CHAPTER 14

Module: PostgreSQL

PostgreSQL is a Database Management System (DBMS). In this module, you'll be shown how to use PostgreSQL to create a new database, as well as accessing other typical DBMS functions.

14.1 Lesson: Introduction to Databases

Before using PostgreSQL, let's make sure of our ground by covering general database theory. You will not need to enter any of the example code; it's only there for illustration purposes.

The goal for this lesson: To understand fundamental database concepts.

14.1.1 What is a Database?

> A database consists of an organized collection of data for one or more uses, typically in digital form. - *Wikipedia*
>
> A database management system (DBMS) consists of software that operates databases, providing storage, access, security, backup and other facilities. - *Wikipedia*

14.1.2 Tables

> In relational databases and flat file databases, a table is a set of data elements (values) that is organized using a model of vertical columns (which are identified by their name) and horizontal rows. A table has a specified number of columns, but can have any number of rows. Each row is identified by the values appearing in a particular column subset which has been identified as a candidate key. - *Wikipedia*

```
 id | name  | age
----+-------+-----
  1 | Tim   |  20
  2 | Horst |  88
(2 rows)
```

In SQL databases a table is also known as a **relation.**

14.1.3 Columns / Fields

> A column is a set of data values of a particular simple type, one for each row of the table. The columns provide the structure according to which the rows are composed. The term field is often used interchangeably with column, although many consider it more correct to use field (or field value) to refer specifically to the single item that exists at the intersection between one row and one column. - *Wikipedia*

A column:

```
| name  |
+-------+
| Tim   |
| Horst |
```

A field:

```
| Horst |
```

14.1.4 Records

A record is the information stored in a table row. Each record will have a field for each of the columns in the table.

```
2 | Horst |  88  <-- one record
```

14.1.5 Datatypes

> Datatypes restrict the kind of information that can be stored in a column. - *Tim and Horst*

There are many kinds of datatypes. Let's focus on the most common:

- String - to store free-form text data
- Integer - to store whole numbers
- Real - to store decimal numbers
- Date - to store Horst's birthday so no one forgets
- Boolean - to store simple true/false values

You can tell the database to allow you to also store nothing in a field. If there is nothing in a field, then the field content is referred to as a **'null' value**.

```
insert into person (age) values (40);

INSERT 0 1
test=# select * from person;
  id | name  | age
 ----+-------+-----
   1 | Tim   |  20
   2 | Horst |  88
   4 |       |  40  <-- null for name
(3 rows)
```

There are many more datatypes you can use - check the PostgreSQL manual![1]

14.1.6 Modelling an Address Database

Let's use a simple case study to see how a database is constructed. We want to create an address database. What kind of information should we store?

Write some address properties in the space provided:

[1] http://www.postgresql.org/docs/current/static/datatype.html

The properties that describe an address are the columns. The type of information stored in each column is its datatype. In the next section we will analyse our conceptual address table to see how we can make it better!

14.1.7 Database Theory

The process of creating a database involves creating a model of the real world; taking real world concepts and representing them in the database as entities.

14.1.8 Normalisation

One of the main ideas in a database is to avoid data duplication / redundancy. The process of removing redundancy from a database is called Normalisation.

> Normalization is a systematic way of ensuring that a database structure is suitable for general-purpose querying and free of certain undesirable characteristics - insertion, update, and deletion anomalies - that could lead to a loss of data integrity. - *Wikipedia*

There are different kinds of normalisation ‘forms’.

Let’s take a look at a simple example:

```
Table "public.people"
  Column  |          Type          |                Modifiers
----------+------------------------+------------------------------------------
 id       | integer                | not null default
          |                        | nextval('people_id_seq'::regclass)
          |                        |
 name     | character varying(50)  |
 address  | character varying(200) | not null
 phone_no | character varying      |
Indexes:
 "people_pkey" PRIMARY KEY, btree (id)
```

```
select * from people;
id |     name      |           address           |  phone_no
 --+---------------+-----------------------------+-------------
 1 | Tim Sutton    | 3 Buirski Plein, Swellendam | 071 123 123
 2 | Horst Duester | 4 Avenue du Roix, Geneva    | 072 121 122
(2 rows)
```

Imagine you have many friends with the same street name or city. Every time this data is duplicated, it consumes space. Worse still, if a city name changes, you have to do a lot of work to update your database.

