference between having and where.

8. Predicates:
   Study the outputs of the following sql statements:
   Logical connectives are and, or, not

```
select not(true);
select false and true ;
select (false and true) or true ;
select True = 'yes';
select True = 'n';
select 't' = 'n';
select true = 'n';
select (true = cast( 1 as boolean));
```

   The following produces all the true and false values for the logical operation called or.

```
select true or true ;
select true or false ;
select false or true ;
select false or false ;
```

   The combined output from the 4 previous commands is often called the truth table for or.

9. Try predicates with numbers

```
select (2<1);
select (1=1);
select (1=2);
select (1=1) or (1=2);
select (1>=1) ;
select (1<=1);
```

10. Try predicates with strings

```
select 'Mary' = 'Mary';
select 'Mary' = 'Mary-Ellen';
select 'Mary-Ellen' LIKE 'Mary%';
select 'Mary-Ellen' LIKE 'Mary_____';
select 'Mary'  ILIKE  'mary';
```

When using the LIKE predicate an underscore (_) in pattern stands for (matches) any single character; a percent sign (%) matches any string of zero or more characters. LIKE is case sensitive, ILIKE is case insensitive,

Lab 1
Spatial Databases
Exercise 2

Some sample databases and queries.

In this section we will create a table, do some inserts, do some queries and drop the table.

Most of the rest of the lab follows steps 1-4 below i.e. we create and drop several staff tables.

```
-- 1
CREATE table staff  (
   employee text,
   dept text,
   salary int4,
   PRIMARY KEY (employee, dept),
  CONSTRAINT
  positive_salary CHECK (salary > 0));
-- 2
INSERT …
-- 3
SELECT * FROM staff;
--4
DROP TABLE staff;
```

employee text	dept text	salary integer
Bernard	Accounting	2100
Marion	Marketing	1000
Ellen	Management	2400
Liam	Marketing	1650
Emily	Management	3250
Liz	Accounting	2950
Joe	IT	3250
Michael	Accounting	1700
John	Management	2950
Paul	Accounting	1650
Phoebe	Accounting	3700
Rachel	Accounting	2950
Ross	IT	1250
Sara	IT	2600
Fred	Management	3250
Pat	IT	3700

Table 1

```
create table staff  (
employee text,
dept text,
salary int4
);
```

```
INSERT INTO staff (employee, dept, salary) VALUES
( 'Bianca', 'it', 3700 ),
( 'Bernard', 'Accounting', 2100 ),
( 'Marion', 'Marketing', 1000 ),
( 'Ellen', 'Management', 2400 ),
( 'Liam', 'Marketing', 1650 ),
( 'Emily', 'Management', 3250 ),
( 'Liz', 'Accounting', 2950 ),
( 'Joe', 'IT', 3250 ),
( 'Michael', 'Accounting', 1700 ),
( 'John', 'Management', 2950 ),
( 'Paul', 'Accounting', 1650 ),
( 'Phoebe', 'Accounting', 3700 ),
( 'Rachel', 'Accounting', 2950 ),
( 'Ross', 'IT', 1250 ),
( 'Sara', 'IT', 2600 ),
( 'Fred', 'Management', 3250 );
```

List the contents of the staff table.

```
SELECT employee,dept,salary FROM staff;
```

Lab 1
Spatial Databases

The department of Bianca was called ‘it’ instead of IT.
To check this try the following query:

```
select employee,dept FROM staff WHERE dept=’IT’;
```

From the results of the previous query you will see that Bianca is not included in the IT staff.
To correct this we need an UPDATE statement.

```
UPDATE staff
SET dept =’IT'
WHERE employee='Bianca';
```

Find the total salary for each department with the most expensive department first

```
SELECT dept,sum(salary) AS “Salary”
FROM staff
GROUP BY dept
ORDER BY “Salary” DESC;
```

Find the total salary for each department with the least expensive department first

```
SELECT dept,sum(salary) AS “Salary”
FROM staff
GROUP BY dept
ORDER BY “Salary” ASC;
```

We wish to move Bianca from the IT department to Accounting.
One way to do this is just to update the department field or attribute of the row containing Bianca.

```
UPDATE staff
SET dept =’Accounting'
WHERE employee='Bianca';
```

This is a similar command to correcting the department name above.

