

PostGIS Training Course

Tim Sutton, Horst Duester, 2010

Contents

1	Introduction to PostgreSQL	3
1.1	What is a Database?	3
1.1.1	Tables	3
1.1.2	Columnns / Fields	3
1.1.3	Records	4
1.1.4	Datatypes	4
1.1.5	Modelling an Address Database	5
1.2	Database Theory	5
1.2.1	Normalisation	6
1.2.2	Indexes	7
1.2.3	Sequences	8
1.2.4	Entity Relationship Diagramming	8
1.2.5	Constraints, Primary Keys and Foreign Keys	10
1.2.6	Transactions	11
1.3	Implementing the Data Model	11
1.3.1	Install PostgreSQL	11
1.3.2	Create a database user	12
1.3.3	Verify the new account	13
1.3.4	Create a database	13
1.3.5	Starting a database shell session	14
1.3.6	Make Tables in SQL	14
1.3.7	Create Keys in SQL	17
1.3.8	Create Indexes SQL	18
1.3.9	Dropping Tables SQL	19
1.3.10	A word on PG Admin III	20
1.4	Adding Data to the Model	20

1.4.1	Insert statement	20
1.4.2	Sequencing data addition according to constraints	21
1.4.3	Select data	22
1.4.4	Update data	23
1.4.5	Delete Data	23
1.5	Queries	24
1.5.1	Ordering results	24
1.5.2	Filtering	25
1.5.3	Joins	26
1.5.4	Subselect	27
1.5.5	Aggregate Queries	28
1.6	Views	30
1.6.1	Creating a View	30
1.6.2	Modifying a view	30
1.6.3	Dropping a View	31
1.7	Rules	31
1.7.1	Materialised Views (Rule based views)	31
2	Introduction to PostGIS	32
2.1	PostGIS Setup	32
2.1.1	Installing under Ubuntu	32
2.1.2	Installing under Windows	32
2.1.3	Install plpgsql	32
2.1.4	Install postgis.sql	33
2.1.5	Install spatial_refsys.sql	33
2.1.6	Looking at the installed PostGIS functions	34
2.2	Simple Feature Model	36
2.2.1	What is OGC	36

2.2.2	What is the SFS Model	36
2.2.3	Add a geometry field to table	37
2.2.4	Add a constraint based on geometry type	38
2.2.5	Populate geometry_columns table	39
2.2.6	Add geometry record to table using SQL	40
2.2.7	View a point as WKT	41
2.3	Spatial Queries	41
2.3.1	Spatial operators	41
2.3.2	Spatial Indexes	42
2.4	Geometry Construction	44
2.4.1	Creating Linestrings	44
2.4.2	Creating Polygons	45
2.4.3	Exercise - Linking Cities to People	45
2.4.4	Looking at our schema	48
2.4.5	Exercise	48
2.4.6	Access Subobjects	48
2.5	Data Processing	49
2.5.1	Clipping	49
2.5.2	Building Geometries from Other Geometries	51
2.5.3	Geometry Cleaning	53
2.5.4	Differences between tables	53
2.6	Import and Export	54
2.6.1	shp2pgsql	54
2.6.2	pgsql2shp	54
2.6.3	ogr2ogr	55
2.6.4	spit	55

1 Introduction to PostgreSQL

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

1.1.1 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**.

1.1.2 Columnns / 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 |
```

1.1.3 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
```

1.1.4 Datatypes

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

There are many kinds of datatypes, lets 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 Horsts 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 - this
data is referred to as 'Null'.

```
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!](#)

1.1.5 Modelling an Address Database

Lets 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 here:

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!

1.2 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.

1.2.1 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'.

Lets 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.

Discussion:

Try to redesign our people table to reduce duplication!

You can read more about database normalisation [here](#)

1.2.2 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 text book and lookig for the explanation of a concept

- and the text book 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)

1.2.3 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

1.2.4 Entity Relationship Diagramming

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

```
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 make 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 streets;
 id |      name      | house_no | street_id | phone_no
----+-----+-----+-----+-----
  1 | Horst Duster  |      4   |          1 | 072 121 122
(1 row)
```

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.

1.2.5 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.

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)
```

1.2.6 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.

