PostGIS in Action
Second Edition

Regina O. Obe
Leo S. Hsu

Foreword by Paul Ramsey
**brief contents**

**PART 1  INTRODUCTION TO POSTGIS ........................................1**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>What is a spatial database? 3</td>
</tr>
<tr>
<td>2</td>
<td>Spatial data types 27</td>
</tr>
<tr>
<td>3</td>
<td>Spatial reference system considerations 60</td>
</tr>
<tr>
<td>4</td>
<td>Working with real data 82</td>
</tr>
<tr>
<td>5</td>
<td>Using PostGIS on the desktop 113</td>
</tr>
<tr>
<td>6</td>
<td>Geometry and geography functions 140</td>
</tr>
<tr>
<td>7</td>
<td>Raster functions 175</td>
</tr>
<tr>
<td>8</td>
<td>PostGIS TIGER geocoder 203</td>
</tr>
<tr>
<td>9</td>
<td>Geometry relationships 218</td>
</tr>
</tbody>
</table>

**PART 2  PUTTING POSTGIS TO WORK ....................................245**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Proximity analysis 247</td>
</tr>
<tr>
<td>11</td>
<td>Geometry and geography processing 261</td>
</tr>
<tr>
<td>12</td>
<td>Raster processing 287</td>
</tr>
<tr>
<td>13</td>
<td>Building and using topologies 309</td>
</tr>
<tr>
<td>14</td>
<td>Organizing spatial data 337</td>
</tr>
<tr>
<td>15</td>
<td>Query performance tuning 369</td>
</tr>
</tbody>
</table>
PART 3 USING POSTGIS WITH OTHER TOOLS ..........................407

16  ▪ Extending PostGIS with pgRouting and procedural languages  409
17  ▪ Using PostGIS in web applications  442
Welcome to *PostGIS in Action, Second Edition*. PostGIS is a spatial database extender for the PostgreSQL database management system. This book will teach you the fundamentals of spatial databases in general, key concepts in geographic information systems (GIS), and more specifically how to configure, load, and query a PostGIS-enabled database. You’ll learn how to perform actions with single lines of SQL code that you thought were possible only with a desktop GIS system. By using spatial SQL, much of the heavy lifting that would require many manual steps in desktop GIS tools can be scripted and automated.

This book is divided into three sections and four appendixes. Part 1 covers the fundamentals of spatial databases, GIS, and working with spatial data. Although part 1 is focused on PostGIS, many of the concepts you’ll learn in part 1 are equally applicable to other spatial relational databases.

Chapter 1 covers the fundamentals of spatial databases and what you can do with a spatially enabled database that you can’t do with a standard relational database. It also introduces features that are fairly unique to PostGIS. It concludes with a fast-paced example of loading fast-food restaurant longitude/latitude data and converting it to geometric points, loading road data from Esri shapefiles, and doing spatial summaries by joining these two sets of data.

Chapter 2 covers all the spatial types that PostGIS has to offer. You’ll learn how to create these using various functions and learn about concepts unique to each spatial type.

Chapter 3 is an introduction to spatial reference systems, and we’ll explain the concepts behind them, why they’re important for working with geometry, raster, and topology, and how to work with them.
Chapter 4 covers how to load spatial data into PostGIS using packaged tools as well as additional third-party open source tools. You’ll learn how to load geometry and geography data using the shp2pgsql command-line tool packaged with all PostGIS distributions, as well as the shp2pgsql-gui GUI loader/exporter that’s packaged with some desktop distributions of PostGIS. You’ll also learn how to load raster data using the PostGIS-packaged raster2pgsql command-line tool and how to import and export both raster and vector data of various formats using the GDAL/OGR suite. You’ll also learn how to load OpenStreetMap data using the commonly available osm2pgsql command-line tool.

Chapter 5 covers some of the more common, open source, desktop tools for viewing and querying PostGIS data.

Chapter 6 starts getting into the simpler core functions that are used with geometry and geography functions. These all take single geometry or geography objects and morph them or take text representations of them and convert them to PostGIS spatial objects.

Chapter 7 is an introduction to raster functions. It covers some functions for creating rasters, interrogating rasters, and setting pixel values.

Chapter 8 covers geocoding with the packaged PostGIS TIGER geocoder. You’ll learn how to load U.S. Census TIGER data using functions packaged with the TIGER geocoder. Once the data is loaded, you’ll learn how to use the packaged functions to normalize, geocode, and reverse geocode data.

Chapter 9 concludes this first part by introducing you to spatial relationships. Spatial relationships are most important when working with sets of data. In later sections of the book, we’ll use these concepts to do things like spatial joins.
What is a spatial database?

Most folks experience their first spatially enabled application when they see pushpins tacked onto points of interest on an interactive map. This provides a glimpse into the vast and varied field of geographic information systems (GIS).

We’ll begin this chapter with a pushpin model. As we demonstrate its limited usefulness, we’ll introduce the need for a spatial database—not just any database, but PostGIS. PostGIS is a spatial database extender for the PostgreSQL database management system. We’ll provide a brief introduction to the entire PostGIS suite and whet your appetite with an example that goes far beyond what you can accomplish with pushpins. The data and code used in this chapter can be found at http://www.postgis.us/chapter_01Edition_2.
1.1 Thinking spatially

Popular mapping sites such as OpenStreetMap, Google Maps, Bing Maps, MapQuest, and Yahoo have empowered people in many walks of life to answer the question “Where is something?” by displaying teardrop shapes on a gorgeously detailed, interactive map. No longer are we restricted to textual descriptions of “where,” like “Turn right at the supermarket and it’ll be the third house on the right with a mangy dog out front.” Nor are we faced with the frustrating problem of not being able to figure out our current location on a paper map.

Going beyond getting directions, organizations large and small have discovered that mapping can be a great resource for analyzing patterns in data. By plotting the addresses of pizza lovers, a national pizza chain can assess where to locate the next grand opening. Political organizations planning grassroots campaigns can easily see on a map where the undecided or unregistered voters are located and target their route walks accordingly. Even though the pushpin model offers unprecedented geographical insight, the reasoning that germinates from it is entirely on visual.

In the pizza example, the chain might be able to see the concentration of pizza lovers in a city by means of adding pushpins. But what if they need to differentiate pizza lovers by income level? If the chain has a gourmet offering, it would be a good idea to locate new restaurants in the midst of mid- to high-income pizza lovers. The pizza chain planners could use pushpins of different colors on an interactive map to indicate various income tiers, but the heuristic visual reasoning will now be much more complicated, as shown in figure 1.1. Not only do the planners need to look at the concentration of pushpins, they must also keep the varying colors or icons of the pins in mind. Add another variable to the map, like households with lactose-intolerant adults, and the problem overwhelms our feeble minds. Spatial databases come to the rescue. A spatial database has column data types specifically designed to store objects in space—these data types can be added to database tables. The information stored is usually geographic in nature, such as a point location or the boundary of a lake. The spatial database also provides special functions and indexes for querying and manipulating that data, which can be called from a query language such as Structured Query Language (SQL). A spatial database is often used as just a storage container for spatial data, but it can do much more than that. Although a spatial database need not be relational in nature, most are.

