



**MEAP Edition  
Manning Early Access Program**

Copyright 2010 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

© Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Licensed to Mihkel Punga <mihkel.punga@jrc.ec.europa.eu>

**Part 1 Learning PostGIS**

- 1. What is a spatial database?**
- 2. Geometry types**
- 3. Data modeling**
- 4. Geometry functions**
- 5. Relating two or more geometries**
- 6. Special reference system considerations**
- 7. Working with real data**

**Part 2 Putting PostGIS to work**

- 8. Techniques to solve spatial problems**
- 9. Performance tuning**

**Part 3 Using PostGIS with other tools**

- 10. Enhancing SQL with add-ons**
- 11. Using PostGIS in web applications**
- 12. Using PostGIS in a desktop environment**
- 13. First look at WKT Raster**

**Appendices**

- Appendix A: Additional resources**
- Appendix B: Installing, compiling, and upgrading**
- Appendix C: SQL primer**
- Appendix D: PostGreSQL features**

# Foreword

As children, we were probably all told at one time or another that "we are what we eat," as a reminder that our diet is integral to our health and quality of life. In the modern world, with location-aware smart phones in our pockets, GPS units in our vehicles, and the internet addresses of our computers geocoded, it has also become true that "who we are is where we are" -- every individual is now a mobile sensor, generating a ceaseless flow of location-encoded data and they move about the planet.

To manage and tame that flow of data, and the parallel flow of data opened up by economical satellite imaging and crowd sourced mapping, we need tools equal to the task. Tools that can both persistently store the data, efficiently access it, and powerfully analyze it. We need spatial databases, like PostGIS.

Prior to the advent of spatial databases, computer analysis of location and mapping data was done with "geography information systems" (GIS), running on desktop workstations. When it was first released in 2001, the project name was just a simple play on words -- naturally a spatial extension of the "PostgreSQL" database would be named "PostGIS".

But the name has come to have further significance as the project matured.

Each year, new functions have been added for data analysis, and each year users have pressed those functions further and further, doing the kinds of work that in earlier years would have required a specialized GIS workstation. PostGIS is actually creating a world that is post-GIS -- we don't need GIS software to do GIS work anymore, a spatial database suffices.

In March of 2002, not even one year after the first release of PostGIS, I asked on the user mailing list for examples of how people were using PostGIS. And in her first post to the list, Regina Obe answered this way:

"We use it here [City of Boston] for proximity analysis. Part of our department is in charge of distributing foreclosed property to developers etc. to build houses, businesses etc.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

We use PostGIS to list properties by proximity [...] so that if a developer wants to develop on a piece of land that is, say, X in size, they will be able to get a better sense of whether it can be done."

Even at that early date in the project, Regina was already testing the capabilities of PostGIS, and creating clever analyses.

In the years that followed, in over 1000 posts to the PostGIS mailing lists, Regina and her husband Leo Hsu have become leaders of the PostGIS community, providing assistance to new users and constantly pushing the boundaries of what is possible. On the strength of her contributions to the project documentation and quality control processes, Regina joined the project steering committee in 2008, and has continued to contribute to the development of the software and reference documentation.

Making the most of a spatial database requires going beyond simple storage and retrieval (though "PostGIS in Action" provides great introductory material to get you started). Once you've mastered the basics, dive right into the advanced material and learn how to analyze your data. Location is the "universal key", it allows you to join and analyze data sets in ways that are impossible using conventional approaches.

Enjoy this book and enjoy the insights it provides in putting location data to work. Regina and Leo have distilled a huge body of information into a concise guide that is truly one of a kind.

Paul Ramsey  
Chair, PostGIS Project Steering Committee

# *About this Book*

PostGIS (pronounced "post-jis") is a spatial database extender for the PostgreSQL open source relational database management system. It is the most powerful open source spatial database engine. It adds to PostgreSQL several spatial data types and over 300 functions for working with these spatial types. It does for PostgreSQL what Oracle Spatial/Locator does for Oracle, IBM DB2/Information spatial datablades do for DB2 and Informix, and what geometry/geography types packaged in Microsoft SQL Server 2008+ do for SQL Server. PostGIS supports many of the OGC/ISO SQL/MM compliant spatial functions you will find in these other OGC compliant databases as well as numerous additional ones that are unique to PostGIS.

For those people coming from other ANSI/ISO compliant spatial databases or other relational databases such as those we have mentioned, you will feel right at home with PostgreSQL/PostGIS. PostgreSQL is probably the most ANSI/ISO SQL compliant database management system around, and supports most of the ANSI-SQL92/2003/ and some of the 2006 standard. In a similar vein PostGIS supports many of the industry standard spatial database functions, types, and operations.

The main raison d'être of this book is to provide a companion volume to the official PostGIS documentation, as a guide book for navigating through the hundreds of functions offered by PostGIS. We wanted to create a book that will catalog many of the common spatial problems we have come across and various strategies for solving them with PostGIS.

Above and beyond our primary mission, we hope to lay the foundations to thinking spatially. We hope that the reader will be able to adapt our numerous examples and recipes to their own field of endeavor and perhaps even to spawn creative scions of their own.

This book is not a substitute for the official PostGIS documentation. The official PostGIS documentation does a good job of introducing you to the myriad of functions available in PostGIS and provides examples on how to use each. It however will not tell you how to combine all these functions into a recipe to solve your problems.

While this book will not cover all 300 some odd functions available in PostGIS, we shall cover the more commonly used or interesting ones and give you the skills you need to combine them together to solve classic and more esoteric but interesting problems in spatial

analysis and modeling. Though you can use this book as a source of reference, we recommend that you do visit the official PostGIS site at <http://www.postgis.org>.

This book will mostly focus on 2-dimensional non-curved cartesian vector geometries. Although this book is mostly about writing spatial queries against 2-D vector geometries, we shall provide introductions to the following ancillary topics:

- Creating 3-D vector geometries
- Creating curved geometries
- Creating and querying geodetic geography data type
- Working with raster data using the companion raster data type (currently packaged separately)

Although the main purpose of this book is the use of PostGIS, we would fall short of our mission if we neglected to provide some perspective on the landscape it lives in. PostGIS is not an island and rarely works alone. To complete the cycle, we also include

- An extensive appendix that covers PostgreSQL in great detail from setup, backup, security management, as well as the fundamentals of SQL and creating functions and other objects in it.
- Several chapters dedicated to the use of PostGIS in web mapping, viewing using desktop tools, PostgreSQL PL/Languages commonly used with PostGIS, and extra open source add-ons such as the Tiger geocoder, PgRouting, PL/R and PL/Python.

## **Who should read this book?**

Although this book provides some introduction to PostGIS, it assumes a basic comfort level with programming and working with data in general. The types of people we have found most attracted to PostGIS and probably best suited for reading this book are listed below.

### **GIS PRACTITIONERS AND PROGRAMMERS**

You know everything about datum, geoids, and projections. You know where to find sources for data. You can create stunning applications with ArcGIS, MapInfo, Google Earth, OpenLayers, Adobe Flex, SilverLight or other AJAX-enabled toolkits. You are adept at generating data sources in ESRI Shape, MapInfo, and creating cartographic masterpieces. You may even be able to add and extract data from a spatially-enabled database, but when asked questions about the data, you are stuck. Being able to draw all the Walmarts in the US on a map is one thing, but being able to answer the question of how many Walmarts are east of the Mississippi without counting individual pushpins is a whole different ball game. Sure you may have used desktop tools and written procedural code to answer these questions, but we hope to show you a much faster way.

So what does a spatially-enabled database offer you that you don't already have at your fingertips?

© Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- Ability to easily intermingle spatial data with other corporate data such as financial information, observation data, and marketing information. Sure you can do these with ESRI shape files and KML files and other output formats, but that requires an extra step and limits your options for joining with other relevant data. A database such as PostgreSQL has features such as a query planner that improves the speeds of your joins and many commonly used statistical functions to make fairly complex questions, and summary stats relatively fast to run and quick to write.
- When collecting data from users – whether that be a user drawing a geometry on the screen and inputting related information or clicking on a point on the map, there is so much infrastructure built around databases, that the task is just so much easier if you are using one. Take for example rolling your own web application whether that be in .NET, PHP, Perl, Python. Each already has a driver for PostgreSQL to make inserting data easy. Add to that mix the text to geometry functions, geometry to SVG, KML and GeoJSON and other processing functions that PostGIS provides, the geometry generation and manipulation functions that things like OpenLayers and GeoServer have, and you have a myriad of options to choose from.
- A relational database provides administrative support to easily control who has access to what whether that be text attribute or geometry.
- Triggers that can allow generation of other things like related geometries in other tables when certain update events happen.
- PostgreSQL has a Multi-version concurrency control (MVCC) Transactional system to insure that when 100 users are reading or updating your data at the same time, your system doesn't come to a screeching halt.
- Ability to write custom functions in the database that can be called from disparate applications. PostgreSQL offers several choices of languages to choose from to write stored functions.
- If you are married to your GIS desktop tool of choice, not to worry. Choosing a spatial DBMS such as PostGIS does not mean you need to abandon your tools of choice. Manifold, CadCorp, MapInfo 10+ and various commonly used desktop tools have built-in support for PostGIS. ArcGIS as well via the SDE offering or via Obtuse ZigGIS Plug-In. While the Autocad FDO driver is not quite production quality, it too is getting there. So even in Autocad 2008+ you can access your PostGIS repository of data. Safe FME, a popular favorite of GIS professionals, has supported PostGIS for a long time.

#### **DB PRACTITIONERS**

At some point in your database career, someone might have asked you a spatially-oriented question about the data. Without a spatially-enabled database, you are forced to limit your thinking in terms of coordinates, location names, or other geographical attributes that can be reduced down to numbers and letters. This works fine for point data, but you are at a complete loss once areas and regions come into play. You may be able to find all the people

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

named Smith within a county, but if we were to ask you to find all the Smiths living within 10 miles of the county, you are stuck.

We want the reader from a pure database background to realize that data is more than just numbers, dates and characters and that amazing feats of SQL can be accomplished against non-textual data. Sure you might have stored images, BLOBs, and other oddities in your relational database, but we doubt you were able to do much in the way of writing SQL joins against these fields.

#### **THE SCIENTIST, RESEARCHER, EDUCATOR AND ENGINEER**

A lot of highly skilled scientists, researchers, educators, and engineers use spatial analysis tools to analyze their collected data, model their inventions, or train students. While we don't consider ourselves one of these people, we admire these people the most because they create knowledge and improve our lives in fundamental ways. They may know a lot about mathematics, biology, chemistry, geology, physics, engineering and so forth, but are not trained in database management, relational database use, or GIS. If you are one of these people, we hope to provide just enough of a framework to get you up to speed without too much fuss. What does PostgreSQL/PostGIS hold for you?

- Ability to integrate with statistical packages such as R and to even write database procedural functions in PL/R that leverage the power of R.
- PostgreSQL also supports PL/Python which allows you to leverage the growing Python libraries for scientific research right in the database where it can work even closer with the data than you can in a plain Python environment.
- While many think of PostGIS as a tool for Geographic Information Systems and that is implied by the name, we see it as a tool for spatial analysis. The distinction is that while geography is focused on earth and the reference systems that bind earth, spatial focuses on space and the use of space. That space and coordinate reference system may be specific to an ant hill, a map of a nuclear plant whose location is yet to be defined or the use of space as a visualization tool to model the inherently non-visual such as in process modeling. So the point is while you may not think of your particular area of interest as being touched by spatial analysis, we challenge you to dig deeper.
- A database is a natural repository for large quantities of data and has a lot of built-in statistical/rollup functions and constructs for producing useful reports and analysis. If you are dealing with data of a spatial nature or using space as a visualization tool, PostGIS provides more functions to extend that analysis.
- A lot of this data can be easily collected by machines (GPS, Alarm systems) and directly piped to the database via automated feeds or standard import formats.
- Easy distribution of portions of data. A relational database is ideal to create what we shall call "data dispensers" – which allows other researchers to easily grab just the subset of data they need for their research or to provide data for easy download by the public.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

The above profiles are the basic groups of spatial database users, but not the only ones.

If you have ever looked at the world and thought, "Wouldn't it be great if I could correlate crime statistics with the locations where we've planted trees or the locations of police stations or determine what demographic profiles seem to give us the best in sales," then PostGIS might be the easiest and most cost-effective tool for you.

## Roadmap

This book is divided into three major parts and several supporting Appendices.

### **Part 1 "Learning PostGIS"**

Part 1 Covers the fundamental concepts of spatial relational databases and PostGIS/PostgreSQL in particular. The goal of this part is to introduce you to industry standard GIS database concepts and practices and working with these. By the end of this chapter, you should have a solid foundation of the various geometry types, basic understanding of spatial reference systems, database storage options, and most importantly, how to load and query spatial data in a PostGIS enabled PostgreSQL database.

Chapter 1 exposes you to the idea of a spatial database and how PostGIS fits into this category. In this chapter you will learn how to load a CSV file into PostgreSQL and convert longitude/latitude coordinates into PostGIS geometry/geography types. You will also experience a fast-paced introduction to doing quantitative analysis with spatial functions.

Chapter 2 goes through all the geometry types that PostGIS has to offer, most of which are standard across most high-end spatial databases. You will learn how to create these on the fly using well-known text representations (WKT). You will also be exposed to the common standard concepts of polygon validity and linestring simplicity.

Chapter 3 covers various data modeling and storage strategies for storing spatial data with other standard relational data types as well as managing data. PostgreSQL supports additional advanced storage options you will not find in most other relational databases. In this chapter we'll explore using table inheritance, heterogeonus/homeogeneous geometry columns and a brief look at the hstore key value data type. We will also demonstrate how to compartmentalize business logic in the database using PostgreSQL rules and triggers.

Chapter 4 covers the easiest to understand of PostGIS functions; functions that work with only one geometry. We cover the key ones and provide brief demonstrations of their use.

Chapter 5 covers the more advanced PostGIS functions. These are functions that take as input one or more geometries.

Chapter 6 is a basic primer to the very important topic of spatial reference systems. It discusses how to determine which one your data is in and selecting suitable ones to store your data.

Chapter 7 is a compendium of the various open source tools and PostGIS/PostgreSQL packaged tools for loading spatial data. It covers how to load various kinds of data from

ESRI Shape, MapInfo, KML, to OpenStreetMap XML format. It also covers how to export data.

## **Part 2 Putting PostGIS to work**

This part focuses on using PostGIS to solve real world spatial problems and optimizing for speed.

Chapter 8 covers classic spatial problems and various techniques for solving them.

Chapter 9 provides approaches for improving the speed of your spatial queries. You will learn about common mistakes people make when writing queries and how to avoid them. You will learn how to take advantage of the various query planner statistics provided by PostgreSQL to troubleshoot problem areas in your queries.

## **Part 3 Using PostGIS with other tools**

Part 3 covers how to use tools most commonly used with PostGIS for building applications.

Chapter 10 covers add-ons you can use with PostGIS directly in spatial queries. It covers the tiger geocoder, PgRouting. In addition it covers the PL/Python and PL/R PostgreSQL procedural languages that are favorites of GIS analysts. Both PL/Python and PL/R have extensive libraries available for working with spatial data.

Chapter 11 is devoted to using PostGIS in conjunction with Web Mapping toolkits. It focuses on the most popular of these GeoServer, MapServer, fundamentals of WMS/WFS OGC webservices, and using OpenLayers and GeoExt JavaScript mapping APIs.

Chapter 12 provides a brief survey of the most commonly used Open Source desktop tools that support PostGIS. You will learn the pros and cons of each as well as quick primers on installing and working with each. Covered are OpenJump, QuantumGIS, uDig and GvSig.

Chapter 13 is an introduction to the PostGIS raster data type loosely referred to as WKT Raster. WKT Raster is not packaged in with PostGIS 1.5 or below, but will be in PostGIS 2.0 as another type called raster. This chapter will teach you how to load raster data using GDAL, doing intersections with geometries, polygonizing rasters, and doing basic analysis with raster pixels.

## **Appendix**

The appendices are divided into 4 sections

Appendix A - Additional resources for getting more help on PostGIS and the ancillary tools discussed in the book.

Appendix B How to get up and running with PostgreSQL and PostGIS

Appendix C An SQL primer that will explain the concepts of JOINS, UNION, INTERSECT and EXCEPT. The fundamentals of rolling up data with aggregate functions and aggregate constructs. The more advanced topic of using window functions and frames.

Appendix D Covers features that are unique to PostgreSQL that are rarely found in other databases.

## **Code and other conventions**

The following typographical conventions are used throughout the book:

- Courier typeface is used in all code listings.
- Courier typeface is used within text for certain code words.
- Sidebars and call outs are used to highlight key points or introduce new terminology.
- Code annotations are used in place of inline comments in the code. These highlight important concepts or areas of the code. Some annotations appear with numbered bullets like this [1] that are referenced later in the text.

## **Code Downloads**

The examples for all chapters for this book can be downloaded via <http://www.postgis.us>

You will find on the book site, chapter code downloads, data downloads, descriptions of each chapter with related links for each chapter.

## **Author Online**

The purchase of PostGIS In Action includes free access to a private forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and other users. You can access and subscribe to the forum at <http://www.manning.com/PostGISinAction>. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct in the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue among individual readers and between readers and authors can take place. It's not a commitment to any specific amount of participation on the part of the authors, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions, lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print. Lastly, there will be additions to the content added to the author's online website for the book, located at <http://www.postgis.us>.

You may also visit the authors at the PostgreSQL and Open Source GIS companion sites - <http://www.postgresonline.com> and <http://www.bostongis.com>.

## **About the title**

By combining introductions, overviews, and how-to examples, the In Action books are designed to help learning and remembering. According to research in cognitive science, the things people remember are things they discover during self-motivated exploration.

Although no one at Manning is a cognitive scientist, we are convinced that for learning to become permanent it must pass through stages of exploration, play, and, interestingly, retelling of what is being learned. People understand and remember new things, which is to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

say they master them, only after actively exploring them. Humans learn in action. An essential part of an In Action book is that it is example- driven. It encourages the reader to try things out, to play with new code, and to explore new ideas.

There is another, more mundane, reason for the title of this book: our readers are busy. They use books to do a job or solve a problem. They need books that allow them to jump in and jump out easily and learn just what they want just when they want it. They need books that aid them in action. The books in this series are designed for such readers.

# Part 1

## *Learning PostGIS*

Welcome to PostGIS in Action. 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 will learn how to perform actions with single lines of SQL code that you thought were only possible 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.

The book is divided into three sections and an extensive appendix. 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 will 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 concludes with a fast-paced example of loading fast food restaurant longitude latitude data and converting them to geometric points, loading roads data from ESRI shapefiles and doing spatial summaries by joining these two sets of data.

Chapter 2 covers all the geometry types that PostGIS has to offer. You will learn how to create these using well known text representations (WKT) and the concepts of validity and simplicity.

Chapter 3 covers different approaches for storing data in PostgreSQL/PostGIS. You'll learn about how to store multiple kinds of geometries in a single geometry column. Controlling what kinds of geometries can be stored in a column using constraints and built in PostGIS management functions. We'll cover PostgreSQL table inheritance and how to use to enhance flexibility and also for partitioning data. We end with a real world example using Paris, France data.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Chapter 4 -is a survey of the most common PostGIS and OGC compliant functions that take as input one geometry. You'll learn about functions for building new geometries, functions for processing that can simplify and morph geometries. You'll learn the fundamental accessor and measurement functions.

Chapter 5 covers relationships functions. These are the most common of functions used between geometries and are often used for SQL Joins. We'll cover intersections, different kinds of equalities, nearest neighbor queries and the industry standard intersection matrix relationship model (DE9IM) that most spatial relationship functions are based on.

Chapter 6- is an introduction to spatial reference systems that will explain the concepts behind them and how to work with them.

Chapter 7 concludes Part 1 of the book with exercises in loading various kinds of spatial and non-spatial data into PostGIS using open source tools such as the PostgreSQL/PostGIS packaged psql, pgsql2shp, shp2pgsql, shp2pgsql-gui, as well as open source tools OGR2OGR, and OSM2PGSQL.

In Part 2 of the book we'll focus on solving common and interesting spatial problems with the functions we learned about in Part 1 as well as optimizing for performance.

In Part 3 we conclude the book with various common open source tools used to enhance the power of PostGIS and PostgreSQL.

# 1

## *What is a spatial database?*

This chapter covers

- Spatial database in problem solving
- Geometry data types
- Modeling with spatial in mind
- Why PostGIS/PostgreSQL for spatial database
- Loading and querying spatial data

In this we will first introduce you to spatial databases. You learn what they are and how they allow you to model space and perform proximity queries that you can't with a plain relational database system. We will then focus on PostGIS, which is a spatial database extender for the PostgreSQL database management system. We will dive in with quick examples of loading spatial data and performing proximity analysis with PostGIS.

### **1.1 Thinking spatially**

The popular mapping sites such as Google Maps, Virtual Earth, MapQuest, and Yahoo have empowered people in many walks of life to answer the question of "Where is something?" by finding it on a gorgeously detailed, interactive map. No longer were they restricted to textual descriptions of "where" like "Turn right at the supermarket and third house on right." nor were they faced with the perennial problem of pulling out a paper map and not being able to figure out where they currently are.

Going beyond getting directions, organizations large and small have learned and discovered that mapping could be a great resource to analyzing patterns in data. Simply by plotting the addresses of pizza lovers, a national pizza chain can visibly see where to locate the next grand opening. Political organization planning on grass-root campaigns can easily

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

get a picture of where the undecided or unregistered voters are located and concentrate their route walks accordingly. Whilst the mapping sites have given unprecedented power to interactive mapping, to use them still requires that users gather point data and place them on the map. More critically, the reasoning that germinates from an interactive map is entirely visual. Back to the pizza example, the chain may be able to see the concentration of pizza lovers in a city or arbitrary sales region via visually inspecting their map with pizza lovers on pushpins, but suppose we further differentiate pizza lovers by income level. If the chain has more of a gourmet offering, it would really want to locate sites in the midst of mid to high income pizza lovers. They could use pushpins of different colors on the interactive map to indicate various income tiers, but the heuristic visual reasoning is now much more complicated. Not only does the planner need to look at the concentration of push pins, but he/she must keep the varying colors or icons of the pin in mind. Add another variable to the map, like households with more than two children, and the problem exceeds the processing power of the average human brain.

Spatial databases can help solve this problem of information overload.

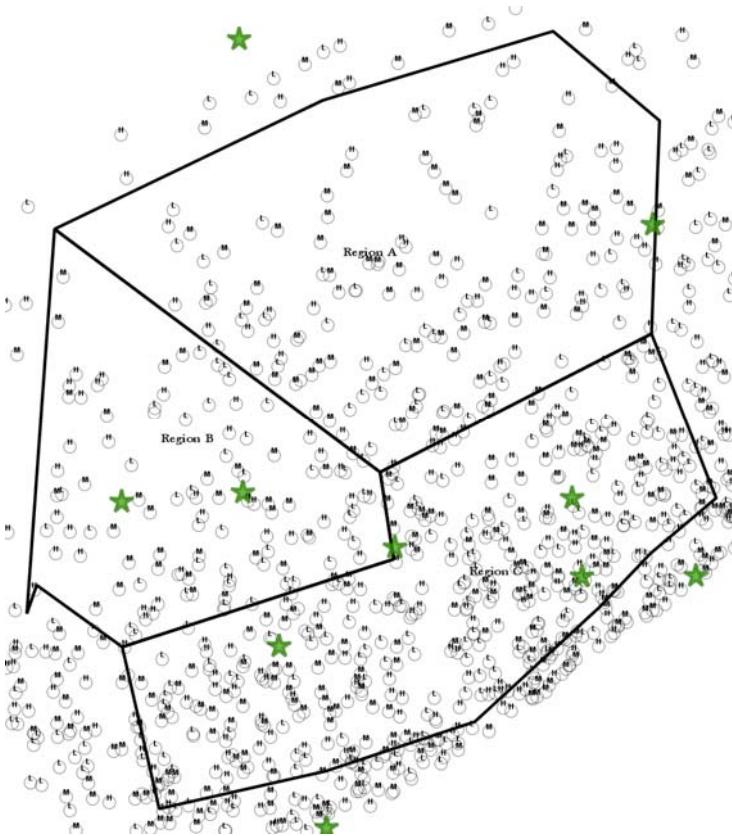


Figure 1.1 Pushpins gone too far

### WHAT IS A SPATIAL DATABASE?

A spatial database is a database that defines special data types for geometric objects and allows you to store geometric data (usually of a geographic nature) in regular database tables. It provides special functions and indexes for querying and manipulating that data using something like Structured Query Language (SQL). While it is often used as just a storage container for spatial data, it can do much more than that. Although a spatial database need not be relational in nature, most of the well-known ones are.

A spatial database gives you both a storage tool and an analysis tool. A spatial database is not constrained by the need to present data visually. The pizza shop planner can store an infinite number of attributes of the pizza loving household, income level, children in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

household, pizza ordering history, and even religious preferences and cultural upbringing (as they relate to topping choices on a pizza). More importantly, the analysis need not be limited to the number of variables that can be juggled in the brain. The planner can ask questions like, "Give me a list of neighborhoods ranked by the number of high-income pizza lovers with more than 2 children." Furthermore, we can add unrelated data such as location of existing pizzerias or even the health-consciousness level of various neighborhoods. Our questions of the database could be as complicated as "Give me a list of locations with the highest number of target households where the average closest distance to any pizza store is greater than 16 kilometers (10 miles). Oh and toss out the health-conscious neighborhoods."

**Table 1.1 Result of a spatial query**

Region	#House Holds	#Restaurants	#Avg Travel
Region A	194	1	17.1 Km

Suppose you are not a mapping user, but more of a data user. You work with data day-in and day-out never needing to plot anything on a map. You are familiar with questions like: "Give me all the employees who live in Chicago." or "Count up the number of customers in each zip code." Suppose that we have the latitude and longitude of all the employees' addresses, we could even 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 questions posed are "Give me the number of houses within 2 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 collecting or deriving additional values for each data point. For the coastline question, we would 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 coast line or use a series of SQL to order all the houses by proximity to the beach and then making a cut. With spatial support all we need to do is slightly reformulate the question to "Find all houses within a 2 mile radius of the coastline." A spatially-enabled database can intrinsically work with data types like coast line (modeled as linestrings), buffer zones modeled as polygons, and houses (modeled as points).

As with most things worth pursuing in life, nothing comes without putting in the effort. You will need to hike up 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 will be introducing you to is completely free.

If you were able to figure out how to get data into your Google map, you will have no problem taking the next step. If you can write queries in non-spatially enabled databases, ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

we'll open your eyes and mind to something beyond the mundane world of numbers, dates and strings. Let's get started.

### **1.1.1 Introducing the geometry data type**

The entirety of 2-D mapping can be accomplished with the three basic geometries of points, linestrings, and polygons. We can begin to model physical geographical entities with these building blocks. This is very intuitive and interesting.

For example, an interstate highway crossing the salt flats of Utah clearly jumps out as linestrings. The salt flat can be represented by a polygon with quite a number of edges. A desolate gas station located somewhere on the interstate can be a point.

We need not limit ourselves to the macro dimensions of road atlases. Take your home; each room could be represented as a rectangular polygon, the wiring and the piping running from room to room could be linestrings, and the location of the dog house in the back yard can be represented by a point. See how interesting this could be? Just by reducing the landscape to 2-dimensional points, linestrings, and polygons, you have the tools to model everything.

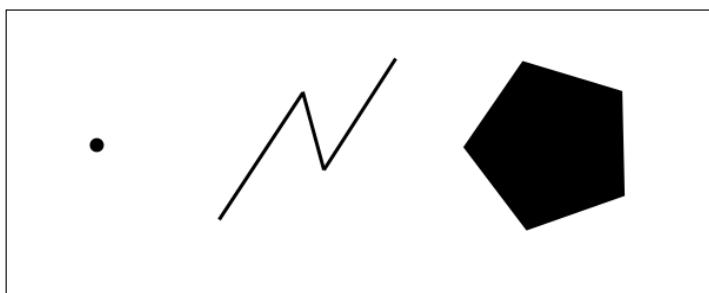


Figure 1.2 The basic geometries: a point, a linestring, and a polygon.

Do not be overly concerned with the rigorous definition of the geometries. Leave that for the mathematical topologists. Points, linestrings, and polygons are simplified models of reality. As such, they will never perfectly mimic the real thing. Do not be overly concerned with geometries that you feel should be included, but are missing. Two good examples are football stadiums or a beltway around a city. The former could be well-represented by an ellipse, the latter could be modeled as a perfect circle. Both of these geometries are missing (at least for now), but they could be approximated closely enough with polygons.

### **1.1.2 Modeling**

We have introduced the building blocks of points, linestrings, and polygons. We have given some example of how to model real world geographies using these basic geometries and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

have invited you to come up with your own. Modeling is not the only step in problem solving. Let us go back to the Salt Flats of Utah. We know the interstate traverses the salt flat from one end to the other. A simple question that any motorist must be thinking would be "How long am I going to be on this thing?" Taking out a map, the motorist may instruct his or her co-pilot to measure the distance from the mile marker when the interstate entered the salt flat to the mile marker when the interstate exits the salt flat. Unbeknownst to the motorists, they have just formulated a spatial query.

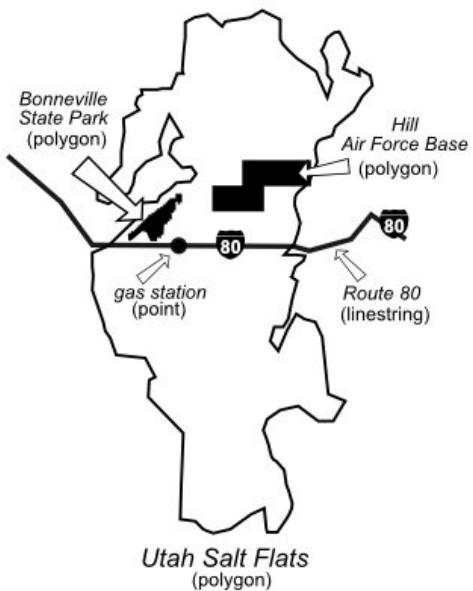


Figure 1.3 The Utah Salt Flats – we can model it with linestrings, points and polygons

With our points, linestrings and polygons in hand, let us dissect this simple act of measurement and then see if we can ask it in a way that a spatially enabled database could easily respond to. To start the measurement, the copilot looks for the point on the map where the interstate first hits the salt flat. This point is the intersection of the linestring with the polygon. Copilot then proceeds to find the place where the interstate will leave the salt flat and comes up with another point of intersection. Given that the highway is completely straight during its cross of the salt flat, the copilot simply has to take a ruler from the first intersection point to the second intersection point. Let us go through this with a higher level of abstraction. We start with two geometries, a linestring and a polygon. We overlay one atop the other and found the intersection of the two geometries. A line intersected with a polygonal area will yield the linestring contained within the polygon. We finish by taking the length of the linestring.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

One linestring through a polygon seems more like an exercise in geometry than databases, but it is the start of something powerful. Suppose the task at hand is to find the total length of all interstate highways in the state of Utah. We find ourselves a table of polygons for all the states and a table of all interstate highways in the US represented as linestrings. Next we pull out the row in the states table for Utah and the polygon that represents the state of Utah and perform an SQL join with the highways table using the geometric intersects function as the join operator and geometric intersection as our output function. The output query would only be those portions of the highway within the state of Utah. Finally we aggregate using the spatial length function and SQL SUM function and we have our answer. Hopefully, with the Utah example we have demonstrated that by introducing geometric data types and functions, we can leverage the querying power of a relational database to easily come up with answers to problems that at first seem insurmountable.

Do not worry if you are not a whiz at SQL. When it comes to spatial databases, learning how to use the additional spatial functions is more important than being able to generate complex SQL statements. In our experience, simple SELECT, INSERT, UPDATE statements will get you through 85% of the spatial queries that you will need to write.

### **1.1.3 Imagine the possibilities**

In the above we demonstrated the power of spatial queries without really telling you what a spatial query is or what it means to be spatial.

## **SPATIAL ANALYSIS, SPATIAL PROCESSING, AND SPATIAL QUERIES**

A spatial query is a database query that uses geometric functions to answer questions about space and objects in space. Spatial Database Extenders such as PostGIS add to the standard SQL language a body of functions that work with geometric objects in a database similar in concept to functions that work with dates. For example with a date, you have functions that tell you how many hours/days/minutes/years/weeks are between two dates or whether this date is in the future or the past. For more sophisticated databases such as PostgreSQL you can even define time and date intervals and answer with the overlaps function if two intervals overlap.

In addition to being able to ask questions about the use of space, spatial functions allow you to create and modify objects in space. This portion of spatial analysis is often referred to as geometric or spatial processing.

In the models we described above we talked about 2-d points, linestrings, and polygons. These are fundamental building blocks. In a spatially enabled database such as PostGIS/PostgreSQL, more complex objects exist such as MULTIPOLYGONS, MULTIPOLYPOINTS, MULTILINESTRINGS, GEOMETRYCOLLECTIONS and curved geometries. In addition to the 2-

d world we have described, you can also have these 2-d objects sitting in 3-D space. This is often referred to as 2.5D and is the first step to real 3D modeling.

There is one major initiative going on in PostGIS land to integrate RASTER support into the database. RASTER data is data stored as individual pixel numeric values in bands and correlated with a location in space. Lots of useful information is coded in RASTER format. We consider RASTER in the database to be the next big thing that happens to PostGIS. This allows analysis of things such as satellite weather data and digital elevation models (DEM) and overlaying these with vector data using SQL as well as slicing these up, creating derivatives of them, and overlaying these on a map. We'll cover these more complex objects including the currently available PostGIS raster functionality in later chapters of this book.

## **1.2 Introducing PostgreSQL and PostGIS**

In the rest of this chapter, we shall talk more about PostgreSQL and PostGIS. PostGIS is a free and open source (FOSS) library which spatially enables the free and open source PostgreSQL Object-Relational database management system.

### **WHAT IS AN OBJECT-RELATIONAL DATABASE?**

An object-relational database is one that can store more complex types of objects in its relational table columns than the basic date, number, text and allows the user to define new custom data types, new functions, and operators that work against these new custom types.

#### **1.2.1 PostgreSQL strengths**

PostgreSQL is an object-relational database system and has a regal lineage that dates back almost to the beginning of the existence of relational databases.

#### **A brief history of PostgreSQL**

If we were to look at the family tree of PostgreSQL it would look something like this

(Ingres, System-R)

Postgres

Illustra

Informix

IBM Informix

Postgres95

PostgreSQL

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In fact PostgreSQL is a cousin of the databases Sybase and Microsoft SQL Server because the people that started Sybase came from UC Berkeley and worked on the Ingres and/or PostgreSQL projects with Michael Stonebraker. Michael Stonebraker is considered by many to be the father of PostgreSQL and 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.

Back in the heyday of relational databases, object-relational was an interesting distinction, but nowadays pretty much any high-end and some low-end relational databases are object-relational. PostgreSQL was one of the few built from the ground up to be that way.