To represent the fact that John has left the company we could use the SQL DELETE command as follows.

```
DELETE
FROM staff
WHERE employee='John';
```

To represent the fact that a new employee Mary has joined the company we could use the SQL INSERT command as follows.

```
INSERT INTO staff (employee, dept, salary)
VALUES ('Mary', 'IT', 2600 );
```

If we did not know Mary’s department or starting salary then we would leave these fields blank as follows;

```
INSERT INTO staff (employee)
VALUES ('Mary');
```

Lab 1
Spatial Databases

If you execute the last two commands then there will be two Mary's in the STAFF table. In general a persons name is not a good unique identifier. To make rows unique we need the concept of a PRIMARY KEY.

Without a primary key, we could also insert a row with no employee name, which is not very useful

```
INSERT INTO staff (dept)
VALUES ('IT');
```

Again concept of a PRIMARY KEY would be useful, it would prevent this.

A PRIMARY KEY is a unique non NULL value. NULL is a special value which can be interpreted in several ways. For example, value not known, value not applicable, value withheld. See handout for further details on NULLs.

We will delete the STAFF table using the DROP command and then we will make a similar table but this time with a primary key.

```
DROP TABLE staff;

create table staff  (
employee text PRIMARY KEY,
dept text,
salary int4
);

INSERT INTO staff (employee, dept, salary) VALUES
( 'Bianca', 'IT', 3700 ),
( 'Bernard', 'Accounting', 2100 ),
( 'Marion', 'Marketing', 1000 ),
( 'Ellen', 'Management', 2400 ),
( 'Liam', 'Marketing', 1650 ),
( 'Emily', 'Management', 3250 ),
( 'Liz', 'Accounting', 2950 ),
( 'Joe', 'IT', 3250 ),
( 'Michael', 'Accounting', 1700 ),
( 'John', 'Management', 2950 ),
( 'Paul', 'Accounting', 1650 ),
( 'Phoebe', 'Accounting', 3700 ),
( 'Rachel', 'Accounting', 2950 ),
( 'Ross', 'IT', 1250 ),
( 'Sara', 'IT', 2600 ),
( 'Fred', 'Management', 3250 );
```

Check that STAFF has a primary key using \d staff
Now if we try to enter a second person called Bianca we will get an error.

```
INSERT INTO staff (employee, dept, salary)
VALUES ('Bianca', 'IT', 2600 );
```

Also we cannot insert an row with no employee name.

```
INSERT INTO staff (dept, salary)
VALUES ( 'IT', 2600 );
```

The primary key can consist of more than one field (aka attribute or column). So again we drop the table and make a new one whose primary key consists of employee and department. This means that names must only be unique within department.

```
DROP TABLE staff;

create table staff  (
employee text,
dept text,
salary int4,
PRIMARY KEY (employee, dept)
);
```

Check that STAFF has a two field primary key using \d staff

Now insert the data

```
INSERT INTO staff (employee, dept, salary) VALUES
( 'Bianca', 'IT', 3700 ),
( 'Bernard', 'Accounting', 2100 ),
( 'Marion', 'Marketing', 1000 ),
( 'Ellen', 'Management', 2400 ),
( 'Liam', 'Marketing', 1650 ),
( 'Emily', 'Management', 3250 ),
( 'Liz', 'Accounting', 2950 ),
( 'Joe', 'IT', 3250 ),
( 'Michael', 'Accounting', 1700 ),
( 'John', 'Management', 2950 ),
( 'Paul', 'Accounting', 1650 ),
( 'Phoebe', 'Accounting', 3700 ),
( 'Rachel', 'Accounting', 2950 ),
( 'Ross', 'IT', 1250 ),
( 'Sara', 'IT', 2600 ),
( 'Fred', 'Management', 3250 );
```

We can add another Bianca but the must be in a different department. So the following is allowed.

```
INSERT INTO staff (employee, dept, salary)
VALUES ('Bianca', 'Management', 2600 );
```

But the next INSERT causes an error.

```
INSERT INTO staff (employee, dept, salary)
VALUES ('Bianca', 'IT', 2600 );
```

Lab 1
Spatial Databases

This is because the pair (employee, dept) should always be unique. Also we will get an update error.