A spatial database gives you a storage tool, an analysis tool, and an organizing tool all in one. Presenting data visually isn’t a spatial database’s only goal. The pizza shop planners can store an infinite number of attributes of the pizza-loving household, including income level, number of children in the household, pizza-ordering history, and even religious preferences and cultural upbringing (as they relate to topping choices on a pizza). More important, the analysis need not be limited to the number of variables that can be juggled in the brain. The planners can make very specific requests, like “Give me a list of neighborhoods ranked by the number of high-income pizza lovers who have more than two children.” Furthermore, they can easily
incorporate additional data from varied sources, such as the location and rating of existing pizzerias from restaurant review sites or the health-consciousness level of various neighborhoods as identified by the local health commission. Their questions of the database could be as complicated as “Show me the region with the highest number of households where the average closest distance to any pizza parlor with a star-ranking below 5 is greater than 16 kilometers (10 miles). Oh, and toss out the health-conscious neighborhoods.”
Table 1.1 shows what the results of such a spatial query might look like.

<table>
<thead>
<tr>
<th>Region</th>
<th>Households</th>
<th>Restaurants</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region A</td>
<td>194</td>
<td>1</td>
<td>17.1 km</td>
</tr>
</tbody>
</table>

Suppose you aren’t a mapping user, but are more of a data user. You work with data day in and day out, never needing to plot anything on a map. You’re familiar with questions like, “Give me all the employees who live in Chicago,” or “Count up the number of customers in each postal code.” Suppose you have the latitude and longitude of all the employees’ addresses; you could ask questions like “Give me the average distance that each employee must travel to work.” This is the extent of the kind of spatial queries that you can formulate with conventional databases, where data types consist mainly of text, numbers, and dates.

Suppose the question posed is “Give me the number of houses within two miles of the coastline requiring evacuation in the event of a hurricane” or “How many households would be affected by the noise of a newly proposed runway?” Without spatial support, these questions would require you to collect or derive additional values for each data point. For the coastline question, you’d need to determine the distance from the beach, house by house. This could involve algorithms to find the shortest distance to fixed intervals along the coastline or require a series of SQL queries to order all the houses by proximity to the beach and then make a cut. With spatial support, all you need to do is reformulate the question slightly as “Find all houses within a two-mile radius of the coastline.” A spatially enabled database can intrinsically work with data types like coastlines (modeled as linestrings), buffer zones (modeled as polygons), and beach houses (modeled as points).

As with most things in life worth pursuing, nothing comes without some effort. You’ll need to climb a gentle learning curve to tap into the power of spatial analysis. The good news is that unlike other good things in life, the database that we’ll introduce you to is completely free—moneywise.

If you’re able to figure out how to get data into your Google map, you’ll have no problem taking the next step. If you can write queries in non-spatially enabled databases, we’ll open your eyes and mind to something beyond the mundane world of numbers, dates, and strings. Let’s get started.

1.2 Introducing PostGIS

PostGIS is a free and open source library that spatially enables the free and open source PostgreSQL object-relational database management system (ORDBMS). We want you to choose PostgreSQL as your relational database and PostGIS as your spatial database extender for PostgreSQL.
1.2.1 Why PostGIS

PostGIS started as a project of Refractions Research and has since been adopted and improved on by government, public organizations, and private companies.

The power of PostGIS is enhanced by other supporting projects:

- **Proj4**—Provides projection support
- **Geometry Engine Open Source (GEOS)**—Advanced geometry-processing support
- **Geospatial Data Abstraction Library (GDAL)**—Provides many advanced raster-processing features
- **Computational Geometry Algorithms Library (CGAL/SFCGAL)**—Enables advanced 3D analysis

Most of these projects, including PostGIS, now fall under the umbrella of the Open Source Geospatial Foundation (OSGeo). The foundation of PostGIS is the PostgreSQL ORDBMS, which provides transactional support, gist index support for spatial objects, and a query planner out of the box. It’s a great testament to the power and flexibility of PostgreSQL that Refractions chose to build on top of PostgreSQL rather than on any other open source database.

**Standards Conformance**

PostGIS and PostgreSQL conform to industry standards more closely than most products. PostgreSQL supports many of the newer ANSI SQL features. PostGIS supports Open Geospatial Consortium (OGC) standards and the new SQL Multimedia Spec (SQL/MM) spatial standard. This means that you aren’t simply learning how to use a set of products; you’re garnering knowledge about industry standards that will help you understand other commercial and open source geospatial databases and mapping tools.

**What are OGC, OSGeo, ANSI SQL, and SQL/MM?**

OGC stands for Open Geospatial Consortium, and it’s the body that exists to standardize how geographic and spatial data is accessed and distributed. Towards that goal, they have numerous specifications that govern accessing geospatial data from web services, geospatial data delivery formats, and querying of geospatial data.

OSGeo stands for Open Source Geospatial Foundation, and it’s the body whose initiative is to fund, support, and market open source tools and free data for GIS. There’s some overlap between the OSGeo and OGC. Both strive to make GIS data and tools available to everyone, which means they’re both concerned about open standards.

You’ll also often hear the term American National Standards Institute (ANSI) or International Organization of Standardization (ISO) SQL. The ANSI/ISO SQL standards define general guidelines that SQL implementations should follow. These guidelines are often year-dated like ANSI SQL 92 and ANSI SQL:2011 and they build upon prior-year
If your data and your APIs implement standards supported by many kinds of software—Cadcorp, Safe FME, AutoCAD, Manifold, MapInfo, Esri ArcGIS, ogr2ogr/GDAL, OpenJUMP, QGIS, deegree, MapGuide, UMN MapServer, GeoServer, or even standard programming tools like PHP, Python, Perl, ASP.NET, SQL, or new emerging tools—then everyone can use the tools that they feel most comfortable with, or that fit their work processes, or that they can afford, and share information with one another. OSGeo tries to ensure that regardless of how small your pocketbook is, you can still afford to view and analyze GIS data. OGC and ANSI/ISO SQL try to enforce standards across all products so that regardless of how expensive your GIS platform is, you can still make your hard work available to everyone. This is especially important for government agencies whose salaries and tools are paid for with tax dollars; for students who have a lot of will, the intelligence to learn, and advanced technology but have small pockets; and even for smaller vendors who have a compelling offering for specific kinds of users but who are often snubbed by larger vendors because they can’t support (or lack access to) the private API standards of the big-name vendors.