PostgreSQL's claim to fame is that it is the most advanced open source database out there. It has the speed and functionality to compete with the functionality provided by the popular commercial enterprise offerings and is used to power databases that are terabytes in size. Some of the compelling features that it has that most other open source databases lack and many commercial ones lack as well are:

#### **POSTGRESQL UNIQUE FEATURES**

- Various choices to choose from for writing database functions that can return simple scalar values as well as data sets and to build aggregate functions with. No open source or commercial database to our knowledge can compete with it in this regard. Commonly used ones are built-in SQL, PL/PGSQL. In addition to the two built in ones PL/Perl, PL/Python , PL/TCL, PL/SH, and PL/R are also often used. These require additional environment installs such as Perl, Python, TCL and R in order to take advantage of them. IBM DB2 and Microsoft SQL Server come close with allowing .NET functions, but its not quite as elegant as being able to write the code right in the database. Oracle only supports PL/SQL and Java.
- Support for Arrays. PostgreSQL, Oracle, IBM DB2 are fairly unique among databases in that arrays are first class citizens. In PostgreSQL, you can define any table column as being composed of an array of say strings, numbers, dates, geometries or even your own data type creations. This comes in pretty handy for matrix like analysis or aggregation. In addition you can convert any single column row list to an array, which comes in particularly useful when manipulating geometries.
- Table Inheritance – PostgreSQL has a feature called Table inheritance which is kind of like object multi inheritance. Table inheritance allows you to treat a whole set of tables as a single table as well as defining nested inheritance hierarchies. It is often used for table partitioning strategies. We'll demonstrate the power and caveats of this later.
- Ability to define aggregate functions that take more than one column. When you think of aggregates you think of them as taking only one column as input. The multi-column feature is something that is not commonly exploited thus is hard to visualize. Multi-column aggregates have existed for some time in PostgreSQL. We have a couple of examples on our Postgres Journal site to demonstrate it:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

How to create multi-column aggregates:

<http://www.postgresonline.com/journal/index.php?/archives/105-How-to-create-multi-column-aggregates.html>

Making SVG Plots with PIPython and Multi-column aggregates -

<http://www.postgresonline.com/journal/index.php?/archives/107-PLPython-Part-5-PLPython-meets-PostgreSQL-Multi-column-aggregates-and-SVG-plots.html>

#### BASIC ENTERPRISE FEATURES

- It is one of the most ANSI-SQL compliant databases around even when you include the commercial offerings. Those familiar with working with other relational database systems should feel at home using PostgreSQL.
- A fairly sophisticated query planner and indexing support for complex objects that is good for optimizing fairly intricate joins and aggregations without the need for hints. The speed is comparable to enterprise class DBMS for even the most hairy of SQL statements.
- Ability to define new data types fairly easily in both C and the built in languages.
- Relational views with ability to write rules against these that allows for updating even non-single table and rollup views.
- Advanced transactional support. It uses a Multi-Version Concurrency Control system which is the same model that Oracle uses and Microsoft SQL Server 2005+. Also has features such as transaction save points.
- Thousands of built-in functions and contributed functions for doing anything from string manipulation, regular expressions, regression analysis, to analyzing astronomical data.
- Those coming from Oracle background's will be surprised how similar Oracle's PL/SQL language is to PostgreSQL's native PL/PgSQL. In addition to PL/PgSQL and numerous other languages, PostgreSQL has a built-in SQL function language. This is something that other databases lack and is much easier to write for simple set returning or calculation functions. Unlike other language functions, an SQL function is not a black box to the PostgreSQL planner. The major benefit of this is that it can be incorporated in the plan strategy. It behaves much like a macro in C where the logic is often in-lined in the query so is often more efficient than a PL/PgSQL or other language function and can still hide the complexity from the person utilizing the function.
- It runs on pretty much any OS you can think of.
- PostgreSQL 8.4 introduced the ability to define column level permissions.
- PostgreSQL 8.4 also introduced ability to write variadic functions. This allows you to write a single function that has a default argument if it is not passed in. So

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

`getMyElephant('blue')`, `getMyElephant()` would use the same function, but `getMyElephant` would use the default color defined in the function.

#### **ADVANCED ENTERPRISE FEATURES**

- Ability to write and write easily your own aggregate function in most any supported language including just SQL. This feature is particularly useful for something like spatial analysis. The simplicity and ease of writing aggregates will come as a breathtaking shock for others who have come from other databases that allow this, but require immense amounts of code to do it.
- In 8.4 windowing functionality was introduced which many of the high-end commercial databases have had such as IBM DB2 and Oracle, and that Microsoft SQL Server introduced in its SQL Server 2005 offering. This is useful for OLAP and data warehouse applications and is particularly useful for nearest neighbor searches as we will demonstrate.
- In 8.4 recursive common table expressions for writing recursive queries (useful for navigating trees) and common table expressions functionality which are found in the popular high-end commercial databases. We shall demonstrate how this functionality is particularly useful in spatial queries in the upcoming chapters. One important thing about this is that it follows the ANSI SQL 2003 standard, so is almost exactly what you would write in Microsoft SQL Server and IBM DB2. Oracle has its own variant for doing hierarchical queries it has had almost since its inception called CONNECT BY that does not follow the standard. Oracle introduced in their 11GR2 offering, ANSI compliant recursive common table expressions that follow the same CTE syntax as PostgreSQL, SQL Server, and IBM DB2.
- In 8.4 the database restore supports parallel restore of tables, which makes the database restore 4 times faster than it was in 8.3 and below. This is important for huge databases.

#### **MORE ENTERPRISE FEATURES IN POSTGRESQL 9.0**

- Support for ANSI SQL compliant column level triggers
- Easier grant/deny level permissions
- Built-in warm standby replication
- Log streaming (another replication feature)
- Anonymous functions in any PL language via the new DO command which allows for running Python, Perl, TCL, PLPgsql code without defining a function
- Ability to run PostgreSQL in 64-bit mode on windows (this has always been available for Linux/Unix variants)
- HStore data type improved speed and functionality
- Even more window function enhancements - row preceding and following logic was

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

added to allow simple moving averages

PostgreSQL was chosen as a suitable companion for PostGIS primarily because of the built-in support to add new custom data types, define functions, operators and indexes that work against these new types and its fairly sophisticated query planner.

### **1.2.2 PostGIS, adding GIS to PostgreSQL**

PostGIS is a project spear-headed by Refractions Research. PostGIS provides over 300 spatial operators, spatial functions, spatial data types and spatial indexing enhancements. If you add to the mix the complimentary features that PostgreSQL and other PostgreSQL related projects provide, then you've got one jam-packed powerhouse at your disposal well suited for hardcore work as well as a valuable training tool for spatial concepts.

The power of PostGIS is enhanced by other supporting projects to include projection support (Proj4), advanced spatial operation support provided by the Geometry Engine Open Source (GEOS) project (a project ported from Vivid Solutions Java Topology Suite (JTS), historically incubated by Refractions Research, and now a project of Open Source Geospatial Foundation OSGEO) The very foundation of PostGIS, the PostgreSQL Object Relational Management system which provides transactional support, gist index support used to index spatial objects, and query planner out of the box for PostGIS is perhaps the most important of all. It is a great testament to the power and flexibility of PostgreSQL that Refractions chose to build on top of PostgreSQL over any other open source database. It goes without saying that PostGIS would not be as useful today without the vast ecosystem that it leveraged and the ecosystem that has grown around it. These include both open source and commercial tools that can work with it and numerous tool kits and application frameworks that use it as a core data storage and manipulation tool.

#### **What are OGC and OSGEO?**

OGC stands for Open Geospatial Consortium and is the body that exists to try to standardize how geographic and spatial data is accessed and distributed. In that mission, 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 is the body whose initiative is to fund, support, and market open source tools and free data for GIS. There is some overlap between the two. Both strive to make GIS data and tools available to everyone which means they are both concerned about Open Standards.

If your data and your APIs can be accessed by standards available to everyone, people using CadCorp, Safe FME, AutoCad, Manifold, MapInfo, ESRI ArcGIS, OGR2OGR/GDAL, OpenJump, QuantumGIS, Deegree, UMN Mapserver, GeoServer, MapDotNet etc, then everyone can use the tools they feel most comfortable with or fits their workprocess and/or can afford and share information between each other. OSGEO tries to ensure that

regardless of how big your pocketbook is, you can still afford to view GIS data. OGC tries 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 all constituents. This is especially important for government agencies where their salaries and tools are paid for with citizen tax dollars, students who have a lot of will and smarts to learn and advance technology, but have small pockets, and even smaller vendors who have a compelling offering for specific kinds of users but who are often snubbed by larger vendors in the market because they can't support or lack access to private API standards of the big named vendors in the industry.

PostGIS and PostgreSQL also conform to industry standards more closely than most products. PostgreSQL supports most of ANSI SQL 92-2003+ and some of ANSI SQL 2006. PostGIS supports OGC standards "SQL/MM Spatial" (ISO JTC1, WG4, 13249-3). This means that you are not simply learning how to use a set of products, but you are garnering knowledge about industry standards that will carry you through grasping other commercial and open source geospatial databases and mapping tools.

PostGIS carries less baggage than most other spatial database engines. The fact that you can see the code and see how it works also makes it an ideal training tool for teaching spatial database concepts and also makes it easier to troubleshoot when things go wrong.

PostGIS has numerous functions you will not find even in the commercial offerings. In general it has more output formats than the commercial offerings and its speed is on par and sometimes better than the commercial ones for common spatial needs.

### **1.2.3 Alternatives to PostgreSQL and PostGIS**

Admittedly, PostGIS/PostgreSQL is not the only spatial database in existence. Most of the high-end commercial relational database systems provide spatial functionality. The first two to do that were Oracle (with their included Locator, priced add-on Oracle Spatial and before those their spatial data option (SDO)), IBM DB2 and IBM Informix with their add-on priced options Spatial DataBlade and Geodetic DataBlade, and very recently SQL Server 2008 with their built-in Geometry and Geodetic Geography types and companion spatial functions.

While PostGIS/PostgreSQL was the first open source database to support OGC compliant spatial SQL, others are coming on board and even building on many of the same foundations that OSGEO developers helped build and extend.

Ingres, the older cousin of PostgreSQL is also enhancing their spatial support and building using some of the same plumbing underlying PostGIS. The new kid on the scene is SpatialLite which is an add-on to the Open source SQLite portable database. SpatialLite is especially interesting because it is well positioned to be used in conjunction with PostGIS and other high-end spatially enabled databases. It can be used to create master/slave applications that can provide basic light-weight spatial support to a client such as a hand held in the field. It can also be used as a more feature rich transport mechanism for spatial data. It also has a companion RasterLite offering mostly focused on storage and display of raster data. This would make a great companion to the PostGIS raster focus on analysis.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

SpatiaLite and RasterLite also leverage many of the core libraries that PostGIS uses - GEOS, Proj and GDAL. This fact makes it an even more admirable companion to PostGIS since many of the conventions are the same and much of the ecosystem supporting 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. As such some spatial queries possible in PostGIS are harder to write in or not even possible in SpatiaLite. SpatiaLite single file and embedded engine makes it a less threatening and easier to deploy database for users new to databases or GIS.

MySQL has spatial support and has for quite sometime, but its spatial development is fairly stagnant and even weaker than SpatiaLite. Now that it is a property of Oracle, its questionable how interested Oracle will be in beefing up MySQL spatial support to compete with its Oracle Locator/Spatial offering. MySQL 4 and 5 have spatial functionality built-in, except their geometric functions have only worked with the bounding boxes of geometries and did not provide indexed access except for MyISAM stored tables. Only recently, before the Oracle acquisition, (and still not released in production), has MySQL started to add functions that work against real geometries rather than just the bounding box caricatures.

In addition to the spatial functionality provided by the popular commercial database vendors, Environmental Systems Research Institute (ESRI), has for a long time provided their spatial database engine (SDE) with their ArcGIS product. The SDE engine is very integrated into the ArcGIS line of products and was often used to spatially enable databases such as Microsoft SQL Server 2005 and below which lacked spatial functionality. It also often competes with, and some would say, castrates the built-in spatial functionality of existing databases such as PostgreSQL, Oracle, or Microsoft SQL Server 2008.

#### **1.2.4 What works with PostGIS**

The following are commercial vendors that currently support PostGIS in their desktop/web offerings. In later chapters we'll go over the free and open source GIS tools out there that support PostGIS as well.

- Cadcorp SIS - who is partially funding the raster support in PostGIS and is a favorite among modelers for both desktop and web-based apps. Cadcorp supports over 160 formats including direct support for the other high-end spatial database offerings
- Safe FME - contributors both monetary and developer support for GEOS and makes Extract Transform Load (ETL) tools for GIS data that makes moving GIS data transport to different formats and databases a simple drag and drop and schedule exercise. They are the favorite for high-end ETL transactions.
- Manifold -- which released support for PostGIS in their version 8.0 and above product and that is a favorite of many spatial database analysts and people who like SQL in all its glory (it supports Oracle Locator/Spatial, PostGIS, SQL Server 2008, IBM DB2, MySQL and its own extender for SQL Server 2005)
- zigGIS - this is a desktop plug-in for ESRI ArcGIS Desktop that works with both 9.2

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

and above and allows you to access PostGIS data without an ArcSDE license. This does not work with ArcGIS server as of this writing.

- In ArcGIS 9.3, ESRI introduced support for PostGIS. Though this requires an ArcSDE Server license for PostGIS so may not be suitable for people on a limited budget. ArcGIS is best known for their great cartography.
- Pitney Bowes MapInfo 10 - Pitney Bowes introduced support for PostGIS in their recent MapInfo 10 offering. MapInfo is a popular tool for GIS VB programmers using its MapBasic interface. It enjoys a rich history of integration with MS Office products. It is a favorite of light-weight GIS users and database analysts because of its rich query options and easy data import menus.

As far as free and open source tools that can work directly with PostGIS data, PostGIS has the strongest support in the Free Open source GIS arena than any other spatial database. It has stronger support than MySQL for both commercial and open source GIS tools. There are just too many GIS open source tools that work with PostGIS to list. We shall cover the more common offerings in our Desktop and Web tools chapters. As you can see, PostGIS has an already very strong and growing commercial support belt as well. Its commercial vendor support is now just as strong as what you will find available for Oracle, SQL Server, or IBM DB2.

### **1.3 Getting Started with PostGIS**

In this section we will demonstrate some simple examples of creating geometries with PostGIS and then in later sections of this chapter we will cover loading and querying spatial data. Before going further, you will need to have a working copy of PostGIS, 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.

#### **1.3.1 Verifying version of PostGIS and PostgreSQL**

If you'd like to see the geometries visually, we recommend that you install one of the desktop utilities available for working with PostGIS which we describe in Chapter 12. For most of these exercises we use OpenJump for visualization. For detailed installation guides on PostgreSQL, PostGIS, PgAdmin III, please refer to Appendix B. The examples below require PostGIS 1.3 or above and PostgreSQL 8.2 or above. In later sections, we will be using some features introduced in PostgreSQL 8.4 and PostGIS 1.5.

Execute the following query in PgAdmin III to verify that you have PostGIS installed successfully and to obtain the version number:

```
SELECT postgis_full_version();
```

If all is well, you should see the version of PostGIS, the GEOS library and the Proj Library installed along with PostGIS displayed, as in below:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

POSTGIS="1.5.2SVN" GEOS="3.2.2-CAPI-1.6.0" PROJ="Rel. 4.6.1, 21  
August 2008" LIBXML="2.7.6" USE\_STATS

Run the following to verify what version of PostgreSQL you are running.

```
SELECT version();
```

If all is well, you should see the PostgreSQL version and Operating System as in the below:

```
PostgreSQL 8.4.2, compiled by Visual C++ build 1400, 32-bit
```

### **1.3.2 Creating geometries with PostGIS**

We will get started creating points.

#### **POINTS**

To create a point at (X,Y), type the following line into your query builder:

```
SELECT ST_Point(1, 2) AS MyFirstPoint;
```

Not much to it, is there? We did not specify a spatial reference system for our simple point. The default coordinate system type used is the basic Cartesian grid you all learned back in early childhood. In most real-world applications, we will need to be explicit about the spatial reference system being used.

#### **WHAT IS A SPATIAL REFERENCE SYSTEM?**

A spatial reference system is a way of denoting the coordinate system that is used to define geometry points. This is a bit of a simplistic definition but will do for now. You can be assured that if 2 geometries are in the same spatial reference system they can be overlaid since they use the same drawing board. PostGIS packages about 3000 of these and these are all denoted by numbers (usually European Petroleum Survey Group (EPSG) standard codes which are common in the industry) and can be looked up in the included spatial reference table. These define how geographic data is flattened and what units of measurement (degrees, meters, feet), the coordinate system uses. Spatial reference systems are in general only good for a specific region of the globe except for some of the degree based ones which need to be transformed to a Cartesian coordinate systems to do useful measurements in PostGIS.

With this in mind, let us create yet another point that has real geographical relevance.

```
SELECT ST_SetSRID(ST_Point(-77.036548, 38.895108), 4326);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Here we added a function to indicate that our point is using a spatial reference system known as WGS 84 Long Lat. This is the longitude and latitude that most people are familiar with. Incidentally, the above point is the Zero Milestone in Washington D.C. When the United States was a young (and small) nation, the city planners of the day intended for all roads to use this point to measure their distance. You can read more about it here. [http://en.wikipedia.org/wiki/Zero\\_Milestone](http://en.wikipedia.org/wiki/Zero_Milestone).

The function ST\_GeomFromText offers a more generic method for creating various types from a textual representation. This function is slower and less accurate than the ST\_Point function, but it is intuitive and applies to all geometric types. Our point creator would look like the following using ST\_GeomFromText:

```
SELECT ST_GeomFromText('POINT(-77.036548 38.895108)', 4326);
```

This approach can be used to create any geometry with what is known as the Well Known Text representation (WKT) of a geometry.

### **WELL KNOWN TEXT (WKT) REPRESENTATION OF GEOMETRIES**

Well Known Text representation is an OGC standard for representing geometries as Text. The ST\_AsText and ST\_GeomFromText are OGC functions found commonly in many OGC compliant spatial databases that convert back and forth between a databases native binary format to a textual display format.

You would have noticed that the result of running all the above geometry constructor statements would look something like this:

```
0101000020E6100000FD2E6CCD564253C0A93121E692724340
```

Most spatial databases store geometries in some sort of binary format which is impossible for the eye to make heads or tails of. This is why most spatial databases have functions to reformat the native binary format to WKT. In PostGIS, this is the ST\_AsEWKT function. Try formatting the above PostGIS binary format using this function:

```
SELECT  
ST_AsEWKT('0101000020E6100000FD2E6CCD564253C0A93121E692724340');
```

You will end up with the more readable WKT like representation of

```
SRID=4326;POINT(-77.036548 38.895108)
```

A variant of the ST\_AsEWKT function is ST\_AsText. This function returns the commonly accepted WKT format, which does not include the spatial reference identifier.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### **Lon/Lat vs. Lat/Lon**

You will note that the points are stored in longitude, latitude and not latitude, longitude. This storage is pretty common for spatial databases. Most people are used to thinking in latitude longitude, so one of the more common mistakes people make is flipping these coordinates and ending up in Antarctica when they mean Washington D.C.

### **LINESTRINGS AND POLYGONS**

Now let us move on to more complex examples of creating linestrings and polygons. PostGIS is primarily used to store and query geographic data. You saw in the previous examples how to represent point data. But we can also use PostGIS to represent any data that can be drawn using a Cartesian coordinate system. We will start by creating a linestring without specifying a spatial reference system. We will use the ST\_GeomFromText and omit the spatial reference parameter.

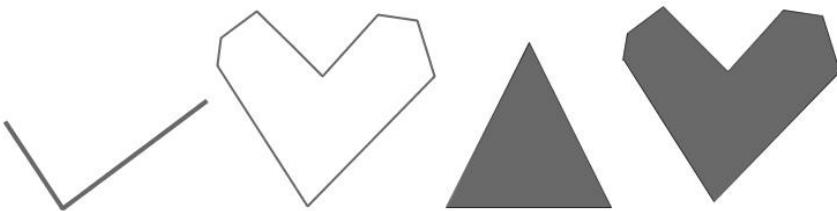


Figure 1.4 Linestrings and Polygons as demonstrated in below code snippets

```
SELECT ST_GeomFromText('LINESTRING(-14 21, 0 35 26)') AS
MyCheckMark;
```

This creates a check mark like linestring going through the origin. As you may be able to observe above, a LINESTRING is nothing more than a sequence of points. How about a heart with jagged edges?

```
SELECT ST_GeomFromText('LINESTRING(52 218, 139 82, 262 207, 245
261, 207 267, 153 207, 125 235, 90 270, 55 244, 51 219, 52 218)')
AS HeartLine;
```

The syntax for creating a polygon is similar to that of the linestring. The key difference is that the polygon must use *closed* linestrings, also known as *rings*. As you might have already guessed, a closed linestring is a linestring where the starting point coincides with the end point. Our heart linestring above is closed. To draw a polygon, we specify the linestring forming its boundary. Here is a triangle:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
SELECT ST_GeomFromText('POLYGON((0 1,1 -1,-1 -1,0 1))') As MyTriangle;
```

Since our heart is closed, we can use it as the boundary for a heart polygon. Our heart polygon would include all the interior points of the geometry as well as the original linestring forming the boundary.

```
SELECT ST_GeomFromText('POLYGON((52 218, 139 82, 262 207, 245 261, 207 267, 153 207, 125 235, 90 270, 55 244, 51 219, 52 218))') As HeartPolygon;
```

First thing interesting about the WKT representation of a polygon is that it has one additional set of parenthesis that a linestring does not have. Why is this? A polygon can have holes. Our triangle and our heart happen to not have any holes so the extra parenthesis appear redundant. When we explore polygons in detail in the next chapter, we will show you how to create donut-like polygons.

Now that we've covered the basics of how to create geometries, we'll cover the more common case of loading preexisting spatial data from other sources and querying that data.

## 1.4 Working with real data

In this section, we will cover how to load data you get in the most common two formats: delimited data and ESRI shapefile data. For delimited data, we will demonstrate how to convert it to spatial points, using the lessons we learned in the previous section. We will load a road linestrings data file using the PostGIS packaged ESRI shapefile loaders.

When working with PostGIS, you often start off by loading data either from a flat file delimited format or a spatial format and then perform various operations on it to get it into a structure suitable for spatial querying. A common task is to convert plain text representations of a location, such as a pairs of longitude/latitude coordinates, into spatial data types such as geometry or geography. For geometry data, we often need to transform the spatial reference system to a planar based spatial reference system that is suitable for planar measurement. This is so that when we do measurements against it, the measurements are in meaningful units instead of degrees.

### PostGIS Geography vs. Geometry measurement

Data in the geography data type must always be stored in WGS 84 long lat degrees. However all measurements in geography are expressed in meters. If your source data is in a planar coordinate such as a State Plane ft or meters, you must load it as geometry, transform it and then cast it to geography if you want to use the geography data type storage.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Geometry data on the other hand can be stored in any spatial reference system. However the measurements are always in the units of the spatial reference system. This means if your data is long lat, your measurements will be in degrees. This is not useful for most kinds of analysis, so for geometry data, people often transform their data to a measurement preserving spatial reference system that is valid for their area of interest. Some common ones are UTM zones meter, State Plane Feet or meters, and thousands more. We'll cover the nuances of this in later chapters and how to choose a spatial reference system.

If you are new to GIS or SQL, many of the terms and syntax we use in this section will be foreign to you. This is a crash course. Don't worry if you don't completely understand what is going on. These processes will be repeated in later chapters and become more clear when you see them again.

#### **1.4.1 Loading comma separated data**

We are going to start off by loading point data from a comma delimited flat file list of fastfood restaurants generously provided to us by fastfoodmaps.com. The tools you need to accomplish this are all available in PostgreSQL and apply to loading any kind of data in PostgreSQL.

Before we load in data, we first create the schema and table to house our data. We also create a lookup table for the franchises so we don't have to remember the codes. The table we create should have the same column ordering and preferably the same number of columns as the data we will be loading.

#### **Listing 1.1 Setup Fastfoods and Franchise lookup**

```
-- 1
CREATE SCHEMA ch01;

-- 2
CREATE TABLE ch01.lu_franchises(
    franchise_code char(1) PRIMARY KEY,
    franchise_name varchar(100));

-- 3
INSERT INTO ch01.lu_franchises(franchise_code, franchise_name)
VALUES ('b', 'Burger King'),
       ('c', 'Carl''s Jr'),
       ('h', 'Hardee''s'),
       ('i', 'In-N-Out'),
       ('j', 'Jack in the Box'),
       ('k', 'Kentucky Fried Chicken'),
       ('m', 'McDonald''s'),
       ('p', 'Pizza Hut'),
       ('t', 'Taco Bell'),
       ('w', 'Wendy''s');

-- 4
CREATE TABLE ch01.fastfoods
(
    franchise char(1) NOT NULL,
    lat double precision,
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

    lon double precision
);
-- 1 create logical container
-- 2 lookup table
-- 3 add franchise types
-- 4 table holds locations

```

We create (1) a schema to hold our data. A schema is a container you will find in most high-end databases. It logically segments objects (tables, views, functions etc) for easier management. Refer to Appendix D for more details. (2) Create a lookup table to map franchise codes to meaningful names (3) Add all the franchises we will be dealing with (4) Create a table to hold the data we will be loading. We only define columns that are in our source dataset. We will add additional columns after the load.

There are two common ways you can copy flat file data into PostgreSQL. You can use the built-in SQL function called COPY, or use the psql \copy command. The SQL function requires that the Postgres daemon process has access to the data path and also requires that you be logged in as a PostgreSQL super user. Other databases have similar SQL constructs. For example in SQL Server, you use the BULK INSERT SQL construct.

When using the SQL copy, the path is relative to the server. You can run the SQL COPY function anywhere you can run SQL. This can be in a .NET or PHP app, PgAdmin III, psql or some other third party database client tool.

The psql \copy command, on the other hand, is a client side feature built into PostgreSQL packaged psql command line tool. It requires the computer you are running psql on has access to the file path and that your OS account also has access to read the file. The path is relative to the client computer. We have these distinctions much more described on our Postgres Journal site -

<http://www.postgresonline.com/journal/index.php?archives/157-Import-fixed-width-data-into-PostgreSQL-with-just-PSQL.html>.

Using the SQL command way, we would do the following. Note that when we use the SQL COPY command, the /data/ path is referencing the path on the PostgreSQL server:

```

COPY ch01.fastfoods
FROM '/data/fastfoods.csv' DELIMITER ',';

```

Using psql, the command line tool, we would do the following and the path would be referencing a folder on our local computer that has psql.

```
\copy ch01.fastfoods from 'C:/data/fastfoods.csv' DELIMITER AS ',';
```

After we are done loading data we add a primary key to the table so we can uniquely identify each fast food restaurant.

```
ALTER TABLE ch01.fastfoods ADD COLUMN ff_id SERIAL PRIMARY KEY;
```

## ACCESSING PSQL FROM PGADMIN III

The easiest way to access psql commandline is via the Plugins menu icon in PgAdmin III. To do so, select a database you want to connect to and then click the PSQL Console menu options.

### 1.4.2 Spatializing flat file data

We need to convert our long lat coordinates into either geometry or a geography data type in order to take advantage of the spatial functionality and spatial index support that PostGIS provides.

If you are using PostGIS 1.4 or lower, you only have the option of the geometry data type to house your spatial data. If you are using PostGIS 1.5 or above, you have the additional option of the geography data type. The advantage of the geography data type is that it returns measurements in meters and allows you to store data using longitude latitude degree coordinates. As such you can cover the whole world with geography data and never have to learn anything about what we call spatial reference systems and projections. The disadvantage of geography is that for localized areas, it may not be as precise as using a geometry type. It also has much fewer functions available to it than geometry does. The speed of functions such as ST\_Intersects and other relationship functions on geography are currently a slower than the geometry data type, though this will change in time. Although there are fewer functions available for geography, it is fairly trivial in PostgreSQL to cast back and forth between these two spatial types. Many of the geometry functions, although they treat all data as planar can be used directly on geodetic coordinates for small regions or by transforming geography data to a suitable spatial reference system and transforming back to WGS 84 and casting back to geography.

### SQL Server 2008 geometry / geography vs. PostGIS geometry / geography

Those who have worked with SQL Server 2008 spatial types may recognize the similarity in naming here. The similarity is not just in naming but in basic use case as well. SQL Server 2008 geometry type is an OGC type similar to PostGIS OGC geometry type. Just like PostGIS geometry type, the SQL Server 2008 geometry type treats all data as planar data (data is measured using Pythagorean math). The PostGIS and SQL Server geography data types are NOT OGC types, though they try to follow many of the same function and naming conventions as OGC geometry functions. The PostGIS geography type was inspired by the SQL Server 2008 geography data type. Both geography types measure data along an ellipsoid instead of a cartesian plane. Where they are different is that: SQL Server 2008 doesn't have built in support for transformation from one spatial reference system to another for its geometry data type, where as PostGIS has very robust support for this for its geometry datatype. On the geography side of the fence, SQL Server 2008 supports many long lat based spatial reference systems. PostGIS geography currently only supports EPSG: 4326 -- also referred to as WGS 84 long lat.

WGS 84 long lat, is the most common spatial reference system for longitude latitude data.

In these next examples we will demonstrate how to define a geometry point data type in our fast foods table, update data to populate this new column and then repeat the same exercise using the geography data type.

#### **USING GEOMETRY DATA TYPE**

If we were to use the geometry data type we would do the following.

#### **Listing 1.2 Using geometry data type to store data**

```
-- 1
SELECT AddGeometryColumn('ch01', 'fastfoods', 'geom', 2163, 'POINT', 2);
-- 2
UPDATE ch01.fastfoods
    SET geom =
        ST_Transform(ST_GeomFromText('POINT(' || lon || ' ' || lat || ')', 4326), 2163);

-- 3
CREATE INDEX idx_fastfoods_geom ON ch01.fastfoods USING gist(geom);
vacuum analyze ch01.fastfoods;
-- 1 add column
-- 2 use equal area planar
-- 3 general maintenance
```

(1) We first add a geometry column to the fastfoods table to house our future spatial points. (2) Since geometry is a planar projection, we want our points to be in a projected coordinate system that covers our area of interest. We have picked EPSG:2163 which is an equal area projection covering continental US. The measurements of its coordinate system are in meters and are a little less accurate than modeling the earth as a sphere. (2) The PostGIS ST\_Transform takes our data from long lat degrees and projects it to North American Equal Area. The main drawback of this particular spatial reference system, is that since it covers a fairly large area, measurements will be worse than if we used geography which by default assumes a spheroid model of the earth, but speed will be faster and we will be free to use the vast array of geometry functions. We then (3) do some general house cleaning by creating a spatial index on our new column and vacuum analyzing. The vacuum analyzing step is a process that gets rid of dead rows and updates planner statistics. Its good practice to do this after bulk uploads. If not done, for newer versions of PostgreSQL, the vacuum daemon process which is enabled by default will eventually do it for you.

#### **USING GEOGRAPHY DATA TYPE**

If we were to use the geography data type we would do the following with our data.

In this example we explicitly use SRID=4326 in our construction. We do this for clarity and also for future use if PostGIS geography type does come to support other long lat spatial reference systems. In practice, you can leave it out and PostGIS will assume 4326.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### **Listing 1.3 Using geography data type to store data**

```
-- 1
ALTER TABLE ch01.fastfoods ADD COLUMN geog geography(POINT,4326);
-- 2
UPDATE ch01.fastfoods
  SET geog =
    ST_GeogFromText('SRID=4326;POINT(' || lon || ' ' || lat ||
')');
-- 3
CREATE INDEX idx_fastfoods_geog ON ch01.fastfoods USING gist(geog);
vacuum analyze ch01.fastfoods;
-- 1 add column
-- 2 use geodetic
-- 3 general maintenance
```

The steps we follow for creating geography columns is similar as what we did for geometry, but there are some differences. (1) geography uses the built in typmod feature introduced in PostgreSQL 8.3 (this is the main reason why you can't install PostGIS 1.5 on anything lower than 8.3). This allows adding geography columns to not require using a function to enforce constraints. (2) Instead of ST\_GeomFromText, we use the parallel ST\_GeogFromText, but the WKT representation is more or less the same. Since geography measures along a spheroid rather than a plane, we don't need to transform to get meaningful measurements. (3) As we did with geometry, we create our spatial index and do some general dead tuple cleanup and update of statistics.

#### **GENERAL RELATIONAL DATABASE MANAGEMENT**

Our fast foods data has no primary key index. Unfortunately nothing in the data file lends itself as a good natural primary key. For our later analysis, we will need to uniquely identify restaurants so that we don't double count them. Also for certain mapping applications and viewers, such as QGIS and MapServer, these have issues if your table has no primary key. They also stupidly have issues if you don't have a primary or unique key that is an integer. So that our data is most useful, we will create an auto number primary key on our fastfoods table. This will be important for example when we need distinct counts.

```
ALTER TABLE ch01.fastfoods ADD COLUMN ff_id SERIAL PRIMARY KEY;
```

Although not really necessary for this particular data set since it will not be updated, we will create a foreign key relationship between our fastfoods franchise column and our lookup table. This is useful to prevent people from introducing franchises we don't know about in our restaurants table. Putting in CASCADE UPDATE DELETE rules and constraints will also allow us to change the codes for our franchises if we want and have those update fastfoods and prevent people from deleting records in our lookup table that have corresponding data in our fastfoods restaurants table. The other benefit of foreign keys is that relational designers such as what you will find in OpenOffice Base and other ERD tools, will automatically draw in lines between the two tables to show the relationship.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
ALTER TABLE ch01.fastfoods
  ADD CONSTRAINT fk_fastfoods_franchise
    FOREIGN KEY (franchise)
    REFERENCES ch01.lu_franchises (franchise_code)
    ON UPDATE CASCADE ON DELETE RESTRICT;
```

We create an index to make the join between the two tables a bit more efficient.

