Geospatial development links your data to locations on the surface of the Earth. Writing geospatial programs involves tasks such as grouping data by location, storing and analyzing large amounts of spatial information, performing complex geospatial calculations, and drawing colorful interactive maps.

This book provides an overview of the major geospatial concepts, data sources, and toolkits. It starts by showing you how to store and access spatial data using Python, how to perform a range of spatial calculations, and how to store spatial data in a database. Further on, the book teaches you how to build your own slippy map interface within a web application, and finishes with the detailed construction of a geospatial data editor using the GeoDjango framework.

By the end of this book, you will be able to confidently use Python to write your own geospatial applications ranging from quick, one-off utilities to sophisticated web-based applications using maps and other geospatial data.

Who this book is written for
This book is for experienced Python developers who want to learn about geospatial concepts, obtain and work with geospatial data, solve spatial problems, and build sophisticated map-based applications using Python.

What you will learn from this book
- Access, manipulate, and display geospatial data from within your Python programs
- Master the core geospatial concepts of location, distance, units, projections, and datums
- Read and write geospatial data in both vector and raster format
- Perform complex, real-world geospatial calculations using Python
- Store and access geospatial information in a database
- Use points, lines, and polygons within your Python programs
- Convert geospatial data into attractive maps using Python-based tools
- Build complete web-based mapping applications using Python

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 6 'Spatial Databases'
- A synopsis of the book’s content
- More information on Python Geospatial Development Third Edition
About the Author

**Erik Westra** has been a professional software developer for over 25 years and has worked almost exclusively in Python for the past decade. Erik's early interest in graphical user interface design led to the development of one of the most advanced urgent courier dispatch systems used by messenger and courier companies worldwide. In recent years, Erik has been involved in the design and implementation of systems matching seekers and providers of goods and services across a range of geographical areas as well as real-time messaging and payments systems. This work has included the creation of real-time geocoders and map-based views of constantly changing data. Erik is based in New Zealand, and he works for companies worldwide.

He is also the author of the Packt titles *Python Geospatial Analysis* and *Building Mapping Applications with QGIS* as well as the forthcoming title *Modular Programming with Python*. 
Preface

With the increasing use of map-based web sites and spatially aware devices and applications, geospatial development is a rapidly growing area. As a Python developer, you can't afford to be left behind. In today's location-aware world, every Python developer can benefit from understanding geospatial concepts and development techniques.

Working with geospatial data can get complicated because you are dealing with mathematical models of the earth's surface. Since Python is a powerful programming language with many high-level toolkits, it is ideally suited to geospatial development. This book will familiarize you with the Python tools required for geospatial development. It walks you through the key geospatial concepts of location, distance, units, projections, datums, and geospatial data formats. We will then examine a number of Python libraries and use these with freely available geospatial data to accomplish a variety of tasks. The book provides an in-depth look at storing spatial data in a database and how you can use spatial databases as tools to solve a range of geospatial problems.

It goes into the details of generating maps using the Mapnik map-rendering toolkit and helps you build a sophisticated web-based geospatial map-editing application using GeoDjango, Mapnik, and PostGIS. By the end of the book, you will be able to integrate spatial features into your applications and build complete mapping applications from scratch.

This book is a hands-on tutorial, teaching you how to access, manipulate, and display geospatial data efficiently using a range of Python tools for GIS development.
Preface

What this book covers

Chapter 1, Geospatial Development Using Python, provides an overview of the Python programming language and the concepts behind geospatial development. Major use cases of geospatial development and recent and upcoming developments in the field are also covered.

Chapter 2, GIS, introduces the core concepts of location, distance, units, projections, shapes, datums, and geospatial data formats, before discussing the process of working with geospatial data by hand.

Chapter 3, Python Libraries for Geospatial Development, explores the major Python libraries available for geospatial development, including the available features, how to install them, the major concepts you need to understand about the libraries, and how they can be used.

Chapter 4, Sources of Geospatial Data, investigates the major sources of freely available geospatial data, what information is available, the data format used, and how to import the data once you have downloaded it.

Chapter 5, Working with Geospatial Data in Python, uses the libraries introduced earlier to perform various tasks using geospatial data, including changing projections, importing and exporting data, converting and standardizing units of geometry and distance, and performing geospatial calculations.

Chapter 6, Spatial Databases, introduces the concepts behind spatial databases before looking in detail at the PostGIS spatially enabled database and how to install and use it from a Python program.

Chapter 7, Using Python and Mapnik to Produce Maps, provides a detailed look at the Mapnik map-generation toolkit and how to use it to produce a variety of maps.

Chapter 8, Working with Spatial Data, works through the design and implementation of a complete geospatial application called DISTAL, using freely available geospatial data stored in a spatial database.

Chapter 9, Improving the DISTAL Application, improves the application written in the previous chapter to solve various usability and performance issues.

Chapter 10, Tools for Web-based Geospatial Development, examines the concepts of web application frameworks, web services, JavaScript UI libraries, and slippy maps. It introduces a number of standard web protocols used by geospatial applications and finishes with a survey of the tools and techniques that will be used to build the complete mapping application in the final three chapters of this book.
Chapter 11, Putting it all Together – a Complete Mapping Application, introduces ShapeEditor, a complete and sophisticated web application built using PostGIS, Mapnik, and GeoDjango. We start by designing the overall application, and we then build the ShapeEditor's database models.