PostGIS is supported by a vast number of GIS proprietary desktop and server tools. You can find a listing of some of these in appendix A, section A.3. PostGIS is also the preferred spatial relational database of most open source geospatial desktop and web-mapping server tools. We’ll cover some of the more common ones in chapters 5 and 17.

**PostGIS is powerful**

PostGIS provides many spatial operators, spatial functions, spatial data types, and spatial indexing enhancements to PostgreSQL. If you add to the mix the complementary features that PostgreSQL and other related projects provide, then you have a jam-packed powerhouse at your disposal that’s well suited for sophisticated GIS analysis and is a valuable tool for learning GIS.
You’ll be hard pressed to find the following features in other spatial databases:

- Functions to work with GeoJSON and Keyhole Markup Language (KML), allowing web applications to talk directly to PostGIS without the need for additional serializing schemes or translations
- Comprehensive geometry-processing functions that go far beyond basic geometric operations, including functions for fixing invalid geometries and for simplifying and deconstructing geometries
- Built-in 3D and topology support
- Over 150 seamless operations for working with vectors and rasters in tandem, as well as for converting between the two families

**GeoJSON and KML data formats**

Geographic JavaScript Object Notation (GeoJSON; [http://geojson.org](http://geojson.org)) and Keyhole Markup Language (KML; [http://en.wikipedia.org/wiki/Keyhole_Markup_Language](http://en.wikipedia.org/wiki/Keyhole_Markup_Language)) are two of the most popular vector formats used by web-mapping applications:

- GeoJSON is an extension of JSON that’s used for representing JavaScript objects. It adds to the JSON standard support for geographic objects.
- KML is an XML format developed by Keyhole (which was purchased by Google), first used in Google’s mapping products and later supported by various mapping APIs.

These are only two of the many formats that PostGIS can output.

**Built on top of PostgreSQL**

The major reason PostGIS was built on the PostgreSQL platform was the ease of extensibility PostgreSQL provided for building new types and operators and for controlling the index operators. PostgreSQL was designed to be extensible from the ground up.

PostgreSQL has a regal lineage that dates back almost to the dawn of relational databases. It’s a cousin of the Sybase and Microsoft SQL Server databases, because the people who started Sybase came from UC Berkeley and worked on the Ingres or PostgreSQL projects with Michael Stonebraker. Michael Stonebraker is considered by many to be the father of Ingres and PostgreSQL and to be one of the founding fathers of object-relational database management systems. The source code of Sybase SQL Server was later licensed to Microsoft to produce Microsoft SQL Server.

PostgreSQL’s claim to fame is that it’s the most advanced open source database in existence. It has the speed and functionality to compete with the popular commercial enterprise offerings, and it’s used to power databases terabytes in size. As time has moved on, new usability features have been added, making it not only the most advanced, but perhaps the most flexible relational database out there. For more details about the features of PostgreSQL and the key enhancements in newer versions that are lacking in most other databases (including expensive proprietary ones), please refer to appendix B.
PostgreSQL is becoming a one-size-fits-all database that doesn’t sacrifice the needs and wants of any database users. Most OS distributions carry a fairly new version that provides a quick and painless install process. Since the last edition of this book, cloud offerings have come on board that provide PostgreSQL with PostGIS out of the box. Some popular cloud versions of PostgreSQL that PostGIS users enjoy are CartoDB, Heroku PostgreSQL, and Amazon RDS for PostgreSQL.

**Free—as in money**
The starting package of Esri’s ArcGIS for Server is well over $10,000 and gets higher as you add more cores. Licenses for SQL Server start at $5,000 and can easily cost you $20,000 for a modest server. The commercial version of Oracle’s Spatial starts at $20,000 per core and doesn’t even include the cost of the Enterprise license.

Oracle Standard ships with Oracle Locator, but this elementary tool lacks most of the medium and advanced functionality that PostGIS offers. The comparable Oracle Spatial requires the purchase of Oracle Enterprise.

PostGIS is free. ’Nuff said.

**Free—as in freedom**
PostGIS and PostgreSQL are open source. PostGIS is under a GPL v2 license; PostgreSQL is under a BSD-style license, which means you can both see and modify the source code. If you find a feature missing, you can contribute a patch or pay a developer to add the feature. Adding features to PostGIS and PostgreSQL generally costs much less than the licensing costs for proprietary counterparts. If you discover a bug in PostGIS, you can fix it instead of just reporting it. You have more freedom to control your destiny with PostGIS and PostgreSQL than you do with comparable proprietary offerings. You can install them on as many servers as you want, and you aren’t limited by artificial restrictions on how many cores you can use.

The openness of PostGIS has spawned an explosion of user-contributed add-ons and community-funded features. These are the most notables ones to date: raster support, geodetic support, topology support, improved 3D support, faster spatial indexes, and Topologically Integrated Geographic Encoding and Referencing (TIGER) geocoder enhancements.

The release cycles for PostGIS and PostgreSQL are radically shorter than those of commercial offerings. With contributions from users, PostgreSQL evolves at a rate of one minor version per year and one micro version per two months, with bugs getting immediate attention. You don’t have to wait years in anticipation of features promised in subsequent releases. If you choose to live on the bleeding edge, you can even download a new build every other week.

### 1.2.2 Alternatives to PostGIS

Admittedly, PostGIS isn’t the only spatial database in use today. Early entrants were dominated by proprietary offerings, and PostGIS broke this mold. Successors to PostGIS are gravitating towards installations with lightweight footprints for use on mobile devices. We’re also beginning to see rudimentary spatial features in NoSQL databases like MongoDB, CouchDB, and Solr.
**Oracle Spatial**

Oracle was the one that started it all. In Oracle 7, joint development efforts with Canadian scientists gave birth to SDO (Spatial Data Option). In later releases, Oracle redubbed this lovechild Oracle Spatial.

Oracle Spatial isn’t available with lower-priced editions of Oracle. Only when you fork out the money for Oracle Enterprise Edition will you have the luxury of being able to buy the Oracle Spatial option.

Standard Oracle installations do come with something called Oracle Locator, which offers the basic geometry types, proximity functions, and some spatial aggregates, but it lacks processing features like union aggregation functions and intersection functions.

**Microsoft SQL Server**

Microsoft introduced spatial support in their SQL Server 2008 offering with its built-in Geometry and Geodetic Geography types and companion spatial functions. To Microsoft’s credit, you’ll get the same feature set with their express, standard, enterprise, and data center offerings. You may just be limited regarding database size, how many processors you can use, and what query plan features you’re allowed.