```
CREATE INDEX fki_fastfoods_franchise ON ch01.fastfoods(franchise);
```

All this code was autogenerated by using PgAdmin GUI. There really isn't any need to memorize how to do this in SQL although it is very standard Data Definition Language (DDL) code across many relational databases.

### **1.4.3 Loading data from spatial data sources**

The most common spatial format that data is distributed in is the ESRI shapefile format. In PostGIS 1.5 and above, there is a GUI tool called **shp2pgsql-gui**, that makes loading of this kind of data as well as plain Dbase DBF data very simple and user friendly. It can also be used as a plugin to PgAdmin III. Details on how to install and get started with it are in Appendix B. It is packaged with the Windows Stack Builder installs as well as the OpenGeo Suite 1.9+ GIS stack installs.

#### **PGADMIN III**

PgAdmin III is the free administrative GUI that comes packaged with PostgreSQL. It can also be downloaded from <http://www.pgadmin.org/> and installed separately on any client computer. There are precompiled binaries available for most OS. PgAdmin III 1.9 and above supports a Plugins architecture which allows you to call shp2pgsql-gui and any other executables you like from within PgAdmin. The psql commandline client is packaged as a plugin with it. All this is configurable by editing the plugins.ini file.

When shp2pgsql-gui is used as a plugin in PgAdmin III, it reads your database credentials directly from PgAdmin III for the selected database. Even if you are using a lower version of PostGIS, you can still use this tool against a PostgreSQL 8.2 or higher with PostGIS 1.3 or higher version installed. You can even use the DBF only loader portion on a database that doesn't even have PostGIS installed.

For our next example we will load US state roads data and will take advantage of the convenience of shp2pgsql-gui.

The shp2pgsql-gui supports loading data into both geometry and geography data types. We will demonstrate loading road network data we downloaded from <http://www.nationalatlas.gov/atlasftp.html#roadtr1>

For this example we will be using it via PgAdmin and accessing from the Plugins menu



Figure 1.5 Plugins menu of PgAdmin III show the PostGIS Shapefile and DBF Loader

The plugins option above will be disabled until you select a database from the PgAdmin database tree. When you click the PostGIS Shapefile and DBF Loader menu option, the following dialog comes up with the credentials of the selected database already filled in.

#### **LOADING DATA INTO GEOMETRY DATA TYPE**

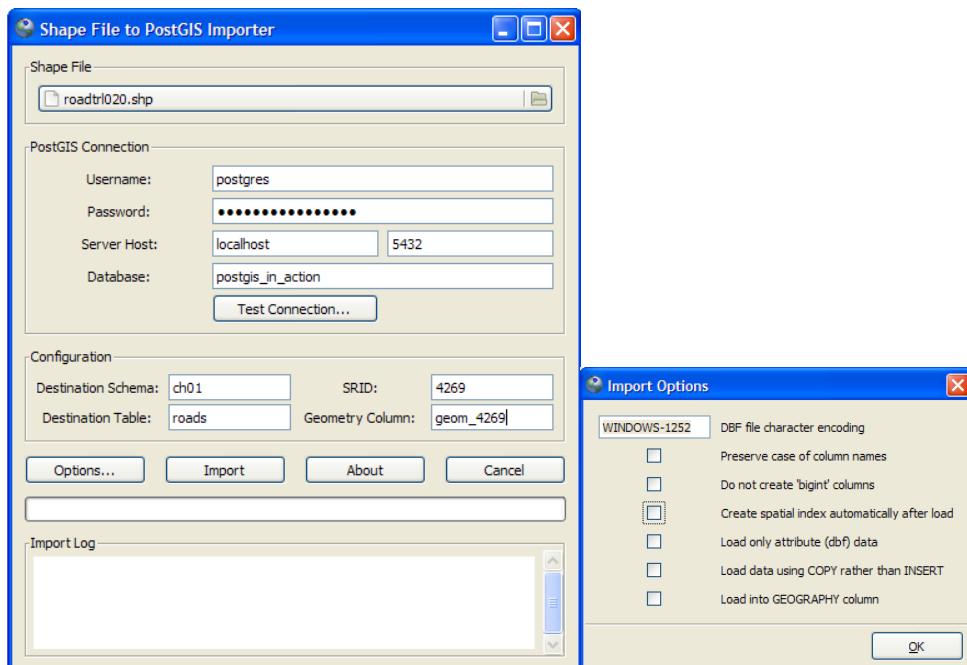


Figure 1.6 Loading into the geometry data type

In order to load into the geometry data type, we browse to the file. Make sure to denote we will be loading into the ch01 schema and a new table called roads. shp2pgsql-gui will create the table for us. Note that we also changed the default geometry column name to geom\_4269. We then click the Options button and uncheck the **create spatial index** and make sure Load into Geography column is unchecked. Normally you want a spatial index created when loading data, but in this case we don't because the geom\_4269 is a temporary column that we will be dropping once we are done with it. In the chapter on spatial reference systems, we'll discuss in more detail the meaning of SRID. Suffice for now that the road data is in a long lat spatial reference system called North American Long Lat Datum 1983 (4269) which is very similar to WGS 84 long lat (4326). So similar that in most cases you can consider them the same. When we are done, setting the options, we click the Import button.

Neither WGS 84 long lat nor NAD 83 long lat is useful for measurement in geometry type. They would get squashed into a rectangular planar grid where the x axis would be longitude and y axis would be latitude. This squashing process is often referred to as a Plate Carrée projection. The other reason we will not be using long lat in geometry is that the measurements would return degrees when loaded in geometry type. With that said, we will be transforming this column to the same spatial reference system we are using for our fast foods so that it will be useful for our measurement and other processing calculations and that we can do comparisons between them and overlay them together on the same map.

#### **Listing 1.4 Using geometry data type to store roads data**

```
-- 1
SELECT AddGeometryColumn('ch01', 'roads', 'geom', 2163,
'MULTILINESTRING', 2);
-- 2
UPDATE ch01.roads
SET geom =
    ST_Transform(geom_4269, 2163);
-- 3
SELECT DropGeometryColumn('ch01', 'roads', 'geom_4269');

-- 4
CREATE INDEX idx_roads_geom ON ch01.roads USING gist(geom);
vacuum analyze ch01.roads;
-- 1 add column
-- 2 use equal area planar meters
-- 3 drop old column
-- 4 general maintenance
```

In order to load the roads data into geometry data type format, we use the shp2pgsql-gui to load it into its native spatial reference system. Neither the shp2pgsql commandline nor the gui has transformation abilities. To make the geometry good for measurement, we transform after we have loaded it in the database. To do so (1) We add a new column that will hold a good for measurement cartesian spatial reference that covers our area of interest (2) We

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

update this new column by transforming our imported geometry data to the new spatial reference system (3) After we do that, we no longer have a need for the original column, so we drop it. (4) Then we do our usual create index and vacuum analyze.

In the next example, we will repeat the same exercise, but loading into the geography data type.

#### **LOADING SPATIAL DATA INTO THE GEOGRAPHY DATA TYPE**

For this particular data set, loading into geography data type is much simpler than what we had to do for geometry. The reason is that our data is already in long lat units and the long lat units are NAD 83 long lat. Although its not WGS 84 long lat, we can pretend it is since they are both almost exactly the same for most areas. If you had data in NAD 27, longlat, (4267) you can not pretend. NAD 27 is different enough in many cases from WGS 84 to screw up all your calculations. For such data, you would have to bring your data in as geometry, transform it to 4326, cast it to geography. Something along the lines of geography(ST\_Transform(geom\_4267,4326))

For this case we simply use the loader.

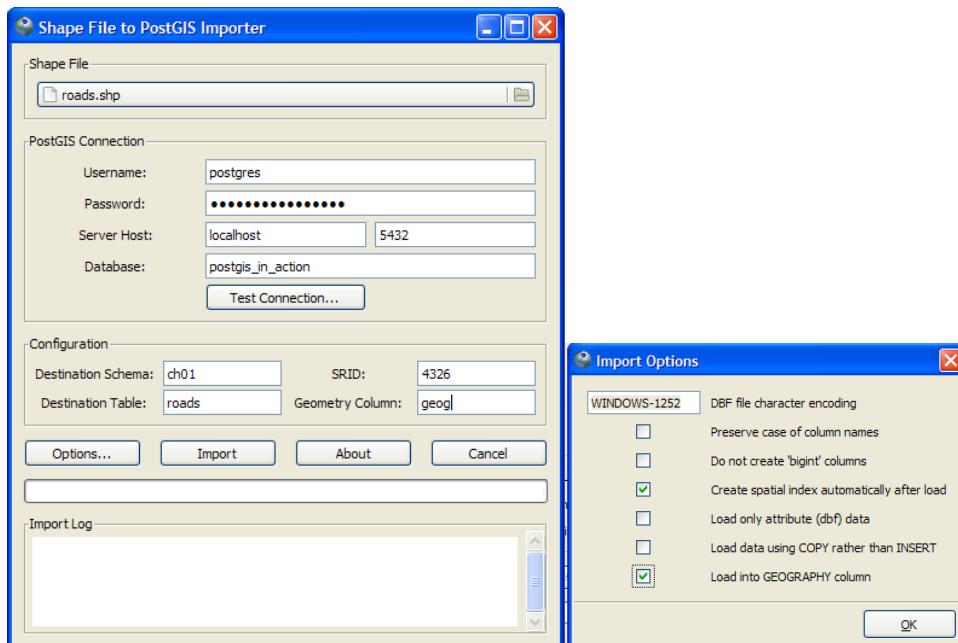


Figure 1.7 Loading data into geography data type. We pretend our data is 4326 instead of 4269 since they are similar. We go ahead and index and check load into Geography.

As you can see in the figure, our settings are more or less the same as we chose for geometry, except that we chose to create the index and load into geography. We also changed the name of the column so we know it holds geography data type data instead of geometry. We lie about the SRID and say its 4326 (close enough to 4269). There is nothing else we need to do after this except to vacuum analyze ch01.roads .

## 1.5 Using spatial queries to analyze data

Now that we have spatial data loaded in our database, we are now able to spatially analyze this data both with standard statistical SQL queries that just yield raw numbers and text as well as with queries that return geometric objects that can be rendered on a map. We will first start off by doing statistical queries that count restaurants by their proximity to roads and also determining which roads have the most number of restaurants.

For these examples since our data is stored using North America Equal Area meter spatial reference system (2163), all our units are in meters. To convert to miles we multiple our desired miles by 1609 to get meters.

### 1.5.1 Proximity Queries

One of the most common uses of PostGIS and other spatial databases is to do tabulations of the proximity of objects to other objects. For the next couple of examples, we will demonstrate how to combine the ST\_DWithin function of PostGIS with standard SQL aggregation functions like COUNT and constructs like JOINS and GROUP BY.

#### HOW MANY FASTFOOD RESTAURANTS BY CHAIN ARE WITHIN 1 MILE OF A MAIN HIGHWAY?

For this case, we answer the question and also order our results by the number of restaurants by franchise, with the most numerous being at the top. Although we are using the geometry type here, the query itself would be written exactly the same for a geography column since there is an ST\_DWithin function available for both geometry and geography types.

#### Listing 1.5 List franchise name, count of restaurants on a Principal Highway

```

SELECT ft.franchise_name,
--1
  COUNT(DISTINCT ff.ff_id) As tot
FROM ch01.fastfoods As ff
--2
  INNER JOIN ch01.lu_franchises As ft
  ON ff.franchise = ft.franchise_code
--3
  INNER JOIN ch01.roads As r
  ON ST_DWithin(ff.geom, r.geom, 1609*1)
WHERE r.feature LIKE 'Principal Highway%'
GROUP BY ft.franchise_name
ORDER BY tot DESC;
--1 distinct count
--2 non-spatial join
--3 spatial join

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In this example we (1) use an ANSI-SQL COUNT(DISTINCT) contract to ensure we count a fastfood restaurant only once even if it is within a mile of more than one highway segment (2) We use a regular non-spatial join with our lookup table to grab the meaningful name of the franchise (3) We use a spatial join between fastfoods and roads to only pick up restaurants within 1 mile of a principal highway.

Which gives us this:

franchise_name	tot
McDonald's	5343
Burger King	3049
Pizza Hut	2920
Wendy's	2446
Taco Bell	2428
Kentucky Fried Chicken	2371
:	

#### WHICH HIGHWAY HAS THE LARGEST NUMBER OF FAST FOOD RESTAURANTS WITHIN A 1/2 MILE RADIUS?

#### **Listing 1.6 Return the principal highway that has the most restaurants and how many**

```
SELECT r.name,
-- 1
  COUNT(DISTINCT ff.ff_id) As tot
FROM ch01.fastfoods As ff
  INNER JOIN ch01.roads As r
-- 2
  ON ST_DWithin(ff.geom, r.geom, 1609*0.5)
WHERE r.feature LIKE 'Principal Highway%'
GROUP BY r.name
-- 3
ORDER BY tot DESC LIMIT 1;
-- 1 distinct count
-- 2 within 1/2 mile
-- 3 one with greatest total
```

(1) Since we are joining with roads that has multiple records, it is possible for a restaurant to be within 1/2 mile of one record, we use the SQL DISTINCT clause to prevent double-counting (2) We use the PostGIS ST\_DWithin to only consider those points within 1/2 mile of a road where the road feature type is a Principal Highway. (3) We only care about the road that has the largest number of restaurants which we can get by ordering by total count per road and picking the one with largest count (DESC).

Which gives us an answer of US Route 1 with a total of 414 restaurants.

### **1.5.2 Viewing spatial data with OpenJump**

Although PostGIS is good for doing quick spatial analysis that are next to impossible to do simply by inspecting a map, it can also be used as a data source for maps or to create additional derivative geometries suitable for highlighting key regions on a map.

One fairly popular function used in PostGIS for visualization is the ST\_Buffer function. You can think of the ST\_Buffer function as the visual companion to the ST\_DWithin function. It will take any geometry and radially expand it  $r$  units, where  $r$  is in units of the spatial reference system for geometry and in meters for geography. The geometry formed by this expansion is called a buffer zone or corridor. For this next example, we'll ask how many Hardee's restaurants are within 10 miles of the portion of US Route 1 that runs thru Maryland.

```
SELECT COUNT(DISTINCT ff.ff_id) As tot
FROM ch01.fastfoods As ff
INNER JOIN ch01.roads As r
ON ST_DWithin(ff.geom, r.geom, 1609*10)
WHERE r.name = 'US Route 1' AND ff.franchise = 'h'
      AND r.state = 'MD';
Which gives us an answer of 3.
```

To spot check our results, we used OpenJump to overlay the portion of US Route 1 that is in Maryland, the Hardee's restaurants within 10 miles of it, and the 10 mile corridor. In order to display geometries with the plain vanilla OpenJump install, you need to use the ST\_AsBinary function to convert the PostGIS geometry to an OGC standard binary format. Details on using OpenJump and other viewing tools that support PostGIS are discussed in Chapter 12.

This first statement will draw the road segments that represent US Route 1 in Maryland.

```
SELECT r.gid, r.name, ST_AsBinary(r.geom) As wkb
FROM ch01.roads As r
WHERE r.name = 'US Route 1' AND r.state = 'MD';
```

Then we overlay the Hardee's restaurants that are within 10 miles of those routes. Note, we don't use INNER JOIN here since it would result in duplicates where a Hardee's restaurant is within 10 miles of more than 1 US route record in MD.

```
SELECT
ST_AsBinary(ff.geom) As wkb
FROM ch01.fastfoods As ff
WHERE EXISTS(SELECT r.gid
            FROM ch01.roads As r
            WHERE ST_DWithin(ff.geom, r.geom, 1609*10))
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
        AND r.name = 'US Route 1' AND r.state = 'MD' AND
ff.franchise = 'h');
```

Then we overlay the 10 mile corridor:

```
SELECT ST_AsBinary(ST_Union(ST_Buffer(r.geom,1609*10))) As wkb
FROM ch01.roads As r
WHERE r.name = 'US Route 1' AND r.state = 'MD';
```

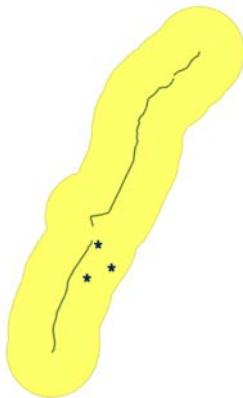


Figure 1.8 US Route 1 in Maryland, with 3 Hardee's restaurants in the 10 mile buffer, and the 10 mile buffer around the route

As you can see, not only is PostgreSQL/PostGIS a good analytical tool for doing simple number stats on spatial data, it can be used to render portions of an area of interest on a map. It can also be used to generate derivative geometries such as buffers that help highlight results.

## 1.6 Summary

In this chapter, we have given you a small taste of what spatial is, how it can be used, what geometry and geography data types are and how they fit in a relational database system. We sowed the budding idea of how to model real world objects and space with spatial constructs.

We then introduced the PostgreSQL database and its spatial companion PostGIS. We demonstrated how PostgreSQL and PostGIS can be used together to analyze spatial patterns in data. We hope we have convinced you that the PostgreSQL/PostGIS combination is one of the best choices if not the best choice for spatial analysis.

Some of the SQL examples we demonstrated were intermediate level. If you are new to SQL or spatial databases, these examples may have seemed daunting. In the chapters that

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

follow, we'll explain the functions we've used here and the SQL constructs. When you revisit this chapter, we hope the strategies behind these examples will become more clear.

Although spatial modeling is an integral part of any spatial analysis, we pointed out that there is no absolute right or wrong answer in spatial modeling or modeling in general. There are models that are a better fit for a particular purpose. Modeling is inherently a balance between simplicity and adequacy. You want to make your model as simple as possible for easy exploration, but you want your model to be complex enough to adequately represent the conditions and laws that govern the questions you are trying to answer and the world you are trying to explore. You also want your model to be simple enough so calculations are fast enough for your specific use cases. In that balance lays the challenge.

Before we can continue our journey, we must first analyze the different geometry types that PostGIS offers us and learn how to create these and when it's appropriate to do so. We shall explore geometries in greater detail in chapter two.

# 2

## *Geometry Types*

This chapter covers

- PostGIS geometry\_columns meta-table
- Geometry types: points, linestrings, polygons
- Geometry collection types: multipoints, multilinestrings, multipolygons
- Curved geometry types and 3D geometry types

In the first chapter we gave you a brief taste of what PostGIS is and what are the basic geometries it supports. In this chapter we continue our discussion by explaining how PostGIS manages geometry data stored in the database. We'll learn about tables that exist in all PostGIS-enabled databases that provide an inventory of geometry table columns and available spatial reference systems. We then show you the defining characteristics of points, linestrings, and polygons and how to work with them in a PostGIS enabled database. After covering these single geometries, we will move onto geometries that are made up of collections of single geometries: multipoints, multilinestrings, multipolygons, and geometrycollections. We then demonstrate creating the less commonly used curved geometries and 3D geometries and outline the issues to consider when using these less common geometry types.

### **2.1 Geometry columns in PostGIS**

PostGIS extends PostgreSQL by introducing a data type called `geometry`. Most of the functions that come packaged with PostGIS work with the core set of geometry types (points, linestrings, polygons, and their multi counterparts), but some are specific to linestrings such as the linear referencing functions, some ignore the 3rd and 4th coordinates, and some reject curved geometries or do not work well with geometry collections. For all intents and purposes, you can treat `geometry` to be on par with other PostgreSQL data types such as dates, numbers, and text.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

## PostgreSQL geometry data types vs. PostGIS geometry data types

PostgreSQL does have its own built-in geometry data types. These are incompatible with the PostGIS geometry data type and have little or no third-party visualization support. These geometry types have existed since the dawn of PostgreSQL and do not follow the OpenGIS Consortium standards nor do they support spatial coordinate systems. These types are broken out into individual types called point, polygon, lseg, box, circle, and path. The built-in PostgreSQL box type vs. PostGIS box2d support type have similar names but different incompatible data types.

### **2.1.1 The *geometry\_columns* table**

PostGIS uses a table named *geometry\_columns* to store meta data associated with the geometry columns in the database. The installation of PostGIS automatically creates this table. The *geometry\_columns* table provides housekeeping information about geometry columns in the database, and is commonly used by third-party tools to gather a list of geometry layers in the database. Other OGC compliant spatial databases also have a table with similar or exact name because this table is defined in the OGC Simple Features for SQL (SF SQL) specs.

### **GEOMETRY COLUMNS VERSUS LAYERS**

When presented to end users in a graphical display tool, geometry columns in a spatial table are often referred to as layers or feature classes.

As of PostGIS 1.4, the *geometry\_columns* table is made up of seven columns. Four of these f\_table\_catalog, f\_table\_schema, f\_table\_name, f\_geometry\_column are used to store the name of the database (also known as the catalog), table schema, table name, and geometry column name. The three final columns merit more discussion: coord\_dimension, srid, and type.

#### **COORD\_DIMENSION**

This is the coordinate dimension of the geometry column, permissible values are 2, 3, and 4. Yes, PostGIS supports up to 4-dimensions. The 4th dimension is non-spatial and often referred to as the m coordinate (The m stands for “measure”). All of the geometry manipulation features supported by PostGIS will treat the 4th dimension as an extra attribute of a point in the geometry rather than as another spatial dimension. The 4th dimension can be a temporal dimension, but you could use it as an index for anything. We will talk more about this m coordinate when we get to points.

#### **What is a dimension?**

In spatial speak, there are 2 kinds of dimensions.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

There is the coordinate dimension which defines the number of axes you have. For example geometries that occupy x, y, z or x, y, m have a coordinate dimension of 3. Those that have x, y, z, m have a coordinate dimension of 4.

The second type of dimension is the geometry type dimension. The geometry type dimension can never be greater than the coordinate dimension. A point and multipoint regardless of what coordinate dimension it has always has a geometric dimension of 0. A linestring and multilinestring are 1-dimensional, and a polygon and multipolygon are 2 dimensional. You will notice that we have no 3-dimensional geometry types. Such types would be volumetric surfaces such as boxes and spheres and amorphous objects you would find in real 3d space. These are not supported in PostGIS 1.\* versions, but in PostGIS 2+, these will be supported in new types called Polyhedral Surfaces and Triangulated Irregular Network (TINS).

#### **SRID**

SRID stands for Spatial Reference Identifier and is an integer that relates back to the primary key of the meta table spatial\_ref\_sys. PostGIS uses this table to catalog all the spatial reference systems available to the database. The spatial\_ref\_sys meta table contains the name of the spatial reference system, details needed to re-project to another system, and the authority of the system.

#### **SRID vs. SRS ID**

In common GIS lingo, there is another term with similar meaning called SRS ID (Spatial Reference System Identified) which is usually represented as the authority name plus the unique identifier the authority uses for a spatial reference system.

For example, the common WGS 84 long lat has an SRS ID of EPSG:4326, where EPSG stands for European Petroleum Survey Group. Most of the spatial reference systems defined in PostGIS are from EPSG so the SRID used in the table is usually the same as the EPSG identifier. This is not the case with all spatial databases and in fact the same spatial reference system can go under multiple identifiers.

Keep in mind that using a different SRID does not change the fact that the coordinate system underlying PostGIS is rectangular Cartesian. This fact comes to prominence when we start to deal with geographical features where the curvature of the earth comes into play. Suppose we are trying to measure the distance between two points that happen to represent New York and Los Angeles using the PostGIS function ST\_Distance(). Because PostGIS is based on a regular X-Y coordinate system, ST\_Distance() will simply return the distance calculated using Pythagorean theorem, not the great circle distance that one might expect. Even if we were to use spherical coordinates like latitudes and longitudes to map our features, the underlying calculations will all assume planar. So to correctly calculate

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

distances when your data is stored in spherical coordinates, you need to first transform to a planar based spatial reference system or use the PostGIS geography data type instead of PostGIS geometry introduced in PostGIS 1.5 to store long lat data. The PostGIS geography data type has similar functions for inserting and uses same text representations, except that all coordinates are input and expressed in WGS 84 long lat (EPSG:4326) , measurements are always in units of meters/square meters, has fewer functions, and uses a view called `geography_columns` instead of `geometry_columns` table that is always in sync with the tables. We shall briefly cover this type in later chapters of this book.

The default SRID in pre-2.0 versions of PostGIS is -1. In PostGIS 2.0, the default will be 0 to conform to OGC standard. Both values of -1 and 0 represent unknown SRID. Should you use the unknown SRID? The answer is no if you are working with geographic data. If you know the spatial reference system of your data and presumably you should if you have real geographic data, then you should explicitly specify it. If you are using PostGIS for non-geographical purposes, such as modeling a localized architecture plan or demonstrating analytic geometry principles, you will be perfectly fine keeping your spatial reference as unknown.

You should also know that even though the `spatial_ref_sys` table has close to 4000 entries, you will encounter plenty of instances where you have to add in SRIDs not already in the table. You can also be adventurous and define your own custom spatial reference system and add it to the `spatial_ref_sys` table in any PostGIS database.

#### **TYPE**

This final column stores the geometry type as a varying character field—‘POINT’, ‘LINESTRING’, ‘POLYGON’, ‘MULTIPOLYGON’ and others. Another admissible data value here is ‘GEOMETRY’. ‘GEOMETRY’ defines a heterogeneous geometry column that can store any geometry type.

The final admonition we need to make about the `geometry_columns` table is that in pre-2.0 versions of PostGIS, it is for informational purposes only. Manipulating the values in the table has no bearing whatsoever on the actual geometry column referred to by the table. For example, you can start off creating a column to be two-dimensional polygons. You can even insert a few rows of polygons into the new column, but should you return to the `geometry_columns` table and change the type from polygons to linestrings, your data will not be revalidated. You will simply end up with meta data that is out of sync with the actual data. For this reason, we do not recommend that you edit the `geometry_columns` tables directly.

### **2.1.2 Interacting with the `geometry_columns` table**

To avoid editing the records directly in the metadata tables, PostGIS offers five functions, when used, will handle any necessary interactions with the `geometry_columns` table.

- `AddGeometryColumn` – adds a geometry column to a table and adds pertinent metadata to the `geometry_columns` table

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- DropGeometryTable – deletes a table with a geometry column and also deletes all metadata about those geometry columns from the geometry\_columns table
- UpdateGeometrySRID – should you stamp the wrong SRID on your table geometry column, this will fix all records and also update the geometry\_columns metadata table.
- Probe\_Geometry\_Columns – has existed for as long as PostGIS. It will not destroy any information already in geometry\_columns table, but will only add new valid entries and give you some stats as to the number it added. It will not inspect views or tables that lack postgis constraints so these kinds of tables and views can not be added with probe\_geometry\_columns. It will also not add missing constraints to tables.
- Populate\_Geometry\_Columns (Introduced in PostGIS 1.4) - a bit more sophisticated than Probe\_Geometry\_Columns – can be used to populate geometry\_columns meta data by inspecting table views and tables that lack geometry constraints. Note that if you call this without any arguments it will delete all entries in geometry\_columns and repopulate the table. As such it may take longer to run than probe\_geometry\_columns. It also adds constraints to the tables it registers that lack srid and type constraints.

Of the five functions described above, only the Populate\_Geometry\_Columns can be used to register views in the PostGIS geometry\_columns, however you are still free to register these and other tables manually by inserting into geometry\_columns directly.

In all of our examples thus far, we used the AddGeometryColumn function to handle the creation of new geometry columns. Although you do not need to use the above functions for creating and maintaining the geometry columns, for pre-2.0 versions of PostGIS, we highly recommend doing so since the functions will automatically register and upkeep entries in the geometry\_columns table. To demonstrate the advantage of using the maintenance functions, we will add a new geometry column of points to a table without using the AddGeometryColumn function. We would need to go through the following steps to achieve the same final result.

1. ALTER TABLE to create a new geometry column
2. Add columnar constraint enforce\_geotype\_poi\_geom to the table to ensure that only points are in the new column.
3. Add columnar constraint enforce\_dims\_poi\_geom to the table to ensure only 2D geometries can be added to the column.
4. Add columnar constraint enforce\_srid\_poi\_geom to the table to only permit SRID of -1.
5. Add an entry to the geometry\_columns metatable noting that the new points column is a coordinate 2-dimensional point layer with an unknown spatial reference system.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

As you can see, using maintenance functions wherever possible greatly simplify matters and reduce the likelihood of you forgetting a step. Now that we have a general idea of the supporting structures and maintenance functions PostGIS offers for geometries, we will explore what kinds of geometries PostGIS offers and how to create and add them to our database.

## **2.2 A panoply of geometries**

PostGIS has a large variety of geometry types to choose from to help you model the real world or, for that matter, anything you can think of that involves shapes. In this section we will explore the geometric data types in PostGIS in detail. We will concentrate on the defining attributes of geometries, addressing questions such as “What makes a linestring a linestring?”

### **2.2.1 What is a geometry?**

In this book, we use the term “geometry” to both indicate the general idea of a geometric shape as used in GIS and to mean PostGIS geometric data types.

#### **POSTGIS GEOMETRIC DATA TYPES COMPLY WITH OGC STANDARDS**

PostGIS geometric data types follow the OpenGIS standard geometry definitions. This compliance will allow you to leverage the knowledge you learn in working with one spatial database to be transferred to another so long as they are both generally OGC compliant.

The best way to think about a geometry data type is to draw an analogy to numeric data types. In general, setting up a column in a data table and declaring that it will store numeric data is not precise enough. We go further to make distinctions between floating point and fixed numbers. With fixed numbers, we can specify more attributes like numbers of places after the decimal point or whether a column will be signed or unsigned. Just like numeric data types, a geometry data type is more akin to a base data type from which we can derive more specific data types. At the root of geometry data types, we find points, linestrings, polygons, and curves each with defining attributes of their own.

Let us delve into some concrete examples. We start by creating a table to store all the geometries that we will be showing.

```
CREATE TABLE my_geometries
(id serial NOT NULL PRIMARY KEY, name varchar(20));
```

For now, our table will have nothing more than an auto-numbered column (also known as a serial column in PostgreSQL parlance), and a column to store the name of the geometries. For sake of brevity, we are placing our new table in the default public schema. Without prefixing the table name, PostgreSQL follows the schema search path. Unless you have tinkered with the search path order, the default search always starts with the \$user and then

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

the public schema. For production work involving databases with many tables, we strongly urge you to create your own schemas and to organize your tables around these schemas. Not only will you keep your tables in logical units, but you will find it easier when it comes time to upgrade PostGIS and for performing selectively backup and restores.

## 2.2.2 Points

All PostGIS geometries are based on the Cartesian coordinate system. As such, in 2D space a point is specified by its x and y coordinate. In 3D space, a point has x, y, z coordinates, in 2DM space a point has x, y, and m coordinates, a PointM geometry (a PostGIS geometry type in its own right to distinguish it from points in 3D space). In 3DM space, a point has an x, y, z, and m coordinate (In OGC nomenclature this is often represented as a distinct data type called PointMZ, but PostGIS does not have this data type since anything with 4 coordinates cannot have anything but x, y, z, and m.)

### What is the M Coordinate?

The M coordinate stands for “measure” and is an additional numeric double-precision value that can be stored for each point in a geometry. It can be negative or positive and its units need not have any relationship to the underlying spatial reference system of the geometry. There are two dimensional variants of such geometries—2DM and 3DM. 2DM has x,y,m and 3DM has x,y,z,m. You can use the extra measure coordinate to store additional information associated with the spatial coordinates. Scientific data often use the extra variable to hold various measurements taken at the point, hence the term “measure.”

The benefit of using M to store additional information directly becomes clear the minute you move beyond points. Suppose that you have a linestring made up of many points, each with its own measure. Without the m coordinate, you would always have to hang an additional table that explodes the linestring into points for the sake of storing the measure data.

There are several functions in PostGIS and also defined by the OGC SFS standard to deal specifically with M coordinate data. We shall explore these in a later chapter.

The following call to the AddGeometryColumn function creates a new column in a table to store points as well as add two pizza parlors and a home to our simple Cartesian world.

### **Listing 2.1: Adding Points**

```
SELECT AddGeometryColumn('public','my_geometries','my_points',-1,'POINT',2);
INSERT INTO my_geometries (name,my_points)
VALUES ('Home',ST_GeomFromText('POINT(0 0)' ));
INSERT INTO my_geometries (name,my_points)
VALUES ('Pizza 1',ST_GeomFromText('POINT(1 1)' )) ;
INSERT INTO my_geometries (name,my_points)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
VALUES ('Pizza 2',ST_GeomFromText('POINT(1 -1)'));
```

The code above adds your home to the origin and two pizza parlors, one at (1,1) and one at (1,-1). Pull out a GIS desktop tool and you can see the three points as in Figure 2.1



Figure 2.1 Three points created using code in Listing 2.1

### **2.2.3 Linestrings**

Linestrings are defined by at least 2 distinct points. Just like points there are 4 dimensional variants of linestrings – linestring with points represented with x,y coordinates, linestrings with points in x,y,z coordinates, linestrings with points in x,y,m coordinates (also known as a LINESTRINGM), and finally linestrings with points represented in x,y,z,m coordinates. Let us use Listing 2.2 to add a simple 2D linestring column with two rows.

#### **Listing 2.2: Adding Linestrings**

```
SELECT AddGeometryColumn ('public','my_geometries','my_linestrings',-1,'LINESTRING',2);
INSERT INTO my_geometries (name,my_linestrings)
VALUES ('Linestring Open',ST_GeomFromText('LINESTRING(0 0,1 1,1 -1)' ));
INSERT INTO my_geometries (name,my_linestrings)
VALUES ('Linestring Closed',ST_GeomFromText('LINESTRING(0 0,1 1,1 -1, 0 0)' ));
```

The first INSERT statement above added a linestring starting at the origin, going to (1,1) and terminating at (1,-1). This is an example of an open line string where the starting and end points are not the same. The second INSERT statement added a closed linestring. A closed linestring is a linestring where the starting and end points are the same. In modeling real world geographic features, open linestrings predominate over closed linestrings. Rivers, streams, fault lines, and roads rarely start where they end. Closed linestrings, as you will soon see, will become the basis for constructing polygons.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

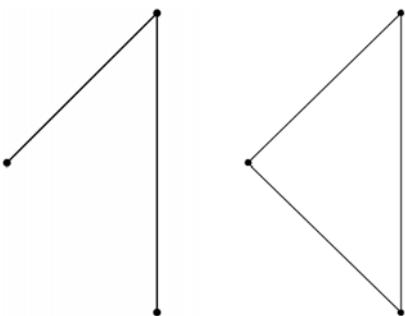


Figure 2.2 Open and Closed Linestrings created using code in Listing 2.2 points that make up line are shown as well

The concept of simple and non-simple geometries comes into play when describing linestrings. A simple linestring cannot have self-intersections except at the starting and end points. Using the definition above, a closed line string with no self-intersections can be considered simple. A linestring that crosses itself is not simple. PostGIS provides a function, `ST_IsSimple` that tests to see if a geometry is simple. The query below will return false.

```
SELECT ST_IsSimple(ST_GeomFromText('LINESTRING(2 0,0 0,1 1,1 -1)'));
```



Figure 2.3 A non-simple linestring tested for simplicity

Another important conceptualization is that although a linestring is defined using a finite set of points, in reality you should think of it as being composed of an infinite number of points. This distinction is obvious when you ask a question such as what point on a linestring is closest to this polygon or other geometry. The answer to such a question rarely coincides with a point used to define the linestring itself.

## 2.2.4 Polygons

Now things get a little more interesting. We will build on top of familiar geometries to form polygons. We start with a simple triangle: Take a closed linestring with at least 3 distinct

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

points. This linestring takes the shape of a triangle. All points enclosed by the linestring and the points on the linestring itself form the polygon. The closed linestring delineating the outer boundary of the polygon is called the ring of the polygon when used in this context. More specifically, the exterior ring.

### **Listing 2.3 Adding Polygon**

```
SELECT AddGeometryColumn('public','my_geometries','my_polygons',-  
1,'POLYGON',2);  
INSERT INTO my_geometries (name,my_polygons)  
VALUES ('Triangle',ST_GeomFromText('POLYGON((0 0, 1 1, 1 -1, 0  
0))'));
```



Figure 2.4 Triangle-shaped polygon as created in Listing 2.3

Many polygons used in geographical modeling consist of a single ring, but polygons can also have multiple rings. To be precise, a polygon can have one exterior ring and zero or more inner rings. Each interior ring creates a hole in the overall polygon. This is why we need the seemingly redundant set of parenthesis in the text representation of polygons. The Well-Known Text (WKT) of a polygon is a set of closed line strings with the first being the exterior ring and all subsequent designating the inner rings.

### **Listing 2.4 Adding polygon with interior rings**

```
INSERT INTO my_geometries (name,my_polygons)  
VALUES ('Square with 2 holes',ST_GeomFromText('POLYGON((-0.25 -1.25,-0.25  
1.25,2.5 1.25,2.5 -1.25,-0.25 -1.25),(2.25 0,1.25 1,1.25 -1,2.25 0),(1 -1,1  
1,0 0,1 -1))'));
```



Figure 2.5 Polygon with interior rings (holes) created by Listing 2.4

Always add the extra set of parenthesis in the WKT, even if your polygon has just a single ring. Some tools may work with single-ringed polygons using only one set of parenthesis without complaining, but not PostGIS.

In the real world, multi-ringed polygons play an important part in excluding bodies of water within geographical boundaries. For example, if we were planning a surface transit system in the greater Seattle area, we can start by outlining a big polygon bounded by the Interstate 5 on the west and Interstate 405 on the east. We can then start to tack down starting and terminal points of popular bus lines and let the computer choose the shortest path within the polygon. Soon enough, we will realize that most of the popular routes are over water; Lake Washington to be specific. To have the computer pick routes correctly, our polygon of greater Seattle needs an inner ring outlining the shape of Lake Washington. This way if we run a query asking for the shortest path between two points on the polygon and completely within the polygon, we will not end up with buses driving into water.

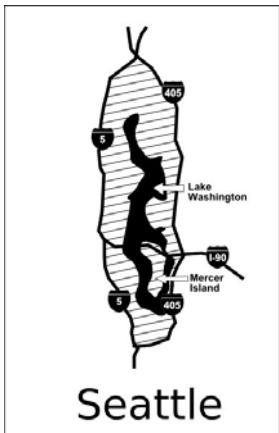


Figure 2.6 We model the Seattle area as a polygon with two rings. Lake Washington fills up the hole. We're also overlooking the existence of Mercer Island in the lake which makes this a multipolygon.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

With polygons we have the concept of validity. The rings of a valid polygon may only intersect at distinct points. What this means is that rings cannot overlap each other and that two rings cannot share a common boundary. A polygon whose inner rings partly lie outside of its exterior ring is also invalid.

Figure 2.7 shows an example of a single polygon with self-intersections. (Visually, you cannot really discern that it is an invalid geometry since such a visual can be created with two valid polygons or one valid multipolygon that happens to be touching at a point.) Not every invalid polygon lends itself to a pictorial representation. Degenerate polygons such as polygons with not enough points and polygons with non-closed rings are difficult to illustrate. Fortunately these polygons are difficult to generate in PostGIS and are pointless in any real world modeling.



Figure 2.7 Example of a self-intersecting polygon with text representation of `POLYGON((2 0,0 0,1 1,-1,2 0))`. This is an example of an invalid polygon, but hard to tell with the naked eye that it is one invalid polygon and not one valid multipolygon or two valid polygons.

### **2.2.5 Collection geometries**

We motivate the concept of collection geometries by asking you to mentally picture the 50 United States of America as polygons. Interior rings allow us to handle states with large bodies of water within its boundaries, such as Utah (The Great Salt Lake), Florida (Lake Okeechobee), and of course Minnesota with its 10,000 plus lakes. There is at least one state that our polygon has trouble handling – Hawaii. Hawaii comes in at least 5 big pieces. We could conceivably model Hawaii as five separate polygons, but this complicates our storage. For example, if we want to create a table of states, we expect to have 50 rows. Breaking states into different polygons would call for storing a state using a state-polygon table where each state could have up to hundreds of geometries depending on how fractured the state is. You would lose the simplicity associated with one geometry per state.

To overcome this problem, PostGIS and the OGC standard offer geometry collections as data types in their own right. A collection of geometries is just that. It groups separate geometries that logically belong together. With the use of collections, each of our fifty states becomes a collection of polygons - a multipolygon.

#### **States as multipolygons**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

To give you a flavor of real world GIS, we will look at state polygon data from US Census Tiger. In the US census tiger data set we have only the following states are modeled as multipolygons: Alaska, California, Hawaii, Florida, Kentucky, New York, and Rhode Island.

In reality we can consider more states to be multipolygons based on geography alone. Almost all states border large bodies of water and have detached islands. Since the census data is more concerned with people living in the states rather than its physical outline, it uses the political boundary of the state for its table of states. State political boundaries extend to adjacent bodies of water and stretches for a few miles into oceans. This more encompassing drawing of boundaries eliminates most states as multipolygons, leaving only the seven above.

