INTRODUCTION TO POSTGIS

Installation - Tutorial - Exercises

09 May 2015

Refractions RESEARCH

THE GEOSPATIAL EXPERTS

Introduction to PostGIS

Installation - Tutorial - Exercises

Prepared By

Paul Ramsey

Refractions Research

Suite 300 – 1207 Douglas Street

Victoria – British Columbia

CANADA – V8W 2E7

Phone: (250) 383-3022

Fax: (250) 383-2140

http://www.refractions.net

Document Tracking

/tmp/PostGIS Workshop.doc

Created: May 2, 2007

Printed: September 13, 2007

Contents

1 INTRODUCTION

PostGIS is a spatial database add-on for the PostgreSQL relational database server. It includes support for all of the functions and objects defined in the OpenGIS "Simple Features for SQL" specification. Using the many spatial functions in PostGIS, it is possible to do advanced spatial processing and querying entirely at the SQL command-line.

This workshop will cover:

- Installation and setup of PostgreSQL,
- Installation of the PostGIS extension,
- Loading sample data,
- Spatial and attribute indexing,
- Performance tuning the database,
- Basic spatial SQL, and
- Some best practices for spatial SQL.

1.1 REQUIREMENTS & SET-UP

This workshop will use the latest Windows native PostgreSQL installer, the latest PostGIS installer, and a data package of shape files – all of these components are included on the workshop CDROM.

Copy the `\postgis-workshop` directory from the CDROM to your `C:` drive. Highlight the `\postgis-workshop` CDROM directory, and hit `CTRL-C`, then move to your `C:` drive, and hit `CTRL-V`.

1.2 SPATIAL DATABASES

Like Oracle Spatial, DB2 Spatial, and SQL Server Spatial, PostGIS adds capabilities to an existing relational database engine, in this case PostgreSQL. In fact, PostGIS could be re-named as "PostgreSQL Spatial", as it functions in the same way as the proprietary spatial database extensions:

- It adds a "geometry" data type to the usual database types (e.g. "varchar", "char", "integer", "date", etc).
- It adds new functions that take in the "geometry" type and provide useful information back (e.g. ST_Distance(geometry, geometry), ST_Area(geometry), ST_Length(geometry), ST_Intersects(geometry, geometry), etc).
- It adds an indexing mechanism to allow queries with spatial restrictions ("within this bounding box") to return records very quickly from large data tables.

The core functionalities of a spatial database are easy to list: types, functions, and indexes. What is impressive is how much spatial processing can be done inside the database once those simple capabilities are present: overlay analyses, re-projections, massive seamless spatial tables, proximity searches, compound spatial/attribute filters, and much more.

1.3 CONVENTIONS

Session instructions that require user-interaction are presented in this document inside of grey boxes – the text to be entered is in **boldface**, and the results are in normal text. **In general, directions to be performed will be presented in boldface.**

1.4 OTHER DOWNLOADS

The PostgreSQL source code is available from: http://www.postgresql.org/.

The PostGIS source code is available from: http://postgis.refractions.net/.

The GEOS source code is available from: http://geos.refractions.net/.

The Proj4 source code is available from: http://proj.maptools.org/.

The PgAdmin administration tool is available from: http://www.pgadmin.org.

2 DATABASE INSTALLATION

2.1 POSTGRESQL INSTALLATION

The PostgreSQL installation package is on the CDROM in:

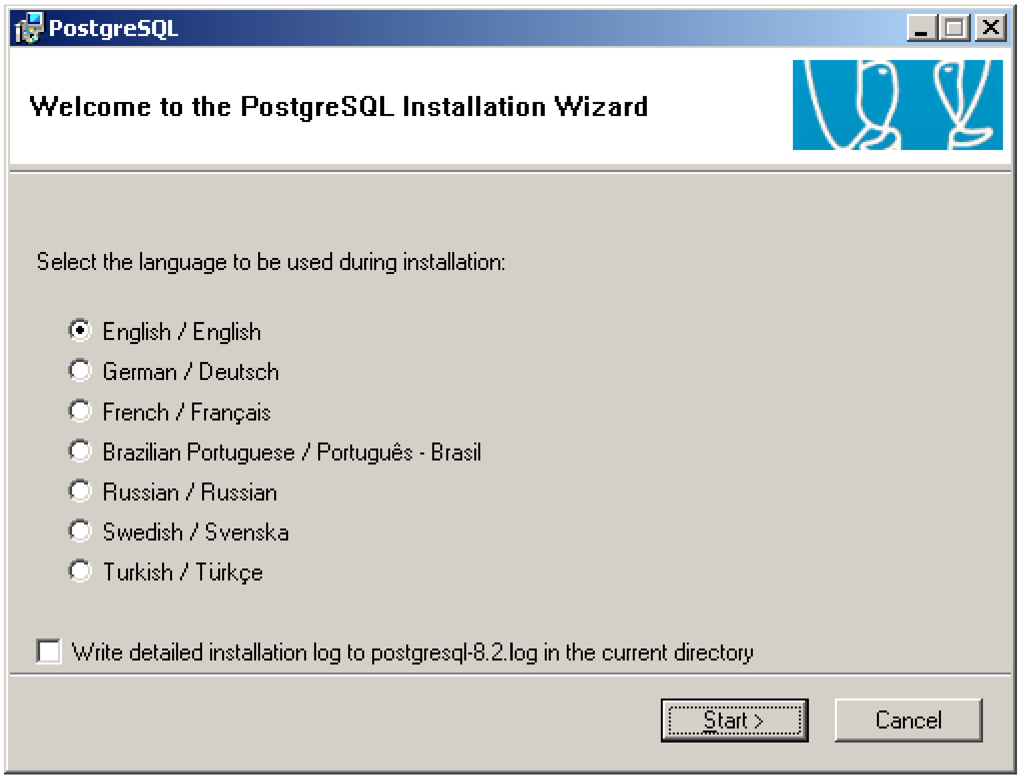
```
\postgis-workshop\software\postgresql-8.2.msi
```

Further information on installing PostgreSQL can be found on the PostgreSQL website:

http://www.postgresql.org/docs/

Steps:

1. **Double-click the file *`postgresql-8.2.msi`***
2. **Select your language:**



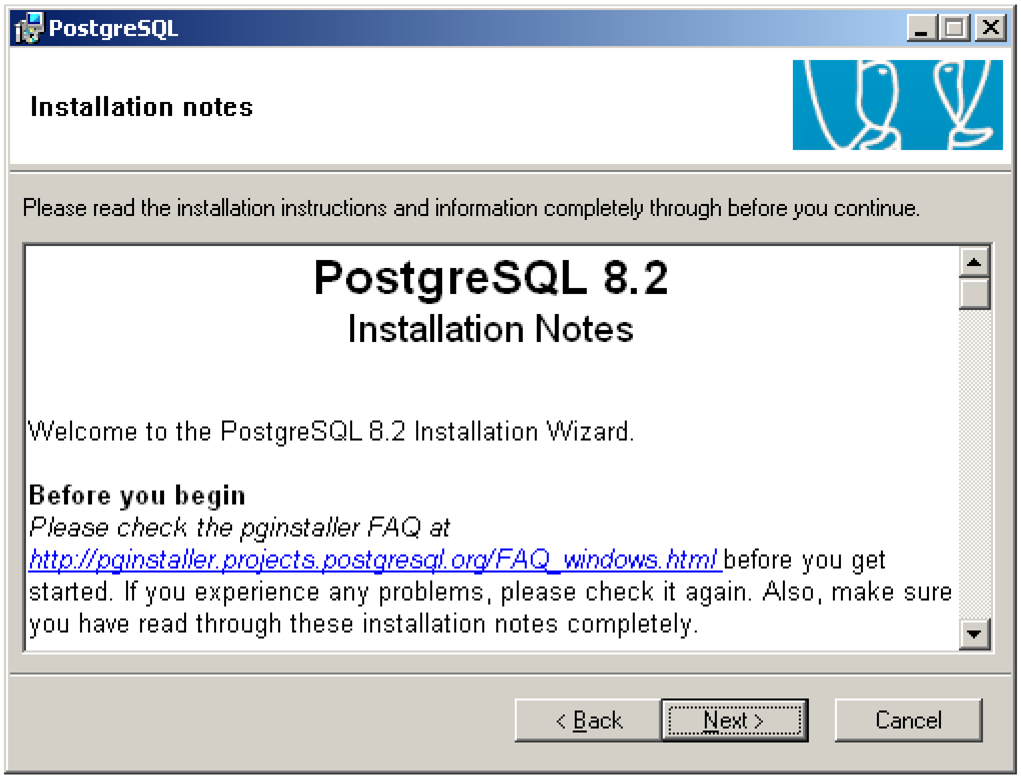
Welcome to the PostgreSQL Installation Wizard

Select the language to be used during installation:

- [✓ English / English]
- German / Deutsch
- French / Français
- Brazilian Portuguese / Português - Brasil
- Russian / Russian
- Swedish / Svenska
- Turkish / Türkçe

☐ Write detailed installation log to postgresql-8.2.log in the current directory

3. The installer will start up, recommend that you close all Windows programs, and then display the license agreement.



Installation notes

Please read the installation instructions and information completely through before you continue.

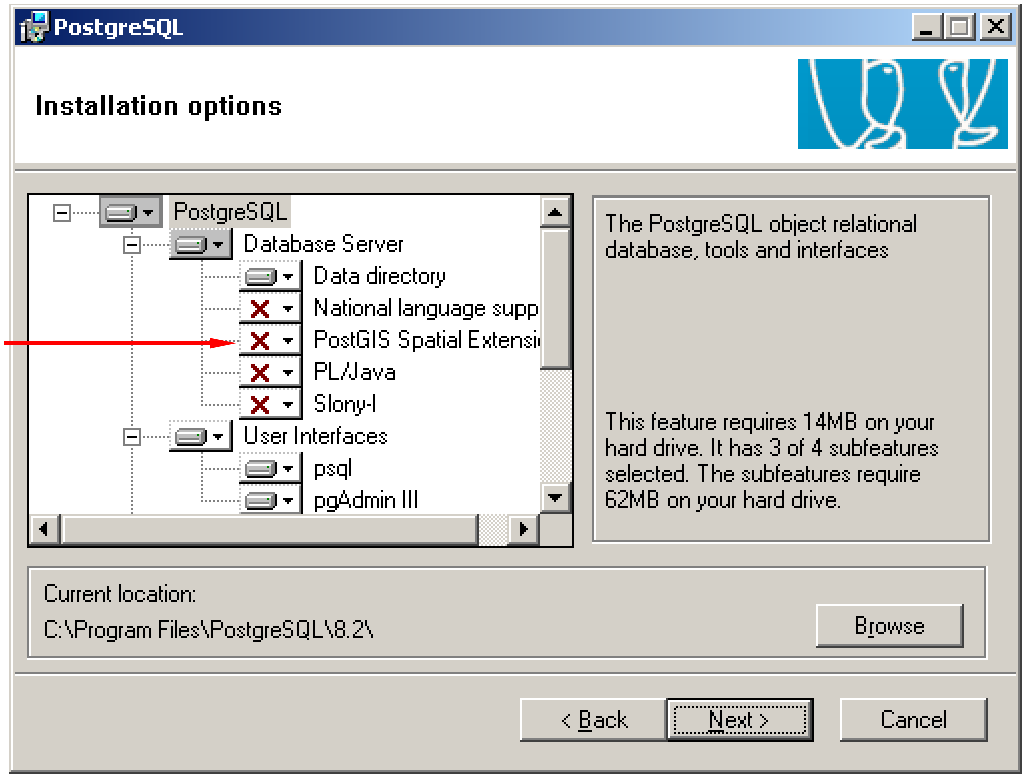
PostgreSQL 8.2
Installation Notes

Welcome to the PostgreSQL 8.2 Installation Wizard.

Before you begin
Please check the pginstaller FAQ at *http://pginstaller.projects.postgresql.org/FAQ_windows.html* before you get started. If you experience any problems, please check it again. Also, make sure you have read through these installation notes completely.

4. When the “Installation options” selector comes up, **do not enable** installation of the PostGIS extension.

The PostGIS Spatial Extension included in the windows installer is often a few versions behind the current stable release. We will install a current PostGIS version separately after completing the PostgreSQL install.


PostgreSQL

Installation options

PostgreSQL
Database Server
Data directory
National language supp
PostGIS Spatial Extensi
PL/Java
Slony-I
User Interfaces
psql
pgAdmin III

The PostgreSQL object relational database, tools and interfaces

This feature requires 14MB on your hard drive. It has 3 of 4 subfeatures selected. The subfeatures require 62MB on your hard drive.

Current location:
C:\Program Files\PostgreSQL\8.2\

5. **Install PostgreSQL as a service and select the account to run the service under.**

> Installing PostgreSQL as a service installs it as a windows service so it starts up automatically when your machine is restarted.