Microsoft’s spatial feature, except their curved and geodetic support, pales in comparison to PostGIS. Admittedly, Microsoft SQL Server 2012 has probably got the best curve and geodetic support of any database—it’s the only one to support curved geometries in geodetic space. But don’t expect to find numerous output/input functions, such as input/output for KML and JSON, or raster support, or the numerous processing functions that PostGIS has.

**SpatiaLite**

Our favorite kid on the block is SpatiaLite, which is an add-on to the open source SQLite portable database. SpatiaLite is especially interesting because it can be used as a low-end companion to PostGIS and other high-end, spatially enabled databases. It can run on an Android smartphone, and it can be used to create master/slave applications to provide basic lightweight spatial support for portable devices.

SpatiaLite also has a companion, RasterLite, that’s mostly focused on raster data storage and display, and so makes a great companion to the raster analysis that’s present in PostGIS. SpatiaLite and RasterLite also use many of the core libraries that PostGIS uses: GEOS, PROJ, and GDAL. This makes it an even more fitting companion to PostGIS, because many of the conventions are the same and much of the ecosystem around PostGIS also supports or is starting to support SpatiaLite/RasterLite.

What SpatiaLite lacks is a strong enterprise database behind it that allows for writing advanced functions and spatial aggregate functions. That’s why some spatial queries possible in PostGIS are harder to write or are not even possible in SpatiaLite.

SpatiaLite/SQLite stores data as a single file that’s easily transportable. This makes it less threatening to deploy for users new to databases or GIS, and easier to deploy as a lightweight offline database companion to a server-side database like PostGIS/PostgreSQL.
MySQL
MySQL has had elementary spatial support since version 4, but as a database MySQL is handicapped by its lack of a powerful SQL engine. Its primary audience is still developers who are looking for a database that will store something, rather than do something. Earlier MySQL spatial support made a fatal mistake of not providing indexing capabilities except on MyISAM tables—spatial queries rely heavily on indexing for speedy performance. In version 5.6, MySQL extended geometric operations to work beyond bounding boxes and also allowed spatial indexes on its InnoDB storage engine.

Oracle MySQL and other MySQL forks like MariaDB have made strides in the 5.6 variants by improving the performance of subqueries, but the query planner and SQL feature set in the MySQL family is still a kid when compared to the likes of PostgreSQL, SQL Server, and Oracle, so MySQL is not suitable for doing anything as complex as most spatial analysis.

ArcGIS by Esri
We must give a nod to Esri, which has long packaged its spatial database engine (SDE) with its ArcGIS for Server product. The SDE engine is integrated into the ArcGIS line of products and is often used to spatially enable or augment legacy or weak database products, such as Microsoft SQL Server 2005 and Oracle Locator.

Older versions of ArcGIS desktop required going through an SDE middle tier to get at the native offerings of your spatial database. Newer versions, starting around ArcGIS 10.0, allow for direct read-only access to PostGIS and other databases. By sidestepping the middleware, you’re free to use any version of PostGIS with ArcGIS desktop. Although their proprietary model doesn’t sit well with us, we must give them credit—a lot of credit, in fact—for being one of the first major companies to introduce GIS analysis to commercial and government organizations. They paved the way, but still stand in the way, for the rise of free and open source GIS.

1.2.3 Installing PostGIS
We encourage you to install the latest version of PostgreSQL and PostGIS—PostgreSQL 9.4 and PostGIS 2.1 at the time of writing. The introduction of the extension model in PostgreSQL 9.1 greatly simplified the installation of add-ons (such as PostGIS) to two steps. First, you locate and install the binaries for your particular OS into your PostgreSQL directories. Second, you individually enable the extensions for each database as needed. For instance, if you have ten databases on your server, but only two require PostGIS, you’d only enable PostGIS for the two.

PostGIS must be enabled in each database
One characteristic of PostgreSQL that confuses many people coming from other database systems is that custom extensions like PostGIS, hstore, PL/JavaScript, and PL/Python must be enabled in each database they will be used in. This isn’t the case for built-in types like Full-Text, XML, PostgreSQL 9.2+ JSON, and so on, which are always present.
Introducing PostGIS

Many of the popular Linux/Unix distributions include PostGIS 2.1 in their repository. Use `yum` or `apt-get` to install the binaries. For Mac users, there are a couple of popular distributions, all itemized on [http://postgis.net/install](http://postgis.net/install). For Windows, we recommend using the EnterpriseDB StackBuilder. Please refer to appendix A for more details on where to obtain binaries for your OS.

Once you’ve successfully installed the binaries, you can create a database with a command such as this:

```sql
CREATE DATABASE postgis_in_action;
```

You next enable PostGIS in your database by connecting to the database and running the single-line SQL command in the following listing. Enabling the extension rarely fails, but you may encounter dependency errors, especially if you have earlier versions of PostGIS floating around.

```sql
CREATE EXTENSION postgis;
```

Two popular tools come packaged with PostgreSQL: `psql` and `pgAdmin`. You use these tools to create databases, users, and compose queries.

Psq is strictly a command-line tool. If you don’t have a GUI, psql is your only option.

If you have the luxury of a graphical interface, we encourage you to use the more newbie-friendly pgAdmin. PgAdmin can be installed separately from PostgreSQL. You can find source code as well as precompiled binaries at the pgAdmin site, [http://www.pgadmin.org](http://www.pgadmin.org).

To enable extensions in pgAdmin, use the extension install section pictured in figure 1.2.

If postgis isn’t listed, you can install it by right-clicking the Extensions branch, choosing New Extension, and picking postgis from the menu. You should see postgis listed in the Add Extension menu if you installed the binaries and don’t have it already installed in your selected database.

**Listing 1.1  Enabling PostGIS in a database**

```sql
CREATE EXTENSION postgis;
```

**VERIFYING VERSIONS OF POSTGIS AND POSTGRESQL**

After a PostGIS install, do a quick verification of the version to make sure the installation succeeded. Execute the following query:

```sql
SELECT postgis_full_version();
```

If all is well, you should see the version of PostGIS, as well as the versions of the supporting GEOS, GDAL, PROJ, LIBXML, and LIBJSON libraries, as shown here:

```
POSTGIS="2.1.2 r12389"
GEOS="3.4.2-CAPI-1.8.2 r3924"
PROJ="Rel. 4.8.0, 6 March 2012"
GDAL="GDAL 1.10.0, released 2013/04/24"
LIBXML="2.7.8" LIBJSON="UNKNOWN" RASTER
```
What is a spatial database?