In PostGIS each of the single geometry data types has a collection counterpart: multipoints, multilinestrings, multipolygons, and multicurves. In addition, PostGIS has a data type called geometrycollection. This data type can contain any kind of geometry as long as all geometries in the set have the same spatial reference system and the same coordinate dimension.

#### **MULTIPOINTS**

We start with multipoints, which is nothing more than a collection of points. The next figure shows an example of a multipoint.

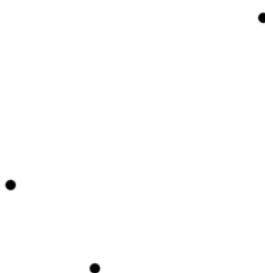


Figure 2.8 A single multipoint geometry. Not three distinct points!

In order to represent a multipoint in WKT syntax, you would use one of the following. If you have only x,y coordinates for a multipoint, each comma delimited value would have 2 coordinates.

MULTIPOINT(-1 1, 0 0, 2 3) (pictured)

For a 3dm multipoints -- those having an x,y,z and m you would have 4 coords  
MULTIPOINT(-1 1 3 4, 0 0 1 2, 2 3 1 2)

For a regular 3d multipoint composed of (x,y,z)

MULTIPOINT(-1 1 3, 0 0 1, 2 3 1)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

For a multipoint where each point is composed of x,y,m you would use MULTIPONTM to distinguish it from a x,y,z multipoint.

```
MULTIPOINTM(-1 1 4, 0 0 2, 2 3 2)
```

### **ALTERNATE WKT SYNTAX FOR MULTIPOINT**

An alternate and acceptable WKT representation for multipoint uses parenthesis to separate each point. Example: MULTIPONT ((-1 1), (0 0), (2 3)). PostGIS will accept this format as input but will output the non-parenthetical version in the ST\_AsText, ST\_AsEWKT output.

We included multipointM in our listings to remind you that all of the geometry collection data types have M dimension type just as their single geometry counterparts.

### **MULTILINESTRINGS**

Of no surprise, multilinestring is a collection of linestrings. Be mindful of the extra set of parenthesis in the WKT representation of a multilinestring that separate each individual linestring in the set. Below are some examples of multilinestring.

```
MULTILINESTRING((0 0,0 1,1 1),(-1 1,-1 -1)) (Pictured)
MULTILINESTRING((0 0 1 1,0 1 1 2,1 1 1 3),(-1 1 1 1,-1 -1 1 2))
MULTILINESTRINGM((0 0 1,0 1 2,1 1 3),(-1 1 1,-1 -1 2))
```

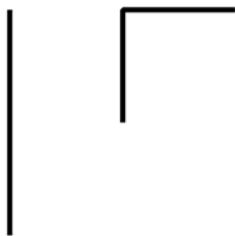


Figure 2.9 Multilinestring generated with WKT of inline examples

Before moving onto multipolygons, let us return to the concept of simplicity. In 2.2.3, we tested a linestring for simplicity. Simplicity holds relevance for any one-dimensional linestring type geometries. We consider multilinestrings to be simple if all constituent linestrings are simple and the collective set of linestrings do not intersect each other at any point except boundary points. For example, if we create a multilinestring with two intersecting simple linestrings, the resultant multilinestring is not simple.

### **MULTIPOLYGONS**

Be careful! The WKT of multipolygon has even more parenthesis than its singular counterpart. Since we use parenthesis to represent each ring of a polygon, we will need

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

another set of outer parenthesis to represent multipolygons. With multipolygons, we highly recommend that you follow PostGIS conventions and not omit any inner parenthesis for single-ringed polygons. Below are some examples of multipolygons.

```
MULTIPOINT(((2.25 0,1.25 1,1.25 -1,2.25 0)),((1 -1,1 1,0 0,1 -1)))
(pictured)
MULTIPOINT(((2.25 0 1,1.25 1 1,1.25 -1 1,2.25 0 1)),((1 -1 2,1 1 2,0 0
2,1 -1 2)))
MULTIPOLYGON(((2.25 0 1 1,1.25 1 1 2,1.25 -1 1 1,2.25 0 1 1)),((1 -1 2 1,1
1 2 2,0 0 2 3,1 -1 2 4)))
MULTIPOLYGONM(((2.25 0 1,1.25 1 2,1.25 -1 1,2.25 0 1)),((1 -1 1,1 1 2,0 0
3,1 -1 4)))
```

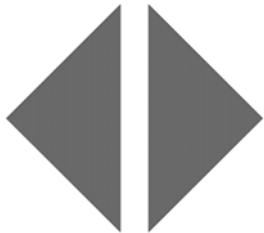


Figure 2.10 Multipolygon generated with WKT of `MULTIPOLYGON(((2.25 0,1.25 1,1.25 -1,2.25 0)),((1 -1,1 1,0 0,1 -1)))`

Recall from the discussion on single polygons, a polygon is considered valid if all its rings do not intersect or only intersect at distinct points. For a multipolygon to qualify as valid, it must pass two tests:

- Each constituent polygon must be valid in their own right
- Constituent polygons cannot overlap. We will define overlap more rigorously in Chapter 4; but even without that we think you get the picture clearly: Once you lay down a polygon, subsequent polygons cannot be sitting atop it and can only share boundaries at finite points.

#### **GEOMETRYCOLLECTION**

Geometrycollection is a PostGIS data type that can contain heterogeneous geometries. Unlike multi-geometries where the constituent geometries must be the same type, the geometrycollection data type can include points, linestrings, polygons, and their collection counterparts. It can even contain other geometrycollections. In short, we can cram every geometry type known to PostGIS into a geometrycollection.

In Listing 2.10, we present the WKT for geometrycollections, but instead of just showing you the WKTs, we include the SQL that generates them. We do this for a reason – In real world applications, you should rarely define a data column as geometrycollection. Although having a heterogeneous collection is perfectly reasonable for storage purposes, most PostGIS

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

functions will not make any sense when applied against these data types. For example, we can ask what the area is of a multipolygon, but we cannot ask what the area is of a geometrycollection that has linestrings and points in addition to just polygons. Geometrycollections generally surface as a result of queries rather than as predefined columns. You should be prepared when you have to work with them, but avoid using them in your table design. For this reason, Listing 2.10 will show you the result of union queries that generate geometrycollections.

### **Listing 2.5 Forming geometrycollections from constituent geometries**

```

SELECT ST_AsText(ST_Collect(the_geom))
FROM (
  SELECT ST_GeomFromText('MULTIPOINT(-1 1, 0 0, 2 3)') As the_geom
  UNION ALL
  SELECT ST_GeomFromText('MULTILINESTRING((0 0,0 1,1 1),(-1 1,-1 -1)))') As the_geom
  UNION ALL
  SELECT ST_GeomFromText('POLYGON((-0.25 -1.25,-0.25 1.25,2.5 1.25,2.5 -1.25,-0.25 -1.25), (2.25 0,1.25 1,1.25 -1,2.25 0),
  (1 -1,1 1,0 0,1 -1))') As the_geom) As foo;

Output:
GEOMETRYCOLLECTION(MULTIPOINT(-1 1,0 0,2 3),MULTILINESTRING((0 0,0 1,1 1),(-1 1,-1 -1)),POLYGON((-0.25 -1.25,-0.25 1.25,2.5 1.25,2.5 -1.25,-0.25 -1.25),(2.25 0,1.25 1,1.25 -1,2.25 0),(1 -1,1 1,0 0,1 -1))) (Pictured)

SELECT ST_AsEWKT(ST_Collect(the_geom)) FROM (
  SELECT ST_GeomFromEWKT('MULTIPOINTM(-1 1 4, 0 0 2, 2 3 2)') As the_geom
  UNION ALL
  SELECT ST_GeomFromEWKT('MULTILINESTRINGM((0 0 1,0 1 2,1 1 3),(-1 1 1,-1 -1 2))') As the_geom
  UNION ALL
  SELECT ST_GeomFromEWKT('POLYGONM((-0.25 -1.25 1,-0.25 1.25 2,2.5 1.25 3,2.5 -1.25 1,-0.25 -1.25 1), (2.25 0 2,1.25 1 1,1.25 -1 1,2.25 0 2), (1 -1 2,1 1 2,0 0 2,1 -1 2))') As the_geom) As foo;

Output:
GEOMETRYCOLLECTIONM(MULTIPOINT(-1 1 4,0 0 2,2 3 2), MULTILINESTRING((0 0 1,0 1 2,1 1 3),(-1 1 1,-1 -1 2)), POLYGON((-0.25 -1.25 1,-0.25 1.25 2,2.5 1.25 3,2.5 -1.25 1,-0.25 -1.25 1), (2.25 0 2,1.25 1 1,1.25 -1 1,2.25 0 2),(1 -1 2,1 1 2,0 0 2,1 -1 2)))

```



Figure 2.11: Geometrycollection formed from code in Listing 2.5

In the above example we use `ST_AsEWKT` and `ST_GeomFromEWKT` to formulate our geometry collection with an M coordinate. (The E stands for “extended.”) The reason for that is that the OGC compliant functions of `ST_AsText` and `ST_GeomFromText` are not really designed for anything above 2D whereas `ST_GeomFromEWKT` and `ST_AsEWKT` are PostGIS constructs and that can be used to display and create geometries of all dimensions. The other benefit of `ST_AsEWKT` over `ST_AsText` is that `ST_AsEWKT` will also return the spatial reference system if it is known. The distinction between the two sets of functions may change in later versions of PostGIS.

Finally, a geometry collection is considered valid if all the geometries in the collection are valid. It is invalid if any of the geometries in the collection are invalid.

### 2.2.6 Curved geometries

PostGIS 1.3 and above have rudimentary support for curved geometries. If you plan to use curved geometries, you definitely should use the latest PostGIS release. Curved geometries were introduced in the OGC SQL-MM Part 3 specs and PostGIS has partial support of what is defined in the specs.

Curved geometries do not share the maturity of other geometries and are not widely supported. Natural terrestrial features rarely manifest themselves as curved geometries. Man-made structures and boundaries do have curves, but for many modeling cases, man made structures can be adequately approximated with lines. Aeronautical charts are full of them since the sweep of radars is circular. Dams, dikes, breakwaters are some other curved macro structures that come to mind. Some highways segments come close to being a curve, but linestrings are often more appropriate in modeling them when processing speed is more important than accuracy. Due to lack of support, we offer the following caveats before you decide to go down the path of using curved geometries.

6. Few third party tools, both open source and commercial, support curved geometries.
7. The advanced spatial library called GEOS that PostGIS uses for many of its functionality such as performing intersections, containments checks, and other spatial

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

relation checks do not support curved geometries. As a work around, you can convert curved geometries to linestrings and regular polygons using the `ST_CurveToLine` function and then convert back with `ST_LineToCurve`. The downside of this method is the loss in speed and the inaccuracies introduced when interpolating arcs using linestrings.

8. Many native PostGIS functions do not support curved geometries. You can find a full listing of functions that do support curved geometries in the PostGIS reference manual. Again for cases where you need to use functions that do not support curved geometries, you can use the `ST_CurveToLine`, apply the function and then `ST_LineToCurve` function to convert back if needed.
9. Curved geometries have not been supported for as long as the other geometries in PostGIS, so you are more likely to run into bugs when working with them. More recent releases of PostGIS have cleaned up many of the bugs and have expanded the number of functions that will support curved geometries.

Given all the drawbacks of curved geometries, you might be wondering why you would ever want to use them. There are a few reasons:

10. You can represent a truly curved geometry with fewer points.
11. More tools will inaugurate support for curved geometries. Java2D graphical rendering library, Adobe Flex, Safe FME, and uDig 1.2 are the ones we know about that have or are working on curved geometry support usable by PostGIS.
12. PostGIS is ramping up support for curved geometries. The complete MM set for curved geometries should be available in PostGIS 2.0. We should also start enjoying complete support of intersections, and relation checks in later versions.
13. Curved geometries are particularly important when modeling man made structures such as buildings and bridges, where true curved objects are commonly found.
14. Even if you don't store your data using curved geometry types, it's often useful as an intermediary source to draw a quarter circle using curved geometry WKT and then convert to regular polygon using the `ST_CurveToLine` function.

Let us now take a closer look at the large varieties of curved geometries. For simplicity, you can think of curved geometries in PostGIS as geometries with arcs. To build an arc, you must have exactly three distinct points. The first and last points denote the starting and end points of the arc respectively. The point in the middle is called the control point since this point controls the degree of curvature of the arc.

#### **CIRCULARSTRING**

A series of one or more arcs where the end point of one is the starting point of another makes up a geometry called a circularstring. Below is a diagram of a simple 5 point circularstring.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

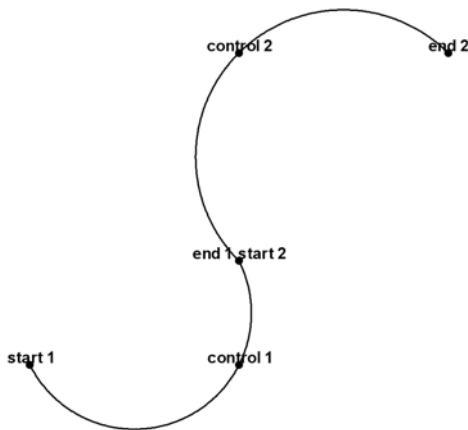


Figure 2.12 A simple 5 point circular string. The WKT CIRCULARSTRING(0 0,2 0, 2 1, 2 3, 4 3). The control points are POINT(2 0) and POINT(2 3).

### **DOES POSTGIS SUPPORT BEZIER CURVES AND SPLINES?**

This is a commonly asked question for people wanting to use curves. The short answer is NO, but there have been talks of introducing such support in much later versions once the basic SQL-MM circularly interpolated curved geometry support is complete. To be completely supported, PostGIS must be able to instantiate all the different types defined in SQL-MM Part 3 and most of the relation and processing functions can work with them.

The circularstring is the simplest of all curved geometries and only contains arcs. Below are more examples of circularstrings and how you would register them in the database.

#### **Listing 2.6 Building circularstrings**

```
SELECT AddGeometryColumn ('public','my_geometries','my_circular_strings',-1,'CIRCULARSTRING',2);

INSERT INTO my_geometries(name,my_circular_strings)
VALUES ('Circle',
        ST_GeomFromText('CIRCULARSTRING(0 0,2 0, 2 2, 0 2, 0 0)'),),
       ('Half Circle',
        ST_GeomFromText('CIRCULARSTRING(2.5 2.5,4.5 2.5, 4.5 4.5)'),),
       ('Several Arcs',
        ST_GeomFromText('CIRCULARSTRING(5 5,6 6,4 8, 7 9, 9.5 9.5, 11 12, 12
12)'));
```

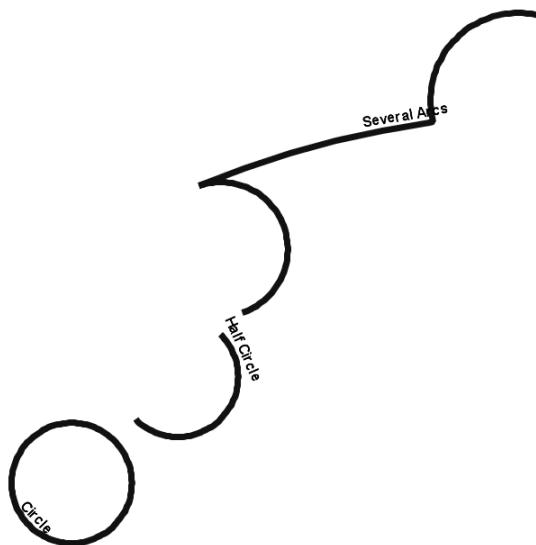


Figure 2.13 Three circular strings generated from code in Listing 2.6

You will discover that not all rendering tools can handle curve geometries. In these instances, ST\_CurveToLine function comes in handy for approximating curved geometries with linestrings. As the code and table below demonstrates, to achieve a reasonable degree of curvature, the linestring is made up of many segments.

```
SELECT name,ST_NPoints(my_circular_strings) As cnpoints,
ST_NPoints(ST_CurveToLine(my_circular_strings)) As lnpoints FROM
my_geometries
WHERE my_circular_strings IS NOT NULL;
```

Table 2.1 Observe how many more points the LINESTRING equivalent takes to approximate a curve

<b>name</b>	<b>cnpoints</b>	<b>Inpoints</b>
Circle	5	129
Half Circle	3	65
Several Arcs	7	113

#### COMPOUNDCURVES

Circularstrings and linestrings in series make up a collection geometry called compoundcurves. A polygon constructed from a compoundcurve is called a curvepolygon. A square with rounded corners is a nice representation of a closed compoundcurve with 4

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

circular strings and 4 straight linestrings. A compoundcurve is a geometry composed of both circularstring and regular linestring segments and where the last point in the prior segment is the first point of the next segment. So for example if the last point of a circularstring is (10 12), then the first point of the linestring that follows would be (10 12). Below is an example of a compoundcurve.

### **Listing 2.7 Compoundcurve composed of an arc sandwiched between 2 linestrings**

```
SELECT AddGeometryColumn ('public','my_geometries','my_compound_curves',-1,'COMPOUNDCURVE',2);
INSERT INTO my_geometries(name,my_compound_curves)
VALUES ('Road with curve', ST_GeomFromText('COMPOUNDCURVE((2 2, 2.5 2.5),
CIRCULARSTRING(2.5 2.5,4.5 2.5, 3.5 3.5), (3.5 3.5, 2.5 4.5, 3 5))'));
```

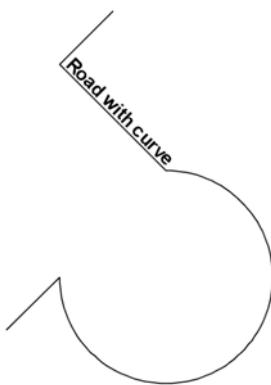


Figure 2.14 A compound curve generated from code in listing 2.7

### **CURVEPOLYGON**

A curvepolygon is a type of polygon that has an exterior or inner ring with circularstrings. In pre-1.4 PostGIS versions, compondcurves cannot be used to form rings even though SQL-MM specs allow for such curvepolygons. Below are some examples of curvepolygons.

### **Listing 2.8 Creating Curved Polygons**

```
SELECT AddGeometryColumn ('public','my_geometries','my_curve_polygons',-1,'CURVEPOLYGON',2);

INSERT INTO my_geometries(name,my_curve_polygons)
VALUES ('Solid Circle', ST_GeomFromText('CURVEPOLYGON(CIRCULARSTRING(0 0,2
0, 2 2, 0 2, 0 0))))', ('Circle t hole',
ST_GeomFromText('CURVEPOLYGON(CIRCULARSTRING(2.5 2.5,4.5 2.5, 4.5 3.5, 2.5
4.5, 2.5 2.5), (3.5 3.5, 3.25 2.25, 4.25 3.25, 3.5 3.5) )'), ('T arcish
hole', ST_GeomFromText('CURVEPOLYGON((-0.5 7, -1 5, 3.5 5.25, -0.5 7),
CIRCULARSTRING(0.25 5.5, -0.25 6.5, -0.5 5.75, 0 5.75, 0.25 5.5))))');
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

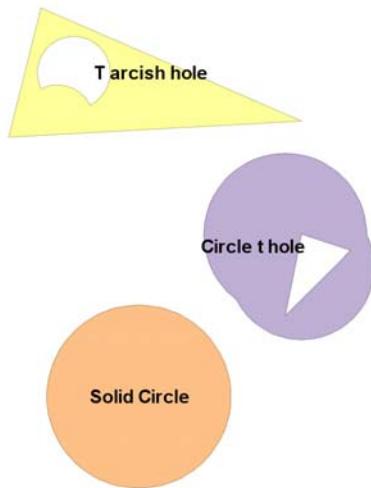


Figure 2.15 Curvepolygons generated with code in Listing 2.8

PostGIS 1.4 introduced support for compoundcurves as rings in a curvepolygon. Below is the WKT of a curvepolygon with a compoundcurve outer ring and a circularstring inner ring.

```
CURVEPOLYGON(COMPOUNDCURVE(CIRCULARSTRING(0 0, 2 0, 2 1, 2 3, 4 3), (4 3, 4 5, 1 4, 0 0)), CIRCULARSTRING(1.7 1, 1.7 0.9, 1.6 0.5, 1.4 0.6, 1.7 1))
```



Figure 2.16 Compound Curve in curve polygon with a circular string hole

### 2.2.7 3D geometries

PostGIS recognizes and stores 3D geometries, but full support is still spotty. As we have shown, you can easily create 3D points, linestrings, polygons and curved geometries, but you must keep in mind that these lack the volumetric sense of 3D. They are basically 2D

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

objects sitting in 3D space (as such, they are sometimes referred to as 2.5D). Be mindful of the following caveats when working with 3D geometries:

1. PostGIS and the underlying GEOS library have spotty support for it. For example all the relationship operators only check if two geometries relate in 2D space and completely ignore the Z dimension. Suppose you have one box floating above another and not touching, asking if the boxes intersect yields a true result since they occupy the same 2D coordinates.
2. Overlay functions such as intersection and union only partially handle the third dimension. When applied to 3D geometries, they perform the expected operation on the 2D portion of the geometry and then interpolate the Z coordinate. This may be acceptable when elevations do not need to be precise, say on a hiking trail.
3. Spatial operators achieve their amazing speed through use of bounding box indexes. Unfortunately, these indexes currently do not extend into the 3rd dimension. The good news is that work is being done to rectify the situation.

## 2.3 Summary

We began this chapter by introducing the `geometry_columns` meta-table. Every PostGIS-enabled database has this meta-table to catalog information about geometry columns in the database. We then introduced five functions that are commonly used to interact with `geometry_columns` meta-table, with `AddGeometryColumn` being the most prominent of these functions. We advised strongly against editing the `geometry_columns` table directly.

We then moved onto the single geometries of points, linestrings, and polygons, giving examples of their WKT representation and the INSERT SQL statements to add them to `geometry_columns`. Combining single geometries gives rise to collection geometries. We covered multipoints, multilinestrings, multipolygons, and geometrycollections.

We finished the chapter with discussions of curved and 3D geometries. Since these two families of geometries are much more complex and less supported in real world GIS modeling, they do not fully enjoy the support of third party tools and are only partially supported by PostGIS itself.

We hope that this chapter will give you the necessary know-how to set up your PostGIS table and the use of meta tables in PostGIS to centralize data type management. You should have a good understanding of the single geometries and the purpose behind using geometry collections. In the next chapter, we apply what we learned in this chapter by exploring the various approaches to organizing your data in a PostGIS database.

Finally, we want to remind you to not be caught up with rigorous definitions when modeling. If it looks, feels, and smells like a polygon, treat it as a polygon; do not worry about inner rings and outer rings unless you need to. Although PostGIS tries to follow standards and the standards try to follow sound mathematical foundations, you will run into definitional inconsistencies. Do not be jarred by these, instead focus on what you are trying to accomplish using PostGIS.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

# 3

## *Organizing spatial data*

This chapter covers

- Options for structuring spatial vector data
- Modeling a real city
- Rule and triggers

In the last chapter we did a walk through of all the possible geometry types PostGIS offers and how you would create and store them. In this chapter we continue our study by demonstrating the different table layouts you can design to store spatial data. Next we apply our various design approaches to a real world example. We finish the chapter with a discussion and examples of using rules and triggers to manage inserts and updates in our tables and views.

### **3.1 Spatial storage approaches**

You should now have a good understanding of all the spatial vector types available in PostGIS. We will move on to the various options for designing your database to store these types. As with any database design, there is a healthy dose of compromise. Many considerations factor into the final structure you settle on for your spatial database such as the analysis it must support, the speed of the queries and so forth. With a spatial database, a few additional considerations enter the design process: availability of data, the precision to which you need to store the data, and mapping tools that you need to be compatible with. Speed of queries take on a new prominence in spatial databases. Unlike databases that store numerical and text data where a poor design leads to slow queries, a poor design in a spatial database could lead to queries that will never finish in one's lifetime. It also goes without saying that many factors cannot be determined at the outset: You may not know exactly how many or what type of geometry will eventually reside in the database. You may not even know how the users will query the data. As with all decision making, you do the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

best you can with the information you have at the time. You can always rework your design as needs change, but as any database practitioner knows, getting the design more or less right the first time saves hassle down the road.

In this section, we cover three common ways to organize data in a spatial database: heterogeneous geometry columns, homogeneous geometry columns, and inheritance. We will explain how you would go about setting up your database structure using each of these approaches and point out the advantages and disadvantages. These approaches are by no means exhaustive and you should feel free to find your own hybrid that fits your specific needs. We will also mainly focus on geometry data types over geography data types. Geometry data types are by far the predominant data types in PostGIS. For one, geometry data types have been around since the dawn of PostGIS whereas geography data types are a recent addition. The number of functions and third-party tools supporting geometry types far out number those for geography types. Finally geometry types are inherently faster for most spatial computations. All this may change as geography data types mature. You will find the general concepts we cover in this section to be applicable to geography types.

### **3.1.1 Heterogeneous geometry columns**

This approach is one that does not care about constraining columns to a specific kind of geometry. For example, to store geographic features in a city, you could create a points-of-interest column in PostGIS; set its data type to be geometry or geography and be done. In this single column you can store points, linestrings, polygons, collections or any other vector type for that matter. You may wish to do this if you are more interested in geographically partitioning the city. For example, Washington DC, as well as many other planned cities are divided into quadrants: NW, SW, SE, NE. A city planner can employ a single table with quadrant names as a text column and another generic geometry column to store the geometries within each quadrant. By leaving the data type as generic geometry type, the column can store polygons for the many polygonal shaped government edifices in DC, linestrings to represent major thoroughfares, and points for Metro stations.

There are varying degrees of the heterogeneous approach. Using a generic geometry column does not necessarily mean having no additional constraints. You should still judiciously apply the constraints to ensure data integrity. We advise that you at least enforce the spatial reference system constraint and the coordinate dimension constraint because the vast majority of all non-unary functions in PostGIS and all aggregate functions require the spatial reference system and the coordinate dimension of the input geometries be the same. Below are the pros and cons of the heterogeneous column approach.

#### **PROS**

- It allows you to run a single query of features of interest together without giving up the luxury of modeling them with the most appropriate geometry type.
- It is simple. You could conceivably cram all your geometries into one table if their non-spatial attributes are more or less the same.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- Table creation is quick because you can just do it by setting the field as a geometry with a single CREATE TABLE statement and not have to worry about the additional need to apply AddGeometryColumn function. This is particularly handy if your are trying to iteratively load data from a large number of tables and not really care or know what geometry types each contains.

**CONS**

- You run the risk of having someone insert an inappropriate geometry for an object. For example, if you have obtained data of subway stations that should be modeled as points, an errant linestring in the data could enter your heterogeneous table. Furthermore, if you do not constrain the spatial reference system or coordinate geometry and unwittingly end up with more than one of each, your queries could be completely incorrect or break.
- Many third-party tools cannot deal with heterogeneous geometry columns. As a workaround, you may need to create views against this table to make it appear as separate tables and add geometry type index or ensure your queries select only a single type of geometry type from the heterogeneous column.
- For cases where you only need to extract a certain kind of geometry, you will need to constantly filter by geometry type. For large tables, this could be slow and annoying to have to keep doing over and over again.
- Throwing all your geometry data into a single table could lead to an unwieldy number of self joins. For example, suppose you placed points of interest in the same table as polygons outlining city neighborhoods, every time you need to identify which points of interest (POIs) fall into which neighborhood, you will need to perform a self join on this table. Not only are self-joins taxing for the processor, they are also taxing for the mind. Imagine a scenario where you have 100 POIs and two neighborhoods for a total of 102 records. To determine which POIs fall into which city requires that a table of 102 rows be joined with a table of 102 rows (itself). If you had separated out the cities into its own table you would only be joining a table of 100 rows with a table with just two rows.

With the disadvantages of heterogeneous storage approach fresh in your mind, we move onto the homogenous geometry columns approach.

### **3.1.2 Homogeneous geometry columns**

This approach avoids the mixing of different geometry types in a single column. Polygons must be stored in a column of only polygons, linestrings in linestring columns, etc. This means that each geometry type must reside in its own column at the very least, but it is also common to break up different geometry types into separate tables altogether.

If in our DC example, we care more about the type of the features than within which quadrant each feature is located, we would employ the homogeneous columns design. One possible table structure would be to define a features table with a name column and three

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

geometry columns. We constrain one column to store only points, one to store only linestrings, and one to store only polygons. If a feature is point data, we populate the point column, leaving the other two columns null; if linestring data, we populate the linestring column only and so on. We do not necessarily need to cram all of our columns into a single table. A more common design would be to use three distinct tables, one to exclusively store each type of geometry.

We now summarize the pros and cons of the homogenous columns approach:

**PROS**

- Enforces consistency and prevents unintended mixing of geometry types.
- Third-party tools rely on consistency in geometry type. Some may go so far as to only allow one geometry column per table. The popular ESRI shapefile only support one geometry per record, so you would need to explicitly state the geometry column should you ever need to dump data into ESRI.
- In general, you get better performance when joining tables with large geometries and few records in with tables of smaller geometries with many records than vice versa.
- When you need to draw different kinds of geometries at different z levels, the speed is much better if you split your data into multiple tables with each geometry type in separate tables.
- Should you be working with monstrous datasets, separate tables also allow you reap benefits from placing your data on separate physical disks for each table if you take advantage of PostgreSQL support for tablespaces.

**What is a PostgreSQL tablespace?**

In PostgreSQL a tablespace is a physical folder location as opposed to a schema which is a logical location. (Oracle also has tablespaces and SQL Server has a similar concept called file groups.) In the default setup, all the tables you create will go into the same tablespace. As your tables grow, you may want to create additional tablespaces each on separate physical disks and distribute your tables across different tablespaces to achieve maximum disk I/O. One common practice is to group rarely-used tables into its own tablespace and place it on a slow disk. In PostgreSQL 9.0 tablespaces were enhanced to allow you to set random\_page\_cost and seq\_page\_cost settings for each table space. In older versions you could only set these at the server or database level. The query planner use these two parameters to determine how to rate query paths based on whether they utilize data running on slower disks or faster disks.

**CONS**

- When you need to run a query that draws multiple geometry types, you have to resort to a union query. This can add to the complexity and the speed of the query. For example, if 99% of the queries you write for the DC example all involve querying by quadrant only, stick with the heterogeneous approach.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- Be careful if you go down the route of having multiple geometry columns in a single table. Wider rows make for slower queries, both on selects and updates.

### **3.1.3 Table inheritance**

The final design approach we offer forth is using table inheritance. This is by far the most versatile of our various storage approaches, but slightly more involved than the previous two. One unique strength of PostgreSQL is support for table inheritance. We can tap into this gem of a feature to distill the positive aspects of both homogeneous and heterogeneous column approaches.

Table inheritance means that a table can inherit its structure from a parent table. The parent table does not actually need to store any data, relegating all the data storage to the child tables. When used this way, the parent table is often referred to as an abstract table (from the OO concept of abstract classes). Each child table inherits all the columns of its parent, but in addition it can have columns of its own that are revealed only when you query the child table directly. Check constraints are also inherited, but primary keys and foreign key constraints are not. PostgreSQL supports multiple inheritance where a child table can have more than one parent table with columns derived from both parents. PostgreSQL also does not place a limit on the number of generations you can have. A parent table can have parents of its own and so forth.

To implement our table inheritance storage approach, we create an abstract table that organizes data along its non-geometric attributes and then create inherited child tables with constrained geometry types. With this pattern, the end user can query from the parent table and see all the child data or query from each child table when they only need data from the child tables or child specific columns. For our DC example, the table of the single generic geometry column would serve as our parent table. We then create three inherited child tables each constrained to hold points, linestrings, and polygons. Now if we need to pull data by quadrants without paying attention to geometry type, we query against the parent table. If we need to pull data of a specific geometry, we query one of the child tables. Remember, only with PostgreSQL can you orchestrate such an elegant solution. No other major database offerings support direct table inheritance, at least not yet.

#### **Constraint exclusion**

PostgreSQL has a configuration option called constraint\_exclusion which is often used in conjunction with table inheritance. When this is enabled, the query planner will check the table constraints of a table to determine if it can skip a table in a query. In PostgreSQL 8.3 and below, the options for this setting are on or off. The on option means that constraints are always checked on tables even if they are not in an inheritance hierarchy or the query is not a union query. For PostgreSQL 8.4 and later, an additional option of partition was introduced. Partition saves query planning cycles by only having constraint exclusion checking happen when doing queries against tables in an inheritance hierarchy or when running a UNION query. For PostgreSQL 8.4+ if you are using table inheritance,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

you should generally keep this at its default of partition. For 8.3 and below, you need to set this to on if you want to use table inheritance effectively.

**PROS**

- Allows you to query a hierarchy of tables as if they were a single table or query them separately as needed.
- If you partition by geometry type, you can, as needed, query for a specific geometry type or query for all geometry types
- With the use of PostgreSQL constraint exclusions, the query planner can cleverly skip over child tables if none of the rows qualify under your filtering condition. For example, if you need to store data organized by countries of the world. By partitioning the data into a child table for each country, any query you write that filters by country name would completely skip unneeded country tables as if they did not exist. This can yield a significant speed boost when you have large numbers of records.
- Inheritance can be set and un-set on the fly, making it convenient when performing data loads. For instance, you can disinherit a child table, load the data, clean the data, add any necessary constraints and then re-inherit the child table. This prevents slowing down of select queries on other data while data loading is happening.
- Most third-party tools will treat the parent table as a bona-fide table even though it may not have any data as long as relevant geometry columns are registered and primary keys are set on the parent table. Inheritance works seamlessly with OpenJump, GeoServer, and MapServer. Any tool that polls the standard PostgreSQL metadata should end up treating parent tables like any other.

**CONS**

- Table inheritance is not supported by other major databases. Should you ever need to switch away from PostgreSQL to another, your application code may not be portable. This is not as big a problem as it may initially appear since most database drivers will simply see a parent table as a single table with all the data of its children. Your opting to desert PostgreSQL is the bigger problem!
- Primary key and foreign key constraints do not pass to child tables, though check constraints do. In our DC example, if we place a primary key constraint on the parent feature table, dictating that each place name must be unique, we cannot expect the child tables to abide by the constraint. Even if we were to assign primary key constraints to the children, we still cannot guarantee unique results when querying multiple child tables or query the parent table together with its child tables.
- To maintain hierarchy, you must take extra steps when adding data to make sure that rows are appropriately added to the parent table or one of its child tables. For table updates, you may want to put in logic that automatically moves a record from one child table to another child table should an update cause a check violation. This

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

generally means having to create rules or triggers to insert into a child table when inserting into a parent table or vice versa. We will cover this in detail in the next section. Thankfully, PostgreSQL inheritance is smart enough to automatically handle updates and deletes for most situations. When you update or delete against a parent table, it will automatically drill down to its child tables. For updates that should move data from one child to another, you need to manage that logic with rules or triggers on the child table. You can still go through the trouble of creating update and delete triggers to figure out which records in child tables need to be updated when an update or delete call is made on the parent table. This often yields a speed improvement over relying on the automated drill down of PostgreSQL inheritance.

- Should you use constraint exclusions to skip tables entirely, you will face an initial performance hit when the query is executed for the very first time.
- Be watchful of the total number of tables in your inheritance hierarchy. Performance begins to noticeably degrade after a couple hundred tables. In PostgreSQL 9.0, the planner will generate statistics for inheritance hierarchy. This should boost performance when querying against inherited tables.

Below is code demonstrating how you would go about implementing a table inheritance model. We first create a parent table for all roads in the United States. In this parent table, we set the spatial reference ID as well as the geometry type. We then beget two child tables. The first will store roads in the six New England states; the second will store roads in the Southwest states. We will only populate the child tables with data, leaving the parent table devoid of any rows.

### **Listing 3.1 Code to partition roads into various states**

```
CREATE TABLE roads(gid serial PRIMARY KEY, road_name character varying(100));  
  
SELECT AddGeometryColumn('public', 'roads', 'geom', 4269, 'LINESTRING', 2);  
  
--[1 BEG]  
CREATE TABLE roads_NE(CONSTRAINT pk PRIMARY KEY (gid))  
INHERITS (roads);  
--[1 END]  
  
--[2 BEG]  
ALTER TABLE roads_NE  
ADD CONSTRAINT chk CHECK (state IN ('MA', 'ME', 'NH', 'VT', 'CT', 'RI'));  
--[2 END]  
  
CREATE TABLE roads_SW(CONSTRAINT pk PRIMARY KEY (gid))  
INHERITS (roads);  
  
ALTER TABLE roads_SW  
ADD CONSTRAINT chk CHECK (state IN ('AZ', 'NM', 'NV'));  
  
--[3 BEG]  
SELECT gid, road_name, geom FROM roads WHERE state = 'MA';
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

--[ 3 END ]

In [1], we create a child table to our roads. [2] We add constraints to our table which will be useful for speeding up queries when we have constraint\_exclusion set to partition or on. It will ensure that roads\_NE table is skipped if the requested state is not in MA, ME, NH, VT, CT, or RI.