1.3 Implementing the Data Model

1.3.1 Install PostgreSQL

Under Ubuntu:

```
sudo apt-get install postgresql-8.4
```

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-8.4 postgresql-client-common postgresql-common
Suggested packages:
oidentd ident-server postgresql-doc-8.4
The following NEW packages will be installed:
postgresql-8.4 postgresql-client-8.4 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]? y
```

Press 'Y' and wait for the download and installation to finish.

1.3.2 Create a database user

Under linux:

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 prompted create the database user. Make sure the user name matches your unix login name as 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
```

1.3.3 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)
```

1.3.4 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: postgres=C
template1    | postgres  | UTF8     | en_ZA.utf8 | en_ZA.utf8 | =c/postgres: postgres=C
(4 rows)
```

1.3.5 Starting a database shell session

You can connect to your database easily like this:

```
psql address
```

To exit out of the psql database shell you type:

```
\q
```

For help in using the shell, type

```
\?
```

For help in using sql commands type:

```
\help
```

To get help on a specific command, do:

```
\help create table
```

See also the [Psql cheat sheet](#) - available online [here](#).

1.3.6 Make Tables in SQL

Lets start making some tables! We will use our ER Diagram as a guide. First lets create a streets table:

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

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 implic
index "streets_pkey" for table "streets"
CREATE TABLE
```

That means your table was created successfully.

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)	

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:

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 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 advanced users, we have purposely omitted the fkey constraint.

1.3.7 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

Lets show you the first way:

```

alter table people
  add constraint people_streets_fk foreign key (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)
```

1.3.8 Create Indexes 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)
```

1.3.9 Dropping Tables 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!

```
drop table people;
```

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

Some thoughts on why you could drop people:

1.3.10 A word on PG Admin III

We are showing you the SQL commands from the psql prompt because its 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' in a GUI.

1.4 Adding Data to the Model

1.4.1 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 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:

1.4.2 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
```

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.

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 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',55,1,'072 882 33 21');
```

1.4.3 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.

1.4.4 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;
```

1.4.5 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';
```

Lets look at our people table now:

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

Excercise:

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

1.5 Queries

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

Lets 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');
```

1.5.1 Ordering results

Lets 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)

```

1.5.2 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 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:

```
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.

1.5.3 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;
```

Here is what the 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.

1.5.4 Subselect

First lets 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;
```

Lets 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 lets 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!

1.5.5 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)

```

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 people.street_id, 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.

1.6 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'.

1.6.1 Creating a View

You can treat a view just like a table, but its data is sourced from a query. Lets 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 road_count_v;
```

Result:

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

1.6.2 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 road_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;
```

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)
```

1.6.3 Dropping a View

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

```
drop view roads_count_v;
```

1.7 Rules

1.7.1 Materialised Views (Rule based views)

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

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);
```

2 Introduction to PostGIS

While working through this section, you may want to keep a copy of the [PostGIS cheat sheet [postgis_cheatsheet.pdf](#)] available at [Boston GIS user group http://www.bostongis.com/postgis_quickguide.bgg].

2.1 PostGIS Setup

2.1.1 Installing under Ubuntu

Postgis is easily installed from apt.

```
sudo apt-get install postgis
```

Really, its that easy...

2.1.2 Installing under Windows

<http://www.postgresql.org/download/>

Now follow this guide:

http://www.bostongis.com/PrinterFriendly.aspx?content_name=postgis_tut01

A little more complicated, but still not hard. Note that you need to be online to install the postgis stack.

2.1.3 Install plpgsql

Note: You can ensure that any database created on your system automatically gets these spatial extensions by running these commands (from this and the next two sections) on the template1 system database "before" you create any of your own databases.