Try to redesign our people table to reduce duplication:

You can read more about database normalisation here[2]

14.1.9 Indexes

> A database index is a data structure that improves the speed of data retrieval operations on a database table. - *Wikipedia*

Imagine you are reading a textbook and looking for the explanation of a concept - and the textbook has no index! You will have to start reading at one cover and work your way through the entire book until you find the information you need. The index at the back of a book helps you to jump quickly to the page with the relevant information.

```
create index person_name_idx on people (name);
```

Now searches on name will be faster:

```
Table "public.people"
  Column  |          Type          |                Modifiers
----------+------------------------+-----------------------------------------
 id       | integer                | not null default
          |                        | nextval('people_id_seq'::regclass)
          |                        |
 name     | character varying(50)  |
 address  | character varying(200) | not null
 phone_no | character varying      |
Indexes:
 "people_pkey" PRIMARY KEY, btree (id)
 "person_name_idx" btree (name)
```

14.1.10 Sequences

A sequence is a unique number generator. It is normally used to create a unique identifier for a column in a table.

In this example, id is a sequence - the number is incremented each time a record is added to the table:

```
id |     name     |           address           |  phone_no
---+--------------+-----------------------------+-------------
 1 | Tim Sutton   | 3 Buirski Plein, Swellendam | 071 123 123
 2 | Horst Duster | 4 Avenue du Roix, Geneva    | 072 121 122
```

14.1.11 Entity Relationship Diagramming

In a normalised database, you typically have many relations (tables). The entity-relationship diagram (ER Diagram) is used to design the logical dependencies between the relations. Remember our un-normalised people table?

[2] http://en.wikipedia.org/wiki/Database_normalization

```
test=# select * from people;
 id |     name     |           address            |  phone_no
----+--------------+------------------------------+-------------
 1  | Tim Sutton   | 3 Buirski Plein, Swellendam | 071 123 123
 2  | Horst Duster | 4 Avenue du Roix, Geneva    | 072 121 122
(2 rows)
```

With a little work we can split it into two tables, removing the need to repeat the street name for individuals who live in the same street:

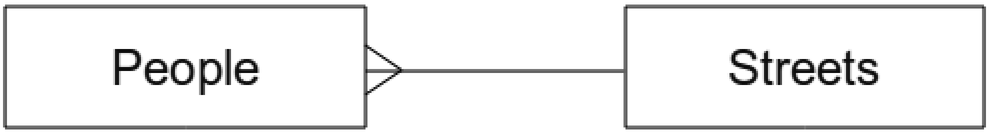
```
test=# select * from streets;
 id |     name
----+--------------
 1  | Plein Street
(1 row)
```

and

```
test=# select * from people;
 id |     name     | house_no | street_id |  phone_no
----+--------------+----------+-----------+-------------
  1 | Horst Duster |        4 |         1 | 072 121 122
(1 row)
```

We can then link the two tables using the 'keys' `streets.id` and `people.streets_id`.

If we draw an ER Diagram for these two tables it would look something like this:



The ER Diagram helps us to express 'one to many' relationships. In this case the arrow symbol show that one street can have many people living on it.

Our people model still has some normalisation issues - try to see if you can normalise it further and show your thoughts by means of an ER Diagram.

14.1.12 Constraints, Primary Keys and Foreign Keys

A database constraint is used to ensure that data in a relation matches the modeller’s view of how that data should be stored. For example a constraint on your postal code could ensure that the number falls between `1000` and `9999`.

A Primary key is one or more field values that make a record unique. Usually the primary key is called id and is a sequence.

A Foreign key is used to refer to a unique record on another table (using that other table’s primary key).

In ER Diagramming, the linkage between tables is normally based on Foreign keys linking to Primary keys.