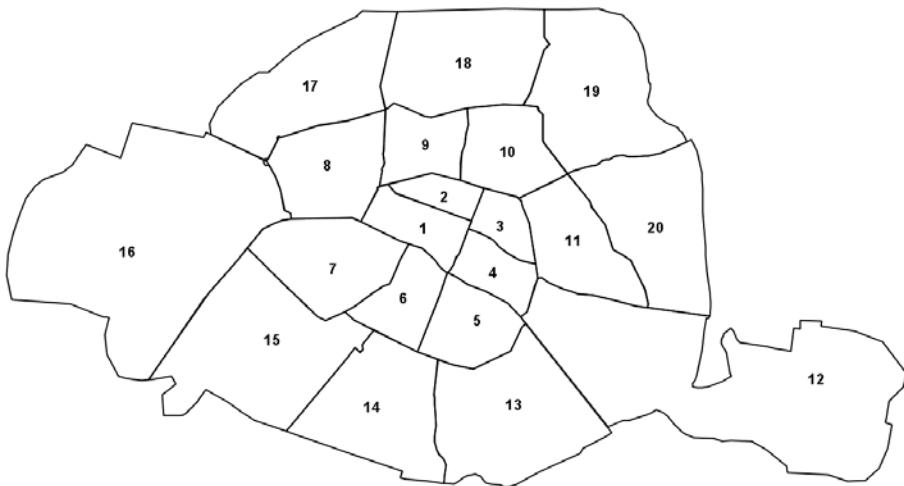
In [3], we write a simple SELECT to pull all roads in Massachusetts . With constraint exclusion, then only the child table with roads in New England will be searched. You can see this by running an explain plan or looking at the graphical explain plan in PgAdmin III.

We have examined three ways of organizing our spatial data. In the next section we put these ideas to task by modeling a real-world city using these approaches.

### **3.2 Modeling a real city**

In this section, we apply what we learned in the previous section by exploring various ways to model a real city. We abandon the quadrants of DC and states of the US and cross the Atlantic to Paris, the city of lights (or love, depending on your preference) for our extended example. We chose Paris because of the importance placed on arrondissements. For those of you unfamiliar with Paris, the city is divided into twenty administrative districts, known as arrondissements. Unlike other major cities, Parisians are keenly aware of the presence of administrative districts. It is not unusual for someone to say that they live in the nth arrondissement fully expecting their fellow Parisians to know perfectly what general area of Paris is being spoken of. Unlike what are often referred to as neighborhoods in other major cities, arrondissements are well-defined geographically and so are well-suited for GIS purposes. It is not often where one finds the exact geographical subdivision of a city to have crept into the common usage of its citizenry. On top of it all, Parisians often refer to them by their ordinal number rather than their ascribed French names, making numerically-minded folk like ourselves extra happy.

The basic Paris arrondissements geography is illustrated in Figure 3.1.



**Figure 3.1 Paris Arrondissements**

The arrondissement arrangement is interesting in that it spirals from the center of Paris and moves clockwise around reflecting the pattern of growth since the 1800s as the city annexed adjacent areas.

We downloaded our Paris data from GeoCommons.com as well as Open Street Map (OSM). We transformed all the data to SRID 32631 (WGS-84 UTM Zone 31). All of Paris fits into this UTM zone. Since UTM is meter-based we have measurements at our disposal without any additional effort. As a starting point, we model each arrondissement as a multipolygon and inserted them into a table called arrondissements. The table contains exactly 20 rows. Not only will this table serve as our base layer, we will use it to geo-tag additional data into specific arrondissements.

### **3.2.1 Modeling using a heterogeneous geometry column**

If we mainly need to query our data by arrondissements without regards to the type of feature, we could employ a single geometry column to store all of our data. Let us create this table.

#### **Listing 3.2 Creating table with heterogeneous geometry column**

```
CREATE TABLE ch03.paris_hetero(gid serial NOT NULL, osm_id integer, geom
geometry, ar_num integer, tags hstore,
CONSTRAINT paris_hetero_pk PRIMARY KEY (gid),
CONSTRAINT enforce_dims_geom CHECK (ndims(geom) = 2),
CONSTRAINT enforce_srid_geom CHECK (srid(geom) = 32631)
);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Notice how a constraint restricting the type of geometry in decidedly missing. Our geometry column will be able to contain points, linestrings, polygons, multigeometries, geometry collections, in short, any geometry type we want to stuff in. We did take the extra step to limit our column to only 2-dimensional geometries and SRID of 32631.

You will also notice a data type called hstore. Hstore is a data type for storing key-value pairs similar in concept to PHP associative arrays. Much like geometry columns, it too can be indexed using the GIST index. OSM makes wide use of tags, for storing properties of features that do not fit elsewhere. To bring in the OSM data without complicating our table, we used the OSM2PGSQL utility with the --hstore switch to map the OSM tags to an hstore column.

### HSTORE DATA TYPE AND POSTGRESQL

The Hstore data type is a contrib module found in versions PostgreSQL 8.2 and above. To enable this module, simply execute the hstore.sql script in the share/contrib folder. In PostgreSQL 9.0+, this data type has been enhanced to allow DISTINCT and GROUP BY operations and to also support storing larger amounts of data per field.

Our table includes a column called ar\_num for holding the arrondissement number of the feature. Unfortunately, this attribute is not one maintained by OSM. No worries, we intersect the OSM data with our arrondissement table to figure out which arrondissement each OSM record fall into. Though we can determine the arrondissements on the fly, having the arrondissements figured out before hand means we can query against an integer instead of having to constantly perform spatial intersections later on.

#### **Listing 3.3 Region tagging and clipping data to a specific arrondissement**

```
--[1 BEG]
INSERT INTO ch03.paris_hetero(osm_id, geom, ar_num, tags)
SELECT o.osm_id, ST_Intersection(o.geom, a.geom) As geom, a.ar_num, o.tags
FROM
(SELECT osm_id, ST_Transform(way, 32631) As geom, tags FROM
planet_osm_line) AS O INNER JOIN ch03.arrondissements AS A ON
(ST_Intersects(o.geom, a.geom));
-- repeat for osm_polygon, osm_line
--[1 END]
--[2 BEG]
CREATE INDEX idx_paris_hetero_geom ON ch03.paris_hetero USING gist(geom);
CREATE INDEX idx_paris_hetero_tags ON ch03.paris_hetero USING gist(tags);
VACUUM ANALYZE ch03.paris_hetero;
--[2 END]
[1] Insert data and clip to specific arrondissement
[2] Add indexes and update statistics
```

In [1] we load in all the OSM data we downloaded. We only listed the insert from planet\_osm\_line table, but you will need to repeat this for OSM points and OSM polygons. Easier yet, download the code. Features such as long linestrings and large polygons will

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

straddle multiple arrondissements, but our intersection operation will clip them so you end up with one record per arrondissement. For example, the famous Boulevard Saint-Germain passes through the 5<sup>th</sup>, 6<sup>th</sup>, and 7<sup>th</sup> arrondissements. After our clipping exercise, our record with a single linestring would have been broken up into three records each with shorter linestrings, one for each of the arrondissements that it passes through. [2] We perform our usual index and update statistics after the bulk load.

As we have demonstrated above, by not putting a geometry type constraint on our geometry column, we can stuff linestrings, polygons, points and even geometry collections if we wanted to in the same table. This model is nice and simple in the sense that if we wanted for mapping purposes or statistical purpose to pick all features or count all features that fit in a particular user defined area, we can do it with one simple query. Here's an example that counts up the number of features within each arrondissement.

```
SELECT ar_num, COUNT(DISTINCT osm_id) As compte
FROM ch03.paris_hetero
GROUP BY ar_num;
```

Which yields an answer of:

ar_num	compte
1	8
:	
8	334
:	
16	302
17	328
18	8

We should mention that for our example, we only extracted the area of Paris surrounding the Arc de Triomphe from OSM. The famous landmark is at the center of arrondissements 8, 16, and 17 hence most of our features tend to be in those three regions. Below is a quick map we generated in OpenJump by overlaying our OSM dataset atop the arrondissement polygons.

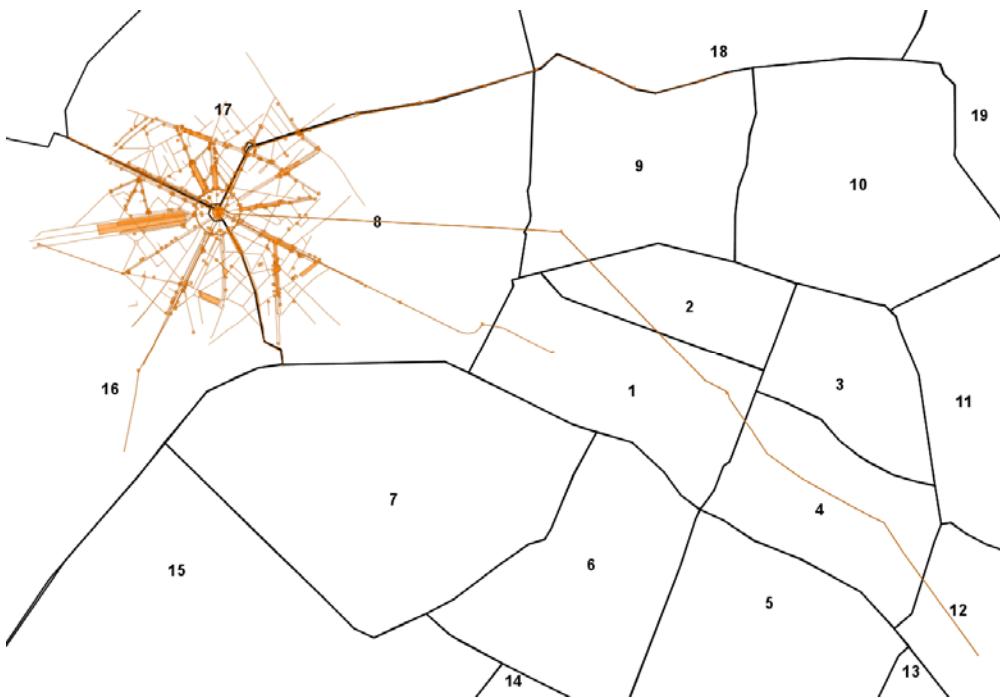


Figure 3.2 Our dataset overlaid on the arrondissements without caring about geometry type

The main advantage of using hstore to hold miscellaneous attribute data is that you do not have to set up bona-fide columns for attributes that could be of little use later on just so you can import data. You can first import the data and then cherry-pick which attributes you would like to promote to actual columns as your needs grow. Using hstore also means that you can also add and remove attributes without fussing with data structure. The drawback is when you do need attributes to be full-fledged columns. You cannot query inside a hstore column as easily as you can a character or numeric column or enforce numeric and other data type constraints on the hstore values. Also remember that hstore is a PostgreSQL data type, not to be found elsewhere. Few mapping tools will accept columns in hstore natively. A simple way to overcome the drawback of hstore columns is to create a view that will map attributes within a hstore column into virtual data columns.

#### **Listing 3.4 Using a view to transform hstore data into columns**

```
CREATE OR REPLACE VIEW ch03.vw_paris_points AS
SELECT gid, osm_id, ar_num, geom,
       tags->'name' As place_name,
       tags->'tourism' As tourist_attraction
  FROM ch03.paris_hetero
 WHERE ST_GeometryType(geom) = 'ST_Point';
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In the above listing, we create a view which promotes the two tags, name and tourism, into two text data columns. Do not forget that you can register views in the geometry\_columns table just as you can with tables. Use the handy populate\_geometry\_columns function to perform the registration.

```
SELECT populate_geometry_columns('ch03.vw_paris_points');
```

Once registered, we challenge any third-party program to treat the view any differently than a regular data table; at least as far as reading the data goes. Now we move onto the homogeneous architecture.

### **3.2.2 Modeling using homogeneous geometry columns**

With the homogeneous columns approach, we want to store each geometry type in its own column or table. This style of storage is more common than the heterogeneous approach. It is the one most supported by third-party tools. Having distinct columns or tables by geometry type allows you to enforce geometry type constraints, preventing different geometry data types from inadvertently mixing. The downside is that your queries will have to enumerate across multiple columns or tables should you ever wish to pull data of different geometry types.

#### **Listing 3.4 Breaking our data into separate tables homogeneous geometry columns**

```
CREATE TABLE ch03.paris_points(gid SERIAL PRIMARY KEY, osm_id integer,
ar_num integer,
    feature_name varchar(200), feature_type varchar(50));
--[1 BEG]
SELECT AddGeometryColumn('ch03', 'paris_points', 'geom', 32631,'POINT', 2);
--[1 END]
--[2 BEG]
INSERT INTO ch03.paris_points(osm_id, ar_num, geom, feature_name,
feature_type)
SELECT osm_id, ar_num, geom, tags->'name' As feature_name,
COALESCE(tags->'tourism', tags->'railway', 'other')::varchar(50) As
feature_type
FROM ch03.paris_hetero
WHERE ST_GeometryType(geom) = 'ST_Point';
--[2 END]
[1] Register the geometry column
[2] Add points
```

We start by creating a table to store point geometry types. [1] We register the geometry column, effectively constraining the column to only store points. Having a geometry type constraint is the essence of the homogeneous approach. [2] Finally we perform the insert, but instead of starting from the OSM data, we take advantage of the fact that we already have the data we need in the paris\_hetero table and we selectively pick out the tags we care about and morph them into the columns we want. If we wanted to have a complete

homogeneous solution, we would create similar tables for paris\_polygons and paris\_linestrings.

If we were to get the counts of all the features by arrondissement, our query would require unioning all the different tables together as seen below:

```
SELECT ar_num, COUNT(DISTINCT osm_id) As compte FROM
(SELECT ar_num, osm_id FROM paris_points
UNION ALL
SELECT ar_num, osm_id FROM paris_polygons
UNION ALL
SELECT ar_num, osm_id FROM paris_linestrings
) AS X
GROUP BY ar_num;
```

### UNION vs. UNION ALL

When performing union operations you generally want to use UNION ALL over UNION. UNION has an implicit DISTINCT clause built-in which automatically eliminate duplicate rows. If you know that the sets you are unioning cannot or need not be deduped in the process, opt for the UNION ALL. It will be faster for sure.

We move next to an inheritance-based storage design where you will see that by expending some extra effort, you reap the benefits of both the heterogeneous and homogeneous approaches.

### 3.2.3 Modeling using inheritance

Table inheritance is a feature fairly unique to PostgreSQL. We gave you a quick overview in Section 3.1.3. Now we will apply it to our Paris example. We begin by creating an abstract parent table to store attributes that all of its children will share.

#### **Listing 3.5 Creating our abstract parent table**

```
CREATE TABLE ch03.paris(gid SERIAL PRIMARY KEY, osm_id integer, ar_num
integer, feature_name varchar(200), feature_type varchar(50), geom
geometry);

ALTER TABLE ch03.paris
ADD CONSTRAINT enforce_dims_geom CHECK (st_ndims(geom) = 2);
ALTER TABLE ch03.paris
ADD CONSTRAINT enforce_srid_geom CHECK (st_srid(geom) = 32631);
```

We took the extra effort of adding a primary key on the parent table even though we never plan to add data to it. Child tables also cannot inherit primary keys so why did we take the extra step? Besides the good practice of having a primary key on every table, abstract or not, many clients tools rely on all tables having a primary key.

With our parent table in place, we create child tables. Keep in mind that you will need to do this for paris\_points and paris\_linestrings or any other geometry type you have data for, but for the sake of brevity we will only create the child table for storing polygons.

### **Listing 3.6 Creating a child table**

```
--[1 BEG]
CREATE TABLE ch03.paris_polygons(tags hstore,
    CONSTRAINT paris_polygons_pk PRIMARY KEY (gid)
)
INHERITS (ch03.paris);
--[1 END]

--[2 BEG]
ALTER TABLE ch03.paris_polygons NO INHERIT ch03.paris;
--[2 END]
--[3 BEG]
INSERT INTO ch03.paris_polygons(osm_id, ar_num, geom, tags, feature_name,
feature_type)
SELECT osm_id, ar_num, ST_Multi(geom) As geom, tags, tags->'name',
COALESCE(tags->'tourism', tags->'railway', 'other')::varchar(50) As
feature_type
FROM ch03.paris_hetero
WHERE ST_GeometryType(geom) LIKE '%Polygon';
--[3 END]
SELECT populate_geometry_columns('ch03.paris_polygons'::regclass);
-- [4 BEG]
ALTER TABLE ch03.paris_polygons INHERIT ch03.paris;
-- [4 END]
-- [1] Create an inherited table
-- [2] Disinherit from parent
-- [3] Load
-- [4] Re-inherit
```

In [1] we create a polygon table and declare it as inheriting from our paris table, we only need to add the additional columns (in this case tags) beyond what is already defined in the parent. We also add a primary key to the child table since primary keys do not automatically inherit. In [2] we disinherit the child from the parent. Disinherit does not remove inherited columns. Once, a child table inherits from a parent table, the structure of the parent is passed down permanently. The disinheritance simply disengages the child from the parent so that queries against the parent do not drill down to the children. We find it a good idea to disinherit a child table prior to performing large bulk loads on the child table. This is to prevent someone from querying a child table while it is in the process of being loaded. In [3], we then load our table taking rows from our paris\_hetero table where the geometry type is polygon or multipolygon. We finish up by calling the populate\_geometry\_columns function to automatically add the geometry constraint and register our geometry column and then in [4] re-inherit from the parent.

We will repeat the same code for linestrings, but we will omit the loading of data and the adding of the tag column. Since we are not populating the table immediately, we constrain

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

the geometry column to store only linestrings so that our `populate_geometry_columns` function can use this check constraint to properly register the geometry column.

### **Listing 3.7 Adding another child and adding additional constraints**

```
-- [1 BEG]
CREATE TABLE ch03.paris_linestrings(
    CONSTRAINT paris_linestrings_pk PRIMARY KEY (gid)
)
INHERITS (ch03.paris);
-- [1 END]
-- [2 BEG]
ALTER TABLE ch03.paris_linestrings
    ADD CONSTRAINT enforce_geotype_geom
        CHECK (geometrytype(geom) = 'LINESTRING'::text);
-- [2 END]
-- [3 BEG]
SELECT populate_geometry_columns('ch03.paris_linestrings'::regclass);
-- [3 END]
-- [1] Create child table
-- [2] Add geometry type constraint
-- [3] Register geometry column
```

As we did with polygons [1] we create a table that inherits from `paris` to store our linestrings. We are not ready to load data yet, but want to constrain the table to just linestrings so in [2] we add a geometry type constraint. We do not need to add a dimension or srid constraint, because check constraints are always inherited from the parent tables and `paris` already has those constraints defined. Since we have a geometry constraint, the `populate_geometry_columns` will use the geometry constraint type to correctly register the table in `geometry_columns`.

At last, we reap the fruits of our labor. With inheritance our count query is identical to the simple one we used for our heterogeneous model.

```
SELECT ar_num, COUNT(DISTINCT osm_id) As compte
FROM ch03.paris
GROUP BY ar_num;
```

With inheritance in place, we have the added flexibility to query just the polygon table should we only care about the counts there.

```
SELECT ar_num, COUNT(DISTINCT osm_id) As compte
FROM ch03.paris_polygons
GROUP BY ar_num;
```

As we see, using inheritance takes an extra step or two to setup properly, but the advantage is that we are able to keep our queries simple by judiciously querying against the parent table or one of the child tables. As one famous Parisienne might have said, "Let them have their cake and eat it too."

#### **ADOPTION**

More often than not, inheritance comes as an after thought rather than as part of the initial table design. As an example, say we already set up a `paris_points` table to store point

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

geometry and have gone to great length to populate the table with data. We would not want to drop our points table and recreate it in order for it be a legitimate child of the paris table.

### **Listing 3.8 Adopt an orphan**

```
--[1 BEG]
ALTER TABLE ch03.paris_points DROP COLUMN gid;
--[1 END]
--[2 BEG]
ALTER TABLE ch03.paris_points ADD COLUMN gid integer PRIMARY KEY NOT NULL
DEFAULT nextval('ch03.paris_gid_seq');
--[2 END]
--[3 BEG]
ALTER TABLE ch03.paris_points INHERIT ch03.paris;
--[3 END]
--[4 BEG]
CREATE INDEX idx_paris_points_geom ON ch03.paris_points USING gist (geom);
--[4 END]
--[1] Drop old gid
--[2] Add new gid based on parent's sequence
--[3] Adoption
--[4] Add a spatial index
```

There are a few considerations when a parent “adopts” a new child table. Before being adopted, the child table must first ensure that its set of columns is a superset of the columns found in the parent’s table. In other words, the new parent must not have any columns not found in the child. Though it is not an absolute necessity, it is useful for all children’s primary keys to be unique across the hierarchy. One way to ensure that is to make them use the same sequence as the parent, a family genetic sequence so to speak. To re-assign the gid of our points table to use the family sequence, we drop the gid column entirely as in [1]. Next, we add the column back except this time, we specify that the sequence must come from the sequence of the parent, see [2]. Finally in [4], we add a spatial index for good measure. If you are doing bulk loads, you may wish to add the index afterwards, not before, so the loading can run as fast as possible.

#### **ADDING COLUMNS TO THE PARENT**

When you add a new column to the parent table, PostgreSQL will automatically add the column to all inherited children. If the child table already has that column, PostgreSQL issues a gentle warning. In our earlier example, we created our paris\_polygons table with a tags column, but the parent table and none of the other child tables have this column. Let us try adding tags to the paris table:

```
ALTER TABLE ch03.paris ADD COLUMN tags hstore;
```

When we do the above, we will get a notice:

```
merging definition of column "tags" for child "paris_polygons"
```

which informs us that the child paris\_polygons already has this column. And remember how we purposely omitted the tag column when creating the paris\_linestrings child table? After

adding tags to the parent, our `paris_linestrings` now also has this column. Check for yourself.

In the next section, we cover the use of rules and triggers. Though these two database utilities can be used wherever the situation calls for them, we find that they are invaluable in working with inheritance hierarchies.

### **3.3 Using rules and triggers**

Sophisticated RDBMS usually offer ways to catch the execution of certain SQL commands on a table or view and allow some form of conditional processing to take place in response to these events. PostgreSQL is certainly not devoid of such features and can perform additional processing when it encounters the four core SQL commands of `SELECT`, `UPDATE`, `INSERT`, and `DELETE`. The two mechanism for handling the conditional processing are rules and triggers.

At this point we count on you to have worked through the example and have in your test database the four tables of: `paris`, `paris_points`, `paris_linestrings`, `paris_polygons`. We will enhance our Paris example by adding in rules and triggers.

#### **3.3.1 Rules versus triggers**

Though both rules and triggers respond to events, this is where their similarity ends. They do overlap in functionality. You could often use a trigger instead of a rule and vice versa, but they were created with different intents. Though there are no steadfast guidelines on when to use one over the other when you have a choice, the underlying motivation behind having two separate event-response mechanism will help you decide,

##### **RULES**

A rule in PostgreSQL is really an instruction of how to rewrite an SQL statement. For this reason people sometimes refer to rules as rewrite rules. A rule is completely passive and only transforms one SQL statement into another SQL statement, nothing more. Unbeknownst to many people, views are really nothing more than one or more rewrite rules nicely packaged together. When you execute a `SELECT` command from a view, the view portion of your SQL statement is rewritten to include the view definition before the command is run. For example, suppose you create a view as follows:

```
CREATE OR REPLACE view some_view AS SELECT * FROM some_table
```

When you then select from the view using a simple statement like

```
SELECT * FROM some_view
```

The rule rewriter substitutes the `some_view` part of the `SELECT` with the definition you used to create the view so that your `SELECT` is essentially re-written to be something along the lines of

```
SELECT * FROM (SELECT * FROM some_table) AS some_view
```

Because a view is nothing more than a packaging of rewrite rules we are free to use views far beyond just a simple SELECT statement. You can have your views actually manipulate data. You are free to use UPDATE, INSERT, AND DELETE commands at will within your view rules. Furthermore, a view need not have just one SQL statement, but can process an entire chain of statements combining SELECT, UPDATE, INSERT, and DELETE statements. Calling it a view in PostgreSQL belies its underlying capability to do much more than simply view data.

### TRIGGERS

A trigger is a piece of procedural code that either

- Prevents something from happening, for example canceling an INSERT, UPDATE or DELETE if certain conditions are not met
- Does something instead of the requested INSERT, UPDATE or DELETE
- Does something else in addition to the INSERT, UPDATE or DELETE command.

Triggers from rules can never be applied to SELECT events.

Triggers are either row-based or statement-based. Row based triggers are executed for each row participating in an UPDATE, INSERT OR DELETE operation. Statement-based triggers are rarely used except for statement logging purposes, so we will not cover them here.

### PostgreSQL 9.0 when trigger clause

PostgreSQL 9.0, introduces a WHEN clause, which allows a trigger to be skipped if it does not satisfy a designated WHEN clause. This improves the performance of triggers, since the trigger function is never entered unless the triggering data satisfies the WHEN clause.

In PostgreSQL, you have many language choices for writing triggers, but unlike with rules, you cannot simply string together a series of SQL statements. Triggers must be stand-alone functions. Popular languages for authoring functions in PostgreSQL are PL/PGSQL, PL/Python, PL/R, PL/TCL, and C. You could even develop your own language should you fancy to do so or have multiple triggers on a table each written in a different language more suited for a particular task.

### WHEN TO USE RULES, WHEN TO USE TRIGGERS

Broadly speaking, triggers are more powerful but must be executed for each row. For bulk loads, rules can often be faster since they are called once per UPDATE or INSERT statement whereas triggers are called for each row needing an UPDATE or INSERT. In situations where only a few rows are involved, the speed difference between rules and triggers are negligible.

In certain situations, you could only use a rule. Should you need to bind to a view, only rules will work. If you want to perform action based on a SELECT command, only rules will work. Then there are situations when you cannot use a rule and have to implement a trigger. Should you need the logic or specialized functions of procedural languages, they

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

only available through triggers. Rules are always written in just plain SQL. You would also need a trigger if you need to execute SQL commands such as CREATE, ALTER, OR DROP. You cannot run data definition language statements with rules. You also do not have the facilities within rules to build SQL statements on the fly.

Here are some general heuristics we follow:

Use rules when

- Creating a select only view
- Making a view updateable
- Bulk loads
- Binding to views

Use a triggers when

- Redirecting inserts from parent tables to child tables
- Preprocessing logic such as converting long lat to geometry/geographies or geo-tagging data
- Complex validation or processing requiring procedural languages
- Need to run create table or other DDL statement in response to change in data

The most important thing to keep in mind is that despite their overlap in achieving the same goal: rules and triggers are fundamentally different. Rules rewrite an SQL statement. Triggers run a function for each affected row. Both have their places.

### **3.3.2 Using rules**

Before providing you with some example use of rules, we need you to understand the distinction between a DO rule versus a DO INSTEAD rule. The default behavior of a rule when not specified is a DO rule. A DO rule takes an SQL statement and tacks additional SQL on to the statement. A DO INSTEAD, on the other hand, throws out the original statement and replaces it with the rewritten SQL. The technical term for breaking apart a query into subparts is often referred to as a query tree. With a DO rule, you add additional branches to the tree. With a DO INSTEAD rule, you supplant the tree completely with a new tree. One last thing to keep in mind is that for views, you can only use DO INSTEAD rules.

In the next example, we will create a simple view and then add rules so that we can use the view to perform insert, update, and delete operations.

#### **Listing 3.9: Making views updateable with rules**

```
--[1 BEG]
CREATE OR REPLACE VIEW ch03.stations
AS
SELECT gid, osm_id, ar_num, feature_name, geom
FROM ch03.paris_points
WHERE feature_type = 'station';
--[1 END]
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
--[2 BEG]
CREATE OR REPLACE RULE ch03.rule_stations_insert AS
    ON INSERT TO ch03.stations
        DO INSTEAD
            INSERT INTO ch03.paris_points(gid, osm_id, ar_num, feature_name,
                feature_type, geom)
                VALUES (DEFAULT, NEW.osm_id, NEW.ar_num, NEW.feature_name, 'station',
                NEW.geom);
--[2 END]
--[3 BEG]
CREATE OR REPLACE RULE ch03.rule_stations_delete AS
    ON DELETE TO ch03.stations
        DO INSTEAD
            DELETE FROM ch03.paris_points
                WHERE gid = OLD.gid AND feature_type = 'station';
--[3 END]
--[4 BEG]
CREATE OR REPLACE RULE ch03.rule_stations_update AS
    ON UPDATE TO ch03.stations
        DO INSTEAD
            UPDATE ch03.paris_points
                SET gid = NEW.gid, osm_id = NEW.osm_id, ar_num = NEW.ar_num, feature_name =
                    NEW.feature_name, geom = NEW.geom
                WHERE gid = OLD.gid AND feature_type = 'station';
--[4 END]
[1] Read-only view
[2] Insert-able view
[3] Delete-able view
[4] Updatable view
```

With [1], we use a simple SELECT to define our view. At this point our view is read-only.

We then define an insert rule in [2] which will allow inserts into this view. We assume that only stations will be added using this view and so deliberately set the feature\_type to 'station.' Our insert rule replaces the original insert with the code you see in [2], effectively redirecting the insert to the paris\_points table. In [3] we create a delete rule. In addition to the primary key field, we have a filter to only delete stations. This ensures that even if you forget a filtering WHERE clause when performing the deletion, the worst you can do is remove all station rows as oppose to all rows or paris\_points. Finally in [4], we have the update rule.

### **NEW AND OLD record variables in rules and triggers**

Both rules and triggers can have available to them two record variables called NEW and OLD. For INSERT FOR EACH ROW events only NEW is available. For DELETE FOR EACH ROW events, only OLD is available. For UPDATE FOR EACH ROW events, both NEW and OLD are available. For statement level triggers neither NEW nor OLD is available.

Both NEW and OLD represent exactly one record and take on the column structure of the triggering table. The OLD represents the record that was deleted or replaced. You can think of an UPDATE as being a combination INSERT and DELETE which is why it has both a NEW and OLD.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

This behavior is similar to other relational databases you may have come across, except that the NEW and OLD always represent one row whereas in some other databases the comparable counterpart are tables consisting of all the records to be inserted or deleted.

Let us take our view for a test drive. We start with a delete from the view as follows:

```
DELETE FROM ch03.stations;
```

The database query engine automatically rewrites the above as:

```
DELETE FROM ch03.paris_points WHERE feature_type = 'station';
```

Our stations have all vanished. We next add back our stations as follows:

```
INSERT INTO ch03.stations(osm_id, feature_name, geom)
SELECT osm_id, tags->'name', geom
FROM ch03.paris_hetero
WHERE tags->'railway' = 'station';
```

With the rewrite, our insert becomes:

```
INSERT INTO ch03.paris_points(osm_id, feature_name, feature_type,
geom)
VALUES (NEW.osm_id, NEW.ar_num, NEW.feature_name, 'stations',
New.geom)
```

As you can see, rules rewrite the original SQL, nothing more. During the rewrite, you are limited to using SQL statements. This does limit the capability of rules in many situations, but for core logic that you wish to apply universally and forever, rules may fit the bill. Now we move onto triggers.

### **3.3.3 Using triggers**

When it comes to triggers, we must expand the three core events of INSERT, UPDATE, DELETE to six: BEFORE INSERT, AFTER INSERT, BEFORE UPDATE, AFTER UPDATE, BEFORE DELETE and AFTER DELETE. BEFORE events fire prior to the execution of the triggering command; AFTER events fire upon completion. Should you wish to perform an alternative action as you can with a DO INSTEAD rule, you would create a trigger and bind it to the BEFORE event, but throw out the resulting record. If you need to modify data that will be inserted/updated, you also need to do this in a BEFORE event. An AFTER trigger would be too late. Similarly, should you wish to perform some operation that depends on the success of your main action, you would need to bind to an AFTER event. Examples of this are if you need to insert or update a related table on success of an insert or update statement.

PostgreSQL triggers are implemented as a special type of function called a trigger function and then bound to a table event. This extra level of indirection means you can re-

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

use the same trigger function for different events and tables. The slight inconvenience is that you face a two step process of first defining the trigger function and then binding it to a table.

PostgreSQL allows you to define multiple triggers per event per table, but each trigger must be uniquely named across the table. Triggers fire in alphabetical sequence. If your database is trigger-happy, we recommend developing a convention for naming your triggers to keep them organized.

We will now move on to a series of examples showcasing how we can use triggers in a variety of situations to fortify our data model. Triggers are powerful tools and your mastery of them will allow you to develop database applications that can control business logic without need of touching the front-end application.

#### **REDIRECTING INSERTS WITH BEFORE TRIGGERS**

For our first trigger example, we will demonstrate a common need when working with inherited tables. This is the ability to redirect inserts done on a parent table to the child tables. Recall that with an inheritance hierarchy with abstract parents, we want people to think they are inserting into the parent table, but do not actually want any data going into it. To accomplish this we use a BEFORE INSERT trigger to redirect inserts into child tables. Our function checks the geometry type of the record being inserted into the table. Depending on its geometry type, we redirect the insert to one of the child tables. For geometry types that do not fit, we toss them into a rejects table. Our code follows:

#### **Listing 3.10 PL/PGSQL BEFORE INSERT Trigger function to redirect insert**

```
CREATE OR REPLACE FUNCTION ch03.trigger_paris_insert() RETURNS trigger AS
$$
DECLARE
    var_geomtype text;
BEGIN
    --[1 BEG]
    var_geomtype := geometrytype(NEW.geom);
    --[1 END]
    IF var_geomtype IN ('MULTIPOLYGON', 'POLYGON') THEN
        NEW.geom := ST_Multi(NEW.geom);
        INSERT INTO ch03.paris_polygons(gid, osm_id, ar_num, feature_name,
        feature_type, geom, tags)
        SELECT gid, osm_id, ar_num, feature_name, feature_type, geom, tags
    --[2 BEG]
    FROM (SELECT NEW.*) As foo;
    --[2 END]
    ELSIF var_geomtype = 'POINT' THEN
        INSERT INTO ch03.paris_points(gid, osm_id, ar_num, feature_name,
        feature_type, geom, tags)
        SELECT gid, osm_id, ar_num, feature_name, feature_type, geom, tags
        FROM (SELECT NEW.*) As foo;
    ELSIF var_geomtype = 'LINESTRING' THEN
        INSERT INTO ch03.paris_linestrings(gid, osm_id, ar_num,
        feature_name, feature_type, geom,tags)
        SELECT gid, osm_id, ar_num, feature_name, feature_type, geom, tags
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
        FROM (SELECT NEW.*) AS foo;
    ELSE
--[3 BEG]
    INSERT INTO ch03.paris_rejects(gid, osm_id, ar_num, feature_name,
feature_type, geom, tags)
        SELECT gid, osm_id, ar_num, feature_name, feature_type, geom, tags
        FROM (SELECT NEW.*) AS foo;
--[3 END]
    END IF;
--[4 BEG]
    RETURN NULL;
--[4 END]

END;
$$
LANGUAGE 'plpgsql' VOLATILE;
[1] Using temporary variables
[2] NEW is alias for the table that contain the new record
[3] Non-standard geometry types go into rejects table
[4] Cancel original insert
```

In [1], we declare a temporary variable to hold intermediary information. This can reduce processing time for long-running functions, plus you end up with clearer code. During an insert operation, PostgreSQL automatically dumps the new record into a single-rowed table, aliased NEW, with the exact structure as the table being inserted into. In [2], we take advantage of this alias to read the values of the geometry type of the new record coming in order to decide which child table to redirect the insert. Normally when you are done with a BEFORE trigger, you return the new record, which you may have changed in the trigger. This signals to the PostgreSQL to continue with the INSERT. But in our case, we want to halt the INSERT into the parent table altogether, so we return NULL instead of NEW. Returning the NEW record is usually only used in a BEFORE trigger, because in an AFTER trigger, there is no hope of being able to change the record being inserted or updated since the event has already happened. This is a common mistake people make—defining an AFTER trigger and then trying to change the NEW record. PostgreSQL will let you do that, but the changes will never make it into the underlying table.