PostgreSQL can use various procedural languages. What is a procedural language? It is an 'in database' language that can be used to extend the functionality of the database. For example you can write database functions that are called when events happen - such as when a record is inserted into the database.

PostGIS requires the PLPGSQL procedural language to be installed so do this:

```
createlang plpgsql address
```

Where the third argument is the name of the database that the procedural language should be installed into.

Note: that you will need administrative permissions for your database to be able to do that.

2.1.4 Install postgis.sql

PostGIS can be thought of as a collection of in database functions that extend the core capabilities of PostgreSQL so that it can deal with spatial data. By 'deal with', we mean, store, retrieve, query and manipulate. In order to do this, a number of functions are installed into the database.

```
psql address < /usr/share/postgresql/8.4/contrib/postgis-1.5/postgis.sql
```

Once that has run (it normally only takes a few seconds), you can view all the functions that were loaded by doing this:

```
psql address  
\df
```

from within the psql prompt. We will discuss these functions in more detail as we proceed with this course.

2.1.5 Install spatial_refsys.sql

In addition to the postgis functions, a second helper sql script needs to be run that will load the database with a collection of spatial reference system (SRS) definitions defined by the European Petroleum Survey Group (EPSG). These are used during operations such as coordinate system conversions.

You can add more to the SRS list later if needed, but the list provided should cover just about every SRS you will need (Google Mercator and lo are notable exceptions).

To load the SRS table do:

```
psql address < /usr/share/postgresql/8.4/contrib/postgis-1.5/spatial_ref_sys.sql
```

The above command adds a table to our database. We can see the schema of this table like this:

```
address=# \d spatial_ref_sys
Table "public.spatial_ref_sys"
  Column      |          Type          | Modifiers
-----+-----+-----
 srid         | integer                | not null
 auth_name    | character varying(256) |
 auth_srid    | integer                |
 srtext       | character varying(2048) |
 proj4text    | character varying(2048) |
Indexes:
"spatial_ref_sys_pkey" PRIMARY KEY, btree (srid)
```

You can use standard SQL queries (as we have learned from our introductory sections), to view and manipulate this table - though we suggest you do not update or delete any records unless you know what you are doing.

One SRID you may be interested in is EPSG:4326 - the geographic / lat lon reference system. Lets take a look at it:

```
select * from spatial_ref_sys where srid=4326;
```

Result

```
srid      | 4326
auth_name | EPSG
auth_srid | 4326
srtext    | GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID["WGS
84",6378137,298.257223563,AUTHORITY["EPSG","7030"]],TOWGS84[0,
0,0,0,0,0],AUTHORITY["EPSG","6326"]],PRIMEM["Greenwich",0,
AUTHORITY["EPSG","8901"]],UNIT["degree",0.01745329251994328,
AUTHORITY["EPSG","9122"]],AUTHORITY["EPSG","4326"]]
proj4text | +proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs
```

The srtext is the projection definition in well known text (you may recognise this from .prj files in your shapefile collection).

2.1.6 Looking at the installed PostGIS functions

Good - our PostgreSQL database is now geospatially enabled. We are going to delve a lot deeper into this in the coming days, but lets give you a quick

little taster. Lets say we want to create a point from text. First we use the psql command to find functions relating to point:

```
\df *point*
```

Here is one that caught my eye: 'st_pointfromtext'

So lets give that a try:

```
address=# select st_pointfromtext('POINT(1 1)');
```

Result:

```
st_pointfromtext
-----
010100000000000000000000F03F000000000000F03F
(1 row)
```

So there are a couple of interesting things going on here:

- we defined a point at position 1,1 (EPSG:4326 is assumed) using 'POINT(1 1)'
- we ran an sql statement, but not on any table, just on data entered from the SQL prompt
- the resulting row looks kinda strange

The resulting row is looking strange because its in the OGC format called 'Well Known Binary' (WKB) - more on that coming in the next section.