1.3 Spatial data types

There are four key spatial types offered by PostGIS: geometry, geography, raster, and topology. PostGIS has always supported the geometry type from its inception. It introduced support for geography in PostGIS 1.5. PostGIS 2.0 raised the bar more by incorporating raster and network topology support. Although PostGIS 2.1 introduces many more functions, perhaps the most important feature it provides is faster speed, particularly for raster and geography operations:

- **Geometry**—The planar type. This was the very first model and it’s still the most popular type that PostGIS supports. It’s the foundation of the other types. It uses the Cartesian math you learned about in high school geometry.
- **Geography**—The spheroidal geodetic type. Lines and polygons are drawn on the earth’s curved surface, so they’re curved rather than straight lines.
- **Raster**—The multiband cell type. Rasters model space as a grid of rectangular cells, each containing a numeric array of values.
- **Topology**—The relational model type. Topology models the world as a network of connected nodes, edges, and faces. Objects are composed of these and may share these with other objects. There are really two related concepts in topology—the network, which defines what elements each thing is composed of, and routing. PostGIS 2+ packages the network topology model, which is often just referred to as topology. Network topology ensures that when you change the edge of an object, other objects sharing that edge will change accordingly. Routing is commonly used with PostGIS via a long-supported add-on called pgRouting. Routing not only cares about connectedness, but also how costly that connectedness is. PgRouting is mostly used for building trip navigation applications (taking into account the cost of tolls or delays due to construction), but it can be used for any application where costs along a path are important. We’ll cover pgRouting in later chapters of this book.

These four types can coexist in the same database and even as separate columns in the same table. For example, you can have a geometry that defines the boundaries of a...
Spatial data types

1.3.1 Geometry type

In two dimensions, you can represent all geographical entities with three building blocks: points, linestrings, and polygons (see figure 1.3). For example, an interstate highway crossing the salt flats of Utah clearly jumps out as linestrings cutting through a polygon. A desolate gas station located somewhere along the interstate can be a point.

But you need not limit yourself to the macro dimensions of road atlases. Look around your home. Use rectangular polygons to represent rooms. The wiring and the piping running behind the walls would be linestrings. You can use either a point or a polygon to stand in for the dog house, depending on its size. Just by abstracting the landscape to 2D points, linestrings, and polygons, you have enough to model everything that could crop up on a map or a blueprint.

Don’t be overly concerned with the rigorous definition of the geometries. Questions such as how many angles will fit into a point, and what is the width of a linestring are best left for mathematicians and philosophers. To us, points, linestrings, and polygons are simplified models of reality. As such, they’ll never perfectly mimic the real thing. Also, don’t worry if you feel that we’re leaving out other geometries. Two good examples are beltways around a metropolis and hippodromes. The former could be well represented by circles; the latter by ellipses. You’ll do fine by approximating them using linestrings with many segments, and polygons with many edges.

The geometry type treats the world as a flat Cartesian grid. The mathematics behind the model requires nothing more than the analytic geometry you learned in high school. The geometry model is intuitively appealing and computationally speedy, but it suffers from one major shortcoming—the flat earth.

1.3.2 Geography type

The curvature of the earth comes into play when you’re modeling anything that extends beyond the visual horizon. Although geometry works for architectural floor plans, city blocks, and runway diagrams, it comes up short when you model shipping lanes, airways, or continents, or whenever you consider two locations that are far apart. You can still perform distance computations without abandoning the Cartesian
underpinnings by sprinkling a few sines and cosines into your formulas, but the minute you need to compute areas, the math becomes intractable.

A better solution is to use a new family of data types based on geodetic coordinates—geography. This new family shields the PostGIS user from the complexity of the math. As a trade-off, geography offers fewer functions, and it trails geometry in speed. You’ll find the same point, linestring, and polygon data types in geography; just keep in mind that the linestrings and polygons conform to the curves of the earth.

Are geometry and geography standard or not?
The geometry type is a long-accepted OGC SQL/MM type that you’ll find in other relational databases. Geography, on the other hand, isn’t a standard type and is only found in PostGIS and SQL Server.

The PostGIS geography type is loosely patterned after the SQL Server 2008+ geography type. For general use cases, you can think of the SQL Server geography type and PostGIS geography type as the same kind of animal.

1.3.3 Raster type

Geometry and geography are vector-based data types. Loosely speaking, anything you can sketch with an ultra-fine pen without running short on ink lends itself to vector representation. Vectors are well suited to modeling designed or constructed features, but suppose you snap a colored photo of the coral-rich Tasmania sea. With its motley colors and fractal patterns, you’re going to have a hard time constructing lines and polygons out of the photo. Your best hope is to quantize the photo into microscopic rectangles and assign a color value to each. Raster data is exactly this—a mosaic of pixels.

Perhaps the best example of a raster is the television you stare at every day, for hours on end. A TV screen is nothing more than a giant raster with some 2 million pixels. Each pixel stores three different color values: the intensity of red, green, and blue (hence the term RGB). In raster-speak, each color is called a band. The scale of each pixel’s edges corresponds to the underlying spatial reference system—if you’re measuring in meters, the pixel represents one square meter of area.

If you’re buying a TV set, the physical number of pixels will matter greatly to you: the larger the number of pixels, the bigger the viewing area and the more money it’ll cost you. For GIS purposes, the pixel has no physical meaning. It’s nothing more than a bundle of data. Terms like pixel length or size don’t apply. Each pixel represents a certain unit of area in reality, nothing more.

Raster data almost always originates from instrumental data collection and often serves as the raw material for generating vector data. As such, you’ll encounter plenty more sources of raster data than vector data. PostGIS will let you overlay vector data atop raster data and vice versa. The satellite view of Google maps is a perfect example of such an overlay. You see roads superimposed on top of the satellite imagery.
Rasters appear in the following applications:

- Land coverage or land use.
- Temperature and elevation variations. This is a single-band raster where each square holds a measured temperature or elevation value.
- Color aerial and satellite photos. These have four bands—one for each of the colors of the RGBA color space.

### 1.3.4 Topology type

When you gaze down at the terrain from your private jet, what you witness is not distinct geometries on a barren terrestrial plane, but an interwoven network of points, linestrings, and polygons. A cornfield abuts a wheat field, abuts a pasture, abuts a large expanse of prairie. Roads, rivers, fences, or other artificial boundaries divide them all. The surface of the earth (at least the parts that host humanity) resembles a completed jigsaw puzzle. Topology models take on this jigsaw perspective of the world. Topology recognizes the inherent interconnection of geographic features and exploits it to help you better manage data.

Consider a historical example where you want to model the United States and Mexico as two large polygons. Prior to the Gadsden Purchase, the northern boundary of Mexico extended well into present day Arizona and parts of New Mexico. For 33 cents per acre, the U.S. “purchased” 30 million acres from Mexico. The U.S. polygon grew as the Mexico polygon shrank. If you were using the geometry family to model the two polygons, you’d have to perform two operations to get your record-keeping straight: enlarge the U.S. and shrink Mexico. Using the topology model, you only need to perform one operation—either the enlargement or the shrinkage—because topology tracks the fact that the U.S. abuts Mexico. If the U.S. grows on its southern border, Mexico must shrink on its northern border. One operation implies the other.