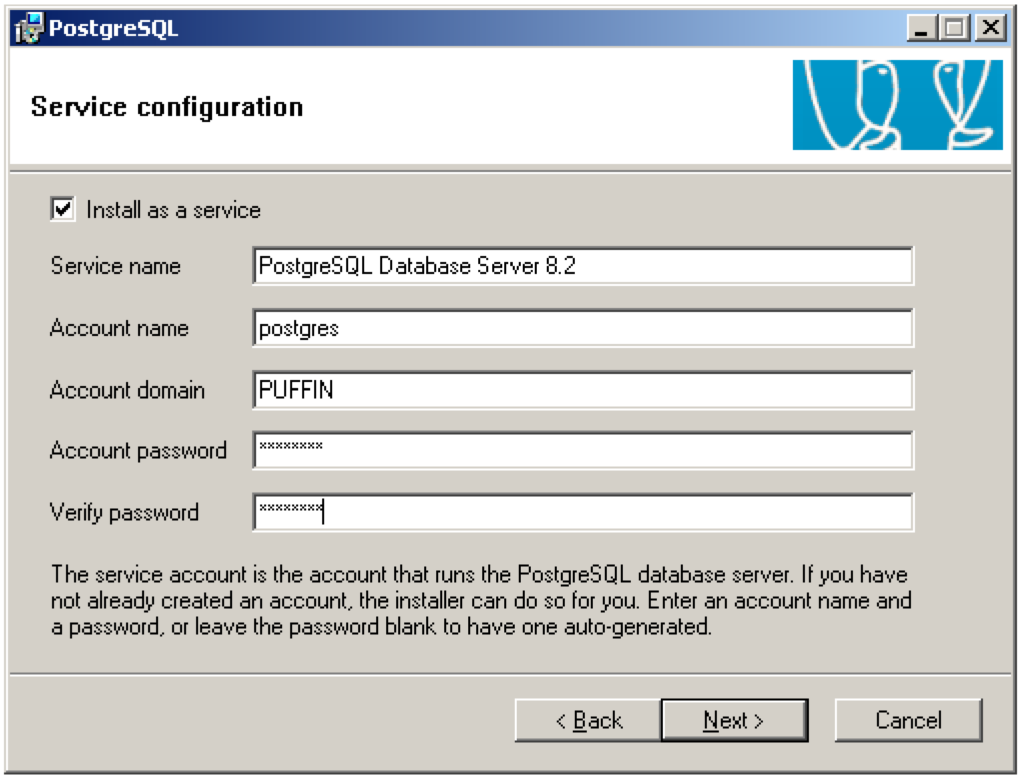
- **Service name** – Name associated with PostgreSQL service.
- **Account name** – User name to run the service under. The default *postgres* is standard. You can enter an existing account or a new one. If you enter a new account the installer will automatically create an account for you. If you enter an existing account the account must not be an administrator account.

> Administrator accounts are not allowed for security reasons. If a hacker were to gain entry to the computer using PostgreSQL they would gain the permissions of account name the service is run under. Ensuring this is a non-administrator account limits the potential harm they can inflict on the system.

- **Account Domain** – This should be the name of your computer.

> If your windows system is setup for domain authentication you will need to enter the domain here. However for our purposes the computer name will suffice.

- **Account password / Verify password** – Password associated with the account name (entered twice for verification).

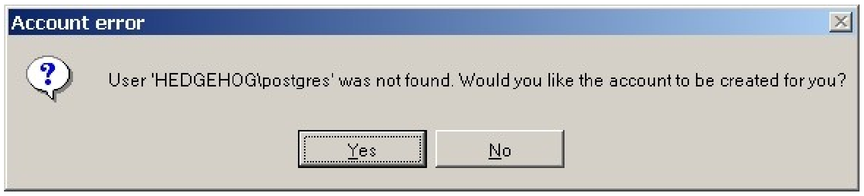


☑ Install as a service

Service name	PostgreSQL Database Server 8.2
Account name	postgres
Account domain	PUFFIN
Account password	********
Verify password	********

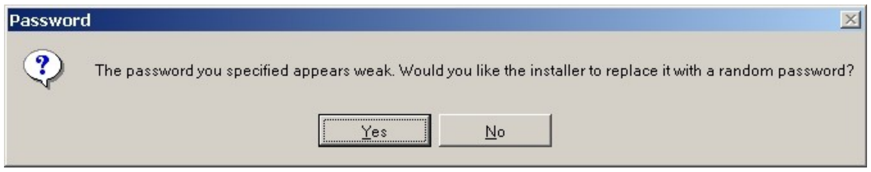
The service account is the account that runs the PostgreSQL database server. If you have not already created an account, the installer can do so for you. Enter an account name and a password, or leave the password blank to have one auto-generated.

- If the user account does not already exist on the computer you will be prompted to create the account.



User 'HEDGEHOG\postgres' was not found. Would you like the account to be created for you?

- Depending on the "strength" of the password you entered the following message might appear – Click '*No*'.



The password you specified appears weak. Would you like the installer to replace it with a random password?

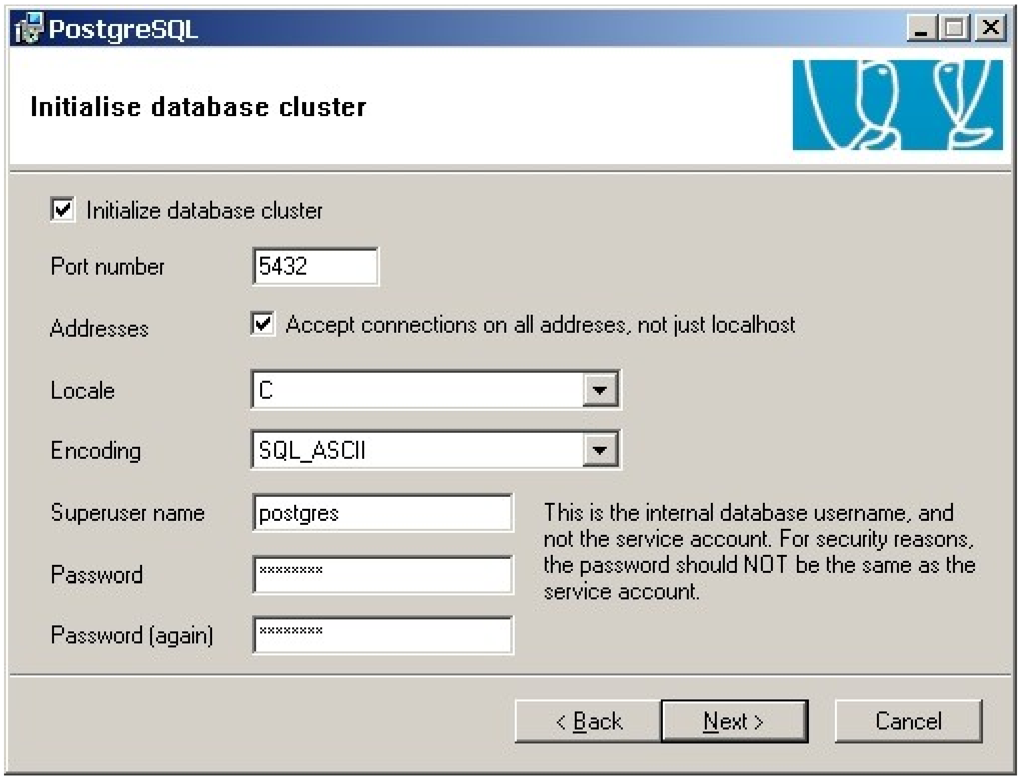
6. **Initialize database cluster.** This sets up a database cluster with the given connection information.

A database cluster is a group of databases managed by a single PostgreSQL server instance.

- **Port Number:** Port Number the database cluster is to reside on.
- **Addresses:** Check the 'Accept connection on all addresses, not just localhost' if you want other machines to be able to access the database from any other machine. Even with this selected you will still have to specifically indicate which hosts or networks you will accept connections from by editing a configuration file. This is covered in the administrative section of the workshop.
- **Locale:** Select a locale. This will become the default locale of all databases. The *C* and *POSIX* locales will cause the system to behave as if it has no locale support. For the purposes of this demonstration we will use the *C* locale.
- **Encoding:** Character encoding to use. For this demonstration we will use *SQL_ASCII*.
- **Superuser Name:** This username is used to connect to the database. We will use *postgres*, although it is not recommended that you use the same name/password as the service account.

This username/password is not associated at all with the service account name/password. The service account name and password is the account the PostgreSQL process is executed within and resides as an operating system user. The Superuser Name/Password is used to connect to the database and perform operations on the database.

- **Password / Password(again):** The password associated with the superuser name. Again we will use *postgres*.


PostgreSQL

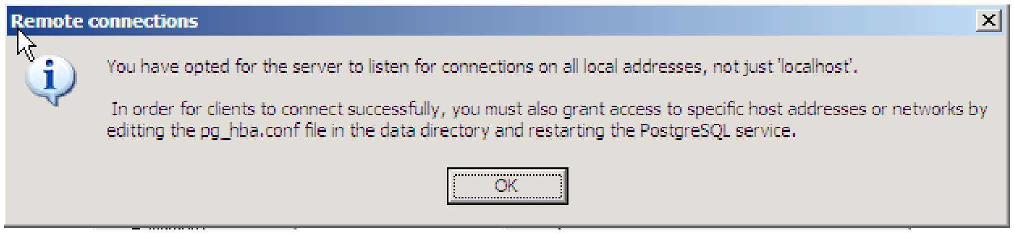
Initialise database cluster

☑ Initialize database cluster

Port number	5432
Addresses	☑ Accept connections on all addreses, not just localhost
Locale	C
Encoding	SQL_ASCII
Superuser name	postgres
Password	********
Password (again)	********

This is the internal database username, and not the service account. For security reasons, the password should NOT be the same as the service account.

- On selecting *'Next'* you will be presented with the following information dialog.

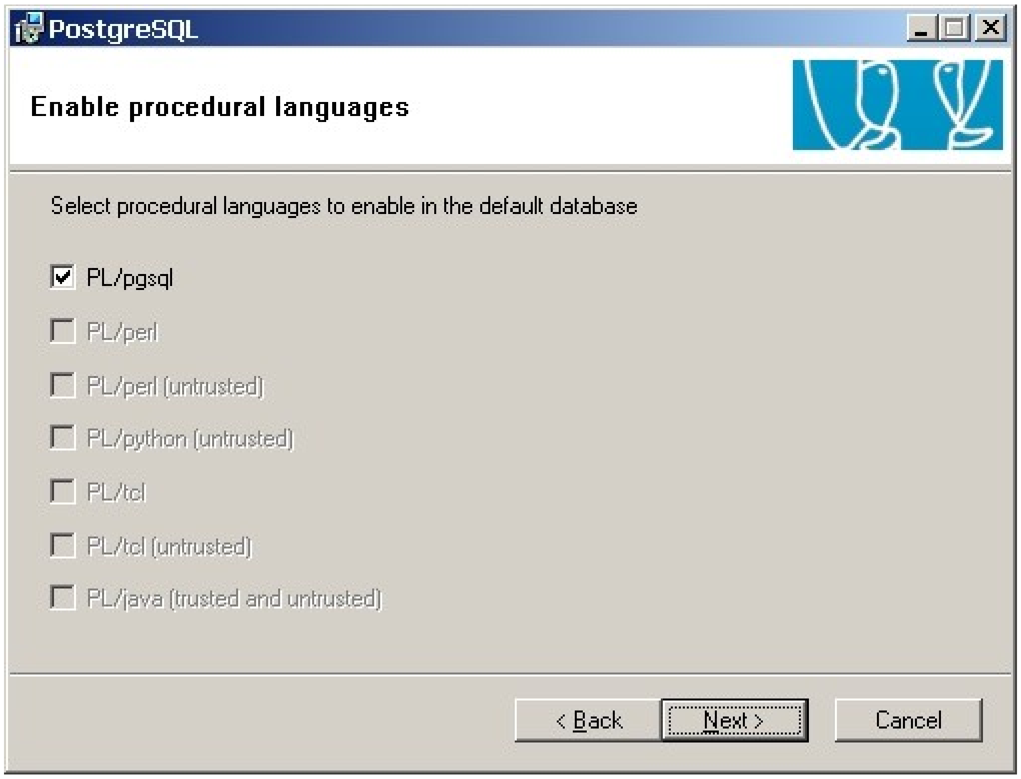

Remote connections

You have opted for the server to listen for connections on all local addresses, not just 'localhost'.

In order for clients to connect successfully, you must also grant access to specific host addresses or networks by editting the pg_hba.conf file in the data directory and restarting the PostgreSQL service.

7. The PL/PgSQL language is required for PostGIS, so ensure it is selected.

PL/Perl is handy for Perl programmers who want to write triggers and functions in Perl, but not required by PostGIS.



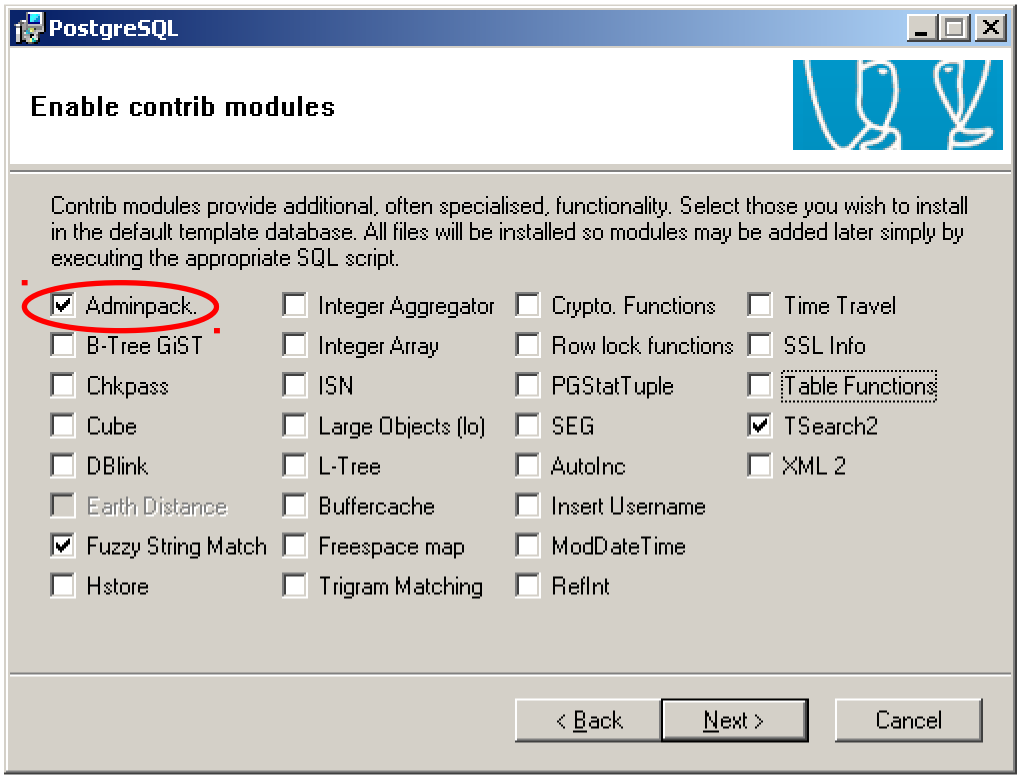
PostgreSQL

Enable procedural languages

Select procedural languages to enable in the default database

- [✓] PL/pgsql
- [ ] PL/perl
- [ ] PL/perl (untrusted)
- [ ] PL/python (untrusted)
- [ ] PL/tcl
- [ ] PL/tcl (untrusted)
- [ ] PL/java (trusted and untrusted)

8. This form allows you to select additional modules to be installed. **Ensure Adminpack is selected.** The Fuzzy String Match (soundex, etc), and Tsearch2 (full text indexing) modules might come in handy if you work with PostgreSQL enough to become a power user.