To get the results back as text, I do a quick scan through the function list for something that returns text:

```
\df *text
```

One that catches my eye is 'astext'. Lets combine it with the previous:

```
select astext(st_pointfromtext('POINT(1 1)'));
```

Result:

```
astext
-----
POINT(1 1)
(1 row)
```

Again that is interesting - we round tripped our point!

One last thing I would like to show you in this little PostGIS taster before we really get into the detail of it:

```
select astext(st_buffer(st_pointfromtext('POINT(1 1)'),1.0));
```

What did that do? It created a buffer of 1 degree around our point. Nifty eh?

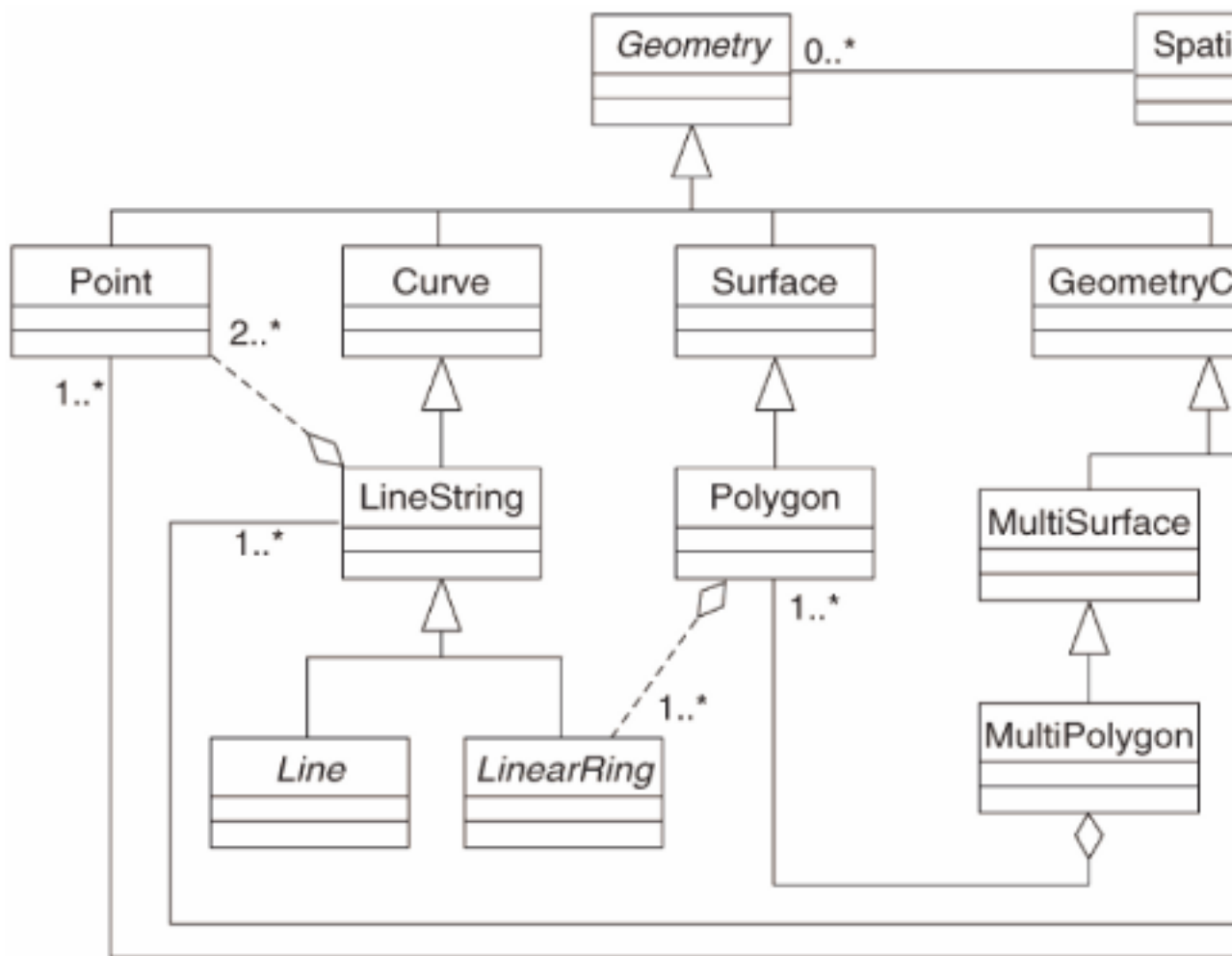
2.2 Simple Feature Model

2.2.1 What is OGC

"The Open Geospatial Consortium (OGC), an international voluntary consensus standards organization, originated in 1994. In the OGC, more than 370+ commercial, governmental, nonprofit and research organizations world-wide collaborate in an open consensus process encouraging development and implementation of standards for geospatial content and services, GIS data processing and data sharing." - Wikipedia

2.2.2 What is the SFS Model

The Simple Feature for SQL (SFS) Model is a non topological way to store geospatial data in a database and defines functions for accessing, operating, and constructing these data.



The model defines geospatial data from type Point, Linestring, Polygon and aggregations of them to Multi objects.

For further information have also a look at [OGC Simple Feature for SQL](#)

2.2.3 Add a geometry field to table

Lets add a point field to our people table:

```
alter table people add column the_geom geometry;
```

2.2.4 Add a constraint based on geometry type

You will notice that the geometry field type does not implicitly specify what "type" of geometry for the field - for that we need a constraint.

```
alter table people
add constraint people_geom_point_chk
    check(st_geometrytype(the_geom) = 'ST_Point'::text OR the_geom IS NULL);
```

What does that do? It adds a constraint to the table that prevents anything except a point geometry or a null.

Now you try:

Create a new table called cities and give it some appropriate columns, including a geometry field for storing polygons (the city boundaries). Make sure it has a constraint enforcing geometries to be polygons