If we look at our people example, the table definition shows that the street column is a foreign key that references the primary key on the streets table:

```
Table "public.people"
  Column   |         Type          |  Modifiers
-----------+-----------------------+--------------------------------------
 id        | integer               | not null default
           |                       | nextval(’people_id_seq’::regclass)
 name      | character varying(50) |
 house_no  | integer               | not null
 street_id | integer               | not null
 phone_no  | character varying     |
Indexes:
"people_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
"people_street_id_fkey" FOREIGN KEY (street_id) REFERENCES streets(id)
```

14.1.13 Transactions

When adding, changing, or deleting data in a database, it is always important that the database is left in a good state if something goes wrong. Most databases provide a feature called transaction support. Transactions allow you to create a rollback position that you can return to if your modifications to the database did not run as planned.

Take a scenario where you have an accounting system. You need to transfer funds from one account and add them to another. The sequence of steps would go like this:

- remove R20 from Joe
- add R20 to Anne

If something goes wrong during the process (e.g. power failure), the transaction will be rolled back.

14.1.14 In conclusion

Databases allow you to manage data in a structured way using simple code structures.

14.1.15 What's next?

Now that we've looked at how databases work in theory, let's create a new database to implement the theory we've covered.

14.2 Lesson: Implementing the Data Model

Now that we've covered all the theory, let's create a new database. This database will be used for our exercises for the lessons that will follow afterwards.

The goal for this lesson: To install the required software and use it to implement our example database.

14.2.1 Install PostgreSQL

Under Ubuntu:

```
sudo apt-get install postgresql-9.1
```

You should get a message like this:

```
[sudo] password for timlinux:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
```

```
postgresql-client-9.1 postgresql-client-common postgresql-common
Suggested packages:
oidentd ident-server postgresql-doc-9.1
The following NEW packages will be installed:
postgresql-9.1 postgresql-client-9.1 postgresql-client-common postgresql-common
0 upgraded, 4 newly installed, 0 to remove and 5 not upgraded.
Need to get 5,012kB of archives.
After this operation, 19.0MB of additional disk space will be used.
Do you want to continue [Y/n]?
```

Press `Y` and `Enter` and wait for the download and installation to finish.

14.2.2 Help

PostgreSQL has very good online[3] documentation.

14.2.3 Create a database user

Under Ubuntu:

After the installation is complete, run this command to become the postgres user and then create a new database user:

```
sudo su - postgres
```

Type in your normal log in password when prompted (you need to have sudo rights).

Now, at the postgres user's bash prompt, create the database user. Make sure the user name matches your unix login name: it will make your life much easier, as postgres will automatically authenticate you when you are logged in as that user.

```
createuser -d -E -i -l -P -r -s timlinux
```

Enter a password when prompted. I normally use a different password to my usual unix login.

What do those options mean?

```
-d, --createdb     role can create new databases
-E, --encrypted    encrypt stored password
-i, --inherit      role inherits privileges of roles it is a member of (default)
-l, --login        role can login (default)
-P, --pwprompt     assign a password to new role
-r, --createrole   role can create new roles
-s, --superuser    role will be superuser
```

Now you should should leave the postgres user's bash shell environment by typing:

```
exit
```

[3] http://www.postgresql.org/docs/9.1/static/index.html

14.2.4 Verify the new account

```
psql -l
```

Should return something like this:

```
timlinux@linfiniti:~$ psql -l
List of databases
Name      |  Owner   | Encoding | Collation  |   Ctype    |
----------+----------+----------+------------+------------+
postgres  | postgres | UTF8     | en_ZA.utf8 | en_ZA.utf8 |
template0 | postgres | UTF8     | en_ZA.utf8 | en_ZA.utf8 |
template1 | postgres | UTF8     | en_ZA.utf8 | en_ZA.utf8 |
(3 rows)
```

Type `q` to exit.