Chapter 12, ShapeEditor – Importing and Exporting Shapefiles, continues with the implementation of the ShapeEditor system, concentrating on displaying a list of imported shapefiles, along with logic for importing and exporting shapefiles via a web browser.

Chapter 13, ShapeEditor – Selecting and Editing Features, concludes the implementation of the ShapeEditor, adding logic to let the user select and edit features within an imported shapefile. This involves the creation of a custom tile map server and the use of the OpenLayers JavaScript library to display and interact with geospatial data.
In this chapter, we will look at how you can use a PostGIS database to store and work with spatial data. In particular, we will cover:

- The concept of a spatially enabled database
- Spatial indexes and how they work
- How PostGIS acts as an extension to the PostgreSQL relational database
- How to install PostgreSQL, PostGIS, and the `psycopg2` Python database adapter onto your computer
- How to set up and configure a spatial database using PostGIS
- How to use the `psycopg2` database adapter to access a spatial database from your Python code
- How to create, import, and query against spatial data using Python
- Recommended best practices for storing spatial data in a database

This chapter is intended to be an introduction to using databases in a geospatial application. Chapter 8, Working with Spatial Data, will build on this to perform powerful spatial queries not possible using shapefiles and other geospatial data files.

**Spatially-enabled databases**

In a sense, almost any database can be used to store geospatial data: simply convert a geometry to WKT format and store the results in a text column. But while this would allow you to store geospatial data in a database, it wouldn't let you query it in any useful way. All you could do is retrieve the raw WKT text and convert it back to a geometry object, one record at a time.
A spatially-enabled database, on the other hand, is aware of the notion of space, and allows you to work with spatial objects and concepts directly. In particular, a spatially-enabled database allows you to:

- Store spatial data types (points, lines, polygons, and so on) directly in the database in the form of a geometry column
- Perform spatial queries on your data, for example, select all landmarks within 10 km of the city named "San Francisco"
- Perform spatial joins on your data, for example, select all cities and their associated countries by joining cities and countries on (city inside country)
- Create new spatial objects using various spatial functions, for example, set "danger_zone" to the intersection of the "flooded_area" and "urban_area" polygons

As you can imagine, a spatially-enabled database is an extremely powerful tool for working with geospatial data. By using spatial indexes and other optimizations, spatial databases can quickly perform these types of operations and can scale to support vast amounts of data simply not feasible using other data-storage schemes.

**Spatial indexes**

One of the defining characteristics of a spatial database is the ability to create and use "spatial" indexes to speed up geometry-based searches. These indexes are used to perform spatial operations, such as identifying all the features that lie within a given bounding box, identifying all the features within a certain distance of a given point, or identifying all the features that intersect with a given polygon.

Spatial indexes are one of the most powerful features of spatial databases, and it is worth spending a moment becoming familiar with how they work. Spatial indexes don't store the geometry directly; instead, they calculate the bounding box for each geometry and then index the geometries based on their bounding boxes. This allows the database to quickly search through the geometries based on their position in space:
The bounding boxes are grouped into a nested hierarchy based on how close together they are, as shown in the following illustration:
The hierarchy of nested bounding boxes is then represented using a tree-like data structure, as follows:

![Diagram of nested bounding boxes]

The computer can quickly scan through this tree to find a particular geometry or compare the positions or sizes of the various geometries. For example, the geometry containing the point represented by the X in the preceding diagram can be quickly found by traversing the tree and comparing the bounding boxes at each level. The spatial index will be searched in the following manner:

![Diagram of spatial index search]

Using the spatial index, it only took three comparisons to find the desired polygon.

Because of their hierarchical nature, spatial indexes scale extremely well and can search through many tens of thousands of features using only a handful of bounding-box comparisons. And, because every geometry is reduced to a simple bounding box, spatial indexes can support any type of geometry, not just polygons.

Spatial indexes are not limited to only searching for enclosed coordinates; they can be used for all sorts of spatial comparisons and for spatial joins. We will be working with spatial indexes extensively throughout this book.
Introducing PostGIS

In this book, we will be working with PostGIS. PostGIS is one of the most popular and powerful geospatial databases and has the bonus of being open source and freely available. PostGIS itself is actually an extension to the PostgreSQL relational database system—to use PostGIS from your Python programs, you first have to install and set up PostgreSQL, then install the PostGIS extension, and then finally install the psycopg2 database adapter for Python. The following illustration shows how all these pieces fit together:

Note that PostgreSQL is often referred to as Postgres. We will regularly use this more colloquial name throughout this book.

PostGIS allows you to store and query against various types of spatial data, including points, lines, polygons, and geometry collections. PostGIS provides two different types of spatial fields that can be used to store spatial data:

- The **geometry** field holds spatial data that is assumed to be in a projected coordinate system. All calculations and queries for geometry fields assume that the spatial data has been projected onto a flat Cartesian plane. This makes the calculations much simpler, but it will only work reliably if the spatial data is in a projected coordinate system.

- The **geography** field holds spatial data that uses geodetic (unprojected) coordinates. Calculations and queries against geography fields assume that the data is in angular units (that is, latitude and longitude values), using sophisticated mathematics to calculate lengths and areas using a spheroid model of the earth.
Because the mathematics involved is much more complicated, not all spatial functions are available for geography fields, and the operations often take a lot longer. However, geography fields are much easier to use if your spatial data uses an unprojected coordinate system such as WGS84.

Let's go ahead and install PostGIS onto your computer and then look at how we can use PostGIS to create and work with a spatial database using Python.