Solution:

```
create table cities (id serial not null primary key,  
                    name varchar(50),  
                    the_geom geometry not null);  
alter table cities  
add constraint cities_geom_point_chk  
check (st_geometrytype(the_geom) = 'ST_Polygon'::text );
```

2.2.5 Populate geometry_columns table

At this point you should also add an entry into the 'geometry_columns' table:

```
insert into geometry_columns values  
(',', 'public', 'people', 'the_geom', 2, 4326, 'POINT');
```

Why? 'geometry_columns' is used by certain applications to be aware of which tables in the database contain geometry data.

Add an appropriate geometry_columns entry for your new cities layer

Solution:

```
insert into geometry_columns values
('','public','cities','the_geom',2,4326,'POLYGON');
```

2.2.6 Add geometry record to table using SQL

Now that our tables are geo-enabled, we can store geometries in them!

```
insert into people (name,house_no, street_id, phone_no, the_geom)
values ('Fault Towers',
       34,
       3,
       '072 812 31 28',
       st_makepoint(33,-33);
```

Note: Now is probably a good time to fire up QGIS and try to view your people table. Also try editing / adding / deleting records and then performing select queries in the database to see how the data has changed.

Formulate a query that shows a person's name, street name and position (from the the_geom column) as plain text.

2.2.7 View a point as WKT

Lets take a look at the solution to the above query:

```
select people.name,  
       streets.name as street_name,  
       astext(people.the_geom) as geometry  
from   streets, people  
where  people.street_id=streets.id;
```

Result:

name	street_name	geometry
Rusty Bedsprings	High street	
QGIS Geek	High street	
Joe Bloggs	New Main Street	
IP Knightly	QGIS Road	
Fault Towers	QGIS Road	POINT(33 -33)

(5 rows)

As you can see, our constraint allows nulls to be added into the database.

2.3 Spatial Queries

Spatial queries are not different to other database queries. You can use the geometry column as every other database column. With the installation of PostGIS in our database we have additional functions to query our database.