### Using NEW.\* without specifying column names in rules and triggers

In trigger functions, you will often see the use of NEW.\* as a shorthand to pick up all the columns of the record being inserted. The single-row NEW not only has the same structure, but also has the exact column order as the triggering table. This often allows us to use the following insert syntax without worrying about listing out each column:

```
INSERT INTO ch03.paris_rejects VALUES(NEW.*);
```

For our example we have chosen not to use this syntax. While, this syntax is extremely powerful because you can use it in any trigger function without knowing before hand what the columns are or will be, it is not without its dangers.

The danger of this approach is when you are redirecting inserts to child tables. A child table may have more columns than its parent or in different order, so this syntax will fail in such cases.

Remember that trigger functions do us no good unless it is bound to a table event. To bind the above trigger function to the BEFORE insert of our paris table, we run this statement:

```
CREATE TRIGGER trigger1_paris_insert BEFORE INSERT
ON ch03.paris FOR EACH ROW
EXECUTE PROCEDURE ch03.trigger_paris_insert();
```

Let us take our new trigger for a test drive. Before we do, we will delete any data we have thus far to get a clean start. As long as we have no foreign-key constraints, we can use the fast SQL TRUNCATE clause to delete data from the parent table and all of its child tables.

```
TRUNCATE TABLE ch03.paris;
```

Now when we perform our insert:

```
INSERT INTO ch03.paris (osm_id, geom, tags)
SELECT osm_id, geom, tags FROM ch03.paris_hetero;
```

With the trigger in place, the records will sort themselves into child tables befitting their respective geometry type. Since we never created a child table for multilinestrings, these records will end up in the paris\_rejects table.

#### **CREATING TABLES ON THE FLY WITH TRIGGERS**

Using triggers allows us to do something we cannot do with rules. We can generate any SQL statements on the fly and execute them as part of our trigger function. We can even run SQL that will create new database objects, which is what we will demonstrate with our next example. Suppose, we want to partition our Paris data by arrondissements in addition to geometry type, but we are too lazy to create all 60 arrondissement tables before hand. We can delegate the work to our trigger function. As we insert new records into the parent table, it will redirect inserts to each geometry type child table, which in turn will redirect inserts to each geometry-arrondissement grandchild table as appropriate. Furthermore, if the particular grandchild geometry-arrondissement table does not exist, our trigger function will create it.

For just the few arrondissements we have or if we know the tables we need beforehand, our dynamic creation in trigger is overkill. It is more efficient to have the tables created at the outset than to have each insert check and then create as needed, but you could imagine cases where this may not be possible. For example, you may have a large amount of financial data that you would like to break out into weekly tables. If you anticipate your

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

database being in use for ten years, you have to prepare 520 tables at the start. Not only that, on the 1<sup>st</sup> day of the eleventh year, your database fails.

### **Listing 3.11 Trigger that dynamically creates tables as needed**

```
CREATE OR REPLACE FUNCTION ch03.trigger_paris_child_insert() RETURNS
TRIGGER AS $$

DECLARE
    var_sql text;
    var_tbl text;
BEGIN
--[1 BEG]
    var_tbl := TG_TABLE_NAME || '_ar' || lpad(NEW.ar_num::text, 2, '0');
--[1 END]
--[2 BEG]
    IF NOT EXISTS(SELECT * FROM information_schema.tables WHERE
table_schema = TG_TABLE_SCHEMA AND table_name = var_tbl) THEN
--[2 END]
--[3 BEG]
    var_sql := 'CREATE TABLE ' ... [See Code Listing]
    EXECUTE var_sql;
--[3 END]
    END IF;
--[4 BEG]
    var_sql := 'INSERT INTO ' || TG_TABLE_SCHEMA || '.' || var_tbl ||
'(gid, osm_id, ar_num, feature_name, feature_type, geom, tags) VALUES($1,
$2, $3, $4, $5, $6, $7)';

    EXECUTE var_sql USING NEW.gid, NEW.osm_id, NEW.ar_num,
NEW.feature_name, NEW.feature_type, NEW.geom, NEW.tags;
--[4 END]
--[5 BEG]
    RETURN NULL;
--[5 END]

END;
$$ language plpgsql;
```

- [1] Assign destination table name to variable
- [2] Check if destination table exists
- [3] Create destination table if absent
- [4] Prepare and execute insert SQL
- [5] Cancel original insert

Before we do anything, we must settle on a naming conventions for all of our geometry-arrondissement grand child tables. We choose paris\_points\_ar01 thru paris\_points\_ar20, paris\_linestrings\_ar01 thru paris\_linestrings\_ar20, and paris\_polygons\_ar01 thru paris\_polygons\_ar20. In [1], we formulate the destination table name of our new record. Notice how PostgreSQL provides a TG\_TABLE\_NAME variable that tells us the table to which the current trigger is bound to. Without this, we would have to further test the geometry type of the new record to figure destination table. In [2], we check to see if the destination

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

table is present. If not, we create it in [3]. By [4], we are assured that the destination table must be present and proceed with the insert.

Once we have our trigger function, we bind it to our three child tables: paris\_points, paris\_linestrings, paris\_polygons.

### **Listing 3.12 Binding same trigger function to multiple tables**

```
CREATE TRIGGER trig01_paris_child_insert BEFORE INSERT
  ON ch03.paris_polygons FOR EACH ROW
  EXECUTE PROCEDURE ch03.trigger_paris_child_insert();

CREATE TRIGGER trig01_paris_child_insert BEFORE INSERT
  ON ch03.paris_points FOR EACH ROW
  EXECUTE PROCEDURE ch03.trigger_paris_child_insert();

CREATE TRIGGER trig01_paris_child_insert BEFORE INSERT
  ON ch03.paris_linestrings FOR EACH ROW
  EXECUTE PROCEDURE ch03.trigger_paris_child_insert();
```

Once in place, these three triggers will prevent data from being inserted into our child tables but instead have the data flow to arrondissement specific grand child tables. If the grand child is missing, we create it on the fly. To test our new trigger, we delete all the data we have inserted and start anew.

```
TRUNCATE TABLE ch03.paris;
TRUNCATE TABLE ch03.paris_rejects;
```

We then perform the insert.

```
INSERT INTO ch03.paris(osm_id, geom, tags)
SELECT osm_id, geom, tags FROM ch03.paris_hetero;
```

After we are all done and provided we have data to fully span all three geometry types and 20 arrondissements, we should end up with 60. Our particular dataset will only require the creation of 9 tables.

Before bringing the discussion of rules and triggers to an end, let us revisit constraint exclusions. Remember how before we began our extended example, we described the usefulness of having constraint exclusion enabled. To test that constraint exclusion is working correctly, we run the following query and look at the PgAdmin graphical explain plan.

```
SELECT * FROM ch03.paris WHERE ar_num = 17;
```

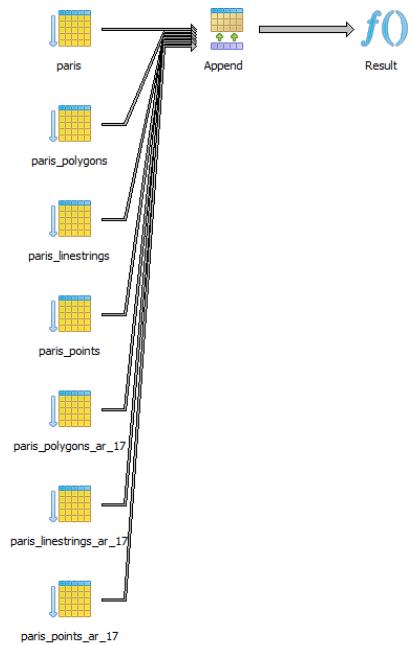


Figure 3.3 Demonstrates only empty parent tables and child tables holding ar17 data are searched

Observe that although there exist tables such as `paris_points_ar01`, `paris_polygons_ar08`, etc, the planner strategically skips over those tables because we only asked for data found in `ar17` tables. Constraint exclusion works!

### **3.5 Summary**

In this chapter we discussed some approaches you can use for storing PostGIS geometry data in PostgreSQL relational tables as well as managing control of this data. We also demonstrated the use of PostgreSQL custom key value `hstore` data type for implementing schemaless models. We followed up by applying these approaches to storing Parisien data.

We demonstrated in this chapter that PostGIS geometry columns are just like other columns you will find in relational tables. They have indexes and are stored along with other related data such as text columns. Just like other data types they take advantage of all the facilities that the database has to offer such as inheritance, triggers, rules, indexes and so forth. In addition you can inspect geometries using various geometry functions designed for them and can even use them in SQL join conditions as you would other relational data types.

This chapter also provided a sneak preview of some of the PostGIS functions we will explore in later chapters. In the next chapter and chapters that follow we shall explore PostGIS functions in more depth. We shall first focus on dealing with single geometries and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

the more common properties, various functions you can use to inspect and modify geometries. After looking at single geometry functions, we will focus on functions that involve interplay between one or more geometries and geometry columns.

# 4

## *Geometry Functions*

This chapter covers

- Core geometry properties
- Geometry functions that take one geometry argument

In the previous chapters we defined the various kinds of geometries that PostGIS provides, how to create them and how to add them to the database. In this chapter and the next we will introduce the core set of functions that you can use to work with geometries. This chapter will concentrate on functions that tend to work with single geometries. In the next, we will work with functions that relate two or more geometries.

PostGIS offers well over 300 functions and operands. To get a handle around all these, we have developed a taxonomy that is driven by intent of use. This is by no means a rigorous classification (say one based on functional signature) nor one which will neatly sort each function into a unique classification without ambiguities. Grouping functions by the types of tasks that you are trying to accomplish has been the handiest approach in our experience. Before delving into the functions themselves, let us go through our classification scheme.

- Constructor functions – Use these functions to create PostGIS geometries from either a well-known-text (WKT), well-known-binary (WKB).
- Output functions – Use these functions to output geometry representations in various well-defined standard formats (WKT, WKB, GML, SVG, KML, GeoJSON).
- Accessor and Setter functions - These are functions that work against a single geometry and return or set attributes of the geometry.
- Decomposition functions – These functions find and return geometries that can be extracted from the input geometry.
- Composition functions – Use these functions to stitch, splice, or group together

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

geometries.

- Measurement functions – Return scalar measurements of the geometry.
- Simplification functions – Sometimes you do not need the full resolution of a geometry. These functions simplify a geometry by removing points, linestrings or rounding the coordinates. The resultant geometry will still have the basic look and feel of the original, but made up of fewer points or coordinates of lower precision.

In keeping with the fundamental mission of this book, which is to show how to use PostGIS rather than serve as a reference volume, we will introduce a dozen or so that are commonly used. You can locate an exhaustive listing of all functions and their usage in the PostGIS official user manual.

### Why ST?

You will notice that almost all functions that we will introduce you to start with the two letters ST. The S stands for spatial and the T stands for temporal, even though support in the temporal dimension never gained much popularity.

ST prefix is usually set aside for SQL/MM functions in other spatial databases, but PostGIS uses the prefix a bit liberally for both SQL/MM and functions unique to PostGIS.

## 4.1 Constructors

As the name implies, constructor functions create geometries. There are two common ways to come up with entirely new geometries. The first way uses raw data in an acceptable format and builds the geometry from scratch. The second way is to take existing geometries and either decompose, splice, slice, dice, or morph them to come up with new ones. In this section, we start with the first approach. We will go through the list of common representation of geometric data and the functions that we use to transform them into bona-fide PostGIS geometry objects. Following that we will introduce some handy functions that create new geometries from other ones.

### 4.1.1 Creating geometries from well known text and binary (WKT, WKB)

#### ST\_GeomFromText

Recall from Chapter One that a common way to represent geometries is through well-known text representations (WKT). PostGIS provides a function called ST\_GeomFromText that can be used to build 2D geometries. This function is an SQL/MM standard function so can be found in other SQL/MM compliant spatial databases. It only supports 2D because the SQL/MM released specs for this function did not define this function to support M and Z coordinates. Below are examples of its use.

**Listing 4.1 Examples of constructing geometries using WKT**

```
INSERT INTO table1 (geom)
VALUES
  (ST_GeomFromText('POINT(-100 28)', 4326)),
  (ST_GeomFromText('LINESTRING(-80 28, -90 29)', 4326)),
  (ST_GeomFromText('POLYGON((10 28, 9 29, 7 30, 10 28));
```

**ST\_GeomFromEWKT**

PostGIS provides another function called ST\_GeomFromEWKT. This is a PostGIS-only function and accepts input in another PostGIS-only creation -- EWKT (extended WKT) -- with the intent of making up for deficiencies in the WKT format. EWKT encodes SRID information directly into the WKT and also supports 3D and 4D geometries. We show you how to use ST\_GeomFromEWKT below. Note that EWKT explicitly prepends the SRID of the geometry.

**Listing 4.2 Examples of constructing geometries from EWKT**

```
INSERT INTO table1 (geom)
VALUES
  (ST_GeomFromEWKT('SRID=4326;POINT(-100 28)'),,
  (ST_GeomFromEWKT('SRID=4326;LINESTRING(-80 28,-90 29)'),,
  (ST_GeomFromEWKT('SRID=4326;POLYGON((10 28, 9 29, 7 30, 10 28));
```

ST\_GeomFromEWKT can accept geometry in just plain WKT format as well, so is often preferred when SQL/MM compliance is not a concern.

**ST\_GeomFromWKB AND ST\_GeomFromEWKB**

On many occasions, you will find yourself needing to import data from a client application where geometries are already stored in binary representations. This is when the functions ST\_GeomFromWKB and ST\_GeomFromEWKB come into play. Again ST\_GeomFromWKB is an SQL/MM defined function and ST\_GeomFromEWKB is a PostGIS extension offering encoding of SRID and support for 3D, 4D geometries. These two functions take in byte arrays instead of text strings. One advantage of the byte arrays is that they are exact where as the ST\_GeomFromText and ST\_GeomFromEWKT truncate at about the 15th digit after the decimal point. Below is an example of using ST\_GeomFromWKB.

```
SELECT
  ST_GeomFromWKB(E'\\001\\001\\000\\000\\000\\321\\256B\\3120\\3040\\
  300\\347\\030\\220\\275\\336%E@',4326);
```

Observe that if you were to output the WKB of this function:

```
SELECT
  ST_AsBinary(ST_GeomFromWKB(E'\\001\\001\\000\\000\\000\\321\\256B\\31
  20\\3040\\300\\347\\030\\220\\275\\336%E@',4326));
```

it would look like this.

```
\001\001\000\000\321\256B\3120\3040\300\347\030\220\275\336%E@
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

The extra slashes we put in when feeding in the value is to escape out the "\\" in the string. This is only needed if you have standard\_conforming\_strings=off, which is the default for PostgreSQL versions before PostgreSQL 9.0.

Below is an example if you have standard\_conforming\_string=on

```
set standard_conforming_strings = on;
SELECT
ST_GeomFromWKB('\'001\001\000\000\000\321\256B\3120\304Q\300\347\030\2
20\275\336%E@');
```

### **Canonical representation**

Try doing a simple select statement from a geometry column, unadorned with any functions and you will end up with something that looks like a long string of digits. This is actually a hexadecimal representation of the EWKB notation. You can create a geometry with this canonical form by doing the ANSI SQL compliant:

```
SELECT    CAST('0101000020E61000008048BF7D1D2059C017B7D100DEB23C40'    As
geometry);
```

or the PostgreSQL short cast notation

```
SELECT '0101000020E61000008048BF7D1D2059C017B7D100DEB23C40'::geometry;
```

To be in conformance with OGC-MM, PostGIS offers other functions such as ST\_PointFromText, ST\_PolyFromText, ST\_GeometryFromText etc. Our advice, as far as using PostGIS is concerned, is to just stay away from them and stick with ST\_GeomFromText, ST\_Point, ST\_MakePoint, ST\_GeomFromWKB etc. The reason for that is that these other functions are just wrappers around ST\_GeomFromText... with a check to make sure that the geometry is actually a polygon, point etc and to nullify it if it isn't. There is no need for such checking if your tables are setup correctly to only accept specific geometry types anyway. These extra functions end up just adding unnecessary overhead to your inserts and updates. Do remember that our advice only applies to PostGIS implementation of these functions.

### **4.1.2 Autocasting in PostgreSQL/PostGIS**

You will encounter instances where someone might take a text representation of a geometry and stuff it right into a function that takes a geometry. While this is convenient, you should exercise it with caution. Below is a demonstration of such a practice.

```
SELECT ST_Centroid('LINESTRING(1 2,3 4)');
```

To see the output of the above – we do this:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```
SELECT ST_AsText(ST_Centroid('LINESTRING(1 2,3 4)'));
```

Which returns this:

```
POINT(2 3)
```

The above practice makes you quickly forget that a centroid works on a geometry not a string. It works because there is an autocast built into PostGIS that takes a string and converts it to a geometry automatically. The more verbose, but clearer, way to write the statement is as follows:

```
SELECT ST_Centroid(ST_GeomFromText('LINESTRING(1 2,3 4)'));
```

Relying on autocasts is convenient but should be used with caution. The reason is that if you have 2 functions that take different data types and both data types have an autocast built-in, you could end up with an ambiguity error. Below is a classic example:

```
SELECT ST_Box3D('BOX(1 2, 3 4)');
```

PostgreSQL will throw a casting error because ST\_Box3D can accept both a box object and a geometry. After autocasting the text representation to a geometry, PostgreSQL no longer knows if you intended to pass in a box or a geometry. Here is another that will fail. ST\_Xmin is a function defined only for box3D. The below will fail because there is no autocast that will take a text representation of a geometry directly to a box3d but there is one that takes a text representation of a box2d to a box3d.

```
SELECT ST_Xmin('LINESTRING(1 2, 3 4)');
```

PostgreSQL throws the error below:

```
ERROR: BOX3D parser - does not start with BOX3D;
```

Bypass the autocasting with the query below:

```
SELECT ST_Xmin('BOX3D(1 2 0, 3 4 0)');
```

In the next section we will discuss output functions, which are the opposite of input functions. PostGIS offers a lot more output functions than input functions to accommodate the ever-growing number of GIS client tools wanting their data in a particular format.

## 4.2 Outputs

Output functions are functions that return a geometry representation in another industry standard format. This allows third-party rendering tools with no knowledge of PostGIS to be repurposed and used as a display tool for PostGIS.

In this section we will summarize the output formats available, general use scenarios, and the PostGIS functions to output them. We will cover some of the more popular output formats, but you should check the official PostGIS site for the ever-growing list. To learn more about the various output format themselves, be sure to visit their own sites. We are not going to go into detail about the various formats.

Finally, we advise that you use your good judgment rather than memorize the intricacies of each function when it comes to determining if the output makes sense for your particular format. For example, if you have only known a particular format to support 2D with SRID 4326, make sure your geometries are all 2D with SRID 4326 prior to using the export function instead of trying your luck. This will save you time from having to remember how each function handles exceptions and will make sure your code still works should the default handling of the output functions change, as they often do with each version of PostGIS.

### 4.2.1 Well known text (WKT) and well known binary (WKB)

Well known text (WKT) is the most common OGC standard format for expressing geometries. We have already used this format quite extensively in the book to show the output of queries because it provides a nice text representation of the underlying geometry.

Two functions that output geometries in this format: ST\_AsText and ST\_AsEWKT. Recall from earlier discussions that ST\_AsEWKT function is a PostGIS specific extension loosely based on the OGC-MM WKT standard, but is not considered OGC-compliant. The OGC-compliant function is ST\_AsText, but this function will not output the SRID, M or Z coordinate. This could change in the future as draft MM standards already propose the addition. Finally, textual representation will always lack the precision of binary representations and will preserve only about 15 significant digits.

Well known binary (WKB) is an OGC standard format. Two functions that output geometries in this format: ST\_AsBinary and ST\_AsEWKB. The ST\_AsEWKB function is a PostGIS specific extension loosely based on the standard, but not OGC compliant. ST\_AsBinary will NOT output M or Z coordinate or the SRID, but ST\_AsEWKB will. Unlike text representation, binary format maintains precision. You can be assured what is stored in your database is what you are outputting and what you export can be read back in using their inverse functions of ST\_GeomFromWKB and ST\_GeomFromEWKB.

### 4.2.2 Keyhole markup language (KML)

Keyhole Markup Language (KML) is an XML based format created by the Keyhole Corporation for rendering their applications. KML gained enormous popularity after Google acquired Keyhole and folded KML into Google's own mapping offerings of Google Maps and

Google Earth. OGC has recently accepted KML as a standard transport format in its own right.

The PostGIS function for exporting to KML is ST\_AsKML. As of PostGIS 1.4, there are 4 variants of this function. The default outputs in KML version 2 with 15-digit precision. Other variants allow you to change the target KML version and precision.

The spatial reference system for KML is WGS-84 Long Lat (SRID 4326). As long as your geometry is already in a known SRID (via membership in the spatial\_ref\_sys meta table), ST\_AsKML functions will automatically convert to SRID 4326 for you.

ST\_AsKML supports both 2D and 3D geometries, but will throw an error in PostGIS 1.4 and above when exporting curved geometry or geometry collections. Prior versions of PostGIS simply returns null for unsupported geometry types. Also keep in mind that while the ST\_AsKML functions will accept geometries containing an M coordinate, it will not output the M coordinate.

#### **4.2.3 Geography markup language (GML)**

Geography Markup Language (GML) is an XML based format and an OGC defined transport format. It is commonly used in Web Feature Services (WFS) to output the columns of a query.

The PostGIS function for exporting to GML is ST\_AsGML. As of PostGIS 1.4, there are 5 variants of this function allowing you to vary target GML versions and precisions. Supported GML versions are 2.1.2 (pass in as 2) and 3.1.1 (pass in 3). If no version parameter is passed, then 2.1.2 is assumed. Two additional parameters control number of significant digits and a bit field indicating whether to use short CRS (Coordinate Reference Systems).

ST\_AsGML supports 2D, 3D for both geometries and geometry collections. If a geometry has an M coordinate, the M is dropped. Passing in curved geometries into will throw an error in PostGIS versions 1.4 and above and return null in older versions.

#### **4.2.4 Geometry JavaScript object notation (GeoJSON)**

Geometry JavaScript Object Notation (GeoJSON) is a recently developed format based on JavaScript object notation (JSON). GeoJSON is geared towards consumption by AJAX-oriented applications (such as OpenLayers) because its output notation is in JavaScript format. JSON is the standard object representation in JavaScript data structures. GeoJSON extends JSON by defining a format for geometry storage within the JSON format. More detail on GeoJSON specification can be found here: <http://geojson.org/geojson-spec.html>.

The PostGIS function for exporting to GML is ST\_AsGeoJSON (first introduced in PostGIS 1.3.5). There are 6 variants of this function as of PostGIS 1.5. The arguments are similar to those for ST\_AsGML—target version, number of decimal places, and an encoded flag denoting if to include the bounding box, short or long CRS, and other options. ST\_AsGeoJSON supports 2D and 3D and geometry collections. It will drop the M coordinate and throw an error for curved geometries.

### **4.2.5 Scalar vector graphics (SVG)**

Scalar Vector Graphics (SVG) has been around for a while and is very popular among high-end rendering tools as well as drawing tools such as Inkscape. Toolkits such as ImageMagick can easily convert SVG to many other image formats. It is also one of the basic formats used by Macromedia Flash/Flex. Microsoft Silverlight's XAML also uses a derivative of the basic SVG format. Most web browsers either have native support for it or via an installable plug-in.

The PostGIS function for exporting to SVG is ST\_AsSVG. As of PostGIS 1.4, this function only outputs 2D geometries without SRIDs, no Z or M coordinates and also no curved geometries. There are 3 variants of the function to indicate if the output points are relative to an origin or relative to the coordinate system, and the level of precision desired.

### **4.2.6 Geohash**

Geohash is a lossy geocoding system for longitudes and latitudes. Its use is more as a hashed encoding tool to ease communicating of coordinates rather than for visual presentation. You can explore its details here: <http://geohash.org>.

PostGIS outputs to Geohash via the ST\_Geohash function. Naturally, ST\_GeoHash must be in Long Lat (WGS-84) output. Your data must have a known SRID so that ST\_GeoHash can automatically transform it for you. ST\_GeoHash can support curved geometries but ignores the Z and M coordinates of geometries. Keep in mind that Geohash is point-based, so if you output anything other than points, ST\_Geohash will just take some interpolation of the bounding box. If you try to pass it too big of an area, it will refuse to give you an output.

### **4.2.7 Examples of Output functions**

It is now time to present a grand example that brings all the output functions together. We will be asking out functions to output a line string in SRID 4326 to a precision of 5 significant digits (The linestring originates in northern France and terminates in southern England.)

#### **Listing 4.3 Example of outputting geometries using different functions**

```
SELECT ST_AsGML(geom,5) as GML, ST_AsKML(geom,5) As KML,
ST_AsGeoJSON(geom,5) As GeoJSON, ST_AsSVG(geom,0,5) As SVG_Absolute,
ST_AsSVG(geom,1,5) As SVG_Relative, ST_GeoHash(geom) As Geohash
FROM (SELECT ST_GeomFromText('LINESTRING(2 48, 0 51)', 4326) As geom) foo;
```

Table 4.1 Results of code in Listing 4.3

Format	Output
GML	<gml:LineString srsName="EPSG:4326"><gml:coordinates>-2,48 1,51</gml:coordinates></gml:LineString>
KML	<LineString><coordinates>2,48 1,51</coordinates></LineString>
GeoJSON	{"type": "LineString", "coordinates": [[2,48],[1,51]]}

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

SVG\_Absolute M 2 -48 L 1 -51

SVG\_Relative M 2 4 L -1 -3

Geohash u

Before moving onto the next section, remember that the output functions we covered only export the geometry fragments necessary to create a fully functional data value in the various formats. Many formats have associated scalar attribute data and PostGIS functions will ignore these. For example, KML, JSON often embed scalar data within JSONed and KMLed wrappers. In the next section, we will cover the scalar setter and accessor functions that will be useful in interfacing with the non-geometric aspects of geometries.

### **4.3 Accessor functions: getters and setters**

If you have exposure to any object-oriented language, accessor functions come as nothing new. We borrow the terms from OO programming to mean any function which gets or sets intrinsic properties of a geometry. Since quite a large number of functions fall under this classification, we decided to only ascribe the term only to properties that are not themselves geometries. For example, if we have a square polygon, we would consider functions that return or set the type, the SRID, and dimension to be accessors. Functions that return the centroid, (a point), the diagonal (linestring), or the boundary (linestring collection) we will call decomposition functions and save for a later section. We will also relegate getter functions that return measurements to its own class.

There are a couple of defining characteristics of geometries are important to know when you are using spatial functions.

- Spatial reference system (SRID) defines the spatial coordinate system, ellipsoid/spheroid, and the datum of the coordinates used in defining the geometry.
- Geometry type defines the kind of geometry: a point, linestring, polygon, multipolygon, multicurves, etc.
- Coordinate dimension is the dimension of the vector space in which our geometry lives. In PostGIS, this can be 2, 3, or 4.
- Geometric dimension is the minimal dimension of the vector space necessary to fully contain the geometry. (Plenty of more rigorous definitions abound, but we are sticking with something intuitive.) In PostGIS, geometry dimensions can be 0 (points), 1 (linestrings), 2 (polygons). In this section, we go into detail about these intrinsic properties of geometries and the various functions to get and set them.

#### **4.3.1 Getting and setting spatial reference system**

For any kind of location based application involving measurements, the concept of a spatial reference system, what spatial reference system your current data is in, and choosing the

right base spatial reference system to store your data are of the utmost importance. Spatial reference system gives rise to meaningful measurement, and allows sharing of data.

In PostGIS, ST\_SRID is the function that retrieves the spatial reference system of a geometry. This is an OGC SQL/MM standard function that you will find in most spatial databases. The companion setter function is ST\_SetSRID(), also a SQL/MM standard. This setter function will replace the spatial reference metadata embedded within a geometry. Remember that all geometries must have a SRID, even if it is the unknown SRID (-1). Let us take a look at uses of this accessor.

#### **Listing 4.4 Example use of ST\_SRID**

```
--[1 Beg]
SELECT ST_SRID(ST_GeomFromText('POLYGON((1 1, 2 2, 2 0, 1 1))', 4326));
--[1 End]
--[2 Beg]
SELECT ST_SRID(geom) As srid, COUNT(*) As number_of_geoms
FROM sometable
GROUP BY ST_SRID(geom);
--[2 End]
--[3 Beg]
SELECT ST_SRID(geom) As srid, ST_SRID(ST_SetSRID(geom,4326)) as srid_new
FROM (VALUES (ST_GeomFromText('POLYGON((70 20, 71 21, 71 19, 70 20))',
4269)), (ST_Point(1,2))) As foo (geom);
--[3 End]

[1] Simple use of ST_SRID
[2] Counts number of distinct SRIDs
[3] Using ST_SetSRID to change SRID
```

If you set up your production tables properly, your geometries should only contain SRIDs found in the spatial\_ref\_sys meta table. While there is nothing in the OGC specification requiring SRIDs to have any real world significance, PostGIS pre-populates the spatial\_ref\_sys meta table with only the EPSG approved SRIDs. You are free to invent your own SRID and add them to the meta table. People commonly add SRIDs associated with ESRI since it is the leading commercial nemesis of PostGIS.

In PostGIS 2.0 version work is planned to support geodetic systems so that they are not treated as planar. While this will not completely replace planar spatial reference systems in use, it will give you the option of using a spherical model, when no planar model can suit your needs. Planar models are in general faster to work with in most spatial databases (in terms of computational intensity) and are more accurate for smaller areas than geodetic, have more spatial functions supporting them, and are better supported by other GIS tools. As a result, it is unlikely they will be completely superseded by geodetic models in the short-term. There are also non-geographic use cases for spatial databases and in these cases geodetic spatial reference systems are not useful.

### 4.3.2 Transform to a different spatial reference

No discussion of spatial reference can be complete without introducing ST\_Transform function, which converts all the points of a given geometry to coordinates in a different spatial reference system. A common application of this function is to take a WGS-84 Lon Lat geometry and transform it to a planar spatial reference system so that we can take meaningful measurements of the geometry of interest. Below is an example that takes a road in somewhere in New York state expressed in WGS-84 Lon Lat and converts it to WGS-84 UTM Zone 18N meters.

```
SELECT
ST_AsEWKT(ST_Transform(ST_GeomFromEWKT('SRID=4326;LINESTRING(-73    41,
-72 42)'), 32618));
```

The output of the above code snippet is

```
SRID=32618;LINESTRING(668207.88519421           4540683.52927698,
748464.920715711 4654130.89132385)
```

Now that we have transformed from geodetic measure to planar measure, obtaining the length is nothing more than a simple application of the Pythagorean theorem.

People often get confused between ST\_SetSRID and ST\_Transform functions. You must remember that ST\_SetSRID does not change the underlying coordinates of a geometry. It simply adds information to the header of the geometry stating that its frame of reference is a particular spatial reference. ST\_SetSRID comes in useful when you realize that you made a mistake during import of data. For example, if you imported your geometries say with WGS-84 Lon Lat (SRID: 4326) and you later realized they were really defined using NAD 27 Lon Lat coordinates (SRID: 4267), just use ST\_SetSRID to correct the mistake.

The ST\_Transform function actually changes the coordinates of each point of a geometry from the geometry's stated SRID to a new SRID using the proj4text defined in the spatial\_ref\_sys for the from spatial reference and the to spatial reference to formulate a conversion formula and changes the SRID meta data as well. We shall explore the use of ST\_Transform later in this chapter. So keep in mind ST\_Transform needs to know what the current correct SRID is since it does math to reproject all the points from the current to the new and reads this information from the geometry structure header, whereas ST\_SetSRID only needs to know the new SRID since it will do nothing to the points in the geometry but will write this new SRID value to the geometry's structure header and ignore whatever one was stated before. If you have the wrong one originally defined, which is quite common, and then transform to another spatial reference system, you will get goofy results.

Just because ST\_Transform is so versatile does not mean that you can use it blindly. When you reproject, you still must make sure that your spatial reference system covers the region under consideration. For example, if you transform from SRID 36932, an Alaska state plane spatial reference to 32130, a Rhode Island state plane reference, you may get an out-

of-bound error. If you do not, you are on your own to discover the folly of what you have just done.

Despite its prowess ST\_Transform is not all that computationally intensive, but if you have choice of SRIDs when storing your data, you should still choose the most popular ones and then create views that transform to other SRIDs. It also does not hurt to add a functional index based on the ST\_Transform to the table for each of the dependent views.

### **4.3.3 Geometry type**

In most situations, you are keenly aware of the geometry types your are working with, but if you import data containing heterogeneous geometry columns, you will need the two functions that PostGIS offers to identify geometry types: GeometryType and ST\_GeometryType. We have mentioned that functions without the ST prefix in PostGIS are deprecated functions, but in the case of GeometryType versus ST\_GeometryType, they are not only difference from each other, both are very much in use.

The GeometryType function is the older function of the two. It is part of the OGC Simple Features for SQL. It returns the geometry types that we are familiar with in all upper case. Its younger counterpart, ST\_GeometryType, is part of the OpenGIS SQL/MM. It outputs the familiar geometry names but prepends ST\_ to comply with the MM geometry class hierarchy naming standards. Below is a listing of code that demonstrates the difference between the two and the results.

#### **Listing 4.5 Differences in output between ST\_GeometryType and GeometryType**

```
SELECT ST_GeometryType(geom) As new_name, GeometryType(geom) As old_name
FROM (VALUES
(ST_GeomFromText('POLYGON((0 0, 1 1, 0 1, 0 0))')),
(ST_Point(1, 2)),
(ST_MakeLine(ST_Point(1, 2), ST_Point(1, 2))),
(ST_Collect(ST_Point(1, 2), ST_Buffer(ST_Point(1, 2),3))),
(ST_LineToCurve(ST_Buffer(ST_Point(1, 2), 3))),
(ST_LineToCurve(ST_Boundary(ST_Buffer(ST_Point(1, 2), 3)))),
(ST_Multi(ST_LineToCurve(ST_Boundary(ST_Buffer(ST_Point(1, 2),3))))))
) As foo (geom);
```

Table 4.3 Results of code in Listing 4.5

<b><u>new_name</u></b>	<b><u>old_name</u></b>
ST_Polygon	POLYGON
ST_Point	POINT
ST_LineString	LINESTRING
ST_Geometry	GEOMETRYCOLLECTION
ST_CurvePolygon	CURVEPOLYGON
ST_CircularString	CIRCULARSTRING

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

ST\_MultiCurve      MULTICURVE

Determining the geometry type comes is particularly useful trying to apply various functions to a heterogeneous geometry column. Remember that some functions accepts certain geometry types and not others or may behave differently for different geometry types. For example, asking the area of a line is pointless. The same goes for asking for the length of a polygon. Using a SQL CASE statement is a compact way to selectively apply functions against a heterogeneous geometry column. Here is an example:

```
SELECT CASE WHEN GeometryType(geom) = 'POLYGON' THEN ST_Area(geom),
    WHEN GeometryType(geom) = 'LINESTRING' THEN ST_Length(geom) END
FROM foo;
```

#### **4.3.4 Coordinate and geometry dimensions**

There are two kinds of dimensions when talking about geometries. Coordinate dimension is the dimension of the space that the geometry lives in and geometry dimension is the smallest dimensional space that will fully contain the geometry. The coordinate dimension is always greater than or equal to the geometric dimension. PostGIS provides ST\_CoordDim and ST\_Dimension to return the coordinate and geometry dimensions respectively. In the following listing we apply these two functions against a variety of geometries and see what we get.