**Installing PostgreSQL**

PostgreSQL is an extremely powerful open source relational database system. The main web site for Postgres can be found at [http://postgresql.org](http://postgresql.org). How you install the Postgres database will depend on which operating system your computer is running:

- For Linux, follow the instructions on the PostgreSQL download page [http://postgresql.org/download](http://postgresql.org/download) to install Postgres onto your computer. Choose the appropriate link for your Linux distribution and you will be presented with the corresponding installation instructions.
- For Mac OS X, you can download an installer for Postgres from the KyngChaos web site [http://www.kyngchaos.com/software/postgres](http://www.kyngchaos.com/software/postgres). Make sure you don't download the client-only version, as you'll need the Postgres server. Once it has been downloaded, open the disk image and double click on the PostgreSQL.pkg package file to install Postgres into your computer.
- For Microsoft Windows, you can download an installer for Postgres from [http://enterprisedb.com/products-services-training/pgdownload](http://enterprisedb.com/products-services-training/pgdownload). Select the appropriate installer for your version of Windows (32 or 64 bit), download the installer file, then simply double click on the installer and follow the instructions.

Once you have installed Postgres, you can check whether it is running by typing `psql` into a terminal or command-line window and pressing the `Return` key. All going well, you should see the Postgres command line:

```
psql (9.4.4)
Type "help" for help.
```

```
postgres=#
```
If the `psql` command-line client can't be found, you may have to add it to your path. For example, on a Mac, you can edit your `.bash_profile` file and add the following:

```
export PATH="$PATH:/usr/local/pgsql/bin"
```

If the `psql` command complains about user authentication, you may need to identify the user account to use when connecting to Postgres, for example:

```
% psql -U postgres
```

Many Postgres installations have a `postgres` user, which you need to select with the `-U` command-line option when accessing the database. Alternatively, you may need to use `sudo` to run `psql` as root, or open a command prompt as an administrator if you are running Microsoft Windows.

To exit the Postgres command-line client, type `\q` and press `Return`.

### Installing PostGIS

Our next task is to install the PostGIS spatial extension for Postgres. The main web site for PostGIS can be found at [http://postgis.net](http://postgis.net). Once again, how you install PostGIS depends on which operating system you are running:

- For Linux-based computers, follow the instructions on the PostGIS installation page ([http://postgis.net/install](http://postgis.net/install))
- For Mac OS X, you should download and run the PostGIS installer from the KyngChaos web site ([http://kyngchaos.com/software/postgres](http://kyngchaos.com/software/postgres))

Note that this PostGIS installer requires the GDAL Complete package, which you should have already installed while working through Chapter 2, GIS.

- For Microsoft Windows, you can download an installer from [http://download.osgeo.org/postgis/windows](http://download.osgeo.org/postgis/windows)

To check whether PostGIS has been successfully installed, try typing the following commands into your terminal window:

```
% createdb test_database
% psql -d test_database -c "CREATE EXTENSION postgis;"
% dropdb test_database
```
Don't forget to add the -U postgres option, or use sudo for each of these commands, if you need to run Postgres under a different user account.

The first command creates a new database, the second one enables the PostGIS extension for that database, and the third command deletes the database again. If this sequence of commands runs without any errors, then your PostGIS installation (and Postgres itself) is set up and running correctly.

## Installing psycopg2

psycopg2 is the Python database adapter for Postgres. This is the Python library you use to access Postgres from within your Python programs. The main web site for psycopg2 can be found at http://initd.org/psycopg.

As usual, how you install psycopg2 will vary depending on which operating system you are using:

- For Linux, you will need to install psycopg2 from source. For instructions on how to do this, refer to http://initd.org/psycopg/docs/install.html.
- For a Mac OS X machine, you can use pip, the Python package manager, to install psycopg2 from the command line:

  ```
  pip install psycopg2
  ```

  Note that you will need to have the Xcode command-line tools installed so that psycopg2 can compile.


To check whether your installation worked, start up your Python interpreter and type the following:

```python
>>> import psycopg2
```

If psycopg2 was installed correctly, you should see the Python interpreter prompt reappear without any error messages, as shown in this example. If an error message does appear, you may need to follow the troubleshooting instructions on the psycopg2 web site.
Setting up a database

Now that we have installed the necessary software, let's see how we can use PostGIS to create and set up a spatial database. We will start by creating a Postgres user account, creating a database, and setting up the user to access that database, and then we will enable the PostGIS spatial extension for our database.

Creating a Postgres user account

Our first task is to set up a Postgres user, who will own the database we create. While you might have a user account on your computer that you use for logging in and out, the PostgreSQL user is completely separate from this account and is used only within Postgres. You can set up a PostgreSQL user with the same name as your computer username, or you can give it a different name if you prefer.

Note that a user is sometimes referred to as a "role" in the Postgres manual.

To create a new PostgreSQL user, type the following command:

```bash
% createuser -P <username>
```

Obviously, replace `<username>` with whatever name you want to use for your new user. You may also need to add the `-U postgres` option or use `sudo` for these commands if you need to run Postgres under a different user account.

The `-P` command-line option tells Postgres that you want to enter a password for this new user. Don't forget the password that you enter, as you will need it when you try to access your database.