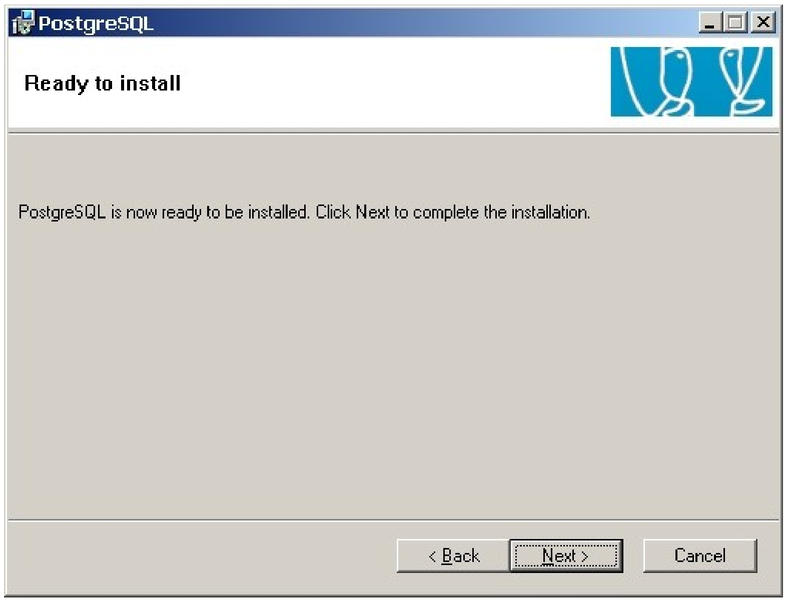


Enable contrib modules

Contrib modules provide additional, often specialised, functionality. Select those you wish to install in the default template database. All files will be installed so modules may be added later simply by executing the appropriate SQL script.

✓ Adminpack.	☐ Integer Aggregator	☐ Crypto. Functions	☐ Time Travel
☐ B-Tree GiST	☐ Integer Array	☐ Row lock functions	☐ SSL Info
☐ Chkpass	☐ ISN	☐ PGStatTuple	☐ Table Functions
☐ Cube	☐ Large Objects (lo)	☐ SEG	✓ TSearch2
☐ DBlink	☐ L-Tree	☐ AutoInc	☐ XML 2
☐ Earth Distance	☐ Buffercache	☐ Insert Username	
✓ Fuzzy String Match	☐ Freespace map	☐ ModDateTime	
☐ Hstore	☐ Trigram Matching	☐ RefInt	

9. That is it for the tough decisions! **Click 'Next' to install PostgreSQL**.

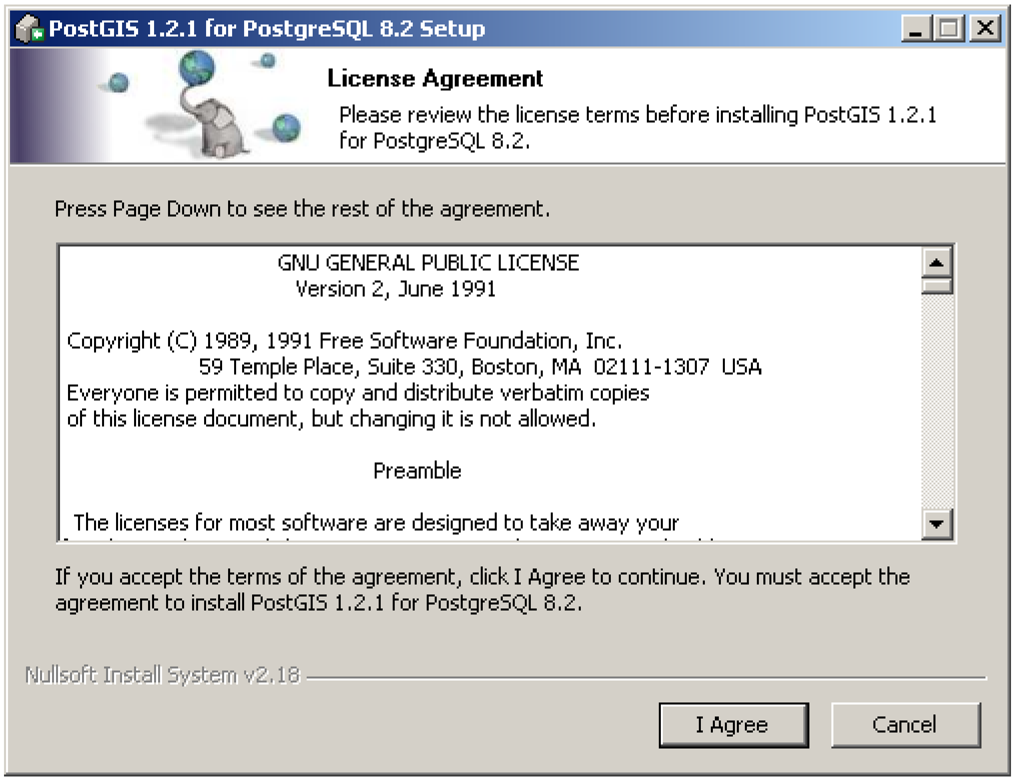

Ready to install

PostgreSQL is now ready to be installed. Click Next to complete the installation.

PostgreSQL is now installed on your computer!

2.2 POSTGIS INSTALLATION

1. **Double-click the `postgis-pg82-setup-1.3.1.exe` file found on the CD at `\postgis-workshop\software\` .**


PostGIS 1.2.1 for PostgreSQL 8.2 Setup

License Agreement

Please review the license terms before installing PostGIS 1.2.1 for PostgreSQL 8.2.

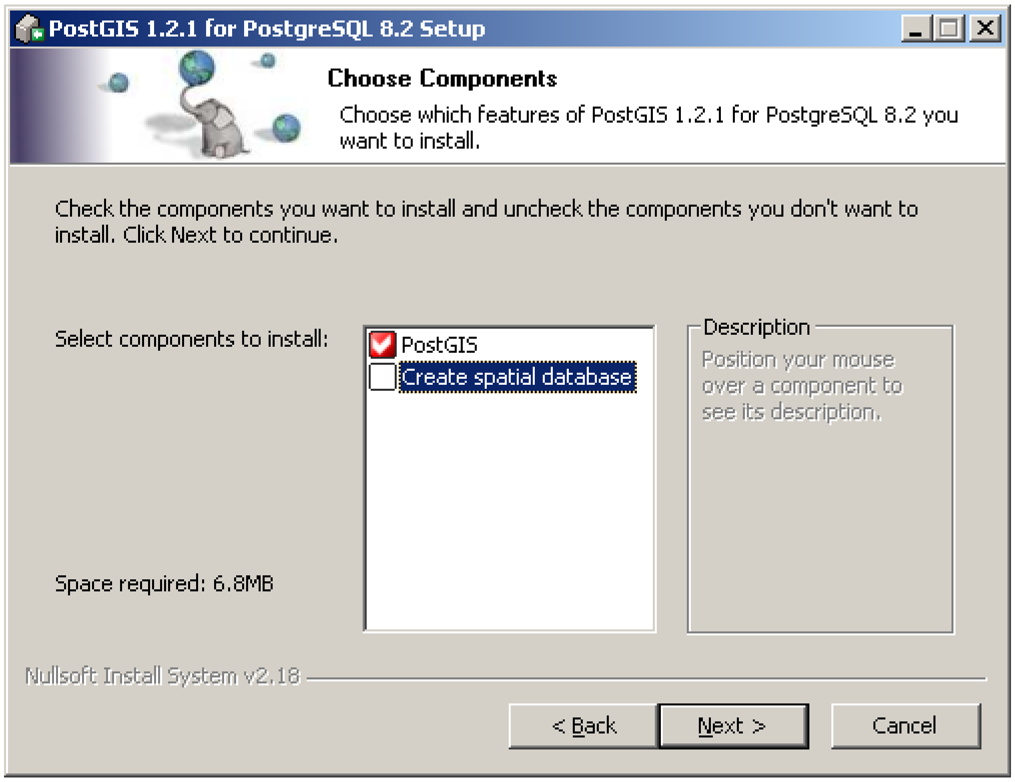
Press Page Down to see the rest of the agreement.

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
                59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