2.3.1 Spatial operators

When you want to know which points are within a distance of 2 degrees to a point(X,Y) you can do this with:

```
select *
from people
where st_distance(the_geom,st_makepoint(32,-34)) < 2;
```

You will notice that `st_makepoint` provides another, different and convenient way to create a point geometry.

Result:

```
id | name | house_no | street_id | phone_no | the_geom
---+-----+-----+-----+-----+-----
 6 | Fault Towers | 34 | 3 | 072 812 31 28 | 01010008040C0
(1 row)
```

Note: the `the_geom` value above was truncated for space on this page.

2.3.2 Spatial Indexes

We also can define spatial indexes. A spatial index makes your spatial queries much faster. To create a spatial index on the geometry column use:

```
CREATE INDEX people_geo_idx
ON people
USING gist
(the_geom);
```

Result:

```
address=# \d people
Table "public.people"
  Column | Type | Modifiers
-----+-----+-----
 id      | integer | not null default nextval('people_id_seq'::reg
 name    | character varying(50) |
 house_no | integer | not null
 street_id | integer | not null
 phone_no | character varying |
 the_geom | geometry |
```

Indexes:

```
"people_pkey" PRIMARY KEY, btree (id)
"people_geo_idx" gist (the_geom) <-- new spatial key added
```

```
"people_name_idx" btree (name)
Check constraints:
"people_geom_point_chk" CHECK (st_geometrytype(the_geom) = 'ST_Point'::text OR th
Foreign-key constraints:
"people_street_id_fkey" FOREIGN KEY (street_id) REFERENCES streets(id)
```

Now you try - modify the cities table so its geometry column is spatially indexed.

Solution:

```
CREATE INDEX cities_geo_idx
  ON cities
  USING gist (the_geom);
```

2.4 Geometry Construction

In this section we are going to delve a little deeper into how simple geometries are constructed in SQL. In reality you will probably use a GIS like QGIS to create complex geometries using their digitising tools, however understanding how they are formulated can be handy for writing queries and understanding how the database is assembled.

2.4.1 Creating Linestrings

Before we start, lets get our streets table matching the others i.e. having a constraint on the geometry, an index and an entry in the geometry_columns table.

Exercise:

- Modify the streets table so that it has a geometry column of type LINESTRING.
- Don't forget to do the accompanying update to the geometry_columns table!
- Also add a constraint to prevent any geometries being added that are not LINESTRINGS or null.
- Create a spatial index on the new geometry column

Solution:

```
alter table streets add column the_geom geometry;
alter table streets add constraint streets_geom_point_chk check
    (st_geometrytype(the_geom) = 'ST_Linestring'::text OR the_geom IS NULL);
insert into geometry_columns values ('', 'public', 'streets', 'the_geom', 2, 4326, 'LINESTRING');
create index streets_geo_idx
    on streets
    using gist
    (the_geom);
```

Now lets insert a linestring into our streets table. In this case I am going to update an existing street record:

```
update streets set the_geom = st_linefromtext('LINESTRING(20 -33, 21 -34, 24 -33)')
where streets.id=2;
```

Take a look at the results in QGIS. Now create some more streets entries - some in QGIS and some from the command line.

2.4.2 Creating Polygons

Creating polygons is just as easy. One thing to remember is that by definition, polygons have at least four vertices, with the last and first being colocated.

2.4.3 Exercise - Linking Cities to People

For this exercise you should do the following:

Delete all data from your people table.
 Add a foreign key column to people that references the primary key of the cities table.
 Use QGIS to capture some cities.
 Use SQL to insert some new people records, ensuring that each has an associated street and city.