Creating a database

You next need to create the database you want to use for storing your spatial data. Do this using the `createdb` command:

```bash
% createdb <dbname>
```

Make sure you replace `<dbname>` with the name of the database you wish to create. Once again, add `-U postgres` or use `sudo` if required.
Allowing the user to access the database
To let the user access this new database, type the following command into a terminal window, adding \texttt{-U postgres} if necessary:
\[
\% \texttt{psql} -c "\texttt{GRANT ALL PRIVILEGES ON DATABASE <dbname> TO <user>;}"\]

Obviously, replace \texttt{<dbname>} with the name of your database and \texttt{<user>} with the name of your new user. Also, remember to add \texttt{-U postgres} or use \texttt{sudo} if you need to.

Spatially enable the database
So far, we have created a Postgres database and an associated user. Our database is just a regular database; to turn it into a spatial database, we have to enable the PostGIS extension for it. To do this, type the following into a terminal window, replacing \texttt{<dbname>} with the name of your new database:
\[
\% \texttt{psql} -d <dbname> -c "\texttt{CREATE EXTENSION postgis;}"\]

Using PostGIS
Now that we have a spatial database, let's see how to access it from Python. Using \texttt{psycopg2} to access a spatial database from Python is quite straightforward. For example, the following code shows how to connect to the database and issue a simple query:
```python
import psycopg2

connection = psycopg2.connect(database="...", user="...",
password="...")
cursor = connection.cursor()
cursor.execute("SELECT id,name FROM cities WHERE pop>100000")
for row in cursor:
    print(row[0], row[1])
```

The \texttt{psycopg2.connect()} statement opens up a connection to the database using the database name, user, and password you set up when you created and configured the database. Once you have a database connection, you then create a \texttt{Cursor} object against which you can execute queries. You can then retrieve the matching data, as shown in this example.
Let's use psycopg2 to store the World Borders Dataset into a spatial database table and then perform some simple queries against that data. Place a copy of the World Borders Dataset into a suitable directory, and create a new Python program called postgis_test.py inside the same directory. Enter the following into your program:

```python
import psycopg2
from osgeo import ogr

cursor = connection.cursor()
```

Don't forget to replace the `<dbname>`, `<user>`, and `<password>` values with the name of the database, the user account, and the password you set up earlier.

So far, we have simply opened a connection to the database. Let's create a table to hold the contents of the World Borders Dataset. To do this, add the following to the end of your program:

```sql
cursor.execute("DROP TABLE IF EXISTS borders")
cursor.execute("CREATE TABLE borders (" +
  "id SERIAL PRIMARY KEY," +
  "name VARCHAR NOT NULL," +
  "iso_code VARCHAR NOT NULL," +
  "outline GEOGRAPHY)"
)
cursor.execute("CREATE INDEX border_index ON borders " +
  "USING GIST(outline)"
)
cursor.execute("DROP TABLE IF EXISTS borders")
cursor.execute("CREATE TABLE borders (" +
  "id SERIAL PRIMARY KEY," +
  "name VARCHAR NOT NULL," +
  "iso_code VARCHAR NOT NULL," +
  "outline GEOGRAPHY)"
)
cursor.execute("CREATE INDEX border_index ON borders " +
  "USING GIST(outline)"
)
cursor.execute("DROP TABLE IF EXISTS borders")
cursor.execute("CREATE TABLE borders (" +
  "id SERIAL PRIMARY KEY," +
  "name VARCHAR NOT NULL," +
  "iso_code VARCHAR NOT NULL," +
  "outline GEOGRAPHY)"
)
cursor.execute("CREATE INDEX border_index ON borders " +
  "USING GIST(outline)"
)
cursor.execute("DROP TABLE IF EXISTS borders")
cursor.execute("CREATE TABLE borders (" +
  "id SERIAL PRIMARY KEY," +
  "name VARCHAR NOT NULL," +
  "iso_code VARCHAR NOT NULL," +
  "outline GEOGRAPHY)"
)
cursor.execute("CREATE INDEX border_index ON borders " +
  "USING GIST(outline)"
)
cursor.execute("DROP TABLE IF EXISTS borders")
cursor.execute("CREATE TABLE borders (" +
  "id SERIAL PRIMARY KEY," +
  "name VARCHAR NOT NULL," +
  "iso_code VARCHAR NOT NULL," +
  "outline GEOGRAPHY)"
)
cursor.execute("CREATE INDEX border_index ON borders " +
  "USING GIST(outline)"
)
cursor.execute("DROP TABLE IF EXISTS borders")
cursor.execute("CREATE TABLE borders (" +
  "id SERIAL PRIMARY KEY," +
  "name VARCHAR NOT NULL," +
  "iso_code VARCHAR NOT NULL," +
  "outline GEOGRAPHY)"
)
cursor.execute("CREATE INDEX border_index ON borders " +
  "USING GIST(outline)"
)
cursor.execute("DROP TABLE IF EXISTS borders")
cursor.execute("CREATE TABLE borders (" +
  "id SERIAL PRIMARY KEY," +
  "name VARCHAR NOT NULL," +
  "iso_code VARCHAR NOT NULL," +
  "outline GEOGRAPHY)"
)
cursor.execute("CREATE INDEX border_index ON borders " +
  "USING GIST(outline)"
)
```

As you can see, we delete the database table if it exists already so that we can rerun our program without it failing. We then create a new table named borders with four fields: an id, a name, and an iso_code, all of which are standard database fields, and a spatial geography field named outline. Because we're using a geography field, we can use this field to store spatial data that uses unprojected lat/long coordinates.

The third statement creates a spatial index on the outline. In PostGIS, we use the GIST index type to define a spatial index.