#### **Listing 4.6 Coordinate and geometry dimensions of various geometries**

```
SELECT item_name, ST_Dimension(geom) As gdim, ST_CoordDim(geom) as cdim
FROM (
    VALUES ('2d polygon',
        ST_GeomFromText('POLYGON((0 0, 1 1, 1 0, 0 0))') ),
    ('2d polygon with hole',
        ST_GeomFromText('POLYGON ((-0.5 0, -1 -1, 0 -0.7, -0.5 0),
            (-0.7 -0.5, -0.5 -0.7, -0.2 -0.7, -0.7 -0.5))') ),
    ('2d point', ST_Point(1,2) ),
    ('2d line', ST_MakeLine(ST_Point(1,2), ST_Point(3,4)) ),
    ('2d collection', ST_Collect(ST_Point(1,2), ST_Buffer(ST_Point(1,2),3)) ),
    ('2d curved polygon', ST_LineToCurve(ST_Buffer(ST_Point(1,2), 3)) ) ,
    ('2d circular string',
        ST_LineToCurve(ST_Boundary(ST_Buffer(ST_Point(1,2), 3))) ) ,
    ('2d multicurve',
        ST_Multi(ST_LineToCurve(ST_Boundary(ST_Buffer(ST_Point(1,2), 3)))) ),
    ('3d polygon', ST_GeomFromText('POLYGON((0 0 1, 1 1 1, 1 0 1, 0 0 1))') ),
    ('2dm polygon',
        ST_GeomFromText('POLYGONM((0 0 1, 1 1 1.25, 1 0 2, 0 0 1))') ),
    ('3d(zm) polygon',
        ST_GeomFromEWKT('POLYGON((0 0 1 1, 1 1 1 1.25, 1 0 1 2, 0 0 1 1))') ),
    ('4d (zm) multipoint',
        ST_GeomFromEWKT('MULTIPOINT(1 2 3 4, 4 5 6 5, 7 8 9 6)') )
) As foo(item_name, geom);
```

**Table 4.4 Results of code in Listing 4.6**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

<b>item_name</b>	<b>gdim</b>	<b>cdim</b>
2d polygon	2	2
2d polygon with hole	2	2
2d point	0	2
2d line	1	2
2d collection	2	2
2d curved polygon	2	2
2d circular string	1	2
2d multicurve	1	2
3d polygon	2	3
2dm polygon	2	3
4d(zm) polygon	2	4
4d (zm) multipoint	0	4

Take note of the exceptional cases from the above table: a point or a multipoint always has a geometry dimension of zero; a line or multiline always one; and a polygon or multipolygon always two.

#### **4.3.5 Geometry validity**

We introduced the concept of validity in Chapter 2. Pathological geometries such as polygons with self intersections and polygons where the holes lie outside the exterior ring are invalid. Generally speaking, the higher the geometry dimension of a geometry the more prone it is to invalidity. The PostGIS function ST\_IsValid tests for validity, but as of PostGIS 1.4, ST\_IsValidReason can provide a brief description about why a geometry is not valid. It will only offer up a description for the first offense encountered, so if your geometry is invalid for multiple reasons, you will only see the first reason. If a geometry is valid, it will return the string "Valid Geometry".

We remind you again that it is important to make sure your geometries are valid. Do not even try to work with geometries unless they are valid. Many of the GEOS-based functions in PostGIS will behave unpredictably against invalid geometries.

#### **4.3.6 Number of Points that defines a geometry**

ST\_NPoints is a function that returns the number of points defining a geometry. It works for all geometries. It is a PostGIS creation so is not guaranteed to be found in other OGC compliant spatial databases. Many people often make the mistake of using the function ST\_NumPoints instead of ST\_NPoints. By PostGIS 2.0, the two functions will probably become interchangeable. Prior to PostGIS 2.0, ST\_NumPoints only works when applied to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

linestrings as dictated by the OGC specification. When used with multilinestrings, only the first linestring in the collection will be considered.

You may be wondering why there are two functions where one can completely perform the duties of another and more. This has to do with the fact that most spatial databases, PostGIS included, will offer functions that adheres strictly to the OGC specification. After meeting the OGC specifications to the letter, spatial databases continue on to extend OGC functions where they find deficiencies.

#### **Listing 4.7 Example of ST\_NPoints and ST\_NumPoints**

```
SELECT atype, ST_NPoints(geom) As npoints, ST_NumPoints(geom) As numpoints
FROM (VALUES ('LinestringM', ST_GeomFromEWKT('LINESTRINGM(1 2 3, 3 4 5, 5
8 7, 6 10 11)'), ('Circularstring', ST_GeomFromText('CIRCULARSTRING(2.5 2.5, 4.5 2.5, 4.5
4.5)'), ('Polygon (Triangle)', ST_GeomFromText('POLYGON((0 1,1 -1,-1 -1,0 1))')), ('Multilinestring', ST_GeomFromText('MULTILINESTRING ((1 2, 3 4, 5 6), (10
20, 30 40)))'), ('Collection', ST_Collect(ST_GeomFromText('POLYGON((0 1,1 -1,-1 -1,0 1))'),
ST_Point(1,3)))
) As foo(type, geom);
```

**Table 4.5 Output results of code in Listing 4.7**

<b>type</b>	<b>npoints</b>	<b>numpoints</b>
LinestringM	4	4
Circularstring	3	
Polygon (Triangle)	4	
Multilinestring	5	3
Collection		5

The output shows that ST\_NPoints returns the number of points used in the definition, not the distinct number of points required to adequately generate the geometry. Look at our triangle; it has three vertices, but ST\_NPoints actually shows four points. ST\_NPoints does not count distinct points, but simply reads the points listed in the definition.

#### **4.4 Measurement functions**

Before taking any measurements in GIS, you must concern yourself with the scale of what you are measuring. This goes back to the fact that we live on a spheroid called earth and you are presumably trying to measure something on its surface. When your measurements cover a small area where the curvature of the earth does not come into play, it is perfectly fine to assume a planar model where we treat the earth as essentially flat. Though what distances should be considered small depends on the accuracy of the measure you are trying

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

to achieve, we have found that planar measure is often the first choice even across very long distances, say distances stretching across an entire continent. People will cling onto the simplicity and intuitiveness that comes with planar measurement even at the expense of accuracy. Measurements based on plane generally have units in meters or feet. Planar models are better supported by GIS tools and faster to process.

Once your distances start to cross continents and oceans, planar measures deteriorate rapidly. You will have to use geodetic measurements where you must consider the spherical nature of the earth. A geodetic measurement models the world as a sphere or spheroid. Coordinates are expressed using degrees or radians. The classic SRID 4326 (WGS-84, Lon Lat) is the most common of the geodetic spatial reference systems in use today.

In this section we cover both kinds of measurements. Prior to PostGIS 1.5, geodetic measurements took a back seat since PostGIS only supported planar geometries. With PostGIS 1.5 came the new geography data types. These new geodetic data types are always in SRID 4326 and PostGIS functions will automatically apply geodetic calculations when using measurement functions against geography data. PostGIS does have dedicated functions that only work on spheroids and can be used with the geometry type. These are most used if most of the work you do requires you keep your data in geometry type, but every once in a while you need to measure using a geodetic model.

One last point to keep in mind: Measurement functions are always used as getters. Setting the measurement of a geometry does not make sense. To change a measurement, you would have to change the geometry itself.

#### **4.4.1 Planar measures for geometry types**

All the planar measurement functions we are about to discuss have units as defined by the spatial reference system that is defined for the geometry. If your spatial reference system is in feet then so are your lengths and your area would be square feet. These functions are ST\_Length, ST\_Length3D, ST\_Area, ST\_Perimeter. If your spatial reference system is in degrees of longitude and latitude (spherical coordinates), then your units will measure will be in degrees after PostGIS naively maps longitude to x coordinate values and latitude to y coordinate values. This may be okay if you are considering small areas where earth curvature does not matter and you have adequate significant digits to work with.

As of PostGIS 1.4, ST\_Length3D is the only one of these measurement functions that considers the z coordinate. Other measurement functions will simply ignore any z coordinate in the input instead of throwing an error.

ST\_Length and ST\_Length3D only apply to linestrings and multilinestrings. ST\_Length3D considers the z coordinate when measuring length where as ST\_Length ignores the z coordinate. As of PostGIS 1.4, there is no distance function for calculating distance between two points in 3D coordinate space, ST\_Length3D applied in series with ST\_MakeLine is the typical work around.

Below is an example demonstrating the 2D and 3D lengths of a 3D linestring. As demonstrated in the below, the length returned by ST\_Length and ST\_Length3D is the same

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

for a linestring in 2D coordinate space. There is discussion going on to revise ST\_Length to return the highest dimension length by default, but compatibility concerns with older PostGIS versions are preventing its adoption.

#### **Listing 4.8 Compare 2D and 3D Lengths of a given linestring**

```
SELECT ST_Length(geom) As length_2d, ST_Length3D(geom) As length_3d
FROM (VALUES(ST_GeomFromEWKT('LINESTRING(1 2 3, 4 5 6)'),,
ST_GeomFromEWKT('LINESTRING(1 2, 4 5)')) As foo(geom);
```

**Table 4.6 Result of Listing 4.8 comparing 3d and 2d lengths**

<b>Length2d</b>	<b>Length3d</b>
4.24264068711928	5.19615242270663
4.24264068711928	4.24264068711928

The two other common measurement functions for area and perimeter are fairly intuitive. Obviously, you should use them against valid polygons and multipolygons. For multi-ringed polygons, ST\_Perimeter calculated the length of all the rings. You should also keep in mind that both ST\_Area and ST\_Perimeter are completely equivalent to ST\_Area2D and ST\_Perimeter2D respectively. Since these two functions only work in 2D coordinate space anyway, we suggest you not use the 2D names to prevent future upgrade problems.

#### **4.4.2 Geodetic measurement for geometry types**

All the measurements we discussed thus far apply to geometries living in the Cartesian coordinate systems. Since the earth as a whole is not flat, a more appropriate coordinate system to use when looking at large swaths of the planet is the spherical coordinate system. Geodetic is simply a fancier-sounding term for spherical coordinate used to map out our earth. Spherical coordinates literally throws a curve into our common sense grasp of length, areas, and perimeters. Take the simple question of what is the length between Mumbai and Chicago. The only straight line passes through the center of the earth. Along the surface of the earth, you have a myriad of curved lines connecting the two cities. Even if you were to always take the shortest curve, you cannot guarantee that it will be unique. Try drawing the shortest line between the two geographic poles. You end up not with one, but many, infinitely many.

As of PostGIS 1.4, the only geodetic measurement functions available is ST\_Length\_Spheroid (also known as ST\_Length3D\_Spheroid). This function always return distance in meters. Should you have a z coordinate value as well to represent elevation, you need to make sure the units are in meters. Before using ST\_Length\_Spheroid, double-check that your geometries are in some type of degree-based spatial reference system; SRID 4326 is by far the most popularly used.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

PostGIS 1.5 introduced a new spatial type called geography, which uses geodetic measurement instead of cartesian measurement. Coordinate points in the geography type are always represented in WGS 84 long lat degrees (SRID: 4326), but measurement functions and relationships ST\_Distance, ST\_DWithin, ST\_Length, and ST\_Area always return answers in meters or assume inputs in meters.

Prior to PostGIS 1.5, the basic geodetic functions that can be used with the geometry cartesian type were very limited. The Length\_Spheroid functions as of PostGIS 1.4 and below only worked with linestring geometries and multilinestrings and the ST\_Distance\_Spheroid and ST\_Distance\_Sphere functions only work with points. From PostGIS 1.5 and above, they also work with polygons, linestrings, and the multi variants of those. The main difference between the \*\_Sphere and \*Spheroid functions is that the Sphere functions use a perfect sphere for calculation where as \*Spheroid use a named spheroid. If you are using a spheroid, you want to make sure your long lat are measured along that spheroid model. In later versions of PostGIS it is planned to have the spheroid be read from the spatial reference system defined for the geometry so that the extra spheroid argument will be unnecessary. WGS 84 and GRS 80 are the most commonly used. Both are so similar that it generally doesn't matter which one you use even if you get it wrong.

In terms of choosing between the geometry and geography type for data storage, you should consider what you will be using it for. If all you do are simple measurements and relationship checks on your data, and your data covers a fairly large area, then most likely you will be better off storing your data using the new geography type.

Although the new geography data type can cover the globe, the geometry type is far from obsolete. The geometry type has a much richer set of functions than geography, relationship checks are generally faster, and it has wider support currently across desktop and web mapping tools. If you only need support for a limited area such as a state, a town, a small country, then you would do better with the geometry type. If you also do a lot of geometric processing such as unioning geometries, simplifying, line interpolation and other kinds of processing, geometry will provide that out of the box for you, whereas for geography, you would need to cast to geometry, transform, process, and cast back to geography to be able to achieve those goals.

#### **Listing 4.9 Examples of calculating length of a multilinestring with different spheroids and compare with different transforms**

```
-- 1
SELECT sp_name, geom_name,
       ST_Length_Spheroid(g.geom, s.the_spheroid) As sp3d_length,
       ST_Length3D(ST_Transform(g.geom, 26986)) As ma_state_m,
       ST_Length3D(ST_Transform(g.geom, 2163)) As us_nat_atl_m
FROM (VALUES ('2d line', ST_GeomFromText('MULTILINESTRING((-71.205 42.531,-
71.204 42.532), (-71.21 42.52, -71.211 42.52))'),4326)),
      ('3d line', ST_GeomFromEWKT('SRID=4326;MULTILINESTRING((-71.205 42.531
10,-71.205 42.531 15,-71.204 42.532 16, -71.204 42.532 18),
(-71.21 42.52 0,-71.211 42.52 0))')) )
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```

) As g(geom_name, geom)
CROSS JOIN
-- 2
  (VALUES ('grs 1980', CAST('SPHEROID["GRS_1980",6378137,298.257222101]' As
spheroid)),
  ('wg 1984', CAST('SPHEROID["WGS_1984",6378137,298.257223563]' As
spheroid))
  ) As s(sp_name, the_spheroid);
-- 1 geometries
-- 2 spheroids

```

- (1)** In the above example we demonstrate the lengths of a 2d and a 3d multilinestring first by the spheroid function that takes each of our spheroids, we also transform our long lat coordinates to Massachusetts state plane meter planar and then use the regular length 3d. We repeat the transform exercise but use US National Atlas meter planar. **(2)** The spheroid object is another PostGIS object – the name ascribed is not relevant for calculations, but what is relevant are the semi major axis (6378137 for both) and inverse flattening (298....). In terms of accuracy, the state plane is most accurate followed by the spheroids. The US national atlas is usually accurate within 10 meters (depending on length/distance), but it covers all of continental US and can be used in all PostGIS planar operations for long range calculations.

**Table 4.7 Results of query in Listing 4.9**

sp_name geom_name	sp3d_length	ma_state_m	us_nat_atl_m
grs 1980 2d line	220.337420025626	220.33319845914	220.759524564227
wg 1984 2d line	220.337387457848	220.33319845914	220.759524564227
grs 1980 3d line	227.341038849482	227.336817351126	227.763097850584
wg 1984 3d line	227.341006282557	227.336817351126	227.763097850584

### Key characteristics about ST\_Length\_Spheroid functions

Considers the z-coordinate (elevation) and z-coordinate is assumed to be in meters

Only works with linestrings and multilinestrings

Units returned are always in meters

Coordinates of the geometry are always assumed to be long lat for PostGIS 1.4 and below.

While there exists no ST\_Perimeter\_Spheroid, its easy enough to simulate one by taking the ST\_Boundary of a polygon and then the ST\_Length\_Spheroid of it. This will only work for 2D polygons since ST\_Boundary will ignore the z coordinate.

#### **4.4.3 Measurement with geography type**

All measurements against geography type presumes a geodetic model. In addition all measurements return meters, but all coordinates are stored as WGS 84 long lat degrees.

Aside from that, the measurement functions you will find for geography, for the most part, parallel those for geometry. There is ST\_Length, ST\_Area, ST\_Distance, ST\_DWithin which work just like they do for geometry. The only difference is that these functions can take an optional argument named use\_spheroid. If this is set to true or not passed in, then the calculations are done using a spheroid. If you pass in false, then all calculations are done using a sphere model. The sphere model is faster than spheroid, but the difference is generally negligible. The measurements do not consider the z axis whatsoever. Unless, you plan on journeying deep into the center of the earth or go on frequent jaunts into outer space, the curvature of the earth outrivals any consideration of height.

#### **POSTGIS 1.5 WHERE IS PERIMETER FOR GEOGRAPHY?**

The ST\_Perimeter function for geography is noticeably absent in PostGIS 1.5. To obtain the perimeter of a polygon geography type, you need to use ST\_Length. This quirk in the function names has been fixed in PostGIS 2.0.

To demonstrate, we will create the same types of objects we had for geometry data types except using geography data types and compare the spheroid versus sphere answers.

#### **Listing 4.10 Compare spheroid and sphere calculations in geography**

```
SELECT name, ST_Length(geog) As sp3d_lengthspheroid, ST_Length(geog, false)
As sp3d_lengthssphere
FROM (VALUES ('2D Multilinestring',
    ST_GeogFromText('SRID=4326;MULTILINESTRING((-71.205 42.531, -71.204
42.532), (-71.21 42.52, -71.211 42.52))'))),
('3D Multilinestring', ST_GeogFromText('SRID=4326;MULTILINESTRING((-71.205
42.531 10, -71.205 42.531 15, -71.204 42.532 16,-71.204 42.532 18), (-71.21
42.52 0, -71.211 42.52 0))'))
) As foo (name, geog);
```

The results of the code run are:

**Table 4.8 Results of query in listing 4.10 demonstrating sphere vs. spheroid lengths**

<b>geom_name</b>	<b>length_spheroid</b>	<b>length_sphere</b>
2d line	220.337435990337	220.080539442185
3d line	220.337435990337	220.080539442185

As demonstrated in the result above, the z coordinate is completely ignored for the geography ST\_Length function. For this particular area the difference between the spheroid and sphere lengths is less than 1 meter.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

Although the geography type has a fairly complete set of measurement functions, the other functions you will find available for the geometry type are for the most part missing for geography. The main exceptions to this rule are that geography does have a ST\_Intersects, ST\_Intersection, and ST\_Buffer. It also has an ST\_Covers and ST\_CoveresBy. The covers family of geography functions in PostGIS 1.5.1 and below only support polygon/point, point/polygon pairs. For geometry, ST\_Covers\*

## 4.5 Decomposition

You will find yourself often needing to extract parts of an existing geometry. You may need to find the closed linestring that encloses a polygon, or the multipoint that constitutes a linestring. We classify functions which perform the extraction and return one or more geometries as decomposition functions.

### 4.5.1 Boxes and envelopes

Boxes are the unsung heroes of geometries. Though rarely useful to model terrestrial features, they play an important role in spatial queries. Often, when comparing the relative spatial orientation of two or more geometries, the faster question to answer is where bounding boxes of the geometries are positioned relative to each other rather than the geometries themselves. By encasing disparate and complicated geometries in bounding boxes, we only need to work with rectangles and can ignore the details of the geometries within. Borrowing from an engineering concept, bounding boxes are the black boxes of spatial analysis.

By definition, a box, or Box2D, is the smallest 2-dimensional box that fully encloses the geometry. (PostGIS also has another kind of box called Box3D, but this is rarely used and does not serve the same purpose as Box2D.) All geometries have boxes, even points! Boxes are not geometries, but you can cast boxes into geometries. Naturally, casting a box to geometry will yield rectangular polygons, but you have to watch out for degenerate cases such as points, vertical lines or horizontal lines, or multipoints along a horizontal or vertical. The syntax for 2D box is as follows:

```
BOX(p1,p2)
```

where p1 and p2 are points of any two opposite vertices.

PostGIS functions that create bounding boxes is ST\_Box2D. Below are some examples of these in action and the corresponding output

#### **Listing 4.11 ST\_Box2D and casting a box to a geometry**

```
SELECT name, ST_Box2D(geom) As box, ST_AsEWKT(CAST(geom As geometry)) As  
box_casted_as_geometry  
FROM (
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

VALUES
('2D linestring', ST_GeomFromText('LINESTRING(1 2, 3 4)'),,
('Vertical linestring', ST_GeomFromText('LINESTRING(1 2, 1 4)'),,
('Point', ST_GeomFromText('POINT(1 2)'),,
('Polygon', ST_GeomFromText('POLYGON((1 2, 3 4, 5 6, 1 2))')))
AS foo(name, geom);

```

The results of the above query are shown below. Vertical line and point produce degenerate boxes and the geometry cast produces the same as the geometry itself.

**Table 4.9 Results of Listing 4.11**

<b>name</b>	<b>box</b>	<b>box_casted_as_geometry</b>
2D linestring	BOX(1 2, 3 4)	POLYGON((1 2, 1 4, 3 4, 3 2, 1 2))
Vertical linestring	BOX(1 2, 1 4)	LINESTRING(1 2, 1 4)
Point	BOX(1 2, 1 2)	POINT(1 2)
Polygon	BOX(1 2, 5 6)	POLYGON((1 2, 1 6, 5 6, 5 2, 1 2))

We mentioned that boxes are not geometries in their own right. If you actually need to obtain the geometry of the smallest rectangular box enclosing your geometry, use `ST_Envelope` function to return the envelope. In cases where the underlying geometry has no width (such as vertical linestrings), no height (such as horizontal linestring), or no width and no height (a point), `ST_Envelope` will simplify the geometry to either linestrings or points. We will revisit the previous example but this time we will include the `ST_Envelope` function.

#### **Listing 4.12 Example of ST\_Envelope**

```

SELECT name, ST_Box2D(geom) AS box, ST_AsEWKT(ST_Envelope(geom)) AS env
FROM (
VALUES
('2D linestring', ST_GeomFromText('LINESTRING(1 2, 3 4)'),,
('Vertical linestring', ST_GeomFromText('LINESTRING(1 2, 1 4)'),,
('Point', ST_GeomFromText('POINT(1 2)'),,
('Polygon', ST_GeomFromText('POLYGON((1 2, 3 4, 5 6, 1 2))')))
AS foo(name, geom);

```

Table 4.9 shows the output of the above query. Observe that for degenerate cases such as a vertical linestring and point, the envelope is the same as the input geometry.

**Table 4.10 Results of code in Listing 4.12**

<b>name</b>	<b>box</b>	<b>env</b>
2D linestring	BOX(1 2, 3 4)	POLYGON((1 2, 1 4, 3 4, 3 2, 1 2))
Vertical linestring	BOX(1 2, 1 6)	LINESTRING(1 2, 1 6)

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

Point	BOX(1 2, 1 2)	POINT(1 2)
Polygon	BOX(1 2, 5 6)	POLYGON((1 2, 1 6, 5 6, 5 2, 1 2))

### 4.5.2 Coordinates

ST\_X and ST\_Y is a pair of functions that you can use to return the underlying coordinates of points. They are generally combined with ST\_Centroid to get the x and y coordinates of a centroid for non-point geometries.

The ST\_Xm\* functions can be applied to all geometries and bounding boxes and are used to return the minimum/maximum x coordinate of each geometry.

They are rarely used alone, but are in general combined with each other to arrive at a pseudo width, height and so forth of a geometry. We will demonstrate their use when we talk about translation.

#### ST\_Xm are BOX3D functions

The ST\_Xm\* functions are really only defined for BOX3D objects, but since there is an autocast in place that converts a geometry to a box3d, you can use it directly on geometries, but you can not use it on the text representation of geometries as demonstrated in our discussion on autocasts.

### 4.5.3 Boundaries

ST\_Boundary works with all geometries and returns the geometry that determines the separation between the points in the geometry and the rest of the coordinate space. This particular way of defining boundary will make matters easy when we discuss interaction between two geometries in Chapter 5. Also note that the boundary of a geometry is at least one dimension lower than the geometry itself. One common use of ST\_Boundary is to break apart polygons and multipolygons into their constituent rings. ST\_Boundary ignores M and Z coordinates and currently does not work with geometry collections or curved geometries. Below are some examples of ST\_Boundary in action.

#### Listing 4.13 Example of ST\_Boundary

```
SELECT name, ST_AsText(ST_Boundary(geom)) As WKT
FROM (VALUES
('Simple linestring', ST_GeomFromText('LINESTRING(-14 21,0 0,35 26)'),),
('Non-simple linestring', ST_GeomFromText('LINESTRING(2 0,0 0,1 1,-1)'),),
('Closed linestring', ST_GeomFromText('LINESTRING(52 218, 139 82, 262 207,
245 261, 207 267, 153 207, 125 235, 90 270, 55 244, 51 219, 52 218)'),),
('Polygon', ST_GeomFromText('POLYGON((52 218, 139 82, 262 207, 245 261, 207
267, 153 207, 125 235, 90 270, 55 244, 51 219, 52 218))'),),
('Polygon with holes', ST_GeomFromText('POLYGON((-0.25 -1.25,-0.25
1.25,2.5 1.25,2.5 -1.25,-0.25 -1.25),(2.25 0,1.25 1,1.25 -1,2.25 0),(1 -1,1
1,0 0,1 -1))')) AS foo(name, geom);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Table 4.11: Output of Listing 4.13

<b>name</b>	<b>WKT</b>
Simple linestring	MULTIPOINT(-14 21,35 26)
Non-simple linestring	MULTIPOINT(2 0,1 -1)
Closed linestring	MULTIPOINT EMPTY
Polygon	LINESTRING(52 218,139 82,262 207,245 261,207 267,153 207,125 235,90 270,55 244,51 219,52 218)
Polygon with holes	MULTILINESTRING((-0.25 -1.25,-0.25 1.25,2.5 1.25,2.5 -1.25,-0.25 -1.25),(2.25 0,1.25 1,1.25 -1,2.25 0),(1 -1,1 1,0 0,1 -1))

Looking at the query and its output, we can surmise the following behavior of ST\_Boundary:

- An open linestring, both simple and non-simple, will return a multipoint made up of exactly two points, one for each of the end points.
- A closed linestring has no boundary points.
- A polygon without holes will return a linestring of the exterior ring
- A polygon with holes will return a multilinestring made up of closed linestrings for each of its rings. The first element of the multilinestring will always be the exterior ring.
- A multipolygon will always return a multilinestring. []

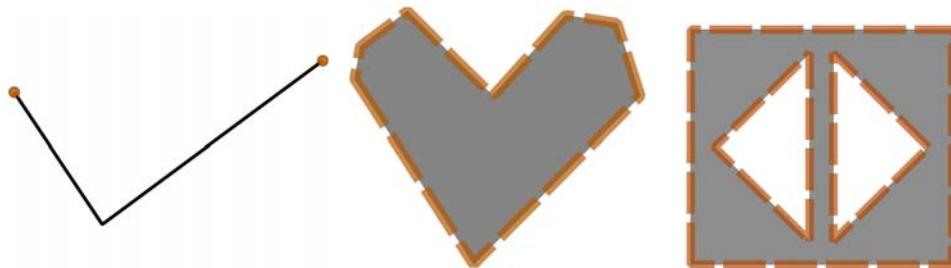


Figure 4.1 Simple Linestring, Polygon, and Polygon with holes overlaid with their boundaries from code in listing 4.13

A more specialized cousin of ST\_Boundary is ST\_ExteriorRing. This function only accepts polygons and returns the exterior ring. If you are trying to find the outer boundary of a polygon, ST\_ExteriorRing will perform faster than ST\_Boundary, but as its name suggest will not return the inner rings. You can use ST\_InteriorRingN to grab individual interior rings.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

#### **4.5.4 Point marker for a geometry: centroid, point on surface, and nth point**

We have all seen maps where geometries with finite areas are reduced down to a single point to unclutter the visual representation. Most maps use a star to indicate capitol cities rather than draw the city boundaries. Should you zoom in enough on any online map, say to the street level, you may find a labeled dot where you expect to see a huge polygon. Try this on a top-secret military installation. You zoom in enough and you will not see any of the details you expect, but just a dot telling you that it's a place the government does not want you to ever visit.

In PostGIS ST\_Centroid and to a lesser extent, ST\_PointOnSurface are often used in to provide a point marker for polygons. You should think of the centroid of a geometry as the center of gravity of a geometry if every point in the geometry had equal mass. The only caveat is that the centroid may not lie within the geometry itself—think donuts or bagels. The ST\_Centroid function works for all valid two dimensional geometries including geometry collections except for curved geometries. For 3D geometries, it will ignore the Z coordinate.

Using ST\_Centroid to return point markers sometimes produce undesirable visual results due to the fact that the point need not be on the geometry itself. Take the island nation of FSM (Federated States of Micronesia); its ST\_Centroid is most like somewhere in the Pacific Ocean. As a mapping service, you probably do not want people sailing to FSM to not end up on dry land. ST\_PointOnSurface comes to the rescue. The somewhat ill-named (since it also works for non-surface geometries like points and linestrings) function guarantees to return an arbitrary point on the geometry. In theory it can return any point in the geometry, but in practice it always returns the same point time and time again. ST\_PointOnSurface works for all geometries except curved geometries. For points, linestrings, multipoints, and multilinestrings it does consider the M and Z coordinates and returns a point that is usually one used to define the geometry. For polygons, it cuts out the M and Z coordinates.

#### **Listing 4.14 Centroid of various geometries**

```
SELECT name, ST_AsEWKT(ST_Centroid(geom)) As centroid,
ST_AsEWKT(ST_PointOnSurface(geom)) As point_on_surface
FROM (VALUES ('Multipoint', ST_GeomFromEWKT('MULTIPOINT(-1 1, 0 0, 2 3'))),
('Multipoint 3D', ST_GeomFromEWKT('MULTIPOINT(-1 1 1, 0 0 2, 2 3 1'))),
('Multilinestring', ST_GeomFromEWKT('MULTILINESTRING((0 0,0 1,1 1),(-1 1,-1 -1)))'),
('Polygon', ST_GeomFromEWKT('POLYGON((-0.25 -1.25,-0.25 1.25,2.5 1.25, 2.5 -1.25,-0.25 -1.25), (2.25 0,1.25 1,1.25 -1,2.25 0), (1 -1,1 1,0 0,1 -1)))))
As foo(name, geom);
```

The code above outputs both the centroid and point on surface of various geometries. What follows are the results of the above query.

**Table 4.12 Output of query in Listing 4.14**

<b>name</b>	<b>centroid</b>	<b>point_on_surface</b>
-------------	-----------------	-------------------------

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Multipoint	POINT(0.3333333333333333 1.333333333333333)	POINT(-1 1)
Multipoint 3D	POINT(0.3333333333333333 1.333333333333333)	POINT(-1 1 1)
Multilinestring	POINT(-0.375 0.375)	POINT(0 1)
Polygon	POINT(1.125 0)	POINT(-0.125 0)

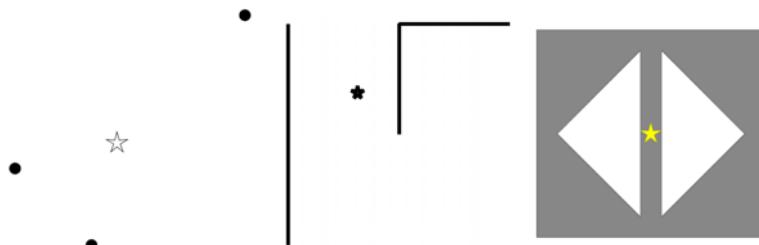


Figure 4.2 Geometries and centroids (denoted by star) generated from code in Listing 4.14. Observe that the centroid is not always a point on the geometry.

The below figure shows the original geometries in Listing 4.13 with the point on surface overlaid.

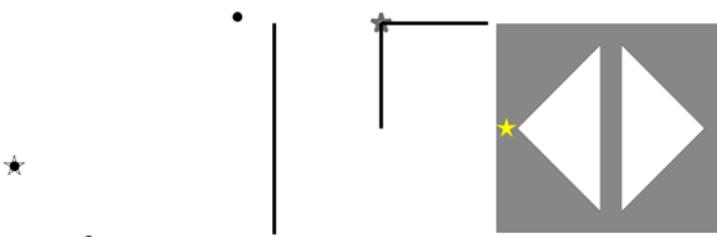


Figure 4.3 Geometries and stars representing point on surface generated from code in Listing 4.14

### ST\_Centroid and ST\_PointOnSurface in other spatial databases

ST\_Centroid and ST\_PointOnSurface are both OGC/MM spatial function but the specification only applied these functions to surfaces geometries, such as polygons and multipolygons. They can be conveniently extended to other geometry types as most databases do, but you have to watch for differences when porting between different databases. PostGIS extends these two functions to work with other geometries. IBM DB II extends ST\_Centroid to apply to other geometries but not ST\_PointOnSurface. SQL Server 2008 does the opposite and supports ST\_Centroid for surface geometries only and ST\_PointOnSurface for all geometries.

A convenient little function that works only with linestrings and circularstrings is ST\_PointN. It returns the nth point on the linestring with indexing starts at 1. Here is a quick example:

```
SELECT ST_AsText(ST_PointN(ST_GeomFromText('LINESTRING(1 2, 3 4, 5
8)'), 2));
returns
POINT(3 4)
```

#### **4.5.6 Breaking down multi and collection geometries**

Both ST\_GeometryN and ST\_Dump are useful for exploding multi and collection geometries into their component geometries. ST\_Dump and ST\_GeometryN do not quite return the same answer with the main difference being that ST\_Dump will recursively dump of multi and collection geometries whereas ST\_GeometryN will only drill down a single level.

Strictly speaking, ST\_Dump does not return a geometry outright but instead rows of geometry\_dump objects. Geometry\_dump object is a custom type installed with PostGIS and has two members. The first member of the dump object is the path. This member is a one-dimensional array indicating the depth from which the extracted geometry was found. The numbering scheme is quite intuitive. For example, if you have a geometry collection of multipolygons, {3, 2} would mean the third element of the collection, second polygon in the multipolygon. The second member of the geometry\_dump is the geom. This contains the exploded geometries themselves. ST\_Dump outputs the path member should you ever need to piece the exploded geometries back together. The other benefit of ST\_Dump is that as of 1.3.6, ST\_Dump can be used to explode Curved Geometries such as COMPOUNDCURVES where as ST\_GeometryN can only explode multicurves curved geometries and other standard MULTI types. Below is a demonstration of ST\_Dump.

#### **Listing 4.15 Example using ST\_Dump**

```
SELECT gid, (ST_Dump(geom)).path As exploded_path,
ST_AsEWKT((ST_Dump(geom)).geom) As exploded_geometry
FROM (VALUES (1, ST_GeomFromEWKT('MULTIPOLYGONM(((2.25 0 3,1.25 1 2,1.25 -1
3,2.25 0 1),((1 -1 1,1 2,0 0 1,1 -1 1))))'), (2,
ST_GeomFromEWKT('GEOMETRYCOLLECTION(MULTIPOLYGON(((2.25 0,1.25 1,1.25 -
1,2.25 0)),((1 -1,1 1,0 0,1 -1))), MULTIPOLYPOINT(1 2, 3 4), LINESTRING(5 6, 7
8), MULTICURVE(CIRCULARSTRING(1 2, 0 4, 2 8), (1 2, 5 6))))')
) As foo(gid, geom);
```

Table 4.13 Results of code in Listing 4.15

gid	exploded_path	exploded_geometry
1	{1}	POLYGONM((2.25 0 3,1.25 1 2,1.25 -1 3,2.25 0 1))
1	{2}	POLYGONM((1 -1 1,1 1 2,0 0 1,1 -1 1))
2	{1, 1}	POLYGON((2.25 0,1.25 1,1.25 -1,2.25 0))

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

2	{1, 2}	POLYGON((1 -1, 1 1, 0 0, 1 -1))
2	{2, 1}	POINT(1 2)
2	{2, 2}	POINT(3 4)
2	{3}	LINESTRING(5 6, 7 8)
2	{4, 1}	CIRCULARSTRING(1 2, 0 4, 2 8)
2	{4, 2}	LINESTRING(1 2, 5 6)

ST\_GeometryN extracts the nth geometry in a multi or collection geometry. It returns the single extracted geometry, does not recursively drill down, and does not report the depth. Use ST\_GeometryN when you have just one geometry to extract. If you find yourself needing to repeatedly call ST\_GeometryN to explode all constituent geometries, you should use ST\_Dump otherwise, you will suffer the performance penalty of having to call ST\_GeometryN over and over again. The listing below demonstrates use of ST\_GeometryN. We use the PostgreSQL generate\_series function combined with the ST\_NumGeometries function to extract all the geometries found in the first level of depth.

#### **Listing 4.16 Example using ST\_GeometryN with generate\_series**