Your updated people schema should look something like this:

```
\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 |
  the_geom | geometry |
  city_id | integer | not null
Indexes:
  "people_pkey" PRIMARY KEY, btree (id)
  "people_name_idx" btree (name)
Check constraints:
  "people_geom_point_chk" CHECK (st_geometrytype(the_geom) =
                                'ST_Point'::text OR the_geom IS NULL)
Foreign-key constraints:
  "people_city_id_fkey" FOREIGN KEY (city_id) REFERENCES cities(id)
  "people_street_id_fkey" FOREIGN KEY (street_id) REFERENCES streets(id)
```

Solution:

```
delete from people;
alter table people add column city_id int not null references cities(id);

(capture cities in QGIS)

insert into people (name,house_no, street_id, phone_no, city_id, the_geom)
  values ('Faulty Towers',
         34,
         3,
         '072 812 31 28',
         1,
         st_makepoint(33,-33);

insert into people (name,house_no, street_id, phone_no, city_id, the_geom)
  values ('IP Knightly',
         32,
         1,
         '071 812 31 28',
         1,
         st_makepoint(32,-34);

insert into people (name,house_no, street_id, phone_no, city_id, the_geom)
  values ('Rusty Bedsprings',
```

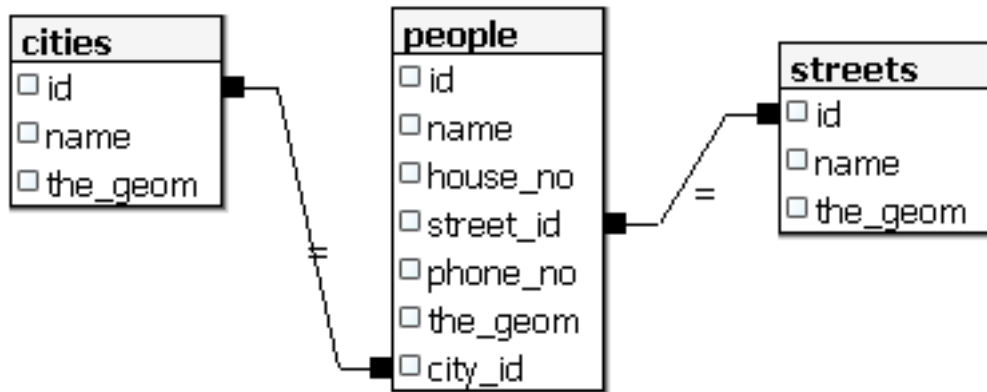
```

39,
1,
'071 822 31 28',
1,
st_makepoints(34,-34);

```

2.4.4 Looking at our schema

By now our schema should be looking like this:



2.4.5 Exercise

Create city boundaries by computing the minimum convex hull of all addresses for that city and computing a buffer around that area.

2.4.6 Access Subobjects

With the SFS-Model functions you have a wide variety of options to access subobjects of SFS Geometries. When you want to select the first vertex point of every polygon geometry in the table myPolygonTable, you have to do this in this way:

- Transform the polygon boundary to a linestring:

```
select st_boundary(geometry) from myPolygonTable
```

- select the first vertex point of the resultant linestring:

```
select st_startpoint(myGeometry)
from (
  select st_boundary(geometry) as myGeometry
  from myPolygonTable) as foo
```

2.5 Data Processing

PostGIS supports all OGC SFS/MM standard conform functions. All these functions are starting with ST_.

2.5.1 Clipping

To clip a subpart of your data you can use the ST_INTERSECT() function. To avoid empty geometries use `where not st_isempty(st_intersection(a.the_geom, b.the_geom))`



```
select st_intersection(a.the_geom, b.the_geom), b.*  
from clip as a, road_lines as b  
where not st_isempty(st_intersection(st_setsrid(a.the_geom,32734), b.the_geom));
```