14.2.5 Create a database

The `createdb` command is used to create a new database. It should be run from the bash shell prompt.

```
createdb address
```

You can verify the existence of your new database by using this command:

```
psql -l
```

Which should return something like this:

```
List of databases
Name      |  Owner   | Encoding | Collation  |   Ctype    |   Access privileges
----------+----------+----------+------------+------------+---------------------
address   | timlinux | UTF8     | en_ZA.utf8 | en_ZA.utf8 |
postgres  | postgres | UTF8     | en_ZA.utf8 | en_ZA.utf8 |
template0 | postgres | UTF8     | en_ZA.utf8 | en_ZA.utf8 | =c/postgres: postgre
template1 | postgres | UTF8     | en_ZA.utf8 | en_ZA.utf8 | =c/postgres: postgre
(4 rows)
```

Type `q` to exit.

14.2.6 Starting a database shell session

You can connect to your database easily like this:

```
psql address
```

To exit out of the psql database shell, type:

```
\q
```

For help in using the shell, type:

```
\?
```

For help in using sql commands, type:

```
\help
```

To get help on a specific command, type (for example):

```
\help create table
```

See also the Psql cheat sheet - available online here[4].

14.2.7 Make Tables in SQL

Let's start making some tables! We will use our ER Diagram as a guide. First, let's create a streets table:

```
create table streets (id serial not null primary key, name varchar(50));
```

`serial` and `varchar` are **data types**. `serial` tells PostgreSQL to start an integer sequence (autonumber) to populate the `id` automatically for every new record. `varchar(50)` tells PostgreSQL to create a character field of 50 characters in length.

You will notice that the command ends with a `;` - all SQL commands should be terminated this way. When you press enter, psql will report something like this:

```
NOTICE:  CREATE TABLE will create implicit sequence "streets_id_seq" for
         serial column "streets.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "streets_pkey"
         for table "streets"
CREATE TABLE
```

That means your table was created successfully, with a primary key `streets_pkey` using `streets.id`.

Note: If you hit return without entering a `;`, then you will get a prompt like this: `address-#`. This is because PG is expecting you to enter more. Enter `;` to run your command.

To view your table schema, you can do this:

```
\d streets
```

Which should show something like this:

```
Table "public.streets"
 Column |         Type          |            Modifiers
--------+-----------------------+--------------------------------------
 id     | integer               | not null default
        |                       | nextval('streets_id_seq'::regclass)
 name   | character varying(50) |
```

[4] http://www.postgresonline.com/special_feature.php?sf_name=postgresql83_psql_cheatsheet&outputformat=html

```
Indexes:
  "streets_pkey" PRIMARY KEY, btree (id)
```

To view your table contents, you can do this:

```
select * from streets;
```

Which should show something like this:

```
id | name
---+------
(0 rows)
```

As you can see, our table is empty!

Use the approach shown above to make a table called people: Add fields such as phone number, home address, name, etc. (these aren't all valid names: change them to make them valid).

> Write the SQL you create here:

Solution:

```
create table people (id serial not null primary key,
                     name varchar(50),
                     house_no int not null,
                     street_id int not null,
                     phone_no varchar null );
```

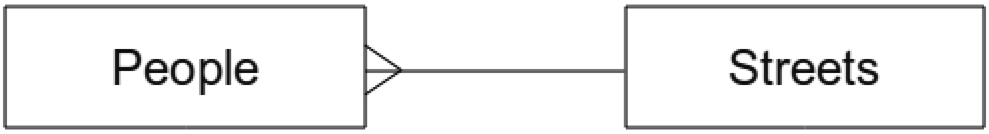
The schema for the table (enter `\d people`) looks like this:

```
Table "public.people"
Column     |         Type          |                      Modifiers
-----------+-----------------------+-------------------------------------
id         | integer               | not null default
           |                       | nextval(’people_id_seq’::regclass)
name       | character varying(50) |
house_no   | integer               | not null
street_id  | integer               | not null
phone_no   | character varying     |
Indexes:
  "people_pkey" PRIMARY KEY, btree (id)
```

Note: For illustration purposes, we have purposely omitted the fkey constraint.

14.2.8 Create Keys in SQL

The problem with our solution above is that the database doesn’t know that people and streets have a logical relationship. To express this relationship, we have to define a foreign key that points to the primary key of the streets table.