Topology isn’t concerned with the exact shape and location of geographic features, but with how they’re connected to each other.

Topology is useful in the following applications:

- Parcel (land lot) data, where you want to ensure that the change of one parcel boundary adjusts all other parcels that share that boundary change as well.
- Road management, water boundaries, and jurisdiction divisions. U.S. Census MAF/Topologically Integrated Geographic Encoding and Referencing system (TIGER) data is a perfect example (http://www.census.gov/geo/www/tiger/).
- Architecture.

### 1.4 Hello real world

In this section, we’ll walk you through a full example from start to finish. Unfortunately, PostGIS is not a programming language where a few lines of code will print a Hello World message on your screen. Instead, to provide you with a true taste of PostGIS, we’ll guide you through the following steps:
CHAPTER 1  

What is a spatial database?

- Digesting a problem and formulating a solution
- Modeling
- Gathering and loading data
- Writing a query
- Viewing the result

If you’re completely new to PostGIS, just perform the tasks we ask of you for now. You won’t understand most of what you’re typing, but you’ll have the rest of this book for that. Right now, we want to give you an overview of the steps involved in writing a spatial query.

Before going further, you’ll need to have a working copy of PostGIS 2.0 or higher and PostgreSQL 9.0 or higher (preferably 9.1+ to take advantage of the new, simpler extension installation feature), as well as ancillary tools such as pgAdmin III to compose and execute your queries. Information about acquiring and installing these can be found in appendix B. As always, if you’re starting from scratch, we recommend that you install the latest versions.

1.4.1 Digesting the problem

Here’s the scenario you’re faced with: you need to find the number of fast-food restaurants within one mile of a highway. As for why someone might want to do this, any of the following reasons could apply:

- A fast-food chain is trying to locate a new store where supply falls short.
- A highway commissioner wants to satisfy the needs of motorists, who will be paying tolls.
- A health-conscious parent is trying to cut down the availability of fast food in the neighborhood.
- Hungry travelers are looking for their next meal.

First, you need to realize that you’re not going to be able to answer this question quickly or accurately with your usual arsenal of Google Maps, MapQuest, or even the latest paper map you picked up from the auto association. Learning PostGIS may not be any quicker, but you’ll have at your disposal the tools and skills to solve any and all problems of this kind in the future. Replace the highway with a lake, and you can determine how many homes surrounding the lake can be considered waterfront property. On a geodetic scale, replace the highway with the continent of Australia, and you can determine the number of islands within territorial waters. From there, you can even go on to a planetary scale and ask how many moons are within 10 million kilometers at perigee.

Once you have an initial understanding of the problem, we recommend that you immediately perform a feasibility study, even if it’s just in your mind. You don’t want to devote time to a solution if the problem itself is impossible to solve, lacking specificity, or, worse, you have no available data source.
Before going further, you need the postgis_in_action database you set up in section 1.2.3.

### 1.4.2 Modeling

You need to translate the real world to a model that is composed of database objects. For this example, you’ll represent the highway as a geometric linestring and the locations of fast-food restaurants as points. You’ll then create two tables: highways and restaurants.

#### USING SCHEMAS

First, you need to create a schema to hold your data for this chapter. A schema is a container you’ll find in most high-end databases. It logically segments objects (tables, views, functions, and so on) for easier management.

```sql
CREATE SCHEMA ch01;
```

In PostgreSQL it’s very easy to back up selected schemas and also to set up permissions based on schemas. You could, for example, have a big schema of fairly static data that you exclude from your daily backups, and also divide schemas along user groups so that you can allow each group to manage their own schema set of data. The postgis_in_action database schemas are chapter-themed so that it’s easy to download just the set of data you need for a specific chapter. Refer to appendix D for more details about schemas and security management.

#### RESTAURANTS TABLE

Next, you need to create a lookup table to map franchise codes to meaningful names. You can then add all the franchises you’ll be dealing with.

```sql
CREATE TABLE ch01.lu_franchises (
  id char(3) PRIMARY KEY,
  franchise varchar(30)
);  
INSERT INTO ch01.lu_franchises(id, franchise)
VALUES
  ('BKG', 'Burger King'),
  ('CJR', 'Carl''s Jr'),
  ('HDE', 'Hardee'),
  ('INO', 'In-N-Out'),
  ('JIB', 'Jack in the Box'),
  ('KFC', 'Kentucky Fried Chicken'),
  ('MCD', 'McDonald'),
  ('PZH', 'Pizza Hut'),
  ('TCB', 'Taco Bell'),
  ('WDY', 'Wendy''s');
```

Finally, you need to create a table to hold the data you’ll be loading.

```sql
CREATE TABLE ch01.restaurants (
  id serial PRIMARY KEY,
  franchise char(3) NOT NULL,
  geom geometry(point,2163)
);  
```

Listing 1.2 Create franchise lookup table

Listing 1.3 Create restaurants table
The restaurant data has no primary key, and nothing in the data file lends itself to a good natural primary key, so you create a dummy one. For your later analysis, you’ll need to uniquely identify restaurants so that you don’t double-count them. Also, certain mapping servers and viewers, such as MapServer and QGIS, balk at tables without integer primary keys or unique indexes, so you need to create an autonumber primary key on the restaurants table.

Next, you use a point geometry column to store your restaurant locations. The second argument to the geometry function indicates the spatial reference ID (SRID) that you’ve selected for the restaurant data. The SRID denotes the coordinate range and how the spherical space is projected on a flat surface. We’ll get into more detail about spatial reference systems in chapter 3. If you’re coming from a GIS background, you’ll know that you must have common projections before you can compare two data sets. This example uses EPSG:2163, which is an equal-area projection covering the continental United States.

Next, you need to place a spatial index on your geometry column:

```
CREATE INDEX idx_code_restaurants_geom ON ch01.restaurants USING gist(geom);
```

As part of the definition of an index in PostgreSQL, you must specify the type of index as we did in the preceding `CREATE INDEX`. PostGIS spatial indexes are of the gist index type.