Finally, because Postgres is a transactional database, we have to commit the changes we have made using the `connection.commit()` statement.
Now that we've defined our database table, let's add some data into it. Using the techniques we learned earlier, we'll read through the contents of the World Borders Dataset shapefile. Here is the relevant code:

```python
shapefile = ogr.Open("TM_WORLD_BORDERS-0.3/TM_WORLD_BORDERS-0.3.shp")
layer = shapefile.GetLayer(0)
for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    name = feature.GetField("NAME")
    iso_code = feature.GetField("ISO3")
    geometry = feature.GetGeometryRef()
    wkt = geometry.ExportToWkt()
```

All of this should be quite straightforward. Our next task is to store this information into the database. To do this, we use the `INSERT` command. Add the following code to your program, inside the `for` loop:

```python
    cursor.execute("INSERT INTO borders (name, iso_code, outline) " +
                   "VALUES (%s, %s, ST_GeogFromText(%s))",
                   (name, iso_code, wkt))
```

Notice that `psycopg2` automatically converts standard Python data types such as numbers, strings, and date/time values to the appropriate format for inserting into the database. Following the Python DB-API standard, `%s` is used as a placeholder to represent a value, and that value is taken from the list supplied as the second parameter to the `execute()` function. In other words, the first `%s` is replaced with the value of the `name` variable, the second with the value of the `iso_code` variable, and so on.

Because `psycopg2` doesn't know about geometry data values, we have to convert the geometry into a WKT-format string and then use the `ST_GeogFromText()` function to convert that string back into a PostGIS geography object.

Now that we have imported all the data, we need to commit the changes we have made to the database. To do this, add the following statement to the end of your program (outside the `for` loop):

```python
    connection.commit()
```
If you run this program, it will take about 30 seconds to import all the data into
the database, but nothing else will happen. To prove that it worked, let's perform
a simple spatial query against the imported data—in this case, we want to find
all countries that are within 500 kilometers of Zurich, in Switzerland. Let's start
by defining the latitude and longitude for Zurich and the desired search radius in
meters. Add the following to the end of your program:

```python
start_long = 8.542
start_lat = 47.377
radius = 500000
```

We can now perform our spatial query using the `ST_DWithin()` query function,
like this:

```python
cursor.execute("SELECT name FROM borders WHERE ST_DWithin(" +
    "ST_MakePoint(%s, %s), outline, %s)",
    (start_long, start_lat, radius))
for row in cursor:
    print(row[0])
```

The `ST_DWithin()` function finds all records within the `borders` table that have an
outline within `radius` meters of the given lat/long value. Notice that we use the
`ST_MakePoint()` function to convert the latitude and longitude value to a Point
gallery, allowing us to compare the outline against the given point.

Running this program will import all the data and show us the list of countries that
are within 500 kilometers of Zurich:

- Luxembourg
- Monaco
- San Marino
- Austria
- Czech Republic
- France
- Germany
- Croatia
- Italy
- Liechtenstein
- Belgium
- Netherlands
- Slovenia
- Switzerland
While there is a lot more we could do, this program should show you how to use PostGIS to create a spatial database, insert data into it, and query against that data, all done using Python code.

**PostGIS documentation**

Because PostGIS is an extension to PostgreSQL and you use psycopg2 to access it, there are three separate sets of documentation you will need to refer to:

- The PostgreSQL manual: [http://postgresql.org/docs](http://postgresql.org/docs)
- The PostGIS manual: [http://postgis.refractions.net/docs](http://postgis.refractions.net/docs)
- The psycopg2 documentation: [http://initd.org/psycopg/docs](http://initd.org/psycopg/docs)

Of these, the PostGIS manual is probably going to be the most useful, and you will also need to refer to the psycopg2 documentation to find out the details of using PostGIS from Python. You will probably also need to refer to the PostgreSQL manual to learn the non-spatial aspects of using PostGIS, though be aware that this manual is huge and extremely complex, reflecting the complexity of PostgreSQL itself.

**Advanced PostGIS features**

PostGIS supports a number of advanced features which you may find useful:

- On-the-fly transformations of geometries from one spatial reference to another.
- The ability to edit geometries by adding, changing, and removing points and by rotating, scaling, and shifting entire geometries.
- The ability to read and write geometries in GeoJSON, GML, KML, and SVG formats, in addition to WKT and WKB.
- A complete range of bounding-box comparisons, including $A$ overlaps $B$, $A$ contains $B$, and $A$ is to the left of $B$. These comparison operators make use of spatial indexes to identify matching features extremely quickly.
- Proper spatial comparisons between geometries, including intersection, containment, crossing, equality, overlap, touching, and so on. These comparisons are done using the true geometry rather than just their bounding boxes.
- Spatial functions to calculate information such as the area, centroid, closest point, distance, length, perimeter, shortest connecting line, and so on. These functions take into account the geometry’s spatial reference, if known.
PostGIS has a well-deserved reputation for being a geospatial powerhouse. While it is not the only freely available spatial database, it is easily the most powerful and useful, and we will be using it extensively throughout this book.

**Recommended best practices**

In this section, we will look at a number of practical things you can do to ensure that your geospatial databases work as efficiently and effectively as possible.