There are two ways to do this:

- adding the key after the table has been created
- defining the key at time of table creation

Our table has already been created, so let's do it the first way:

```
alter table people
  add constraint people_streets_fk foreign key (street_id) references streets(id
```

That tells the `people` table that its `street_id` fields must match a valid street `id` from the `streets` table.

The more usual way to create a constraint is to do it when you create the table:

```
create table people (id serial not null primary key,
                     name varchar(50),
                     house_no int not null,
                     street_id int references streets(id) not null,
                     phone_no varchar null);
```

After adding the constraint, our table schema looks like this now:

```
\d people
Table "public.people"
  Column   |         Type          |            Modifiers
-----------+-----------------------+---------------------------------
 id        | integer               | not null default
           |                       | nextval('people_id_seq'::regclass)
 name      | character varying(50) |
 house_no  | integer               | not null
 street_id | integer               | not null
 phone_no  | character varying     |
Indexes:
  "people_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
  "people_streets_fk" FOREIGN KEY (id) REFERENCES streets(id)
```

14.2.9 Create Indexes in SQL

We want lightning fast searches on peoples names. To provide for this, we can create an index on the name column of our people table:

```
create index people_name_idx on people(name);
```

```
address=# \d people
Table "public.people"
  Column   |         Type          |                      Modifiers
-----------+-----------------------+-----------------------------------
 id        | integer               | not null default nextval
           |                       | ('people_id_seq'::regclass)
 name      | character varying(50) |
 house_no  | integer               | not null
 street_id | integer               | not null
 phone_no  | character varying     |
Indexes:
 "people_pkey" PRIMARY KEY, btree (id)
 "people_name_idx" btree (name)    <-- new index added!
Foreign-key constraints:
 "people_streets_fk" FOREIGN KEY (id) REFERENCES streets(id)
```

14.2.10 Dropping Tables in SQL

If you want to get rid of a table you can use the drop command:

```
drop table streets;
```

In our example, this would not work - why?

> Some deep and inspired thoughts as to why...

Sometimes you just can't stand having a table any more. Maybe you are sick of all your friends. How can you get rid of them all in one easy step? Drop the table of course! Of course, right now too much hard work has gone into our table to get rid of it on a whim, but if you really hate your friends that much, nothing's stopping you from ridding yourself of them forever:

```
drop table people;
```

This time it works fine! Why? Are people less important than streets?

> Some thoughts on why you could drop people:

Note: If you actually did enter that command and dropped the `people` table, now would be a good time to rebuild it, as you will need it in the next exercises.

14.2.11 A word on PG Admin III

We are showing you the SQL commands from the psql prompt because it's a very useful way to learn about databases. However, there are quicker and easier ways to do a lot of what we are showing you. Install PGAdminIII and you can create, drop, alter etc tables using 'point and click' operations in a GUI.

Under Ubuntu, you can install it like this:

```
sudo apt-get install pgadmin3
```

14.2.12 In conclusion

You have now seen how to create a brand new database, starting completely from scratch.

14.2.13 What's next?

Next you'll learn how to use the DBMS to add new data.

14.3 Lesson: Adding Data to the Model

The models we've created will now need to be populated with the data they're intended to contain.

The goal for this lesson: To learn how to insert new data into the database models.

14.3.1 Insert statement

How do you add data to a table? The sql `INSERT` statement provides the functionality for this:

```
insert into streets (name) values ('High street');
```

A couple of things to note:

- after the table name (`streets`), you list the column names that you will be populating (in this case only the `name` column).
- after the `values` keyword, place the list of field values.
- strings should be quoted using single quotes.
- you will note that I did not insert a value for the `id` column - that is because it is a sequence and will be autogenerated.
- if you do manually set the `id`, you may cause serious problems with the integrity of your database.

You should see `INSERT 0 1` if it is successful.

You can see the result of your insert action by selecting all the data in the table:

```
select * from streets;
```

result:

```
select * from streets;
 id |    name
----+-------------
  1 | High street
(1 row)
```

Now you try:

Use the `INSERT` command to add a new street to the `streets` table.