We are allowed enter a negative salary:

```
UPDATE staff
SET salary = -2600
WHERE employee='Bianca';
```

Recall different data types:

Name	Storage Size	Range
smallint	2 bytes	-32768 to +32767
integer	4 bytes	-2147483648 to +2147483647
bigint	4 bytes	-9223372036854775808 to 9223372036854775807
decimal	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	6 decimal digits precision
double	8 bytes	15 decimal digits precision

Also, check the postgresql documentation on datatypes:
https://www.postgresql.org/docs/9.5/static/datatype.html

And postgis:
http://postgis.net/docs/reference.html

```
SELECT 100 * (0.08875)::numeric;
8.875
SELECT 100 * (0.08875)::numeric(7,2);
9.0
SELECT (100 * 0.08875)::numeric(7,2);
8.88
```

The primary key concept is an example of a CONSTRAINT.
Data types are a way to limit the kind of data that can be stored in a table.
For many applications, however, the constraint they provide is too coarse.
For example, a column containing a salary should only accept positive values.
But there is no standard data type that accepts only positive numbers (i.e. $n \in \mathbb{N}$).
Another issue is that you might want to constrain column data with respect to other columns or rows. For example, in a table containing staff records, there should only be one row for each staff member. With our current schema we could enter the following negative pay rate.

Lab 1

Spatial Databases

```
INSERT INTO staff (employee, dept, salary)
VALUES ('Mike', 'IT', -2600 );
```

To prevent this we can add the constraint that all salary must be positive

```
DROP TABLE staff;


CREATE table staff  (
employee text,
dept text,
salary int4,
PRIMARY KEY (employee, dept),
CONSTRAINT positive_salary CHECK (salary > 0)
);
```

Now if we try to enter a negative salary we get an error message;

```
INSERT INTO staff (employee, dept, salary)
VALUES ('Mike', 'IT', -2600 );
```

A **correlated sub-query** is a query nested inside another query that uses values from the outer query in its WHERE clause. The sub-query is evaluated once for each row processed by the outer query. Example: for each department find the employee having a higher salary than the average salary of all employees in that employee's department.

```
SELECT dept,employee,salary
 FROM staff AS p1
 WHERE salary > (SELECT avg(salary)
    FROM staff as p2
    WHERE p2.dept = p1.dept)
    order by dept;
```

Check the answer by looking at the average of each dept.

```
SELECT avg(salary) FROM staff where dept='IT';
```

A non-correlated sub query.

```
SELECT dept,employee FROM staff WHERE dept =
(SELECT dept FROM staff WHERE employee = 'Phoebe');

-- could be done as a JOIN
SELECT e1.dept, e1.employee
FROM staff e1, staff e2
WHERE e1.dept = e2.dept AND  e2.employee = 'Phoebe';

SELECT e1.dept, e1.employee
FROM staff e1, staff e2
WHERE e1.dept = e2.dept AND  e2.employee = 'Phoebe';
```

A correlated sub query refers to a column from a table in the parent query, whereas a non-correlated sub query doesn't. This means that a non-correlated sub query is

executed just once for the whole SQL statement, whereas correlated sub queries are executed once per row in the parent query.