                            Preamble

 The licenses for most software are designed to take away your
```

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install PostGIS 1.2.1 for PostgreSQL 8.2.

2. **De-select the "*Create Database*" option.**

The *Create Database* option will create a new database with PostGIS installed to the database. Because you will often want to create PostGIS enabled databases without running the installer we will show how to create a PostGIS enabled database manually.



PostGIS 1.2.1 for PostgreSQL 8.2 Setup

Choose Components

Choose which features of PostGIS 1.2.1 for PostgreSQL 8.2 you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

- [✓] PostGIS
- [ ] Create spatial database

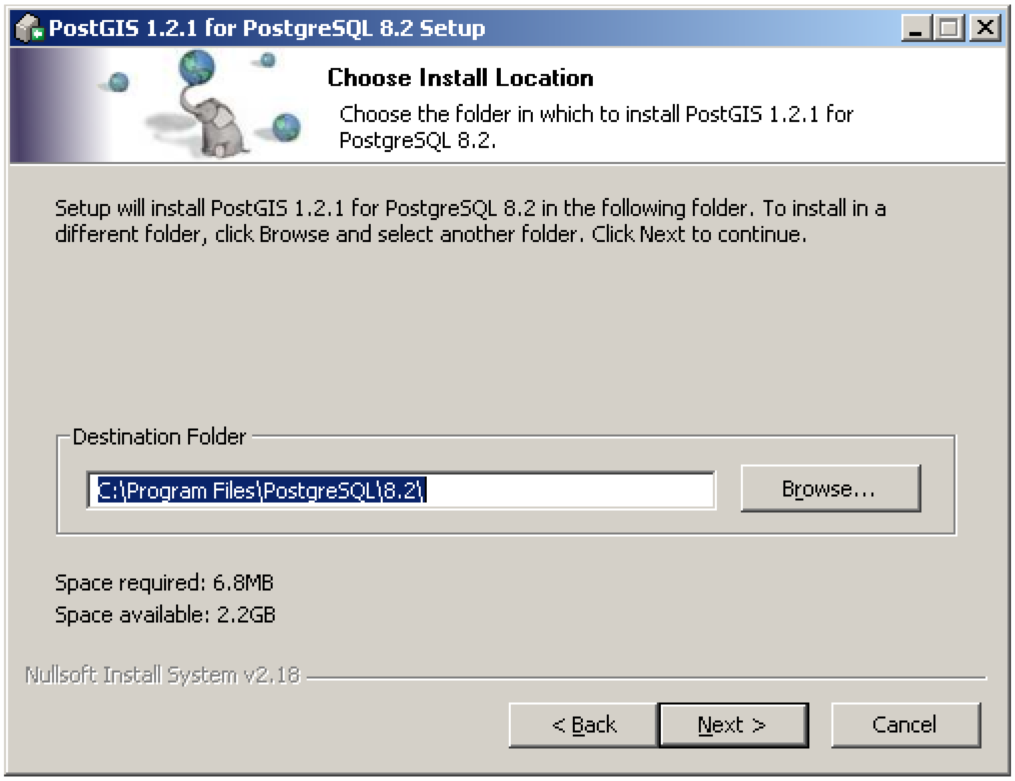
Description

Position your mouse over a component to see its description.

Space required: 6.8MB

Nullsoft Install System v2.18

3. **Accept the default install location.**



PostGIS 1.2.1 for PostgreSQL 8.2 Setup

Choose Install Location

Choose the folder in which to install PostGIS 1.2.1 for PostgreSQL 8.2.

Setup will install PostGIS 1.2.1 for PostgreSQL 8.2 in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder

C:\Program Files\PostgreSQL\8.2\

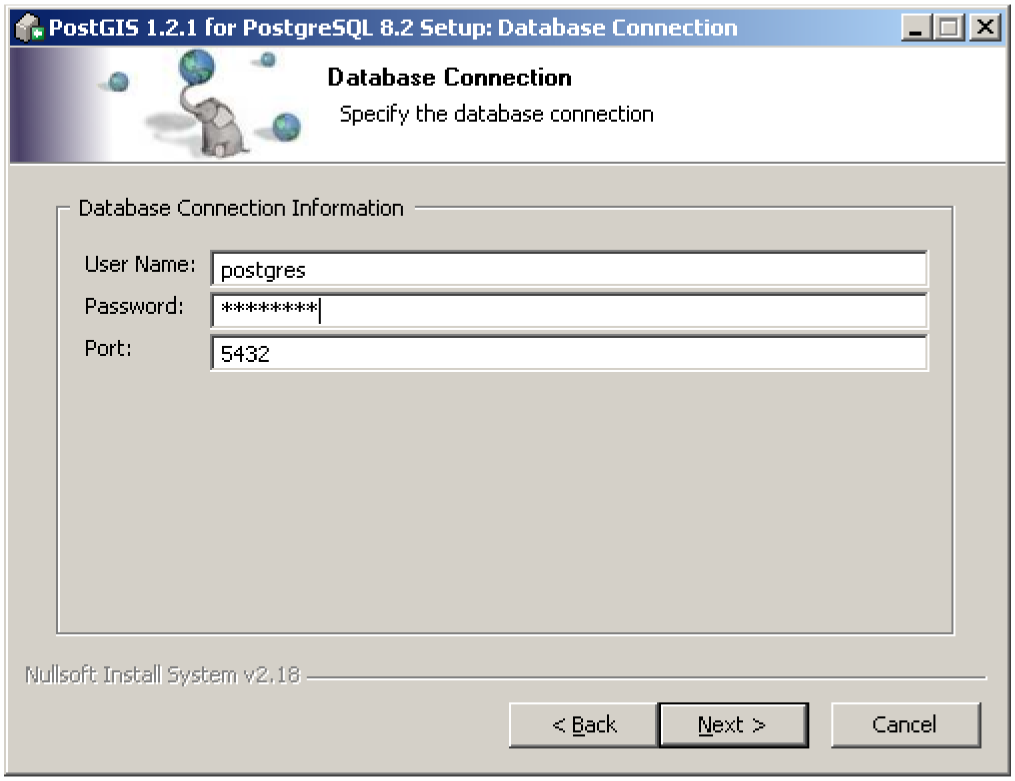
Space required: 6.8MB

Space available: 2.2GB

Nullsoft Install System v2.18

4. **Accept the default values.**

Because the "*Create Database*" option was not selected the information on this form has no effect on the PostGIS install. All these fields can be left blank. However if you selected the "*Create Database*" option you will need to enter the user name, password, database name, and port. If you do not enter valid values the installer will hang.


PostGIS 1.2.1 for PostgreSQL 8.2 Setup: Database Connection

Database Connection

Specify the database connection

Database Connection Information

User Name: postgres

Password: *********

Port: 5432

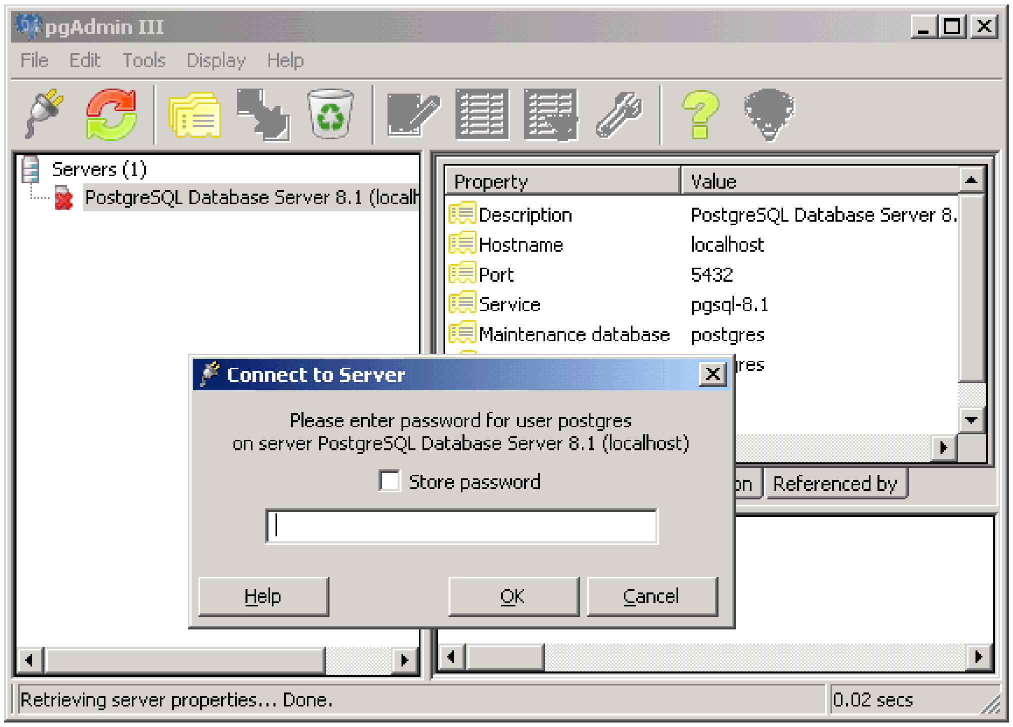
Click '*Install'* to complete the installation.

PostGIS is now installed on your computer.

2.3 SPATIALLY-ENABLE POSTGRESQL

The installer will add a PostgreSQL menu to your *Start* menu.

- **Navigate to the PostgreSQL menu item and run PgAdmin III.**
- **Double click on the "PostgreSQL Database Server" tree entry**. You will be prompted for the super user password to connect to the "template1" database.


pgAdmin III
File Edit Tools Display Help
Servers (1)
PostgreSQL Database Server 8.1 (localh
Property
Value
Referenced by
Help
OK
Cancel

Property	Value
Description	PostgreSQL Database Server 8.
Hostname	localhost
Port	5432
Service	pgsql-8.1
Maintenance database	postgres

Connect to Server

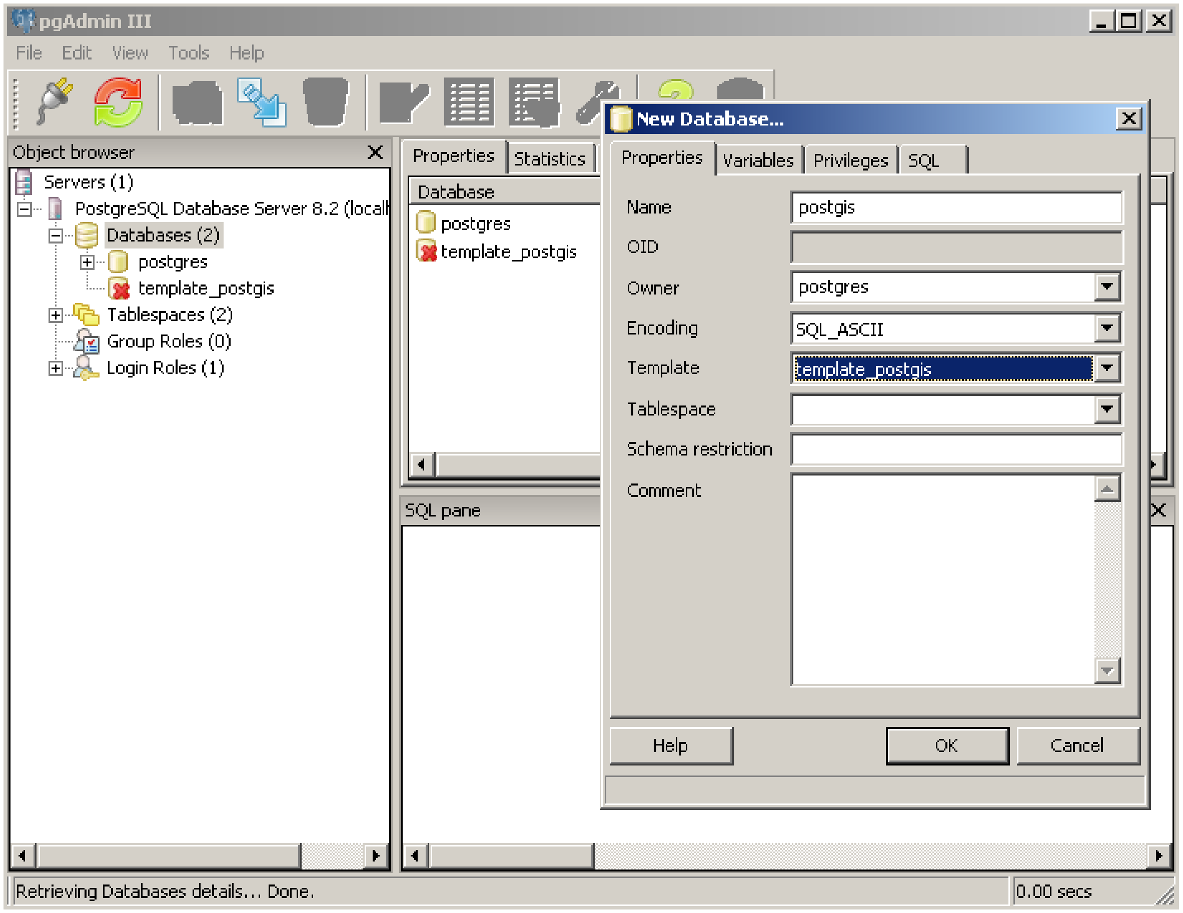
Please enter password for user postgres
on server PostgreSQL Database Server 8.1 (localhost)

☐ Store password

Retrieving server properties... Done.   0.02 secs

- **Navigate to the "Databases" section of the database tree and open "Edit ⇨ New Object ⇨ New Database". Add a new database named "postgis", with "postgres" as the owner "template_postgis" as the template.**

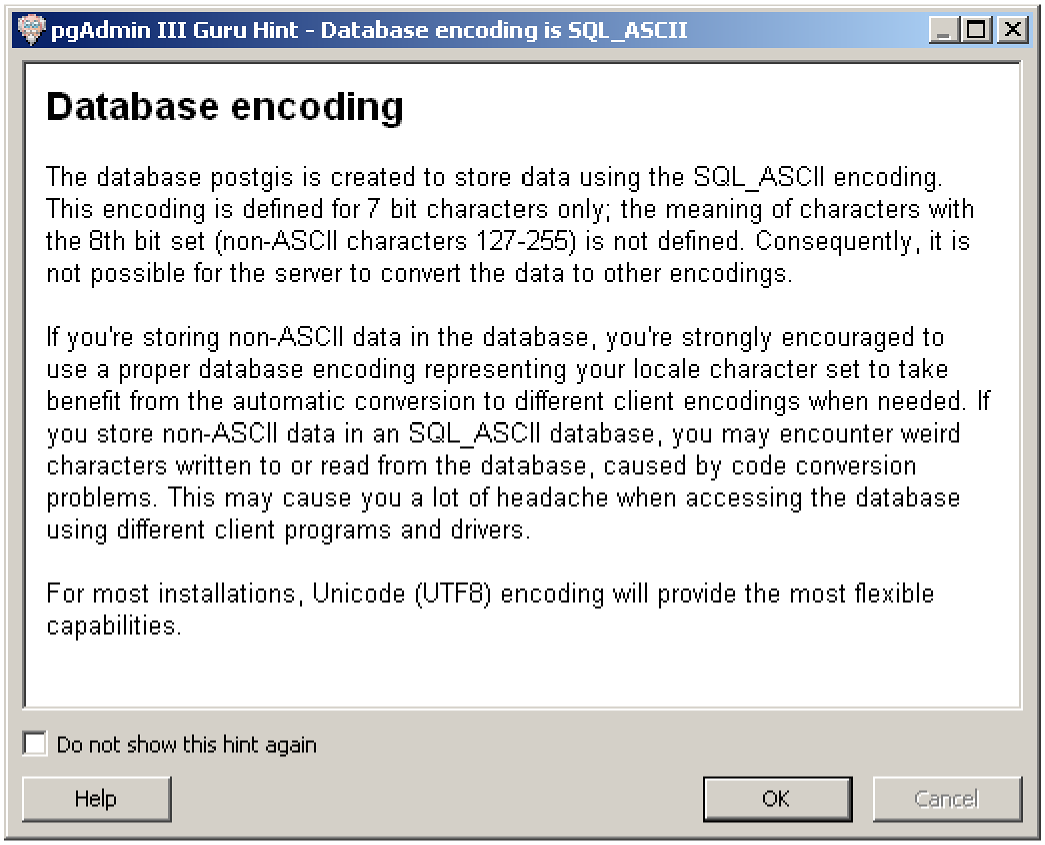
  By using the "template_postgis" database as the template, you get a new database with spatial capabilities already installed and enabled.



Name	postgis
OID	
Owner	postgres
Encoding	SQL_ASCII
Template	template_postgis
Tablespace	
Schema restriction	
Comment	

Note on table-spaces: PostgreSQL installs with two default table-spaces, pg_default and pg_global. The pg_global table-space is used for system tables and other objects that need cluster-wide visibility, **do not** use it for your working databases, use pg_default (which is what will be chosen if you do not specify a table-space explicitly).

- **Open up the new "postgis" database.**



pgAdmin III Guru Hint - Database encoding is SQL_ASCII

Database encoding

The database postgis is created to store data using the SQL_ASCII encoding. This encoding is defined for 7 bit characters only; the meaning of characters with the 8th bit set (non-ASCII characters 127-255) is not defined. Consequently, it is not possible for the server to convert the data to other encodings.

If you're storing non-ASCII data in the database, you're strongly encouraged to use a proper database encoding representing your locale character set to take benefit from the automatic conversion to different client encodings when needed. If you store non-ASCII data in an SQL_ASCII database, you may encounter weird characters written to or read from the database, caused by code conversion problems. This may cause you a lot of headache when accessing the database using different client programs and drivers.

For most installations, Unicode (UTF8) encoding will provide the most flexible capabilities.

Note on encodings: Encodings are used to consistently store "special characters", to associate a particular byte (201) with a particular character (É). Different encodings map the same bytes to different characters, so if you are using any special characters, it is important to be explicit about what encoding they are in.

The "SQL_ASCII" encoding is not actually an encoding at all, it is an "I do not know or care what the encoding is" encoding. If you use an explicit encoding, like "UTF-8", the database can transparently convert your data into different encodings for you. If you use "SQL_ASCII", you will have to do the translations yourself.

"UTF-8" is the recommended database encoding because it can handle every known character.

- **Navigate to postgis ⇨ Schemas ⇨ public ⇨ Tables and see what tables exist**. You should see “geometry_columns” and “spatial_ref_sys” tables, that are standard tables created by PostGIS. The tables starting in “pg_ts_” are used by the full-text-search module and can be ignored.

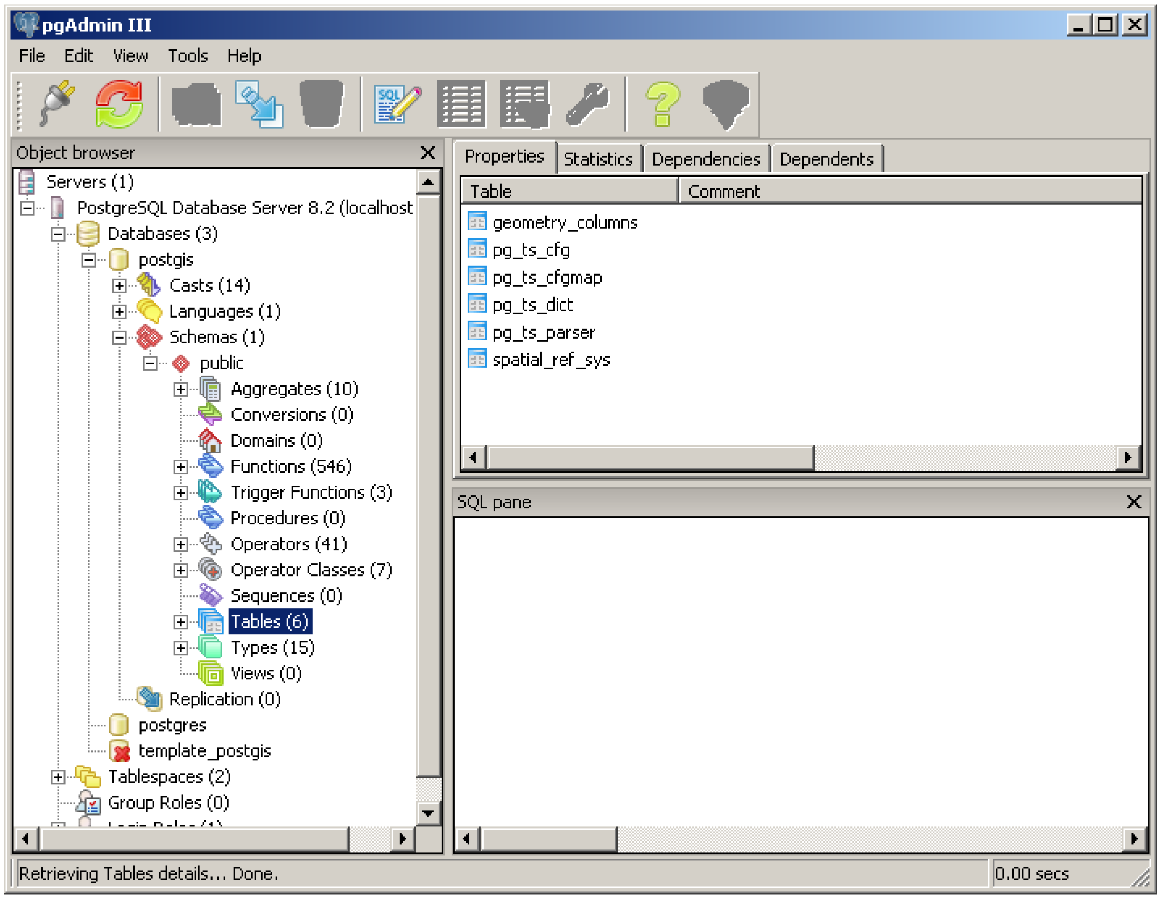


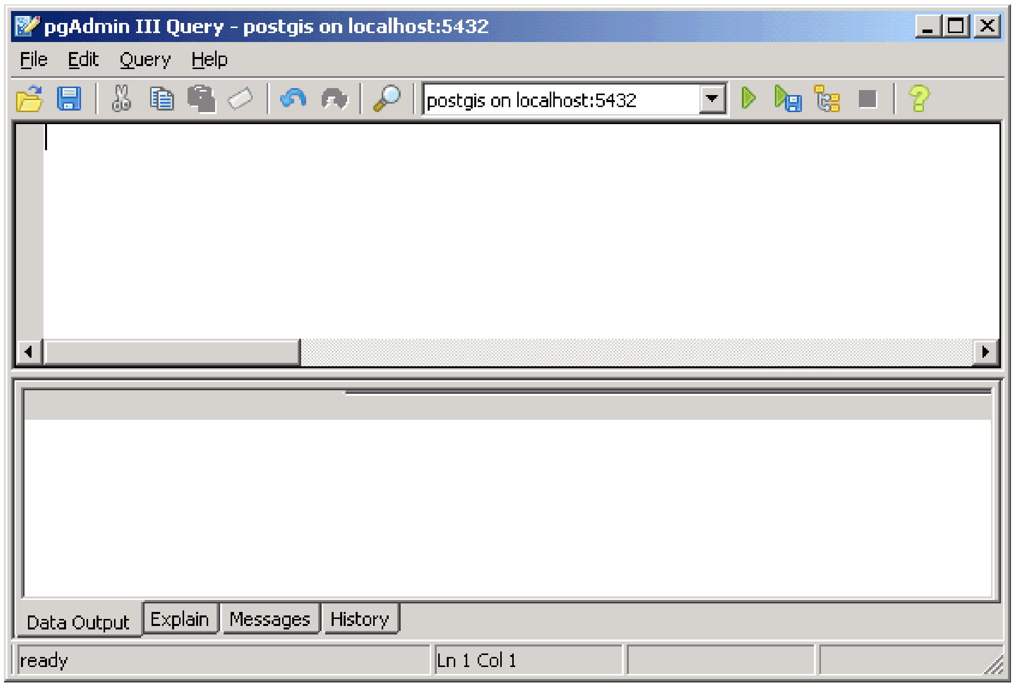
Table	Comment
geometry_columns	
pg_ts_cfg	
pg_ts_cfgmap	
pg_ts_dict	
pg_ts_parser	
spatial_ref_sys	

Retrieving Tables details... Done. 0.00 secs

2.3.1 Enabling PostGIS Without template_postgis

If your database does not have a template_postgis to use as a database creation template, you can load PostGIS manually, by invoking a pair of SQL scripts that install the PostGIS functions and types.

- **In PgAdmin, open up the SQL window by clicking the SQL button** (the one with the pencil).


pgAdmin III Query - postgis on localhost:5432
File
Edit
Query
Help
postgis on localhost:5432
Data Output
Explain
Messages
History
ready
Ln 1 Col 1

- **Choose "File ⇨ Open…" and navigate to**

  `C:\Program Files\PostgreSQL\8.2\share\contrib\lwpostgis.sql`

- **Press the "Run" button**. (The green triangle.) The `lwpostgis.sql` file will execute, loading the PostGIS functions and objects into the "postgis" database.

- **Choose "File ⇨ Open…" and navigate to**

`C:\Program Files\PostgreSQL\8.2\share\contrib\spatial_ref_sys.sql`

- **Press the "Run" button again**. The `spatial_ref_sys.sql` file will execute, loading the EPSG coordinate reference systems into the PostGIS spatial reference table.

The “postgis” database is now set up and ready for data to be loaded. There are a few things that we should remember for other applications:

Database Name:	postgis
User Name:	postgres

Administration Note: We have created our database as the “postgres” super-user. In a real multi-user system, you will probably have different users with different access privileges to various tables and functions. Setting up users and access permissions can be a complicated DBA exercise, so for workshop purposes we are using the all-powerful super-user.

Unix Note: If you install PostgreSQL and PostGIS on a Unix server, you can still use PgAdmin to administer your database. However, when loading the `lwpostgis.sql` file, you must use the `lwpostgis.sql` file from your Unix distribution, the one created during the build and install of PostGIS. This is because the `lwpostgis.sql` file includes references to machine library files that include the spatial functions. The Windows version of the file will not make sense to a Unix machine because the references will not point to the right files.

3 USING POSTGIS

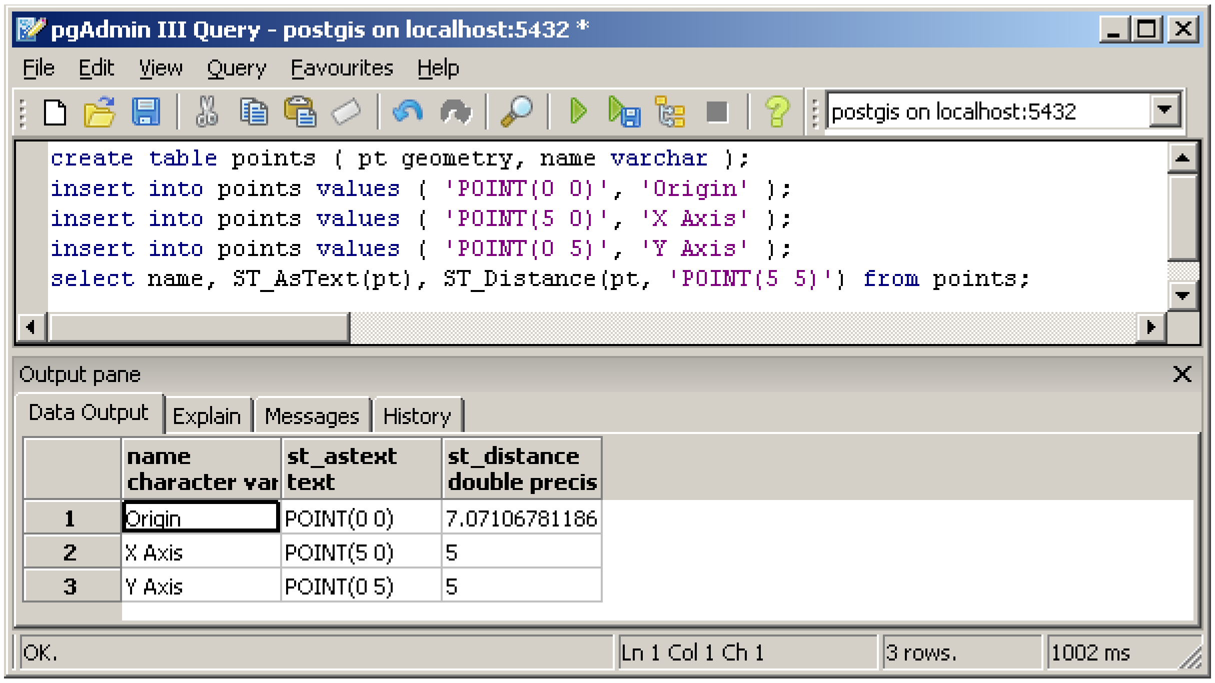
Connect to the "postgis" database with PgAdmin, and open up the query tool window, using the Tools ⇨ Query Tool or the button with the "SQL" icon on it.

3.1 SIMPLE SPATIAL SQL

Now we will test creating a table with a geometry column, adding some spatial objects to the table, and running a spatial function against the table contents.

Paste the following SQL into the query tool window, and then hit the green triangle "Execute" button (or the F5 shortcut key).