Write the sql you used here:

14.3.2 Sequencing data addition according to constraints

Try to add a person to the people table with the following details:

```
Name: Joe Smith
House Number: 55
Street: Main Street
Phone: 072 882 33 21
```

Remember, we defined phone numbers as strings.

> What problems did you encounter?

You should have an error report if you try to do this without first creating a record for Main Street in the `streets` table.

What error did you get?

You should have noticed that:

- You can’t add the street using its name
- You can’t add a street using a street `id` before first creating the street record on the streets table

Remember that our two tables are linked via a Primary/Foreign Key pair. This means that no valid person can be created without there also being a valid corresponding street record.

Here is how we made our friend:

```
insert into streets (name) values('Main Street');
insert into people (name,house_no, street_id, phone_no)
  values ('Joe Smith',55,2,'072 882 33 21');
```

If you look at the streets table again (using a select statement as before), you’ll see that the `id` for the `Main Street` entry is `2`. That’s why we could merely enter the number `2` above. Even though we’re not seeing `Main Street` written out fully in the entry above, the database will be able to associate that with the `street_id` value of `2`.

14.3.3 Select data

We have already shown you the syntax for selecting records. Lets look at a few more examples:

```
select name from streets;
```

```
select * from streets;
```

```
select * from streets where name='Main Street';
```

In later sessions we will go into more detail on how to select and filter data.

14.3.4 Update data

What is you want to make a change to some existing data? For example a street name is changed:

```
update streets set name='New Main Street' where name='Main Street';
```

Be very careful using such update statements - if more than one record matches your `WHERE` clause, they will all be updated!

A better solution is to use the primary key of the table to reference the record to be changed:

```
update streets set name='New Main Street' where id=2;
```

It should return `UPDATE 1`.

Note: the `WHERE` statement criteria are case sensitive `Main Street` <> `Main street`

14.3.5 Delete Data

Some times you fall out of friendship with people. Sounds like a job for the DELETE command!

```
delete from people where name = 'Joe Smith';
```

Let's look at our people table now:

```
address=# select * from people;
  id | name | house_no | street_id | phone_no
 ----+------+----------+-----------+----------
(0 rows)
```

Exercise: Use the skills you learned earlier to add some new friends to your database:

```
       name        | house_no | street_id |   phone_no
-------------------+----------+-----------+---------------
 Joe Bloggs        |        3 |         2 | 072 887 23 45
 IP Knightly       |       55 |         3 | 072 837 33 35
 Rusty Bedsprings  |       33 |         1 | 072 832 31 38
 QGIS Geek         |       83 |         1 | 072 932 31 32
```

14.3.6 In conclusion

Now you know how to add new data to the existing models you created previously. Remember that if you want to add new kinds of data, you may want to modify and/or create new models to contain that data.

14.3.7 What's next?

Now that you've added some data, you'll learn how to use queries to access this data in various ways.

14.4 Lesson: Queries

When you write a SELECT ... command it is commonly known as a query - you are interrogating the database for information.

The goal of this lesson: To learn how to create queries that will return useful information.

14.4.1 Follow-up from previous lesson

Let's check that you added a few people to the database successfully:

```
insert into people (name,house_no, street_id, phone_no)
            values ('Joe Bloggs',3,1,'072 887 23 45');
insert into people (name,house_no, street_id, phone_no)
            values ('IP Knightly',55,1,'072 837 33 35');
insert into people (name,house_no, street_id, phone_no)
            values ('Rusty Bedsprings',33,1,'072 832 31 38');
insert into people (name,house_no, street_id, phone_no)
            values ('QGIS Geek',83,1,'072 932 31 32');
```

14.4.2 Ordering results

Let's get a list of people ordered by their house numbers:

```
select name, house_no from people order by house_no;
```

Result:

```
       name       | house_no
------------------+----------
 Joe Bloggs       |        3
 Rusty Bedsprings |       33
 IP Knightly      |       55
 QGIS Geek        |       83
(4 rows)
```

You can sort by more than one column:

```
select name, house_no from people order by name, house_no;
```

Result:

```
       name       | house_no
------------------+----------
 IP Knightly      |       55
 Joe Bloggs       |        3
 QGIS Geek        |       83
 Rusty Bedsprings |       33
(4 rows)
```

14.4.3 Filtering

Often you won't want to see every single record in the database - especially if there are thousands of records and you are only interested in seeing one or two. Never fear, you can filter the results!