Although it’s not necessary for this particular data set, because it won’t be updated, you’ll create a foreign key relationship between the franchise column in the restaurants table and the lookup table. This helps prevent people from mistyping franchises in the restaurants table. Adding `CASCADE UPDATE DELETE` rules when you add foreign key relationships will allow you to change the franchise ID for your franchises if you want, and have those changes update the restaurants table automatically. By restricting deletes, you prevent inadvertent removal of franchises with extant records in the restaurants table. (One added benefit of foreign keys is that relational designers, such as those you’ll find in OpenOffice Base and other ERD tools, will automatically draw lines between the two tables to visually alert you to the relationship.)

```
ALTER TABLE ch01.restaurants
ADD CONSTRAINT fk_restaurants_lu_franchises FOREIGN KEY (franchise)
REFERENCES ch01.lu_franchises (id) ON UPDATE CASCADE ON DELETE RESTRICT;
```

You can then create an index to make the join between the two tables more efficient:

```
CREATE INDEX fki_restaurants_franchises ON ch01.restaurants (franchise);
```

**Highways table**

Next you need to create a highways table to contain the road segments that are highways.

```sql
CREATE TABLE ch01.highways (gid integer NOT NULL,
                             ...);
```

**Listing 1.4 Create highways table**
feature character varying(80),
name character varying(120),
state character varying(2),
geom geometry(multilinestring,2163),
CONSTRAINT pk_highways PRIMARY KEY (gid)
); 
CREATE INDEX idx_highways ON ch01.highways USING gist(geom);

In this case, you’re creating the spatial index before loading the data, but for large tables that are loaded only once, it’s more efficient to create the indexes after you have loaded the data.

1.4.3 Loading data

To give this example some real-world flavor, we’ll scope out real data sources.

In this chapter, you first created the data tables and are now chasing after data to populate them. Ideally, these are the steps you’d want to take. In reality, though, you’ll sometimes find yourself subservient to the available data and begrudgingly have to alter your ideal table structure to fit what’s available.

But don’t surrender to the availability of real data too easily. You can often create SQL scripts that will translate the less-than-perfect data from your source into your perfected data structure. Always give primacy to your model. A well-thought-out model can often ride out the vagaries of a data source. We’ll follow this mantra as we continue.

Importing a CSV file

Fastfoodmaps.com graciously provided us with a comma-delimited file of all fast-food restaurants circa 2005. To import a CSV file, you need to create a table beforehand. After quickly studying the CSV file, you can create a staging table:

CREATE TABLE ch01.restaurants_staging ( 
    franchise text,
    lat double precision,
    lon double precision
 ); 

Use the psql \copy command to import the CSV file into your staging table:

copy ch01.restaurants_staging
    FROM '/data/restaurants.csv' DELIMITER as ',';

Your purpose here is to get the CSV data into a table so you can scrutinize it more carefully and write any additional queries to sanitize the data before you insert it into the production table. In this case, the data passes the quality check, so you can proceed with the insert:

INSERT INTO ch01.restaurants (franchise, geom) 
SELECT 
    franchise,
    ST_Transform(ST_SetSRID(ST_Point(lon,lat),4326),2163) As geom 
FROM ch01.restaurants_staging;
In this example, we use SRID 4326 (which corresponds to WGS 84 lon/lat), but then transform all the data to our desired planar projection for faster analysis.

**Spatial reference IDs (SRID) and spatial reference systems (SRS)**

You’ll often find number identifiers such as 4326 and 2163 in PostGIS and other spatial database code. These refer to records in the `spatial_ref_sys` table, where `srid` is the column that uniquely identifies the record. The ID 4326 is the most popular and refers to an SRS that often goes by the name WGS 84 lon/lat. We’ll go into spatial reference systems in more detail in chapter 3.

**Importing from an Esri shapefile**

You’ll find Esri shapefiles to be a common storage format for spatial data, mostly due to Esri’s early preponderance in GIS. To load data from shapefiles into a PostGIS database, use the `shp2pgsql` command-line utility that comes with all PostGIS installations. If you’re on Windows or Linux/Unix with a graphical desktop, you can also use the GUI version of the utility, dubbed `shp2pgsql-gui`. Both `shp2pgsql` and `shp2pgsql-gui` can load DBF files in addition to the Esri shapefile format.

We’ll use `shp2pgsql-gui` to demonstrate loading the road network data we downloaded from the National Atlas website and packaged with this chapter’s download. Using pgAdmin III, you can first highlight the database and then fire up `shp2pgsql-gui` using the plug-in menu or button. See figure 1.4.

**What if the shp2pgsql-gui plug-in doesn’t show in plug-ins?**

There are generally two reasons why the PostGIS Shapefile and Dbf-Loader option doesn’t show in your pgAdmin plug-ins menu. The first is that your distribution doesn’t install this GUI. The second is that the `.ini` file that pgAdmin is looking for isn’t in the right place. Refer to the “Configured shp2pgsql-gui in pgAdmin III” page (www.postgresonline.com/journal/archives/180-shp2pgsql-gui.html) for troubleshooting help.

You can use the Add File button to add one or more shapefiles, and each file appears in the Import List. Each shapefile will load into its own table. Clicking in any cell in the file list allows you to change the preset value.

You know your projection to be NAD 83 lon/lat, so you indicate this by changing the SRID to 4269, but be careful here! You’re simply telling the importer what the SRID is for the data coming in. You’re not transforming it! In this example, we also changed the name of the imported table to `highways_staging`. Click the Import button once you’re ready.

Once the import finishes, you should see the new `highways_staging` table in your database. You may have to refresh the browse tree in pgAdmin. Both `shp2pgsql-gui` and its command-line sibling automatically add a column named geom during the
import and set its data type by reading information contained in the shapefile. If you’re unfamiliar with the raw data, this is the time to study it. Perform general sanity checks, such as checking the total record count, inspecting columns that came in without data, and so on.

After you’re satisfied that the importer did its job without dropping any information, you can write an insert query to move the data from your staging table to the production table. In the query, you want to transform the SRID from 4269 to 2163 and only select columns that you defined in your production table. You can also filter the data to only the needed rows. The highway data has approximately 47,000 rows and includes every major and state highway in the U.S., and you’re only going to be looking at major highways, so you can add a filter that will bring the row count down to about 14,000.

```sql
INSERT INTO ch01.highways (gid, feature, name, state, geom)
SELECT gid, feature, name, state, ST_Transform(geom, 2163)
FROM ch01.highways_staging
WHERE feature LIKE 'Principal Highway%';
```

If you’d rather use the command-line shp2pgsql tool, open up the psql console and type in the following command:

```
shp2pgsql -s 4269:2163 -g geom
-I /data/roadtr1020.shp ch01.highways_staging
| psql -h localhost -U postgres -p 5432 -d postgis_in_action
```
As you might have noticed, the command line lets you transform the SRID with an additional \(-s\) switch, so you could skip the \texttt{ST_Transform} step in your code or use the data more or less as is. You can also index with the \(-I\) switch.