```
create table points ( pt geometry, name varchar );
insert into points values ( 'POINT(0 0)', 'Origin' );
insert into points values ( 'POINT(5 0)', 'X Axis' );
insert into points values ( 'POINT(0 5)', 'Y Axis' );
select name, ST_AsText(pt), ST_Distance(pt, 'POINT(5 5)') from points;
```


pgAdmin III Query - postgis on localhost:5432 *
File Edit View Query Favourites Help
postgis on localhost:5432
Output pane
Data Output Explain Messages History

```
create table points ( pt geometry, name varchar );
insert into points values ( 'POINT(0 0)', 'Origin' );
insert into points values ( 'POINT(5 0)', 'X Axis' );
insert into points values ( 'POINT(0 5)', 'Y Axis' );
select name, ST_AsText(pt), ST_Distance(pt, 'POINT(5 5)') from points;
```

	name character var	st_astext text	st_distance double precis
1	Origin	POINT(0 0)	7.07106781186
2	X Axis	POINT(5 0)	5
3	Y Axis	POINT(0 5)	5

OK. Ln 1 Col 1 Ch 1 3 rows. 1002 ms

Note that there are two spatial database functions being used in the example above: ST_Distance() and ST_AsText(). Both functions expect geometry objects as arguments. The ST_Distance() function calculates the minimum Cartesian distance between two spatial objects. The ST_AsText() function turns geometry into a simple textual representation, called "Well-Known Text".

3.1.1 Examples of Well-Known Text

```
POINT(1 1)
MULTIPOINT(1 1, 3 4, -1 3)
LINESTRING(1 1, 2 2, 3 4)
POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))
MULTIPOLYGON((0 0, 0 1, 1 1, 1 0, 0 0), (5 5, 5 6, 6 6, 6 5, 5 5))
MULTILINESTRING((1 1, 2 2, 3 4),(2 2, 3 3, 4 5))
```

3.2 OGC METADATA TABLES

When a database has been spatially enabled with PostGIS, two metadata tables are created, as specified by the Open Geospatial Consortium Simple Features for SQL specification, the SPATIAL_REF_SYS table, and the GEOMETRY_COLUMNS table.

GEOMETRY_COLUMNS serves as a directory of what spatially enabled tables exist in the database. It is not kept up-to-date automatically, so running a simple CREATE TABLE including a GEOMETRY type, as we did above, will not add an entry to the table. The special "AddGeometryColumn()" procedure can be used to simultaneously add a geometry column to a non-spatial table while updating the GEOMETRY COLUMNS table. Or you can just insert the relevant record yourself.

The structure of GEOMETRY_COLUMNS is:

```
             Table "public.geometry_columns"
      Column       |          Type          | Modifiers
-------------------+------------------------+-----------
 f_table_catalog   | character varying(256) | not null
 f_table_schema    | character varying(256) | not null
 f_table_name      | character varying(256) | not null
 f_geometry_column | character varying(256) | not null
 coord_dimension   | integer                | not null
 srid              | integer                | not null
 type              | character varying(30)  | not null
```

Each spatial column is uniquely identified by the combination of schema/table/column. Extra metadata about the dimensionality of the data (COORD_DIMENSION), the spatial referencing system (SRID) and the geometry type (TYPE) is provided in the table.

SPATIAL_REF_SYS serves as a directory of spatial referencing systems, both planar map projections and geodetic systems. Every geometry in the spatial database has an associated SRID, or "spatial referencing identifier", that is an integer. The SRID can be read using the ST_SRID(geometry) function.

The structure of `SPATIAL_REF_SYS` is:

```
         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) |
```

The SRID column is the unique identifier, implicitly referenced as a foreign key by both GEOMETRY_COLUMNS and the SRID embedded in every geometry. The AUTH_NAME refers to the "authority" or organization that defines and uses the referencing system (for example, for EPSG systems, the authority is "ESPG"). The AUTH_SRID is the number assigned by the authority, and the SRTEXT and PROJ4TEXT are the system definition strings, in OGC "Well-Known Text" and PROJ4 syntax, respectively.

Note: The default `SPATIAL_REF_SYS` table shipped with PostGIS is derived from the ESPG spatial referencing systems database. As such the SRID in the PostGIS `SPATIAL_REF_SYS` table and the ESPG identifier are always the same. This is a **coincidence**, and **not required** by the OGC specification **in any way**. The database "`SRID`" field is meant to be local to the database and not have any meaning external to it. The `AUTH_SRID`, on the other hand, is expected to potentially have a global meaning beyond the context of the database. In the shipping PostGIS `SPATIAL_REF_SYS` table, the `SRID` and the `AUTH_SRID` happen to be the same, and the `AUTH_NAME` is always "EPSG".

3.3 LOADING SHAPE FILES

Now we will load our example data (**C:\postgis-workshop\data\**) into the database.

The sample data is in Shape files, so we will need to convert it into a loadable format using the `shp2pgsql` tool and then load it into the database. The conversion tools run on the Windows (or UNIX) command line, so open a command line terminal window first (in Windows, Start -> Run… **`cmd.exe`**).

- **Run the `pg_setenv.bat` file first, to add the PostgreSQL install data to your path, so you can easily use the shp2pgsql tool.**

Our data is in projected coordinates, the projection is "BC Albers" and is stored in the SPATIAL_REF_SYS table as SRID 3005. When we create the loadable format, we will specify the SRID on the command line, so that the data is correctly referenced to a coordinate system. This will be important later when we try the coordinate re-projection functionality.

We may either create the load format as a file, then load the file with `psql`, or pipe the results of `shp2pgsql` directly into the `psql` terminal monitor. We will use the first option for the bc_pubs data, and then bulk load the remaining data.

```
C:\> cd \postgis-workshop\data

C:\postgis-workshop\data> dir *.shp

Directory of C:\postgis-workshop\data>

06/26/2007  11:15a            278,184 bc_municipality.shp
06/26/2007  11:15a         19,687,612 bc_roads.shp
06/26/2007  11:15a          1,537,144 countries.shp
06/26/2007  11:15a             10,576 bc_pubs.shp
06/26/2007  11:15a            466,308 timezone.shp
06/26/2007  11:15a             71,024 cities.shp
06/26/2007  11:15a              1,332 bc_hospitals.shp
06/26/2007  11:15a          3,662,668 newyork_census.shp
08/22/2007  10:21a          8,359,672 bc_voting_areas.shp
06/26/2007  11:15a          1,196,124 bc_border.shp
06/26/2007  11:15a          1,518,248 usa_counties.shp
              11 File(s)     36,788,892 bytes
               0 Dir(s)  539,488,157,696 bytes free

C:\postgis-workshop\data> pg_setenv.bat
C:\postgis-workshop\data> shp2pgsql -i -D -s 3005 bc_pubs.shp bc_pubs >
bc_pubs.sql
C:\postgis-workshop\data> psql -U postgres -f bc_pubs.sql -d postgis
Password:
. . . . . .
C:\postgis-workshop\data> pg_shpsql.bat
C:\postgis-workshop\data> psql -U postgres -f bc_data.sql -d postgis
Password:

BEGIN
INSERT 215525 1
. . . . . .
COMMIT
```

3.3.1 SHP2PGSQL Command Line Options

The shp2pgsql command has a number of options available, some of which are very useful:

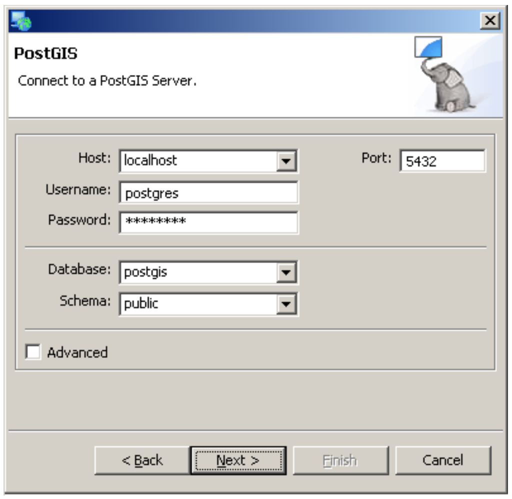
-D	Use the “database dump” format. By default, the “insert” format is used, which requires the database to parse each insert line. The dump format is much faster to load than the default.
-s <#>	Use this SRID (spatial reference system identifier) number when creating the tables and geometries. This is important to specify, as a known SRID is required to support coordinate transformations inside the database.
-i	Use 32 bit integers for all integer values. The DBF portion of the shape file can include header information about the size of integers that is radically larger than the integers in the data file. If you know all your integer values will be less that 2 billion, use this switch.
-W <encoding>	When working with database instances with explicit encoding (any encoding other than SQL_ASCII) and internationalized data (characters with accents, Cyrillic, Greek, special symbols like ©, € and ®, etc) it is important to specify what encoding the attribute data is stored in. For North Americans and western Europeans on Windows, “WINDOWS-1252” is probably what your system uses.
-a	Append mode. The default mode, create, will first create a blank table, then fill it with data. Append mode assumes a correct table already exists (with columns of the right type for each attribute) and simply loads the data. This option is useful for loading multiple files with identical attributes.

3.4 VIEWING DATA IN POSTGIS

There are a number of open source options for desktop viewers / editors of PostGIS data:

- QGIS, a C++ / Qt program;
- uDig, a Java / Eclipse program; and,
- gvSIG, a Java / Swing program.

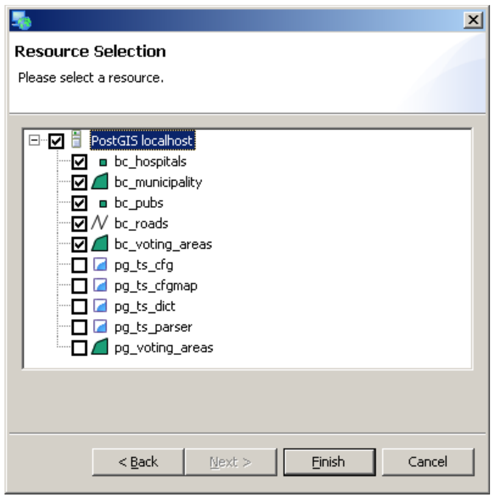
To view the data in uDig fire up the uDig application and select: **File→New->Map, then File→New→Layer.** Select a PostGIS layer type, then enter the connection information into the dialog box:



PostGIS

Connect to a PostGIS Server.

Host: localhost
Port: 5432
Username: postgres
Password: ********
Database: postgis
Schema: public
☐ Advanced



Resource Selection

Please select a resource.

- ☑ PostGIS localhost
  - ☑ bc_hospitals
  - ☑ bc_municipality
  - ☑ bc_pubs
  - ☑ bc_roads
  - ☑ bc_voting_areas
  - ☐ pg_ts_cfg
  - ☐ pg_ts_cfgmap
  - ☐ pg_ts_dict
  - ☐ pg_ts_parser
  - ☐ pg_voting_areas

Once you have selected the layers you want to view, they will be added to the map, and you can zoom, pan, edit, do table views, styling, and all the other desktop functionality you generally get in a GIS viewer.

3.5 SPATIAL INDEXES

3.5.1 Creating Spatial Indexes

Indexes are extremely important for large spatial tables, because they allow queries to quickly retrieve the records they need. Since PostGIS is frequently used for large data sets, learning how to build and (more importantly) how to use indexes is key.

PostGIS indexes are R-Tree indexes, implemented on top of the general GiST (Generalized Search Tree) indexing schema. R-Trees organize spatial data into nesting rectangles for fast searching.

```
CREATE INDEX bc_roads_gidx ON bc_roads USING GIST ( the_geom );
CREATE INDEX bc_pubs_gidx ON bc_pubs USING GIST ( the_geom );
CREATE INDEX bc_voting_areas_gidx ON bc_voting_areas USING GIST
( the_geom );
CREATE INDEX bc_municipality_gidx ON bc_municipality USING GIST
( the_geom );
CREATE INDEX bc_hospitals_gidx ON bc_hospitals USING GIST
( the_geom );
```

Now, clean up your database and update the index selectivity statistics (we will explain these in more detail in a couple sections).

```
VACUUM ANALYZE;
```

3.5.2 Using Spatial Indexes

It is important to remember that spatial indexes are not used automatically for every spatial comparison or operator. In fact, because of the “rectangular” nature of the R-Tree index, spatial indexes are only good for bounding box comparisons.

This is why all spatial databases implement a “two phase” form of spatial processing.

- The first phase is the indexed bounding box search, which runs on the whole table.
- The second phase is the accurate spatial processing test, which runs on just the subset returned by the first phase.

In PostGIS, the first phase indexed search is activated by using the “&&” operator. “&&” is a symbol with a particular meaning. Just as the symbol “=” means “equals”, the symbol “&&” means “bounding boxes overlap”. After a little while, using the “&&” operator will become second nature.

The spatial comparison functions of PostGIS (`ST_Intersects()`, `ST_DWithin()`, `ST_Contains()`, etc) automatically use the && operator.

Un-indexed versions of the same functions also exist, that use the "_" as a prefix (`_ST_Intersects()`, `_ST_Contains()`, etc). Generally, you will not use the un-indexed versions, but you might, if you are writing a complex query and want to be explicit about where to use indexed bounding boxes.

For example, the definition for the `ST_Intersects()` function is actually a SQL expansion function that re-writes the query in terms of an indexed operation (&&) and an un-indexed function (`_ST_Intersects`):

```
CREATE OR REPLACE FUNCTION ST_Intersects(geometry, geometry)
    RETURNS Boolean
    AS 'SELECT $1 && $2 AND _ST_Intersects($1,$2)'
    LANGUAGE 'SQL' IMMUTABLE;
```

3.5.3 Spatial Index Test

Let's compare the performance of a query that uses a two-phase index strategy and a query that does not.

First, time the non-indexed query (this looks for the roads that cross a supplied linestring, the example linestring is constructed so that only one road is returned, note that we are using the un-indexed _ST_Crosses() function):

```
SELECT gid, name
FROM bc_roads
WHERE
  _ST_Crosses(
    the_geom,
    ST_GeomFromText('LINESTRING(1220446 477473,1220417 477559)', 3005)
  );

  gid  |     name
-------+--------------
 64556 | Kitchener St
(1 row)
```

Now, time the two-phase strategy, that includes an index search as well as the crossing test:

```
SELECT gid, name
FROM bc_roads
WHERE
  ST_Crosses(
    the_geom,
    ST_GeomFromText('LINESTRING(1220446 477473,1220417 477559)', 3005)
  );

  gid  |     name
-------+--------------
 64555 | Kitchener St
(1 row)
```

You will have to be pretty fast with your stopwatch to time the second query.

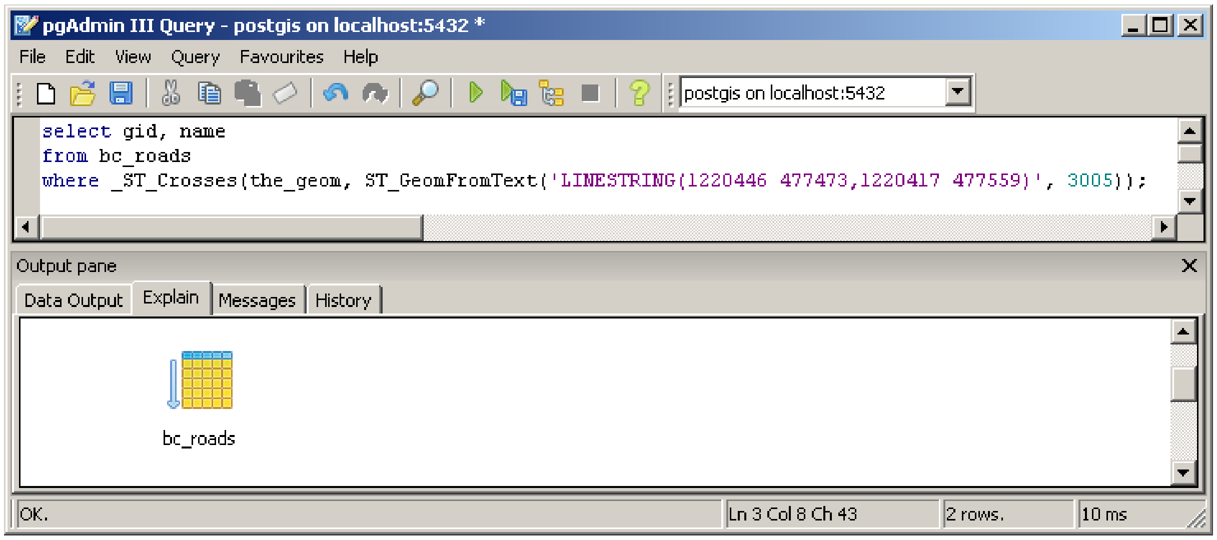
3.5.4 Indexes and Query Plans

Databases are fancy engines for speeding up random access to large chunks of data. Large chunks of data have to be stored on disk, and disk access is (compared to memory access) very, very slow. At the core of databases are algorithms tuned to search as much data as possible with as few disk accesses as possible.

Query plans are the rules used by databases to convert a piece of SQL into a strategy for reading the data. In PostgreSQL, you can see the estimated query plan for any SQL query by pre-pending "EXPLAIN" before the query. You can see the actual observed performance by pre-pending "EXPLAIN ANALYZE" before the query.

In PgAdmin, you can see a visual representation of the query plan, by running a query with the "Explain" button instead of the "Run" button.

Using an un-indexed query function, `_ST_Crosses()`, the explain shows that the result is returned by running a full scan on the data table (`bc_roads`):

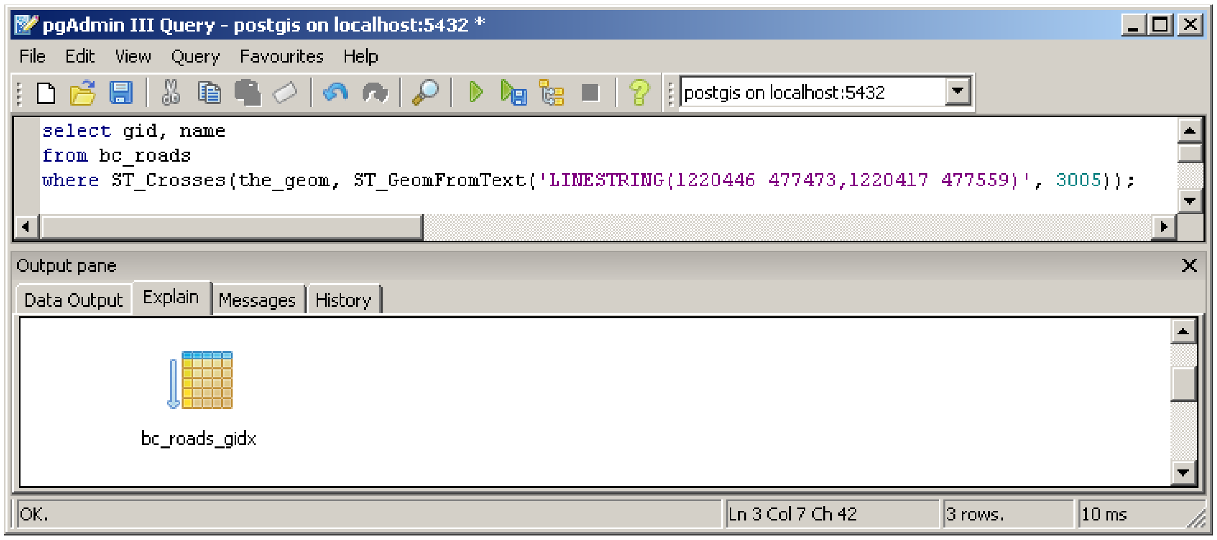

pgAdmin III Query - postgis on localhost:5432 *
File Edit View Query Favourites Help
postgis on localhost:5432
Output pane
Data Output Explain Messages History

```
select gid, name
from bc_roads
where _ST_Crosses(the_geom, ST_GeomFromText('LINESTRING(1220446 477473,1220417 477559)', 3005));
```

bc_roads

OK. Ln 3 Col 8 Ch 43 2 rows. 10 ms

Using an indexed query function, ST_Crosses(), the explain shows that the result is returned by running an index scan on the spatial index (bc_roads_gidx):


pgAdmin III Query - postgis on localhost:5432 *
File Edit View Query Favourites Help
postgis on localhost:5432
Output pane
Data Output Explain Messages History

```
select gid, name
from bc_roads
where ST_Crosses(the_geom, ST_GeomFromText('LINESTRING(1220446 477473,1220417 477559)', 3005));
```

bc_roads_gidx

OK. Ln 3 Col 7 Ch 42 3 rows. 10 ms

3.5.5 When Query Plans Go Bad

A database attempts to minimize disk accesses. Indexes can lower disk accesses for queries that only return a few rows, but can actually increase accesses for queries that return a large number of rows. The database attempts to generate a "query plan" that reduces disk accesses to a minimum by using the "most selective" indexes.

So, if you are making a query using a filter that specifies two indexed columns, the database will attempt to pick the index with the "greatest selectivity". That is, the index that returns the fewest rows. It does this by using statistics gathered from the table and indexes about the makeup of the data.

As of PostgreSQL version 8.0, the spatial selectivity analysis for PostGIS indexes is integrated into the main selectivity system of PostgreSQL. So, to ensure that your statistics are kept up to date as you change your data, you just have to run the "ANALYZE" command occasionally.

If the spatial index is (incorrectly) chosen, the database will have to sequence scan every road from the south of the province to see if it is named 'Bob', instead of sequence scanning every road named 'Bob' to see if it is in the south half of the province.

So, it is important to ensure that selectivity statistics are generated using either "ANALYZE" manually or turning on pg_autovacuum to automatically gather statistics regularly.

As of PostgreSQL 8.1, the query planner can merge the index scans of two indexes on the same table, which dramatically reduces selectivity problems, particularly for cases where two moderately selective indexes can be used to create a very selective query. (e.g. "in Prince George and named Main Street", there are lots of roads in Prince George and lots of roads named Main Street, but very few roads that meet both conditions at once).

3.6 POSTGRESQL OPTIMIZATION

PostgreSQL is shipped by the development team with a conservative default configuration. The intent is to ensure that PostgreSQL runs without modification on as many common machines as possible. That means if your machine is uncommonly good (with a large amount of memory, for example) the default configuration is far too conservative.

True database optimization, particularly of databases under transactional load, is a complex business, and there is no one size fits all solution. However, simply boosting a few PostgreSQL memory use parameters can increase performance for the most common spatial database workload: loading a lot of data in once, then querying it repeatedly.

The database configuration file is in the database's `\data` area, and is named `postgresql.conf`. The Windows install includes a handy link to the file in **Start→Programs→PostgreSQL 8.2→Configuration Files→Edit postgresql.conf**

Changing the following lines offer quick boosts if you have the hardware:

```
shared_buffers = 256MB       # min 128kB or max_connections*16kB
```

Increase the shared buffers as much as possible, but not so much that you exceed the amount of physical memory available for you on the computer. You will have to do a little math to calculate your ideal buffer size. Take your physical memory, subtract the amount of memory used by non-database processes, and use 75% of the remaining memory for database shared buffers.

Shared buffers are (surprise!) shared between all PostgreSQL back-ends, so you do not have to worry about how many back-ends you spawn.

```
work_mem = 16MB                 # min 64, size in KB
maintenance_work_mem = 16MB     # min 1024, size in KB
```

Working memory is used for sorting and grouping. It is used per-backend, so you have to guesstimate how many back-ends you will have running at a time. The default value is only 1MB.

Maintenance memory is used for vacuuming the database. It is also used per-backend, but you are unlikely to have more than one backend vacuuming at a time (probably). The default value is 16MB, which might be enough, depending on how often you vacuum, how big your tables are, and how often your data changes.

NOTE: Some optimizations, like `shared_buffers` have to be synchronized with operating system kernel tuning. For example, the default number of shared buffers allowed by Linux is fairly low, and that number needs to be increased before a tuned-up PostgreSQL will be able to run. See http://www.postgresql.org/docs/8.2/static/kernel-resources.html.

```
wal_buffers = 1MB                  # min 32kB
```

Write-ahead log (WAL) buffers are used to buffer write information in memory. More WAL buffers are useful for systems with lots of write activity.

```
checkpoint_segments = 6            # in logfile segments, min 1, 16MB each
```

Checkpoint segments are used in flushing the write-ahead logs to the actual tables. Increasing this number is a good idea for systems with write activity.

```
random_page_cost = 2.0             # arbitrary scale
```

The random page cost is a value-less scalar. Changing this value will do nothing for raw performance, but will improve the plans the planner creates for complex, multi-index, multi-table queries. Generally reducing the number from the default value of 4.0 to 2.0 achieves improved plans for complex queries on systems that have a healthy amount of memory.

3.7 SPATIAL ANALYSIS IN SQL

A surprising number of traditional GIS analysis questions can be answered using a spatial database and SQL. GIS analysis is generally about filtering spatial objects with conditions, and summarizing the results – and that is exactly what databases are very good at. GIS analysis also tests interactions between spatially similar features, and uses the interactions to answer questions – with spatial indexes and spatial functions, databases can do that too!

3.7.1 Exercises

These exercises will be easier if you first peruse the data dictionary and functions list for information about the data columns and available functions. You can enter these exercises in order, or for a challenge, cover up your page with another piece of paper and try to figure out the answer yourself before looking at the SQL.

“What is the total length of all roads in the province, in kilometers?”

```
SELECT Sum(ST_Length(the_geom))/1000 AS km_roads
FROM bc_roads;

     km_roads
------------------
 70842.1243039643
(1 row)
```

”How large is the city of Prince George, in hectares?”

```
SELECT ST_Area(the_geom)/10000 AS hectares
FROM bc_municipality
WHERE name = 'PRINCE GEORGE';

     hectares
------------------
 32657.9103824927
(1 row)
```

”What is the largest municipality in the province, by area?”

```
SELECT
  name,
  ST_Area(the_geom)/10000 AS hectares
FROM bc_municipality
ORDER BY hectares DESC
LIMIT 1;

     name      |    hectares
---------------+-----------------
 TUMBLER RIDGE | 155020.02556131
(1 row)
```

The last one is particularly tricky. There are several ways to do it, including a two step process that finds the maximum area, then finds the municipality that has that area. The suggested way uses the PostgreSQL “LIMIT” statement and a reverse ordering to pull just the top area.

3.8 BASIC EXERCISES

“What is the perimeter of the municipality of Vancouver?”

```
SELECT ST_Perimeter(the_geom)
FROM bc_municipality
WHERE name = 'VANCOUVER';

    perimeter
------------------
 57321.7782018048
(1 row))
```

”What is the total area of all voting areas in hectares?”

```
SELECT Sum(ST_Area(the_geom))/10000 AS hectares
FROM bc_voting_areas;

     hectares
------------------
 94759319.6833071
(1 row)
```

“What is the total area of all voting areas with more than 100 voters in them?”

```
SELECT Sum(ST_Area(the_geom))/10000 AS hectares
FROM bc_voting_areas
WHERE vtotal > 100;

     hectares
------------------
 36609425.2114911
(1 row)
```

”What is the length in kilometers of all roads named ‘Douglas St’?”

```
SELECT Sum(ST_Length(the_geom))/1000 AS kilometers
FROM bc_roads
WHERE name = 'Douglas St';

    kilometers
------------------
 19.8560819878386
(1 row)
```

4 ADVANCED POSTGIS

4.1 DATA INTEGRITY

PostGIS spatial functions require that input geometries obey the “Simple Features for SQL” specification for geometry construction. That means LINESRINGS cannot self-intersect, POLYGONS cannot have their holes outside their boundaries, and so on. Some of the specifications are quite strict, and some input geometries may not conform to them.

The `ST_IsValid()` function is used to test that geometries conform to the specification. This query counts the number of invalid geometries in the voting areas table:

```
SELECT gid
FROM bc_voting_areas
WHERE NOT ST_IsValid(the_geom);

  gid
------
 4897
(1 row)
```

So we have one invalid polygon in our data set. How do we fix it?

PostGIS has no special “cleaning” functionality at the moment, but it turns out that the geometry processing routines required to build a buffer around features also do a good job of rebuilding valid polygons from invalid inputs. The trick? Setting the buffer distance to zero, so that the output is the same size as the input.

Does the buffer-by-zero trick work for our geometry?

```
SELECT ST_IsValid(ST_Buffer(the_geom, 0.0))
FROM bc_voting_areas
WHERE gid = 4897;

  st_isvalid
-------------
  t
(1 row)
```

Good, let’s apply that fix to our table.