Here is an example of a numerical filter:

```
address=# select name, house_no from people where house_no < 50;
       name       | house_no
------------------+----------
```

```
 Joe Bloggs       |        3
 Rusty Bedsprings |       33
(2 rows)
```

You can combine filters (defined using the `WHERE` clause) with sorting (defined using the `ORDER BY`)

```
address=# select name, house_no from people where house_no < 50 order by
address-# house_no;
       name       | house_no
------------------+----------
 Joe Bloggs       |        3
 Rusty Bedsprings |       33
(2 rows)
```

You can also filter based on text data:

```
address=# select name, house_no from people where name like '%i%';
       name       | house_no
------------------+----------
 IP Knightly      |       55
 Rusty Bedsprings |       33
(2 rows)
```

Here we used the `LIKE` clause to find all names with an `i` in them. If you want to search for a string of letters regardless of case, you can do a case insensitive search using the `ILIKE` clause:

```
address=# select name, house_no from people where name ilike '%k%';
    name     | house_no
-------------+----------
 IP Knightly |       55
 QGIS Geek   |       83
(2 rows)
```

That found everyone with a `k` or `K` in their name. Using the normal `ILIKE` clause, you'd get:

```
address=# select name, house_no from people where name like '%k%';
    name    | house_no
 -----------+----------
  QGIS Geek |       83
  (1 row)
```

14.4.4 Joins

What if you want to see the person's details and their street name (not its id)? In order to do that, you need to join the two tables together in a single query. Lets look at an example:

```
select people.name, house_no, streets.name
from people,streets
where people.street_id=streets.id;
```

Note: With joins, you will always state the two tables the information is coming from, in this case people and streets. You also need to specify which two keys must match (foreign key & primary key). If you don't specify that, you will get a list of all possible combinations of people and streets, but no way to know who actually lives on which street!

Here is what the correct output will look like:

```
       name       | house_no |    name
------------------+----------+-------------
 Joe Bloggs       |        3 | High street
 IP Knightly      |       55 | High street
 Rusty Bedsprings |       33 | High street
 QGIS Geek        |       83 | High street
(4 rows)
```

We will revisit joins as we create more complex queries later. Just remember they provide a simple way to combine the information from two or more tables.

14.4.5 Subselect

First, let's do a little tweaking to our data:

```
insert into streets (name) values('QGIS Road');
insert into streets (name) values('OGR Corner');
insert into streets (name) values('Goodle Square');
update people set street_id = 2 where id=2;
update people set street_id = 3 where id=3;
```

Let's take a quick look at our data after those changes - we reuse our query from the previous section:

```
select people.name, house_no, streets.name
from people,streets
where people.street_id=streets.id;
```

Result:

```
       name       | house_no |      name
------------------+----------+-----------------
 Rusty Bedsprings |       33 | High street
 QGIS Geek        |       83 | High street
 Joe Bloggs       |        3 | New Main Street
 IP Knightly      |       55 | QGIS Road
(4 rows)
```

Now let's show you a subselection on this data. We want to show only people who live in `street_id` number `1`.

```
select people.name
from people, (
    select *
    from streets
    where id=1
  ) as streets_subset
where people.street_id = streets_subset.id;
```

Result:

```
       name
------------------
 Rusty Bedsprings
 QGIS Geek
(2 rows)
```

This is a contrived example and in the above situations it may have been overkill. However when you have to filter based on a selection, subselects are really useful!

14.4.6 Aggregate Queries

One of the powerful features of a database is its ability to summarise the data in its tables. These summaries are called aggregate queries. Here is a typical example:

```
select count(*) from people;
```

Result:

```
 count
-------
     4
(1 row)
```

If we want the counts summarised by street name we can do this:

```
select count(name), street_id
from people
group by street_id;
```

Result:

```
 count | street_id
-------+-----------
     1 |         2
     1 |         3
     2 |         1
(3 rows)
```

Note: Because there is no `ORDER BY` clause, on your computer these data may not be in the same order as shown here.