What is the average salary?

```
select avg(salary) from staff;
```

A correlated query example.
Get employees, but exclude the poorest paid.

```
SELECT employee, dept, salary
 FROM staff AS p1
 WHERE salary > ANY (SELECT salary
    FROM staff as p2
    WHERE p2.dept = p1.dept)
ORDER BY dept;
```

Constraint example

```
CREATE table staff  (
employee text,
dept text,
salary int4,
PRIMARY KEY (employee, dept),
CONSTRAINT salary_must_exist CHECK (salary IS NOT NULL)
);


INSERT INTO staff (employee, dept, salary) VALUES
( 'Bianca', 'IT', 3700 ),
( 'Bernard', 'Accounting', 2100 ),
( 'Marion', 'Marketing', 1000 ),
( 'Ellen', 'Management', 2400 ),
( 'Liam', 'Marketing', 1650 ),
( 'Emily', 'Management', 3250 ),
( 'Liz', 'Accounting', 2950 ),
( 'Joe', 'IT', 3250 ),
( 'Michael', 'Accounting', 1700 ),
( 'John', 'Management', 2950 ),
( 'Paul', 'Accounting', 1650 ),
( 'Phoebe', 'Accounting', 3700 ),
( 'Rachel', 'Accounting', 2950 ),
( 'Ross', 'IT', 1250 ),
( 'Sara', 'IT', 2600 ),
( 'Fred', 'Management', 3250 );
```

What errors do we get when we try these INSERTs?

```
INSERT INTO staff (employee, dept, salary) VALUES
                 ( 'Bianca', 'IT', 3700 );

INSERT INTO staff (employee, dept) VALUES
                 ( 'John', 'IT' );
```

Lab 1
Spatial Databases

Exercise 3

JOINs

So far, the queries have only accessed one table at a time.
Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time.
A query that accesses multiple rows of the same or different tables at one time is called a join query.

Joins use columns with **common types** and values **satisfying some predicate** (usually equality =)We will now look at how two distinct tables can be joined based on some common values in the department column. We create a second table that contains the locations of each department.

```
DROP TABLE staff;
CREATE TABLE staff  (
employee text PRIMARY KEY,
dept text,
salary int4);

INSERT INTO staff (employee, dept, salary) VALUES
( 'Bianca', 'IT', 3700 ),
( 'Bernard', 'Accounting', 2100 ),
( 'Marion', 'Marketing', 1000 ),
( 'Ellen', 'Management', 2400 ),
( 'Liam', 'Marketing', 1650 ),
( 'Emily', 'Management', 3250 ),
( 'Liz', 'Accounting', 2950 ),
( 'Joe', 'IT', 3250 ),
( 'Michael', 'Accounting', 1700 ),
( 'John', 'Management', 2950 ),
( 'Paul', 'Accounting', 1650 ),
( 'Phoebe', 'Accounting', 3700 ),
( 'Rachel', 'Accounting', 2950 ),
( 'Ross', 'IT', 1250 ),
( 'Sara', 'IT', 2600 ),
( 'Fred', 'Management', 3250 );


CREATE TABLW location  (
location  text PRIMARY KEY,
dept text
);

INSERT INTO location  (location  , dept) VALUES
('Dublin', 'IT'),
('Cork', 'Accounting' ),
('Sligo', 'Marketing' ),
('Galway', 'Management');
```

Now perform a Join combining the staff and location tables.
The following will find the location of all employees.

```
SELECT employee, location FROM staff AS s , location AS l where s.dept =
l.dept;
```

This is the most basic of joins; there are additional concepts such as foreign keys that provide more sophisticated joining capabilities.
NOTE: The above join is **not** a spatial join; it combines that tables purely on the basis of the text values in the dept columns of both. The tables do not contain coordinates or spatial objects. To quit the SQL Shell type \q

Exercise 4

FUNCTIONS

Functions can be written in several languages e.g. C, Java, or R. Here is an example written in the built in procedural language PL/pgSQL[1].

```
CREATE FUNCTION inc(val integer) RETURNS integer AS $$
BEGIN
RETURN val + 1;
END; $$
LANGUAGE PLPGSQL;
select inc(5);
```