**Best practice: use the database to keep track of spatial references**

As we've seen in earlier chapters, different sets of geospatial data use different coordinate systems, datums, and projections. Consider, for example, the following two geometry objects:

The geometries are represented as a series of coordinates, which are nothing more than numbers. By themselves, these numbers aren't particularly useful—you need to position these coordinates onto the earth's surface by identifying the spatial reference (coordinate system, datum, and projection) used by the geometry. In this case, the Polygon is using unprojected lat/long coordinates in the WGS84 datum, while the LineString is using coordinates defined in meters using the UTM zone 12N projection. Once you know the spatial reference, you can place the two geometries onto the earth's surface. This reveals that the two geometries actually overlap, even though the numbers they use are completely different:
In all but the simplest databases, it is recommended that you store the spatial reference for each feature directly in the database itself. This makes it easy to keep track of which spatial reference is used by each feature. It also allows the queries and database commands you write to be aware of the spatial reference and enables you to transform geometries from one spatial reference to another as necessary in your spatial queries.

Spatial references are generally referred to using a simple integer value called a **spatial reference identifier (SRID)**. While you could choose arbitrary SRID values to represent various spatial references, it is strongly recommended that you use the **European Petroleum Survey Group (EPSG)** numbers as standard SRID values. Using this internationally recognized standard makes your data interchangeable with other databases and allows tools such as OGR and Mapnik to identify the spatial reference used by your data.

To learn more about EPSG numbers, and SRID values in general, refer to [http://epsg-registry.org](http://epsg-registry.org).

PostGIS automatically creates a table named `spatial_ref_sys` to hold the available set of SRID values. This table comes preloaded with a list of over 3,000 commonly used spatial references, all identified by EPSG number. Because the SRID value is the primary key for this table, tools that access the database can refer to this table to perform on-the-fly coordinate transformations using the **PROJ.4** library.

When you create a table, you can specify both the type of geometry (or geography) data to store and the SRID value to use for this data. Here’s an example:

```sql
CREATE TABLE test (outline GEOMETRY(LINestring, 2193))
```

When defined in this way, the table will only accept geometries of the given type and with the given spatial reference.

When inserting a record into the table, you can also specify the SRID, like this:

```sql
INSERT INTO test (outline) VALUES (ST_GeometryFromText(wkt, 2193))
```

While the SRID value is optional, you should use this wherever possible to tell the database which spatial reference your geometry is using. In fact, PostGIS requires you to use the correct SRID value if a column has been set up to use a particular SRID. This prevents you from accidentally mixing spatial references within a table.
Best practice: use the appropriate spatial reference for your data

Whenever you import spatial data into your database, it will be in a particular spatial reference. This doesn't mean, though, that it has to stay in that spatial reference. In many cases, it will be more efficient and accurate to transform your data into the most appropriate spatial reference for your particular needs. Of course, what you consider appropriate depends on what you want to achieve.

When you use the geometry field type, PostGIS assumes that your coordinates are projected onto a Cartesian plane. If you use this field type to store unprojected coordinates (latitude and longitude values) in the database, you will be limited in what you can do. Certainly, you can use unprojected geographic coordinates in a database to compare two features (for example, to see whether one feature intersects with another), and you will be able to store and retrieve geospatial data quickly. However, any calculation that involves area or distance will be all but meaningless.

Consider, for example, what would happen if you asked PostGIS to calculate the length of a LineString geometry stored in a geometry field:

```sql
# SELECT ST_Length(geom) FROM roads WHERE id=9513;

ST_Length(geom)
-----------------
0.364491142657260
```

This "length" value is in decimal degrees, which isn't very useful. If you do need to perform length and area calculations on your geospatial data (and it is likely that you will need to do this at some stage), you have three options:

- Use a geography field to store the data
- Transform the features into projected coordinates before performing the length or distance calculation
- Store your geometries in projected coordinates from the outset

Let's consider each of these options in more detail.
Option 1: Using GEOGRAPHY fields
While the GEOGRAPHY field type is extremely useful and allows you to work directly with unprojected coordinates, it does have some serious disadvantages. In particular:

- Performing calculations on unprojected coordinates takes approximately an order of magnitude longer than performing the same calculations using projected (Cartesian) coordinates
- The GEOGRAPHY type only supports lat/long values on the WGS84 datum (SRID 4326)
- Many of the functions available for projected coordinates are not yet supported by the GEOGRAPHY type

Despite this, using GEOGRAPHY fields is an option you may want to consider.

Option 2: Transforming features as required
Another possibility is to store your data in unprojected lat/long coordinates and transform the coordinates into a projected coordinate system before you calculate the distance or area. While this will work, and will give you accurate results, you should beware of doing this because you may well forget to transform the data to a projected coordinate system before making the calculation. In addition, performing on-the-fly transformations of large numbers of geometries is very time consuming. Despite these problems, there are situations where storing unprojected coordinates makes sense. We will look at this shortly.

Option 3: Transforming features from the outset
Because transforming features from one spatial reference to another is rather time consuming, it often makes sense to do this once, at the time you import your data, and store it in the database already converted to a projected coordinate system.

By doing this, you will be able to perform your desired spatial calculations quickly and accurately. However, there are situations where this is not the best option, as we will see in the next section.

When to use unprojected coordinates
As we saw in Chapter 2, GIS, projecting features from the three-dimensional surface of the earth onto a two-dimensional Cartesian plane can never be done perfectly. It is a mathematical truism that there will always be errors in any projection.
Different map projections are generally chosen to preserve values such as distance or area for a particular portion of the earth's surface. For example, the Mercator projection is accurate at the tropics but distorts features closer to the poles.