```
UPDATE bc_voting_areas
SET the_geom = ST_Buffer(the_geom, 0.0)
WHERE gid = 4897;
```

4.2 DISTANCE QUERIES

It is easy to write inefficient queries for distances, because it is not always obvious how to use the index in a distance query. Here is a short example of a distance query using an index.

“How many BC Unity Party supporters live within 2 kilometers of the Tabor Arms pub in Prince George?”

First, we find out where the Tabor Arms is:

```
SELET ST_AsText(the_geom)
FROM bc_pubs
WHERE name ILIKE 'Tabor Arms%';

 POINT(1209385.41168654 996204.96991804)
```

Now, we use that location to pull all the voting areas within 2 kilometers and sum up the Unity Party votes. The “obvious” way to do the query will not use the index:

```
SELECT Sum(unity) AS unity_voters
FROM bc_voting_areas
WHERE
  ST_Distance(
    the_geom,
    ST_GeomFromText('POINT(1209385 996204)',3005)
  ) < 2000;

  unity_voters
  ------------
  421
```

The “optimized” way to do the query is to use the `ST_DWithin()` function, that silently adds an index operator to the mix. Here is the query:

```
SELECT Sum(unity) AS unity_voters
FROM bc_voting_areas
WHERE
  ST_DWithin(
    the_geom,
    ST_GeomFromText('POINT(1209385 996204)',3005),
    2000
  );
```

And here is the definition of `ST_DWithin()`:

```
CREATE FUNCTION ST_DWithin(geometry, geometry, float8)
  RETURNS boolean
  AS '
    SELECT
      $1 && ST_Expand($2,$3) AND
      $2 && ST_Expand($1,$3) AND
      ST_Distance($1, $2) < $3
  ' LANGUAGE 'SQL' IMMUTABLE;
```

Note that `ST_DWithin()` still uses the `ST_Distance()` function, it just adds two index conditions (the && operators) that quickly winnow the candidate geometries down using bounding box tests. Any geometry that is not inside a box expanded by N units in all directions cannot be within N units of the candidate geometry.

4.2.1 Distance Not Buffer

For completeness, here is the way you **never ever, ever** want to use to answer a distance query:

```
SELECT Sum(unity) AS unity_voters
FROM bc_voting_areas
WHERE
  ST_Contains(
    ST_Buffer(the_geom,2000),
    ST_GeomFromText('POINT(1209385 996204)',3005)
  )
```

This query will return the same result as the other queries, but much slower, because it has to individually buffer every voting area in order to calculate the result.

4.3 SPATIAL JOINS

A standard table join puts two tables together into one output result based on a common key. A spatial join puts two tables together into one output result based on a spatial relationship.

The previous distance query ("How many BC Unity Party supporters live within 2 kilometers of the Tabor Arms pub in Prince George?") can be re-written to run in one step as a join:

```
SELECT Sum(unity) AS unity_voters
FROM
  bc_voting_areas,
  bc_pubs
WHERE
  ST_DWithin(
    bc_voting_area.the_geom,
    bc_pubs.the_geom,
    2000
  )
AND
  bc_pubs.name ilike 'Tabor Arms%';
```

Note that the join condition (where both the voting area and pubs occur, is actually the spatial function, `ST_DWithin()`).

For another example, we will find the safest pubs in British Columbia. Presumably, if we are close to a hospital, things can only get so bad.

"Find all pubs located within 250 meters of a hospital."

```
SELECT bc_hospitals.name, bc_pubs.name
FROM bc_hospitals, bc_pubs
WHERE ST_DWithin(bc_hospitals.the_geom, bc_pubs.the_geom, 250);
```

Just as with a standard table join, values from each input table are associated and returned side-by-side.

Using indexes, spatial joins can be used to do very large scale data merging tasks.

For example, the results of the 2000 election are usually summarized by riding – that is how members are elected to the legislature, after all. But what if we want the results summarized by municipality, instead?

“Summarize the 2000 provincial election results by municipality.”

```
SELECT
  m.name,
  Sum(v.ndp) AS ndp,
  Sum(v.liberal) AS liberal,
  Sum(v.green) AS green,
  Sum(v.unity) AS unity,
  Sum(v.vtotal) AS total
FROM
  bc_voting_areas v,
  bc_municipality m
WHERE ST_Intersects(v.the_geom, m.the_geom)
GROUP BY m.name
ORDER BY m.name;


           name            |  ndp  | liberal | green | unity | total
---------------------------+-------+---------+-------+-------+-------
100 MILE HOUSE             |   398 |     959 |     0 |    70 |  1527
ABBOTSFORD                 |  1507 |    9547 |    27 |   575 | 12726
ALERT BAY                  |   366 |      77 |    26 |     0 |   500
ANMORE                     |   247 |    1299 |     0 |     0 |  1644
ARMSTRONG                  |   433 |    1231 |   199 |   406 |  2389
ASHCROFT                   |   217 |     570 |     0 |    39 |   925
BELCARRA                   |    93 |     426 |    37 |     0 |   588
BURNABY                    | 15500 |   31615 |  8276 |  1218 | 58316
BURNS LAKE                 |   370 |     919 |   104 |    92 |  1589
CACHE CREEK                |    76 |     323 |     0 |    27 |   489
CAMPBELL RIVER             |  2814 |    6904 |  1064 |     0 | 11191
. . . . . . . .
```

4.4 OVERLAYS

Overlays are a standard GIS technique for analyzing the relationship between two layers. Particularly where attributes are to be scaled by area of interaction between the layers, overlays are an important tool.

In SQL terms, an overlay is just a spatial join with an intersection operation. For each polygon in table A, find all polygons in table B that interact with it. Intersect A with all potential B's and copy the resultants into a new table, with the attributes of both A and B, and the original areas of both A and B. From there, attributes can be scaled appropriately, summarized, etc. Using sub-selects, temporary tables are not even needed – the entire overlay-and-summarize operation can be embedded in one SQL statement.

Here is a small example overlay that creates a new table of voting areas clipped by the Prince George municipal boundary:

```
CREATE TABLE pg_voting_areas AS
SELECT
  ST_Intersection(v.the_geom, m.the_geom) AS intersection_geom,
  ST_Area(v.the_geom) AS va_area,
  v.*,
  m.name
FROM
  bc_voting_areas v,
  bc_municipality m
WHERE
  ST_Intersects(v.the_geom, m.the_geom) AND
  m.name = 'PRINCE GEORGE';


postgis=# SELECT Sum(ST_Area(intersection_geom)) FROM pg_voting_areas;
       sum
------------------
 326579103.824927
(1 row)

postgis=# SELECT ST_Area(the_geom) FROM bc_municipality WHERE name =
'PRINCE GEORGE';
       area
------------------
 326579103.824927
(1 row)
```

Note that the area of the sum of the resultants equals the area of the clipping feature, always a good sign.

4.5 COORDINATE PROJECTION

PostGIS supports coordinate re-projection inside the database.

Every geometry in PostGIS has a “spatial referencing identifier” or “SRID” attached to it. The SRID indicates the spatial reference system the geometry coordinates are in. So, for example, all the geometries in our examples so far have been in the British Columbia Albers reference system.

You can view the SRID of geometries using the srid() function:

```
postgis=# SELECT ST_SRID(the_geom) FROM bc_roads LIMIT 1;
```

The SRID of 3005 corresponds to a particular entry in the SPATIAL_REF_SYS table. Because PostGIS uses the PROJ4 library for re-projection support, the SPATIAL_REF_SYS table includes a PROJ4TEXT column that gives the coordinate system definition in PROJ4 parameters:

```
postgis=# SELECT proj4text FROM spatial_ref_sys WHERE srid=3005;
                              proj4text
------------------------------------------------------------------
+proj=aea +ellps=GRS80 +datum=NAD83 +lat_0=45.0 +lon_0=-126.0 +lat_1=58.5
+lat_2=50.0 +x_0=1000000 +y_0=0
(1 row)
```

Coordinate re-projection is done using the transform() function, referencing an SRID that exists in the SPATIAL_REF_SYS table. For example, in the panel below we will view a geometry in the stored coordinates, then re-project it to geographic coordinates using the transform() function:

```
postgis=# SELECT ST_AsText(the_geom) FROM bc_roads LIMIT 1;
                          astext
----------------------------------------------------------
 MULTILINESTRING((1004687.04355194 594291.053764096,1004729.74799931
594258.821943696,1004808.0184134 594223.285878035,1004864.93630072
594204.422638658,1004900.50302171 594200.005856311))
(1 row)

postgis=# SELECT ST_AsText(ST_Transform(the_geom,4326)) FROM bc_roads
LIMIT 1;
                               astext
--------------------------------------------------------------------
MULTILINESTRING((-125.9341 50.3640700000001,-125.9335 50.36378,
-125.9324 50.36346,-125.9316 50.36329,-125.9311 50.36325))
(1 row)
```

Note the longitude/latitude coordinates in the transformed geometry.

4.6 ADVANCED EXERCISES

”What is the length in kilometers of ‘Douglas St’ in Victoria?”

```
SELECT Sum(ST_Length(r.the_geom))/1000 AS kilometers
FROM
  bc_roads r,
  bc_municipality m
WHERE
  ST_Contains(m.the_geom, r.the_geom) AND
  r.name = 'Douglas St' AND
  m.name = 'VICTORIA';

    kilometers
------------------
 4.44341090725
(1 row)
```

”What two pubs have the most Green Party supporters within 500 meters of them?”

```
SELECT
  p.name,
  p.city,
  Sum(v.green) AS greens
FROM
  bc_pubs p,
  bc_voting_areas v
WHERE
  ST_DWithin(v.the_geom, p.the_geom, 500)
GROUP BY p.name, p.city
ORDER BY greens desc limit 2;

      name       |   city    | greens
----------------+-----------+--------
 Bimini's        | Vancouver |   1407
 Darby D. Dawes | Vancouver |   1104
(2 rows)
```

”What is the latitude of the most southerly hospital in the province?”

```
SELECT ST_Y(ST_Transform(the_geom,4326)) AS latitude
FROM bc_hospitals
ORDER BY latitude ASC
LIMIT 1;

     latitude
------------------
 48.4657953714625
(1 row)
```

”What were the percentage NDP and Liberal vote within the city limits of Prince George in the 2000 provincial election?”

```
SELECT
  100*sum(v.ndp)/sum(v.vtotal) AS ndp,
  100*sum(v.liberal)/sum(v.vtotal) AS liberal
FROM
  bc_voting_areas v,
  bc_municipality m
WHERE
  ST_Contains(m.the_geom, v.the_geom) AND
  m.name = 'PRINCE GEORGE';

 ndp | liberal
-----+---------
  17 |      59
(1 row)
```

”What is the largest voting area polygon that has a hole?”

```
SELECT
  gid,
  id,
  ST_Area(the_geom) AS area
FROM bc_voting_areas
WHERE ST_NRings(the_geom) > 1
ORDER BY area DESC
LIMIT 1;

 gid  |   id   |       area
------+--------+------------------
 6130 | SKN 80 | 12552212259.4344
(1 row)
```

“How many NDP voters live within 50 meters of ‘Simcoe St’ in Victoria?”

```
SELECT
  Sum(v.ndp) AS ndp
FROM
  bc_voting_areas v,
  bc_municipality m,
  bc_roads r
WHERE
  ST_DWithin(r.the_geom, v.the_geom, 50) AND
  ST_Contains(m.the_geom, r.the_geom) AND
  r.name = 'Simcoe St' AND
  m.name = 'VICTORIA';

 ndp
------
 2558
(1 row)
```

5 MAPSERVER & POSTGIS

The University of Minnesota MapServer (aka "MapServer") can read spatial data directly out of PostGIS databases. This provides a quick and easy way to publish database data directly to the internet without a "staging" process of dumping data out of the database for publication.

5.1 BASIC MAPSERVER CONFIGURATION

The most basic configuration of MapServer using PostGIS simply references a spatial table and indicates to MapServer which column contains the spatial information to be mapped. All other configuration is standard MapServer map file syntax:

```
LAYER
 CONNECTIONTYPE postgis
 NAME "bc_roads"
 CONNECTION "user=postgres password=postgres dbname=postgis
host=localhost"
 DATA "the_geom FROM bc_roads"
 STATUS ON
 TYPE LINE
 CLASS
   COLOR 200 0 0
 END
END
```

The example above takes roads from the `bc_roads` table in our PostGIS database and maps them all as red lines.

5.2 MAPSERVER FILTERS AND EXPRESSIONS

The only “standard” MapServer map file constructs that work differently when using PostGIS as a data source are the FILTER parameter and the EXPRESSION parameter.

When using shape files as a data source, FILTER takes in MapServer “expression format” logical strings. When using PostGIS, the FILTER takes in SQL “where clause” expressions. That is, SQL fragments that would be legal in a SQL “where” expression. Note that the example below does not have square brackets [] around the attribute name, as it would if it where using MapServer syntax.

```
LAYER
 CONNECTIONTYPE postgis
 NAME "bc_roads"
 CONNECTION "user=postgres password=postgres dbname=postgis
host=localhost"
 DATA "the_geom FROM bc_roads"
 FILTER “type = 5”
 STATUS ON
 TYPE LINE
 CLASS
   COLOR 200 0 0
 END
END
```

MapServer expressions are only slightly different when using PostGIS. When using shape files, the attributes in MapServer expressions are expressed all in UPPER CASE. When using PostGIS, the attributes are expressed using all lower case.

```
LAYER
 CONNECTIONTYPE postgis
 NAME "bc_roads"
 CONNECTION "user=postgres password=postgres dbname=postgis
host=localhost"
 DATA "the_geom FROM bc_roads"
 STATUS ON
 TYPE LINE
CLASS
   COLOR 200 0 0
   EXPRESSION ([type] = 6)
 END
CLASS
   COLOR 200 200 0
   EXPRESSION ([type] < 6)
 END
END
```

5.3 MAPSERVER WITH SQL

Sometimes the information you want to map is not directly available in your table – it is the result of a calculation or a comparison with some other data.

In these cases, you can construct an arbitrary piece of SQL, and have MapServer render the result into a map. Once you get used to the power of spatial SQL, you will see how useful this MapServer/PostGIS capability is.

As a simple example, consider the `bc_voting_areas` table. It includes a `vtotal` column which records the total number of votes, and a `vregist` column that records the total number of registered voters. Mapping the total number of votes would be interesting, but a more useful map would be of the percentage voter turnout. We can calculate percentage turnout on the fly in SQL and have MapServer render that. The mapping file fragment below shows the data statement:

```
DATA "the_geom from
  (select gid, the_geom, 100 * vtotal::real / vregist::real
  as percent from bc_voting_areas) as query
  using srid=3005 using unique gid"
```

The statement includes the following components:

- **`the_geom`** indicates the spatial column to use.
- **`(select … ) as query`** is the arbitrary SQL query.
- **`using srid=3005`** tells MapServer what SRID to use when building a bounding box to subset the data. This should be the SRID of the data table in the database.
- **`using unique gid`** tells MapServer what column in the SQL query to use as a unique key. This is needed to support map querying against your SQL.