TRIGGERS

A database trigger is procedural code that is automatically executed in response to certain events on a particular table or view in a database. The trigger is mostly used for maintaining the integrity of the information on the database. For example, when a new record (representing a new worker) is added to the employees table, new records should also be created in the tables of the taxes, holidays and salaries[2].
The trigger can be specified to fire before the operation is attempted on a row (before constraints are checked and the INSERT, UPDATE, or DELETE is attempted); or after the operation has completed (after constraints are checked and the INSERT, UPDATE, or DELETE has completed); or instead of the operation (in the case of inserts, updates or deletes on a view). If the trigger fires before or instead of the event, the trigger can skip the operation for the current row, or change the row being inserted (for INSERT and UPDATE operations only). If the trigger fires after the event, all changes, including the effects of other triggers, are "visible" to the trigger[3]. Note that date and time are special data types in Postgres called TIMESTAMP.

[1] http://www.postgresql.org/docs/9.0/static/plpgsql.html
[2] http://en.wikipedia.org/wiki/Database_trigger
[3] http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html

Lab 1

Spatial Databases

```
DROP TABLE staff;

CREATE TABLE staff (
    employee text,
    dept text,
    salary integer,
    last_date timestamp,
    last_user text
);

INSERT INTO staff (employee, dept, salary, last_date,last_user) VALUES
( 'Bianca', 'IT', 3700, current_timestamp, current_user ),
( 'Bernard', 'Accounting', 2100, current_timestamp, current_user ),
( 'Marion', 'Marketing', 1000 , current_timestamp, current_user),
( 'Ellen', 'Management', 2400, current_timestamp, current_user),
( 'Liam', 'Marketing', 1650, current_timestamp, current_user ),
( 'Emily', 'Management', 3250, current_timestamp, current_user ),
( 'Liz', 'Accounting', 2950 , current_timestamp, current_user),
( 'Joe', 'IT', 3250, current_timestamp, current_user),
( 'Michael', 'Accounting', 1700, current_timestamp , current_user),
( 'John', 'Management', 2950, current_timestamp, current_user ),
( 'Paul', 'Accounting', 1650 , current_timestamp, current_user),
( 'Phoebe', 'Accounting', 3700 , current_timestamp, current_user),
( 'Rachel', 'Accounting', 2950, current_timestamp, current_user ),
( 'Ross', 'IT', 1250 , current_timestamp, current_user),
( 'Sara', 'IT', 2600, current_timestamp , current_user),
( 'Fred', 'Management', 3250, current_timestamp , current_user);

drop trigger "staff_check" on staff;
CREATE FUNCTION staff_check() RETURNS trigger AS $staff_check$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.employee IS NULL THEN
            RAISE EXCEPTION 'employee name cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION 'cannot have null salary', NEW.empname;
        END IF;
        -- No negative salaries
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;
        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$staff_check$ LANGUAGE plpgsql;

CREATE TRIGGER staff_check BEFORE INSERT OR UPDATE ON staff
    FOR EACH ROW EXECUTE PROCEDURE staff_check();

-- test
INSERT INTO staff (employee, dept, salary) VALUES ( 'Liz2', 'IT', (-3700) );
```

Lab 1

Spatial Databases

Triggers can also be used to maintain logs of all transactions. We log changes to a table by creating a new table that holds a row for each insert, update, or delete that occurs.

```
CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    empname           text      NOT NULL,
    salary integer);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
-- Create a row in emp_audit to reflect the operation performed on emp,
-- make use of the special variable TG_OP to work out the operation.
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(),OLD.employee;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), NEW.employee;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), NEW.employee;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON staff
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
-- test
INSERT INTO staff (employee, dept, salary, last_date,last_user) VALUES
( 'Bianca2', 'IT', 3700, current_timestamp, current_user ),
( 'Bernard2', 'Accounting', 2100, current_timestamp, current_user );
```

Postgresql triggers : https://www.postgresql.org/docs/9.1/static/sql-createtrigger.html