After you’ve finished loading the data, it’s good to follow up with a vacuum analyze so the statistics are up to date:

\texttt{vacuum analyze ch01.highways;}

### 1.4.4 Writing the query

It’s now time to write the query. Remember the question we set out to answer: \textit{How many fast-food restaurants are within one mile of a highway?} The query that will answer this question is shown in the following listing.

```
SELECT f.franchise, COUNT(DISTINCT r.id) As total 
FROM ch01.restaurants As r INNER JOIN ch01.lu_franchises As f ON r.franchise = f.id INNER JOIN ch01.highways As h ON ST_DWithin(r.geom,h.geom,1609) GROUP BY f.franchise ORDER BY total DESC;
```

The crux of this example is where you join the restaurants table with the highways table using the \texttt{ST_DWithin} function. This commonly used function accepts two geometries and returns true if the minimum distance between the two geometries is within the specified distance. In this case, you pass in a point for the restaurant, a multilinestring for the highway, and 1609 meters as the distance. All restaurant-highway pairs matching the join condition will filter through.

The join condition does allow for duplicate restaurants. For example, a McDonald’s located at the intersection of two major highways would show up twice. To only count each restaurant once, you use the \texttt{COUNT(DISTINCT)} construct.

The rest of the code is elementary SQL. If you’re a little rusty on SQL, please see appendix C for a refresher. As fair warning, the SQL we use in this book will get harder.

Finally, here’s the fruit of your labor.

```
<table>
<thead>
<tr>
<th>franchise_name</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>McDonald’s</td>
<td>5343</td>
</tr>
<tr>
<td>Burger King</td>
<td>3049</td>
</tr>
<tr>
<td>Pizza Hut</td>
<td>2920</td>
</tr>
<tr>
<td>Wendy’s</td>
<td>2446</td>
</tr>
<tr>
<td>Taco Bell</td>
<td>2428</td>
</tr>
<tr>
<td>Kentucky Fried Chicken</td>
<td>2371</td>
</tr>
</tbody>
</table>
```

```
1.4.5 Viewing spatial data with OpenJump

What’s more gratifying than to see your query output displayed on a map? You don’t want to display some 20,000 dots on a map of the U.S.—you can find that on each chain’s restaurant locator. Instead, you’re going to draw a buffer zone around highway segments and see how many dots fall within them.

For this you’ll use the `ST_Buffer` function. This function will take any geometry and radially expand it by a specified number of units. The post-expansion polygonal geometry is called a buffer zone or corridor.

**INSTALLING OPENJUMP** If you haven’t installed OpenJump, do so now before continuing. Chapter 5 discusses the installation and use of OpenJump, among other tools.

For this example, we’ll locate Hardee’s restaurants within a 20-mile buffer of U.S. Route 1 in the state of Maryland. Here’s the query to get the count:

```
SELECT COUNT(DISTINCT r.id) As total
FROM ch01.restaurants As r INNER JOIN ch01.highways As h
ON ST_DWithin(r.geom, h.geom, 1609*20)
WHERE r.franchise = 'HDE' AND h.name = 'US Route 1' AND h.state = 'MD';
```

Let’s see where the three Hardee’s are located. Fire up OpenJump and connect to your PostgreSQL database. You can first draw U.S. Route 1 using the following query:

```
SELECT gid, name, geom
FROM ch01.highways
WHERE name = 'US Route 1' AND state = 'MD';
```

Next, overlay the 20-mile corridor:

```
SELECT ST_Union(ST_Buffer(geom, 1609*20))
FROM ch01.highways
WHERE name = 'US Route 1' AND state = 'MD';
```

Finally, position the Hardee’s in the buffer zone routes.

```
SELECT r.geom
FROM ch01.restaurants r
WHERE EXISTS (
    SELECT gid
    FROM ch01.highways
    WHERE
        ST_DWithin(r.geom, geom, 1609*20) AND
        name = 'US Route 1' AND
        state = 'MD' AND
        r.franchise = 'HDE'
);
```

The results are shown in figure 1.5.

Play around with this example. Use your home state and your favorite restaurant chain to see how far you have to go to grab your next nutritious meal.
1.5 Summary

In this chapter, we’ve given you a small taste of how a spatially enabled database can fit in with a relational database system. We sowed the idea of how you can model real-world objects in space with spatial constructs.

We championed PostgreSQL and its spatial companion PostGIS. We demonstrated how PostgreSQL and PostGIS can be used together to analyze spatial patterns in data. We hope we’ve convinced you that the PostgreSQL/PostGIS combination is one of the best choices (if not the best) for spatial analysis.

Some of the SQL examples we demonstrated were at an intermediate level. If you’re new to SQL or spatial databases, these examples may have seemed daunting. In the chapters that follow, we’ll explain the functions we used here and the SQL constructs in greater detail. For now, we hope that you focused on the general steps we followed and the strategies that we chose.

Although spatial modeling is an integral part of any spatial analysis, there’s no right or wrong answer in modeling. Modeling is inherently a balance between simplicity and adequacy. You want to make your model as simple as possible so you can focus on the problem you’re trying to solve, but you must retain enough complexity to simulate the world you’re trying to model. Therein lies the challenge.

Before we can continue our journey, we must first analyze the different spatial types that PostGIS offers and show you how to create these and when it’s appropriate to do so. We’ll explore the geometry, geography, and raster types in greater detail in the next chapter and revisit topology in chapter 13.
Processing data tied to location and topology requires specialized know-how. PostGIS is a free spatial database extender for PostgreSQL, every bit as good as proprietary software. With it, you can easily create location-aware queries in just a few lines of SQL code and build the back end for a mapping, raster analysis, or routing application with minimal effort.

PostGIS in Action, Second Edition teaches you to solve real-world geodata problems. It first gives you a background in vector-, raster-, and topology-based GIS and then quickly moves into analyzing, viewing, and mapping data. You’ll learn how to optimize queries for maximum speed, simplify geometries for greater efficiency, and create custom functions for your own applications. You’ll also learn how to apply your existing GIS knowledge to PostGIS and integrate with other GIS tools.

What’s Inside

- An introduction to spatial databases
- Geometry, geography, raster, and topology spatial types, functions, and queries
- Applying PostGIS to real-world problems
- Extending PostGIS to web and desktop applications
- Updated for PostGIS 2.x and PostgreSQL 9.x

Familiarity with relational database and GIS concepts is helpful but not required.

Regina Obe and Leo Hsu are database consultants and authors. Regina is a member of the PostGIS core development team and the Project Steering Committee.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/PostGISinActionSecondEdition

"A huge body of information distilled into a concise guide."
—From the Foreword by Paul Ramsey, Chair PostGIS Steering Committee

"A more-than-worthy update to the definitive book on PostGIS."
—Jonathan DeCarlo Bentley Systems, Inc.

"The most comprehensive guide to spatial data on PostgreSQL."
—Sergio Arbeo, codecantor.com

"Provides the science and the tools needed to create innovative applications for the new digital age."
—Guy Ndjeng, NTSP