Exercise:

> Summarise the people by street name and show the actual street names instead of the street_id's

Solution:

```
select count(people.name), streets.name
from people, streets
where people.street_id=streets.id
group by streets.name;
```

Result:

```
 count |      name
-------+-----------------
     1 | New Main Street
     2 | High street
     1 | QGIS Road
(3 rows)
```

Note: You will notice that we have prefixed field names with table names (e.g. people.name and streets.name). This needs to be done whenever the field name is ambiguous.

14.4.7 In conclusion

You've seen how to use queries to return the data in your database in a way that allows you to extract useful information from it.

14.4.8 What's next?

Next you'll see how to create views from the queries that you've written.

14.5 Lesson: Views

When you write a query, you need to spend a lot of time and effort formulating it. With views, you can save the definition of a sql query in a reusable 'virtual table'.

The goal for this lesson: To save a query as a view.

14.5.1 Creating a View

You can treat a view just like a table, but its data is sourced from a query. Let's make a simple view based on the above.

```
create view roads_count_v as
  select count(people.name), streets.name
  from people, streets where people.street_id=streets.id
  group by people.street_id, streets.name;
```

As you can see the only change is the `create view roads_count_v as` part at the beginning. The nice thing is that we can now select data from that view:

```
select * from roads_count_v;
```

Result:

```
 count |      name
-------+-----------------
     1 | New Main Street
     2 | High street
     1 | QGIS Road
(3 rows)
```

14.5.2 Modifying a View

A view is not fixed, and it contains no 'real data'. This means you can easily change it without impacting on any data in your database.

```
CREATE OR REPLACE VIEW roads_count_v AS
  SELECT count(people.name), streets.name
  FROM people, streets WHERE people.street_id=streets.id
  GROUP BY people.street_id, streets.name
  ORDER BY streets.name;
```

(This example also shows the best practice convention of using UPPER CASE for all SQL keywords.)

You will see that we have added an ORDER BY clause so that our view rows are nicely sorted:

```
 count |      name
-------+-----------------
     2 | High street
     1 | New Main Street
     1 | QGIS Road
(3 rows)
```

14.5.3 Dropping a View

If you no longer need a view, you can delete it like this:

```
drop view roads_count_v;
```

14.5.4 In conclusion

Using views, you can save a query and access its results as if it were a table.

14.5.5 What's next?

Sometimes, when changing data, you want your changes to have effects elsewhere in the database. The next lesson will show you how to do this.

14.6 Lesson: Rules

> Rules allow the "query tree" of an incoming query to be rewritten. One common usage is to implement views, including updatable view. - *Wikipedia*

The goal for this lesson: To learn how to create new rules for the database.

14.6.1 Materialised Views (Rule based views)

Say you want to log every change of phone_no in your people table in to a people_log table. So you set up a new table

```
create table people_log (name text, time timestamp default NOW());
```

In the next step create a rule, that logs every change of a phone_no in the people table into the people_log table:

```
create rule people_log as on update to people
  where NEW.phone_no <> OLD.phone_no
  do insert into people_log values (OLD.name);
```

To test that the rule works, let's modify a phone number:

```
update people set phone_no = '082 555 1234' where id = 2;
```

Check that the table was updated correctly:

```
 id |    name    | house_no | street_id |   phone_no
----+------------+----------+-----------+--------------
  2 | Joe Bloggs |        3 |         2 | 082 555 1234
(1 row)
```

Now, thanks to the rule we created, the `people_log` table will look like this:

```
    name    |            time
------------+----------------------------
 Joe Bloggs | 2012-04-23 15:20:56.683382
(1 row)
```

Note: The value of the `time` field will depend on the current date and time.

14.6.2 In conclusion

Rules allow you to automatically add or change data in your database to reflect changes in other parts of the database.

14.6.3 What's next?

The next module will introduce you to PostGIS, which takes these database concepts and applies them to GIS data.