Because of this inevitable distortion, projected coordinates work best when your geospatial data only covers a part of the earth's surface. If you are only dealing with data for Austria, then a projected coordinate system will work very well indeed. But if your data includes features in both Austria and Australia, then using the same projected coordinates for both sets of features will once again produce inaccurate results.

For this reason, it is generally best to use a projected coordinate system for data that covers only a part of the earth's surface, but unprojected coordinates will work best if you need to store data covering large parts of the earth.

Of course, using unprojected coordinates leads to problems of its own, as we discussed earlier. This is why it is recommended that you use the appropriate spatial reference for your particular needs; what is appropriate for you depends on what data you need to store and how you intend to use it.

The best way to find out what is appropriate would be to experiment: try importing your data in both spatial references, and write some test programs to work with the imported data. That will tell you which is the faster and easier spatial reference to work with, rather than having to guess.

**Best practice: avoid on-the-fly transformations within a query**

Imagine that you have a `cities` table with a `geom` column containing `POLYGON` geometries in UTM 12N projection (EPSG number 32612). Being a competent geospatial developer, you have set up a spatial index on this column.

Now, imagine that you have a variable named `pt` that holds a `POINT` geometry in unprojected WGS84 coordinates (EPSG number 4326). You might want to find the city that contains this point, so you issue the following reasonable-looking query:

```sql
SELECT * FROM cities WHERE
  ST_Contains(ST_Transform(geom, 4326), pt);
```
This will give you the right answer, but it will take an extremely long time. Why? Because the `ST_Transform(geom, 4326)` expression is converting every polygon geometry in the table from UTM 12N to long/lat WGS84 coordinates before the database can check to see whether the point is inside the geometry. The spatial index is completely ignored, as it is in the wrong coordinate system.

Compare this with the following query:

```sql
SELECT * FROM cities WHERE
  Contains(geom, Transform(pt, 32612));
```

A very minor change, but a dramatically different result. Instead of taking hours, the answer should come back almost immediately. Can you see why? Since the `pt` variable does not change from one record to the next, the `ST_Transform(pt, 32612)` expression is being called just once, and the `ST_Contains()` call can then make use of your spatial index to quickly find the matching city.

The lesson here is simple: be aware of what you are asking the database to do, and make sure you structure your queries to avoid on-the-fly transformations of large numbers of geometries.

### Best practice: don't create geometries within a query

While we are discussing database queries that can cause the database to perform a huge amount of work, consider the following (where `poly` is a polygon):

```sql
SELECT * FROM cities WHERE
  NOT ST_IsEmpty(ST_Intersection(outline, poly));
```

In a sense, this is perfectly reasonable: identify all cities that have a non-empty intersection between the city's outline and the given polygon. And the database will indeed be able to answer this query—it will just take an extremely long time to do so. Hopefully, you can see why: the `ST_Intersection()` function creates a new geometry out of two existing geometries. This means that for every row in the database table, a new geometry is created and is then passed to `ST_IsEmpty()`. As you can imagine, these types of operations are extremely inefficient. To avoid creating a new geometry each time, you can rephrase your query like this:

```sql
SELECT * FROM cities WHERE ST_Intersects(outline, poly);
```
While this example may seem obvious, there are many cases where spatial developers have forgotten this rule and have wondered why their queries were taking so long to complete. A common example is using the `ST_Buffer()` function to see whether a point is within a given distance of a polygon, like this:

```
SELECT * FROM cities WHERE
    ST_Contains(ST_Buffer(outline, 100), pt);
```

Once again, this query will work, but it will be painfully slow. A much better approach would be to use the `ST_DWithin()` function:

```
SELECT * FROM cities WHERE ST_DWithin(outline, pt, 100);
```

As a general rule, remember that you never want to call any function that returns a `GEOMETRY` or `GEOGRAPHY` object within the `WHERE` portion of a `SELECT` statement.

**Best practice: use spatial indexes appropriately**

Just like ordinary database indexes can make an immense difference to the speed and efficiency of your database, spatial indexes are also an extremely powerful tool for speeding up your database queries. Like all powerful tools, though, they have their limits:

- If you don't explicitly define a spatial index, the database can't use it. Conversely, if you have too many spatial indexes, the database will slow down because each index needs to be updated every time a record is added, updated, or deleted. Thus, it is crucial that you define the right set of spatial indexes: index the information you are going to search on, and nothing more.

- Because spatial indexes work on the geometries' bounding boxes, they can only tell you which bounding boxes actually overlap or intersect; they can't tell you whether the underlying points, lines, or polygons have this relationship. Thus, they are really only the first step in searching for the information you want. Once the possible matches have been found, the database still needs to check the geometries one at a time.
Spatial indexes are most efficient when dealing with lots of relatively small geometries. If you have large polygons consisting of many thousands of vertices, a polygon's bounding box is going to be so large that it will intersect with lots of other geometries, and the database will have to revert to doing full polygon calculations rather than just using the bounding box. If your geometries are huge, these calculations can be very slow indeed—the entire polygon will have to be loaded into memory and processed one vertex at a time. If possible, it is generally better to split large geometries (and in particular, large Polygons and MultiPolygons) into smaller pieces so that the spatial index can work with them more efficiently.