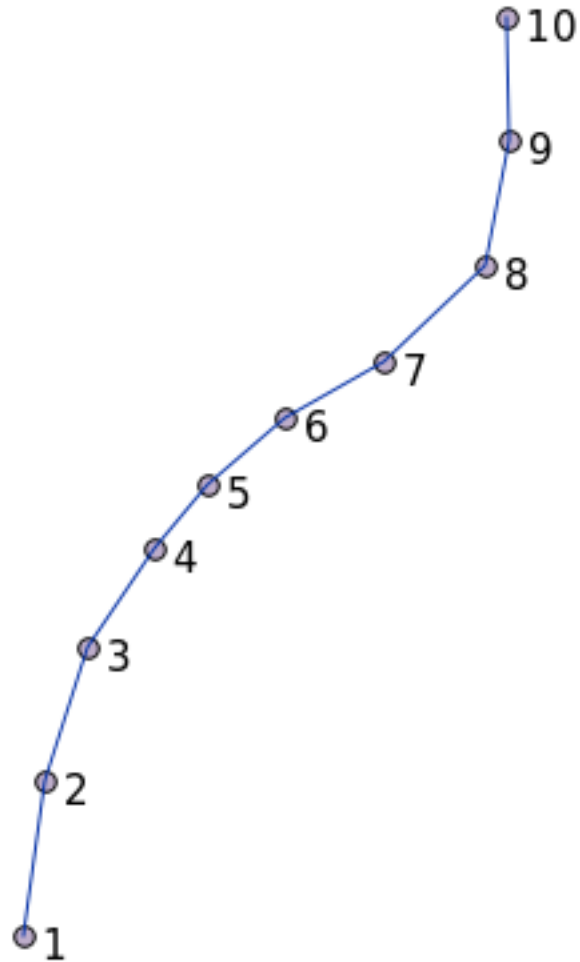
2.5.2 Building Geometries from Other Geometries

From a given point table you want to generate a linestring. The order of the points is defined by their id. Another ordering method could be a timestamp like you get when you capture waypoints with GPS-Receivers.



To create a linestring you can run the following command:

```
select ST_LineFromMultiPoint(st_collect(the_geom)), 1 as id
from (
  select the_geom
  from points
  order by id
) as foo
```



2.5.3 Geometry Cleaning

You can get more information for this topic at Tim's wonderful blog article you can find here:

<http://linfiniti.com/?s=cleangeometry>

2.5.4 Differences between tables

To detect the difference between two tables with the same structure you can use the PostgreSQL keyword `EXCEPT`.

```
select * from table_a
except
select * from table_b;
```

As the result you will get all records from table.a which are not stored in table.b.

2.6 Import and Export

Of course a database with no easy way to migrate data into it and out of it would be no fun. More so for your spatial data! Fortunately there are a number of tools that will let you easily move data into and out of PostGIS.

2.6.1 shp2pgsql

shp2pgsql is a commandline tool to import ESRI shapefiles to the database. Under Unix you can use the following command for importing a new PostGIS table:

```
shp2pgsql -s <SRID> -c -D -I <path to shapefile> <schema>.<table> | \
psql -d <databasename> -h <hostname> -U <username>
```

under Windows you have to perform the import process in two steps:

```
shp2pgsql -s <SRID> -c -D -I <path to shapefile> <schema>.<table> > import.sql
psql psql -d <databasename> -h <hostname> -U <username> -f import.sql
```

2.6.2 pgsq2shp

pgsq2shp is a commandline tool to export PostGIS Tables, Views or SQL select queries. To do this under Unix:

```
pgsq2shp -f <path to new shapefile> -g <geometry column name> \
-h <hostname> -U <username> <databasename> <table | view>
pgsq2shp -f <path to new shapefile> -g <geometry column name> \
-h <hostname> -U <username> "<query>"
```

2.6.3 ogr2ogr

ogr2ogr is a very powerful tool to convert data into and from postgis to many dataformats. ogr2ogr is part of the GDAL/OGR Software and has to be installed seperately. To export a table from PostGIS to GML you can use this command:

```
ogr2ogr -f GML export.gml PG:'dbname=<databasename> user=<username> host=<hostname>
```

2.6.4 spit

SPIT is a QGIS plugin, which is delivered with QGIS. You can use SPIT for uploading ESRI shapefiles to postgis.