Best practice: know the limits of your database's query optimizer

When you send a query to the database, it automatically attempts to optimize the query to avoid unnecessary calculations and make use of any available indexes. For example, if you issued the following (non-spatial) query, the database would know that `Concat("John ","Doe")` yields a constant, and so would only calculate it once before issuing the query. It would also look for a database index on the `name` column and use it to speed up the operation.

```
SELECT * FROM people WHERE name=Concat("John ","Doe");
```

This type of query optimization is very powerful, and the logic behind it is extremely complex. In a similar way, spatial databases have a spatial query optimizer that looks for ways to precalculate values and make use of spatial indexes to speed up the query. For example, consider this spatial query from the previous section:

```
select * from cities where ST_DWithin(outline, pt, 12.5);
```

In this case, the PostGIS function `ST_DWithin()` is given one geometry taken from a table (`outline`) and a second geometry that is specified as a fixed value (`pt`), along with a desired distance (12.5 "units", whatever that means in the geometry's spatial reference). The query optimizer knows how to handle this efficiently, by first precalculating the bounding box for the fixed geometry plus the desired distance (`pt ±12.5`) and then using a spatial index to quickly identify the records that may have their `outline` geometry within that extended bounding box.

While there are times when the database's query optimizer seems to be capable of magic, there are many other times when it is incredibly stupid. Part of the art of being a good database developer is to have a keen sense of how your database's query optimizer works, when it doesn't, and what to do about it.
The PostGIS query optimizer looks at both the query itself and at the contents of the database to see how the query can be optimized. In order to work well, the PostGIS query optimizer needs to have up-to-date statistics on the databases’ contents. It then uses a sophisticated genetic algorithm to determine the most effective way to run a particular query.

Because of this approach, you need to regularly run the VACUUM ANALYZE command, which gathers statistics on the database so that the query optimizer can work as effectively as possible. If you don’t run VACUUM ANALYZE, the optimizer simply won’t be able to work.

Here is how you can run this command from Python:

```python
import psycopg2

connection = psycopg2.connect("dbname=... user=...")
cursor = connection.cursor()

old_level = connection.isolation_level
connection.set_isolation_level(0)
cursor.execute("VACUUM ANALYZE")
connection.set_isolation_level(old_level)
```

Don’t worry about the isolation_level logic here; it just allows you to run the VACUUM ANALYZE command from Python using the transaction-based psycopg2 adapter.

It is possible to set up an "autovacuum daemon" that runs automatically after a given period of time or after a table’s contents have changed enough to warrant another vacuum. Setting up an autovacuum daemon is beyond the scope of this book.

Once you have run the VACUUM ANALYZE command, the query optimizer will be able to start optimizing your queries. To see how the query optimizer works, you can use the EXPLAIN SELECT command. For example:

```
psql> EXPLAIN SELECT * FROM cities
    WHERE ST_Contains(geom,pt);
```

**QUERY PLAN**
Spatial Databases

Seq Scan on cities  (cost=0.00..7.51 rows=1 width=2619)

Filter: ((geom && '010100000000000000000000000000000000000000'::geometry) AND _st_ contains(geom, '010100000000000000000000000000000000000000'::geometry))

(2 rows)

Don't worry about the Seq Scan part; there are only a few records in this table, so PostGIS knows that it can scan the entire table faster than it can read through an index. When the database gets bigger, it will automatically start using the index to quickly identify the desired records.

The cost= part is an indication of how much this query will cost, measured in arbitrary units that by default are relative to how long it takes to read a page of data from disk. The two numbers represent the start up cost (how long it takes before the first row can be processed) and the estimated total cost (how long it would take to process every record in the table). Since reading a page of data from disk is quite fast, a total cost of 7.51 is very quick indeed.

The most interesting part of this explanation is the Filter. Let's take a closer look at what the EXPLAIN SELECT command tells us about how PostGIS will filter this query. Consider the first part:

(geom && '010100000000000000000000000000000000000000'::geometry)

This makes use of the && operator, which searches for matching records using the bounding box defined in the spatial index. Now consider the second part of the filter condition:

_st_contains(geom, '010100000000000000000000000000000000000000'::geometry)

This uses the ST_Contains() function to identify the exact geometries that contain the desired point. This two-step process (first filtering by bounding box, then by the geometry itself) allows the database to use the spatial index to identify records based on their bounding boxes and then check the potential matches by doing an exact scan on the geometry itself. This is extremely efficient, and as you can see, PostGIS does this for us automatically, resulting in a quick but also accurate search for geometries that contain a given point.
Summary

In this chapter, we took an in-depth look at the concept of storing spatial data in a database, using the freely available PostGIS database toolkit. We learned that spatial databases differ from ordinary relational databases in that they directly support spatial data types and use spatial indexes to perform queries and joins on spatial data. We saw that spatial indexes make use of the geometries' bounding boxes to quickly compare and find geometries based on their position in space.

We then looked at the PostGIS spatial extension to PostgreSQL and how the psycopg2 library can be used to access PostGIS spatial databases using Python. After installing the necessary software, we configured a spatial database and used psycopg2 to create the necessary database tables, import a set of spatial data, and perform useful queries against that data.

Next, we looked at some of the recommended best practices for working with spatial databases. We saw that it is important to store a spatial reference ID along with the data and looked at how you can select an appropriate spatial reference for your application.

We then looked at some of the mistakes that can kill the performance of a geospatial database, including creating geometries and performing transformations on the fly and using spatial indexes inappropriately so that the database cannot use them.

Finally, we learned about the PostGIS query optimizer and how we can use the EXPLAIN command to see exactly how PostGIS will execute a spatial query.

In the next chapter, we will learn how to use the Mapnik library to convert raw geospatial data into good-looking map images.
Where to buy this book

You can buy Python Geospatial Development Third Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.