```
SELECT gid, ST_AsEWKT(ST_GeometryN(geom,
generate_series(1,ST_NumGeometries(geom)))) AS extracted_geometry
FROM (VALUES (1, ST_GeomFromEWKT('MULTIPOLYGONM((2.25 0 3, 1.25 1 2, 1.25
-1 3, 2.25 0 1)), ((1 -1 1, 1 1 2, 0 0 1, 1 -1 1)))'), (2,
ST_GeomFromEWKT('GEOMETRYCOLLECTION(MULTIPOLYGON(((2.25 0, 1.25 1, 1.25 -1,
2.25 0)), ((1 -1, 1 1, 0 0, 1 -1)), MULTIPOINT(1 2, 3 4), LINESTRING(5 6,
7 8), MULTICURVE(CIRCULARSTRING(1 2, 0 4, 2 8), (1 2, 5 6))))')
) AS foo(gid, geom);
```

**Table 4.14 Results of code in Listing 4.16**

gid	extracted_geometry
1	POLYGONM((2.25 0 3, 1.25 1 2, 1.25 -1 3, 2.25 0 1))
1	POLYGONM((1 -1 1, 1 1 2, 0 0 1, 1 -1 1))
2	MULTIPOLYGON(((2.25 0, 1.25 1, 1.25 -1, 2.25 0)), ((1 -1, 1 1, 0 0, 1 -1)))
2	MULTIPOINT(1 2, 3 4)
2	LINESTRING(5 6, 7 8)
2	MULTICURVE(CIRCULARSTRING(1 2, 0 4, 2 8), (1 2, 5 6))

ST\_DumpRings is less commonly used function than ST\_Dump but invaluable when it comes to breaking up multi-ringed polygons into smaller polygons. Unlike ST\_ExteriorRing and ST\_InteriorRingN functions which return the exterior ring and nth ring of a polygon as linestrings, ST\_DumpRings converts them to single-ringed polygons. ST\_DumpRings is tremendously useful for polygons with lots of holes, especially if you need all the rings. The ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

alternative is to dump each ring using ST\_InteriorRingN and then ST\_BuildArea to form the polygon.

Since the output of the function could contain multiple rows, ST\_DumpRings returns geometry\_dump objects. Because a valid polygon can only have one exterior ring, the path array uses zero to denote the exterior ring and then starts numbering at one. In our example below, we will use ST\_DumpRings to extract the exterior ring and the first ring followed by an example of ST\_ExteriorRing and ST\_InteriorRingN to do the same.

#### **Listing 4.17 ST\_DumpRings to return a polygon exterior and first interior ring as cross tab**

```
SELECT MAX(CASE WHEN path[1] = 0 THEN ST_AsText(geom) ELSE NULL END) As exterior_ring_polygon, MAX(CASE WHEN path[1] = 1 THEN ST_AsText(geom) ELSE NULL END) As interior_ring1_polygon
FROM ST_DumpRings(ST_GeomFromText('POLYGON((-0.25 -1.25, -0.25 1.25, 2.5 1.25, 2.5 -1.25, -0.25 -1.25), (2.25 0, 1.25 1, 1.25 -1, 2.25 0), (1 -1, 1 0, 0 1, 1 -1))')) WHERE path[1] IN(0,1);
```

**Table 4.15 Results of query in Listing 4.17**

<b>exterior_ring_polygon</b>	<b>interior_ring1_polygon</b>
POLYGON((-0.25 -1.25, -0.25 1.25, 2.5 1.25...))	POLYGON((2.25 0, 1.25 1, 1.25 -1, 2.25 0))

We now perform the same extraction using ST\_ExteriorRing and ST\_InteriorRingN. Remember that these two function return the rings just as linestrings.

#### **Listing 4.18 Return polygon exterior and first interior ring as linestrings**

```
SELECT ST_AsText(ST_ExteriorRing(geom)) As exterior_ring,
ST_AsText(ST_InteriorRingN(geom,1)) As interior_ring1
FROM ST_GeomFromText('POLYGON((-0.25 -1.25,-0.25 1.25,2.5 1.25, 2.5 -1.25,-0.25 -1.25), (2.25 0,1.25 1,1.25 -1,2.25 0), (1 -1,1 0,0 1,1 -1))') As geom;
```

**Table 4.16 Result of query in Listing 4.18**

<b>exterior_ring</b>	<b>interior_ring1</b>
LINESTRING(-0.25 -1.25, -0.25 1.25, 2.5 1.25...)	LINESTRING(2.25 0, 1.25 1, 1.25 -1, 2.25 0)

Now that we know how to take geometries apart, you need to know how to put geometries together. We will move onto composition functions in the next section.

## **4.6 Composition**

We already covered how to create geometries from non-geometry data such as text and binaries. In this section, we will show you how to put together geometries from other geometries.

#### **4.6.1 Making points**

Since points are the most elemental geometries, creating points from another geometry is by definition impossible or at best superfluous. We will mention two functions that do however engender point geometry from the underlying coordinates—`ST_Point` and `ST_MakePoint`. Coordinates are not geometries, but we feel they are more related to geometries than text representations. Hence, we classify `ST_Point` and `ST_MakePoint` as composition functions.

`ST_Point` works only for 2D coordinates but is found in most spatial databases. `ST_MakePoint` and a variant, `ST_MakePointM`, can accept 2DM, 3D, and 4D coordinates in addition to 2D, but these two functions are PostGIS-specific. Syntax is the same for all three. The first argument is the coordinates separated by commas; the second is the SRID.

You may be wondering what these two additional functions offer beyond the common `ST_GeomFromText` besides a different import format. In short, it is speed and precision. Creating a handful of points or even a few dozen points may not matter much, but once you find yourself needing to load files containing millions of point data with many significant digits (common task when working with data collected via instrumentation), you will want to choose `ST_Point` or `ST_MakePoint` over `ST_GeomFromText`. To illustrate use of these two functions, we will simulate a few readings from tracking devices attached to grey whales as they make their annual migration from Baja California to the Bering Sea. Depending on the interval of reads and the number of whales we track, the number of data points coming into our database can be quite overwhelming, making speed a consideration in how we import the data.

#### **Listing 4.19 Point constructor functions**

```
SELECT whale, ST_AsEWKT(spot) As spot
FROM
  (VALUES
    ('Mr. Whale', ST_SetSRID(ST_Point(-100.499, 28.7015), 4326)),
    [1 Beg]
    ('Mr. Whale with M as time', ST_SetSRID(ST_MakePointM(-100.499, 28.7015,
      5), 4326)),
    [1 End]
    [2 Beg]
    ('Mr. Whale with Z as depth', ST_SetSRID(ST_MakePoint(-100.499, 28.7015,
      0.5), 4326)),
    [2 End]
    ('Mr. Whale with M and Z', ST_SetSRID(ST_MakePoint(-100.499, 28.7015, 0.5,
      4326)))
  ) As foo(whale, spot);
```

The code above demonstrates various overloads to the `ST_Point` and `ST_MakePoint` functions. In [1], we employ an extra unit M to store time as a serial. This serial can be based on say if we take readings every 5 hours then M=1 would mean the reading was taken 5 hours from the start time, M=2, 10 hours, and so on. If you are keeping data as individual points, this is not terribly useful, but if you later decide to stitch them together into a LINESTRINGM then you can have the time slots encoded in the line and have just one record

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

for each whale instead of needing to create a separate array of for time. [2] We may be interested in knowing how far Mr. Whale dived before coming to surface for air. We appropriately use the Z coordinate to store the depth. SRID 4326 is unprojected data, so the Z coordinate is unit-less. You can choose your own units to record the depth: feet, meters, fathoms, etc.

**Table 4.17 Output from query in Listing 4.19**

<b>whale</b>	<b>spot</b>
Mr. Whale	SRID=4326; POINT(-100.499 28.7015)
Mr. Whale with M as time	SRID=4326; POINTM(-100.499 28.7015 5)
Mr. Whale with Z as depth	SRID=4326; POINT(-100.499 28.7015 0.5)
Mr. Whale with M and Z	SRID=4326; POINT(-100.499 28.7015 0.5 5)

### **4.6.2 Making polygons**

`ST_MakePolygon`, `ST_BuildArea`, and `ST_Polygonize` all build polygons but vary slightly varying differences in purpose.

#### **ST\_MAKEPOLYGON**

`ST_MakePolygon` builds a polygon from a closed linestring representing the exterior ring. Optionally, it can accept a second argument of an array of closed linestrings to build interior rings. `ST_MakePolygon` does not validate the input linestrings whatsoever. This means that if you are not careful, and pass in open linestrings or linestrings that cannot form polygons, you could end up with an error or fairly goofy polygon, such as polygons where the holes lie outside the exterior ring or the interior rings are not completely contained by the exterior ring. The complete absence of validation does provide an advantage in speed. `ST_MakePolygon` runs much quicker than other functions for creating polygons and is the only one that will not ignore Z and M coordinates. `ST_MakePolygon` does require that you give it closed linestrings only, no multilinestrings, no collections of linestrings.

#### **ST\_BUILDAREA**

You can think of `ST_BuildArea` as the neater roommate of `ST_MakePolygon`. Unlike its more reckless counterpart, you can toss it whatever you like and it will take the time to organize what you have into valid polygons.

`ST_BuildArea` will accept linestrings, multilinestrings, polygons, multipolygons, and geometrycollections. You do not have to worry about the order or the validity of the geometries that you are feeding `ST_BuildArea`. It will take the time to check for validity of each input geometry, determine which geometry should be interior rings and which should be exterior rings and reshuffle them to output polygons or multipolygons. `ST_BuildArea` will not work with arrays. But this shortcoming is mitigated by the fact that it will accept multilinestrings and geometrycollection geometries. If you intend on feeding the function an

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

assortment of linestrings and polygons, perform a ST\_Collect first to gather all the loose pieces into a single geometry.

All this neatness comes at a price of course—you sacrifice performance. If you have already sanitized your input geometries using another procedure and speed is of utmost importance, use ST\_MakePolygon. If your input geometry came from suspect sources and you just want to see what area comes out, the sanitizing feature of ST\_BuildArea will be worth the wait.

### **ST\_Polygonize**

ST\_Polygonize is a database aggregate function. As a database aggregate, its use only makes sense when used against existing table with geometry columns. This function takes rows of linestrings and returns a geometry collection consisting of the possible polygons you can form from such linestrings. It is often used when trying to formulate polygons from edge linestrings and then passed to ST\_Dump to dump out the individual polygons as separate rows.

We demonstrate the use of all three polygon-making functions in the listing below. You will quickly notice that we are using the common table expressions (CTE) feature introduced in PostgreSQL 8.4. If you do not have at least 8.4—upgrade! If you cannot upgrade, you will have to repeat the expression or use temporary tables. CTE is an ANSI SQL feature now found in most SQL databases.

#### **Listing 4.20 ST\_Polygonize , ST\_BuildArea, ST\_MakePolygon**

```
--[1 Beg]
WITH example(geom) AS
  (VALUES(ST_GeomFromText('LINESTRING(1 2, 3 4, 4 4, 1 2)'), 
  (ST_GeomFromEWKT('MULTILINESTRING((0 0, 4 4, 4 0, 0 0), (2 1, 3 1, 3 2, 2
1))))))
--[1 End]
--[2 Beg]
SELECT 'ST_MakePolygon (1)' As function, ST_AsEWKT(ST_MakePolygon(geom)) As
polygon
FROM example
WHERE ST_GeometryType(geom) = 'ST_LineString'
--[2 End]
UNION ALL
--[3 Beg]
SELECT 'ST_MakePolygon (2)' As function,
ST_AsEWKT(ST_MakePolygon(ST_GeometryN(geom, 1), ARRAY[ (SELECT
ST_GeometryN(geom, n) FROM generate_series(2, ST_NumGeometries(geom)) As n
)])) As polygon
FROM example
WHERE ST_GeometryType(geom) = 'ST_MultiLineString'
--[3 End]
UNION ALL
SELECT 'ST_BuildArea' As function, ST_AsEWKT(ST_BuildArea(geom)) As polygon
FROM example
UNION ALL
SELECT 'ST_Polygonize' As function, ST_AsEWKT(ST_Polygonize(geom)) As
polygon
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```
FROM example;
```

```
[1] CTE
```

ST\_MakePolygon has two variants. [2] The simpler version just takes an outer ring and forms a polygon without holes. In the second version [3], we are using the ST\_MakePolygon(outer ring, array of inner rings) to form a polygon with holes. We are also using two SQL constructs somewhat unique to PostgreSQL. The first is the generate\_series function which generates a number between start and end (for this trivial example it will generate a set of numbers between 2 and 2 since there are only 2 linestrings in our multilinestring example and the first is reserved for the exterior ring. We are then using this to extract out the second linestring. (If there were more linestrings or more multilinestrings then the generate series could be 2 to 3 or 2 to 4 etc.) We then use the array[...] constructor in PostgreSQL that can take a list of element or an SQL statement to populate the array (in our case we are using the SQL). ST\_MakePolygon can now accept our array of linestrings as the second argument and use it to form the interior rings of our polygon.

**Table 4.18 Results of query in Listing Code 4.20**

function	polygon
ST_MakePolygon (1)	POLYGON((1 2, 3 4, 4 4, 1 2))
ST_MakePolygon (2)	POLYGON((0 0, 4 4, 4 0, 0 0),(2 1, 3 1, 3 2, 2 1))
ST_BuildArea	POLYGON((1 2, 3 4, 4 4, 1 2))
ST_BuildArea	POLYGON((0 0, 4 4, 4 0, 0 0),(2 1, 3 1, 3 2, 2 1))
ST_Polygonize	GEOMETRYCOLLECTION(POLYGON((1 2, 3 4, 4 4, 1 2)), POLYGON((0 0, 4 4, 4 0, 0 0), (2 1, 3 1, 3 2, 2 1)), POLYGON((2 1, 3 2, 3 1, 2 1)))

In the above listing, ST\_MakePolygon and ST\_BuildArea return the same answers when a linestring and multilinestrings form well formed geometries, however for ST\_MakePolygon; we had to break our selects to separate the linestrings from multilinestrings. ST\_Polygonize is an aggregate function; it takes rows of geometries and returns one geometry collection. It is incapable of creating polygons with holes so every ring in the multilinestring becomes a polygon in their own right.

### 4.6.3 Promoting single to multi geometries

The ST\_Multi function is one that is used quite often in PostGIS mostly to promote points, linestrings, and polygons to their multi counterparts even if they have only one single geometry. If a geometry is already a multi variety, then it remains unchanged. Its main use case is to ensure that all geometries in a table column are of the same geometry type for consistency. For instance, suppose you obtained polygons for all nations. The Kingdom of Lesotho could come in as a single polygon since it is a tiny landlocked enclave whereas

Indonesia will come in a multipolygon. To keep your column consistent, you would promote Lesotho to a multipolygon.

## **4.7 Simplification**

For this section we will cover the three functions: ST\_SnapToGrid, ST\_Simplify, and ST\_SimplifyPreserveTopology. These three functions behave quite differently from one another, but they all try to achieve the same goal—reducing the bytes necessary to describe a geometry. Simplification functions tend to take center stage when passing geometries across the internet. Despite recent advances, bandwidth is still a precious commodity especially with the recent proliferation of wireless devices. If someone is sporting a tiny, black and white, 200x300 resolution GPS screen, what a waste it would be to send over detailed geometries with thousands of vertices or coordinates with monstrous number of significant digits.

### **4.7.1 Coordinate rounding using ST\_SnapToGrid**

ST\_SnapToGrid reduces the weight of a geometry by rounding the coordinates. If after rounding, two or more adjacent coordinates become indistinguishable, it will automatically keep only one of them, thus reducing vertices.

There four variants of this function. The most common one takes one argument for tolerance and rounds just the x and y coordinate while leaving z and m intact. ST\_SnapToGrid does not remove z and m coordinates. Other variants can round all four coordinates or allow you to specify offsets to indicate where the grid starts.

One common use of ST\_SnapToGrid is to trim the extra floating point decimals introduced by ST\_Transform. Holding onto the extra digits can degrade performance and is a general nuisance if the precision is not needed. Another use of ST\_SnapToGrid is to group distinct points close to one another into a single representational point. For example, if you obtained point data for every school in the country, but only care about the location of school districts. Collapsing all the schools down to a single point would be the way to go, especially if you are going to look at the data on a national scale.

As with most simplification operations, you should exercise restraint. If you are too ambitious with your rounding, you can inadvertently round a valid polygon to an invalid one.

#### **Listing 4.21 Example of ST\_SnapToGrid coordinate rounding**

```
SELECT pow(10, -1*n)*5 As tolerance,
ST_AseWKT(ST_SnapToGrid(ST_GeomFromEWKT('SRID=4326;LINESTRING(-73.81309
41.74874, -73.81276 41.74893, -73.812765 41.74895, -73.81307 41.74896)'), 
pow(10, -1*n)*5)) As simplified_geometry
FROM generate_series(3,6) As n
ORDER BY tolerance;
```

Table 4.19 Results of query in Listing 4.21

tolerance	simplified_geometry
5	LINESTRING(-73.8130941.74874, -73.8127641.74893, -73.81276541.74895, -73.8130741.74896)
10	LINESTRING(-73.8130941.74874, -73.8127641.74893, -73.81276541.74895, -73.8130741.74896)
15	LINESTRING(-73.8130941.74874, -73.8127641.74893, -73.81276541.74895, -73.8130741.74896)
20	LINESTRING(-73.8130941.74874, -73.8127641.74893, -73.81276541.74895, -73.8130741.74896)
25	LINESTRING(-73.8130941.74874, -73.8127641.74893, -73.81276541.74895, -73.8130741.74896)

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```
0.000005 SRID=4326; LINESTRING(-73.81309 41.74874, -73.81276 41.74893, -73.812765 41.74895, -73.81307 41.74896)
```

```
0.00005 SRID=4326; LINESTRING(-73.8131 41.74875, -73.81275 41.74895, -73.81305 41.74895)
```

```
0.0005 SRID=4326; LINESTRING(-73.813 41.7485, -73.813 41.749)
```

```
0.005 NULL
```

In this example we generate a number between 3 and 6 and then use that to round the coordinates or our linestring. Notice that when we reach rounding tolerance of 0.005 our linestring disappears. This is because ST\_SnapToGrid will always return the same output geometry type as the input, but if you round to .005, the input geometry has collapsed into a single point and is no longer a linestring.

### 4.7.2 Simplifying geometries

ST\_Simplify and ST\_SimplifyPreserveTopology both reduce the weight of a geometry by reducing the number of vertices used to describe the geometry using some variant of Douglas-Peucker algorithm. ST\_SimplifyPreserveTopology function is newer than ST\_Simplify and has safeguards against oversimplification. In extreme cases of oversimplification, your geometry could very well vanish into thin air or become invalid. As such ST\_SimplifyPreserveTopology is generally preferred over the older ST\_Simplify even though it is a tad slower.

Both ST\_Simplify and ST\_SimplifyPreserveTopology take a second argument which we will term tolerance. This can be roughly treated as the unit of length between the vertices at which you would want to collapse the vertices into one. For example, if you set the argument to 100, the two functions will try to collapse any vertices spaced 100 units apart. As you increase the tolerance, you will experience more simplification. Putting it another way, the more tolerant you are of losing vertices, the more simplification you can achieve.

These two simplify functions, unlike ST\_SnapToGrid, do not preserve m and z coordinates and will get rid of them if present. They also only work for linestrings, multilinestrings, polygons, multipolygons and geometry collections containing these. For other geometries, namely multipoints, it will return the same input geometry without any simplification. The reason for this is because ST\_Simplify and ST\_SimplifyPreserveTopology require edges (lines between vertices) to achieve simplification. Multipoints do not have edges.

#### **DO NOT CALL ST\_SIMPLIFY FUNCTIONS WITH LON LAT DATA**

ST\_Simplify and ST\_SimplifyPreserveTopology assume planar space. Should you apply these functions against with lon-lat data (SRID: 4326) the resultant geometry can range from being slightly askew to completely goofy. First transform your lon-lat to a planar coordinate, apply the ST\_Simplify, and then transform back to lon-lat.

#### **Listing 4.22 Comparing ST\_Simplify with ST\_SimplifyPreserveTopology**

```
SELECT pow(2, n) as tolerance, ST_AsText(ST_Simplify(geom, pow(2, n))) As  
©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
http://www.manning-sandbox.com/forum.jspa?forumID=565
```

```
ST_Simplify, ST_AsText(ST_SimplifyPreserveTopology(geom, pow(2, n))) As
ST_SimplifyPreserveTopology
FROM (SELECT ST_GeomFromText('POLYGON((10 0, 20 0, 30 10, 30 20, 20 30, 10
30, 0 20, 0 10, 10 0)') As geom
) As foo CROSS JOIN generate_series(2,4) As n;
```

Table 4.20 results of query in Listing 4.22 (split into two sections for readability)

**toleranceST\_Simplify**

---

4	POLYGON((10 0, 20 0, 30 10, 30 20, 20 30, 10 30, 0 20, 0 10, 10 0))
8	POLYGON((10 0, 30 10, 20 30, 0 20, 10 0))
16	NULL

**toleranceST\_SimplifyPreserveTopology**

---

4	POLYGON((10 0, 20 0, 30 10, 30 20, 20 30, 10 30, 0 20, 0 10, 10 0))
8	POLYGON((10 0, 30 10, 20 30, 0 20, 10 0))
16	POLYGON((10 0, 30 10, 20 30, 0 20, 10 0))

In looking at the output of the query, notice that with ST\_Simplify once you reach a tolerance of 16, your geometry vanishes. However, ST\_SimplifyPreserveTopology reduces the 8-sided polygon to a 4-sided polygon and stops there regardless of how high you raise the tolerance.

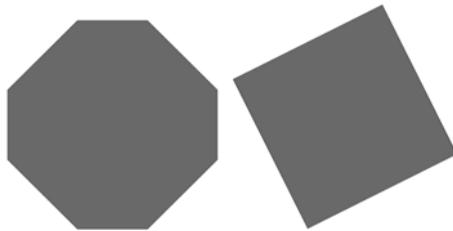


Figure 4.4 ST\_Simplify and ST\_SimplifyPreserveTopology, going from an 8-sided polygon to 4-sided

## 4.8 Summary

In this chapter we begin covering the most commonly used functions in PostGIS. For now, we concentrated on functions that take a single geometry as arguments. To organize the myriad of unary functions in PostGIS, we developed a loose classification scheme based on purpose. Starting with constructors, we then moved onto getters and setters, followed by decomposition and composition functions. We ended the chapter with simplification functions. The popular functions we touched on only constitute a small subset of the entirety of unary functions available in PostGIS. We highly recommend that you peruse the official

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

PostGIS documentation to see all that is available. You may find the number of functions overwhelming at first, but upon closer examination, many functions are closely related and fit nicely into our taxonomy. We also advise that you refer to the documentation prior to using any of the functions we described, especially if you are running older versions of PostGIS or intend to overload the functions.

In the next chapter we will continue our exploration of PostGIS functions by covering functions that take two or more geometries as input, binary functions. You will find binary functions to be far more useful, and perhaps more interesting, in answering questions regarding your data, but do not sidetrack the unaries. In almost all queries, you cannot use binary functions outright without applying some type of unary function to prepare your geometry.

# 5

## *Relationships between geometries*

This chapter covers

- Intersections and differences
- Intersect relationship types
- Equality
- Nearest Neighbor
- Arbitrary relationships
- Dimensionally Extended 9 Intersection Model (DE9IM)

As the old saying goes “No man is an island,”; the same holds true for geometries. In the previous chapter we concentrated on describing geometries in isolation. We described common properties used to describe geometries and various functions that can measure, morph, or transform single geometries. From this chapter forward, we will no longer entertain ourselves with one geometry at a time. The richness and the power of spatial queries really come to light when we start working with more than a single geometry. Those readers coming from a database background know this quite well. If we liken geometries to tables, an SQL statement that queries from a single table can only go so far. It is only when more than one table get involved; when we have join operations at our disposal that things become interesting. Mastery of join operations is what separates the casual database user from the serious database analyst.

Spatial databases have a similar jumping-off point; the casual consumer of a spatial database may use PostGIS to store geometry data or to filter geometries befitting certain conditions. The serious spatial database analyst will be able to write queries that join and morph multiple geometries to solve seemingly intractable problems with brisk elegance.

Although spatial databases are thought of as a tool for geographic information systems, the problems they can tackle are not limited to geographic ones. In this chapter we will

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

explore the fundamental underpinnings of spatial databases. Spatial databases are all about space, how objects occupy space, and interact with other objects in space. Any problem you can state using the physical or abstract concept of vector space is one that is a potential use case for a spatial database. For these exercises we will be using the unknown spatial reference system. Later on we will delve into spatial reference systems when loading geographic data and the special considerations involved with dealing with geographic data.

As the old saying goes "No pain, no gain," working with more than one geometry introduces a new level of conceptual challenges. In non-spatial databases, disparate data interact through various mathematical or string operations. When one number meets another number, you can add, subtract, multiply, divide or some combination thereof. When one string meets another, you can concatenate, "substring" one against the other. In spatial databases, when one geometry meets another, things heat up quite a bit. In PostGIS, there are many ways in which the relationship can be consummated. This chapter explores the most commonly used of these relationships. We will describe each relationship separately as much as possible in this chapter so that you can gain a solid understanding of what each means. Keep in mind that the full analytical power of spatial SQL usually entail multiple relationship functions, operators, and processing functions being applied in unison.

## **5.1 Introducing spatial relationship functions**

Spatial relationship functions in PostGIS accept two input geometries and return either true or false, or another geometry. As the name implies, relationship functions describe how the two input geometries relate to each other spatially. For example, if we want to see if one geometry encloses another, we could use ST\_Contains. If we want to see if two geometries rub up against each other, we can use ST\_Touches.

### **Functionally Speaking**

To avoid muddling the meaning while speaking the functions out loud, we suggest that instead of literally saying "ST\_SomeRelationship(A,B)" to simply say "A SomeRelationship B". For example,

`ST_Contains(geom1,geom2)`

would be read as: geom1 contains geom2

Not all relationship functions are commutative. Reversing the order of geometries in non-commutative relationships is a fairly common mistake. For instance if you want to know if A contains B, reversing the input arguments will give you the B contains A answer except possibly in the case of invalid geometries.

When using spatial relationship functions, the two geometries being compared must both be in the same spatial reference system or both be in the unknown spatial reference system. If they are not, the function may return an error. Keep in mind that all spatial relationship functions for geometry data type presuppose a planar projection (Cartesian coordinates), so ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

when using against long lat data (spherical coordinates), be very careful and use the geography type instead if you are comparing large areas. The geography data type will model the spatial relationships using a true geodetic model, but is not as rich in the number of spatial relationship functions you can use with it. For small degree differences, the Cartesian spatial relationship assumptions will generally work fine, but as the degree differences increase or you approach the poles, the assumption of longlat being flat is no longer correct and you may end up with incorrect results.

### **Relationship and output functions support for curved and 3D geometries**

While you can create geometries with X,Y,Z,M and curved geometries in PostGIS, as far as relationships go (those that return true/false), PostGIS currently ignores the third dimension and forth dimensions and the GEOS relationship functions will simply reject curved geometries.

However the geometry relationship output functions – things like ST\_Intersection, ST\_Difference, ST\_SymDifference – don't completely ignore the z-coordinate, but they apply the z-coordinate after doing a 2-D relationship process. The results are sometimes less than desireable.

As a work around for the lack of support for curves you can approximate a curve with a non-curve by converting to non-curve and then applying the relationship ST\_Intersects(ST\_CurveToLine(a.the\_geom,100), ST\_CurveToLine(a.the\_geom, 100)), where 100 is the number of segments to approximate a quarter circle; default if left out is 32. The 3D issue is harder to compensate for.

## **5.2 Intersections**

We start with intersections, as this is by far the most commonly used relationship between two geometries. The idea of intersection encompass a wide range of ways in which geometries can interact. We will delve into the nuances in time, but let us start with the basic definition of intersection: Two geometries intersect when they share space together.

PostGIS has two functions that work with intersections. The first is ST\_Intersects. ST\_Intersects is a function that takes two geometries and returns true if any part of those geometries is shared between the two. The other function is ST\_Intersection. This function returns a geometry that represents the shared part of the two input geometries. If the geometries do not intersect, then the intersection is an empty geometry. Both functions are defined in the OGC/SQL-MM specs so can be found in most databases that follow the ISO SQL-MM model.

### **WHAT IS AN EMPTY GEOMETRY?**

An empty geometry is a geometry with no points, but it is not null. You can create an empty geometry by doing this `ST_GeomFromText('GEOMETRYCOLLECTION EMPTY');`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Although you can have an EMPTY POLYGON, PostGIS silently converts it to an empty geometry collection. This may change in versions after PostGIS 1.4

We will demonstrate the two functions in action with some examples.

### 5.2.1 Segmenting linestrings with polygons

In Listing 5.1, we start with a polygon and a line string and see if they intersect and what the resultant geometry looks like. This example is quite common in real-world scenarios. The linestring can represent the planned route for a new roadway. The polygon can represent private property. Our query quickly tells us if the new road will cut through the private property. If so, we can determine which part of the road falls within the boundaries to determine the cost associated with an eminent domain takeover. Although we only show a simple example, you can imagine how useful this can be if we have all the private properties in a city and determine which properties the road will cut through. The route planner can virtually trace any path through the city and obtain an immediate calculation for the eminent domain purchase.

Below is a figure showing a planned roadway (linestring) and our land mass (polygon) and the resulting intersection geometry - that portion we need to take over by eminent domain.

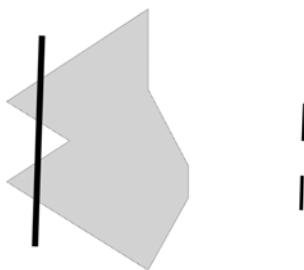


Figure 5.1 The first image is the POLYGON (our property) overlaid with the LINESTRING (our planned road) and the second is the intersection of the two. This results in a MULTILINESTRING which represents that portion of the property we need to take over.

The code to generate the above is shown in Listing 5.1

#### **Listing 5.1 Linestring segmented by a polygon**

```
SELECT ST_Intersects(g1.geom1,g1.geom2) As they_intersect,  
GeometryType(ST_Intersection(g1.geom1, g1.geom2)) As intersect_geom_type  
FROM (SELECT ST_GeomFromText('POLYGON((2 4.5,3 2.6,3 1.8,2 0,-1.5 2.2,0.056  
3.222,-1.5 4.2,2 6.5,2 4.5))') As geom1,  
ST_GeomFromText('LINESTRING(-0.62 5.84,-0.8 0.59)') As geom2) AS g1;
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

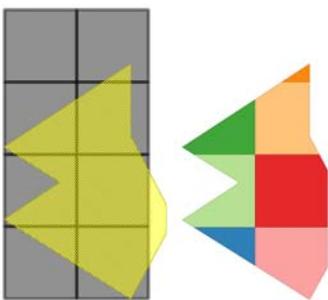
In the above listing we end up with a MULTILINESTRING because the line is cut by the polygon.

**Table 5.1 Result of query in Listing 5.1**

they_intersect	intersect_geom_type
t	MULTILINESTRING

### **5.2.2 Clipping polygons with polygons**

Should you be unimpressed by the previous example, the next one ought to change your mind. One of the most common uses of the ST\_Intersection function is for clipping polygons. Clipping loosely refers to the process of breaking up a geometry into smaller segments or regions. For instance, if you were in charge of sales for a city and have a dozen sales representatives on your staff. You could clip the polygon of the city into 12 sales regions so each representative can have his or her own. Another common use is to make your spatial database queries faster by breaking up your geometries beforehand. If you have data covering more area than you generally need to work with. It behooves you to clip the original geometry so that you query against a smaller geometry. For example, if you are working with data covering the entire island of Hispaniola, but only need to report on Haiti, you could clip the island using a linestring so you only query against the data covering the Haitian half of the island. We start with an example where we break up an arbitrarily-shaped polygon (the one we used in Listing 5.1) into square regions.



**Figure 5.2 Result of code in Listing 5.2 the first shows a region overlaid against square tiles. The second shows the result of intersection of square tiles with the region**

In order to perform the above trick we take a rectangle, break it into 8 equal size cells, and then intersect with a polygon.

#### **Listing 5.2 Return our sales region diced up**

```
-- (1)
SELECT x || ' ' || y As grid_x_y,
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

CAST(ST_MakeBox2d(
    ST_Point(-1.5 + x, 0 + y),
    ST_Point(-1.5 + x + 2, 0 + y + 2)) As geometry) As geom2
FROM generate_series(0,3,2) As x
CROSS JOIN generate_series(0,6,2) As y;

-- (2)
SELECT ST_GeomFromText('POLYGON((2 4.5,3 2.6,3 1.8,2 0,-1.5 2.2,0.056
3.222,-1.5 4.2,2 6.5,2 4.5))') As geom1

-- (3)
SELECT CAST(x AS text) || ' ' || CAST(y AS text) As grid_xy ,
ST_AsText(ST_Intersection(g1.geom1, g2.geom2)) As intersect_geom
FROM (SELECT ST_GeomFromText('POLYGON((2 4.5,3 2.6,3 1.8,2 0,-1.5 2.2,0.056
3.222,-1.5 4.2,2 6.5,2 4.5))') As geom1) As g1
INNER JOIN (SELECT x, y, CAST(ST_MakeBox2d(ST_Point(-1.5 + x, 0 + y),
ST_Point(-1.5 + x + 2, 0 + y + 2)) As geometry) As geom2
FROM generate_series(0,3,2) As x CROSS JOIN generate_series(0,6,2) As y) As g2
ON ST_Intersects(g1.geom1,g2.geom2) ;

```

- [1] The squares to dice with
- [2] The region
- [3] The result of dicing – note you get multiple records one for each polygon

We [1] use the PostgreSQL generate\_series to generate from min/max x and min to max y and skip every two steps so each square is 2 units wide and high. [2] Represents the region we want to cut. [3] We combine the two and take the intersection which results in our geometry diced.

The well-known text representation of each slice is shown in Table 5.2.

**Table 5.2 The result of last query in Listing 5.2**

grid\_xy intersect\_geom

---

0 0	POLYGON((0.5 0.942857142857143 .....,))
2 0	POLYGON((2.5 0.9,2 0,0.5 0.942857142857143,0.5 2,2.5 2,2.5 0.9))
0 2	POLYGON((-1.18181818181818 2,-1.5 2.2,...))
2 2	POLYGON((2.26315789473684 4,2.5 3.55,...))
0 4	POLYGON((-1.18179959100204 4,-1.5 4.2,0.5 5....))
2 4	POLYGON((2 4.5,2.26315789473684 4,0.5 4,0.5 5.51428571428571...))
2 6	POLYGON((1.23913043478261 6,2 6.5,2 6,1.23913043478261 6))

The above example shows how intersections can be useful for partitioning a single geometry into separate records. Notice that your cutting squares do not need to completely

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

cover the polygon. In the above we left out the last sliver by making our grid not completely cover the extent of the region.

Another important thing to keep in mind is that the geometry type returned by ST\_Intersection may look rather different than the input geometries, but it is guaranteed to be of equal or lower dimension of the lowest dimension geometry. For example if you have 2 polygons that share an edge, then the intersection of the 2 will be the linestring representing the shared edge. Similarly if you are intersecting a road with a parcel of land, then the intersection would be possibly a linestring that represents that portion of the road that runs thru the parcel of land.

In summary, if A and B are the input geometries to ST\_Intersection:

1. ST\_Intersection returns that portion shared by A and B
2. ST\_Intersection and ST\_Intersects are both commutative – meaning  
 $ST\_Intersection(A,B) = ST\_Intersection(B,A)$  and  $ST\_Intersects(A,B) = ST\_Intersects(B,A)$
3. A and B need not be of the same geometry type
4. The geometry returned by ST\_Intersection is of equal or lower dimension of the lowest dimension of A and B geometry.
5. If the A and B do not intersect, then the intersection is an Empty geometry

Now that we have covered the basic concept of intersects and intersection, we'll delve into the finer details of intersecting relationships.

## **5.3 Specific intersection relationships**

Recall that the definition of intersection involves two geometries sharing space. Sometimes, you may want to have more detail about how the space is shared and have to say something about the space not being shared. For these situations, we have at our disposal many PostGIS functions that focus on the subtleties of the intersection. Again these functions rely on the fact that the spatial reference system (SRID) of both geometries are the same, though the geometric dimension of them need not be the same. The geometries should also be valid otherwise the results can not be trusted.

### **5.3.1 Interior, exterior, and boundary of a geometry**

Most of the intersection relationship functions rely on the concepts of Interior, Exterior, and Boundary of a geometry and whether these intersect with Interior, Exterior, and Boundary of the second geometry. In the case of intersection of these 3 parts, the geometric dimension of the resulting geometry is also important. Will the intersection result in a geometry of 0, 1, or 2 dimensions?

We shall cover these concepts in more detail in a later section of this chapter. For brevity – these are what the terms mean

- Interior – that portion of a geometry that is inside the geometry and not on

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

the boundary.

- Exterior – The coordinate space outside of a geometry but not including the boundary
- Boundary – The coordinate space neither interior nor exterior of the geometry – it is the space that separates the interior from the exterior (rest of coordinate space). Recall we covered in the prior section ST\_Boundary which tells you what the boundary geometry is of a geometry.

### **How do you represent interior and exterior?**

The boundary of a geometry is another geometry. In the case of finite points the boundary is an empty geometry. You can get the boundary of a geometry with the ST\_Boundary function and this resultant geometry also has an interior, exterior and boundary. The model of an interior and exterior is a bit harder to fathom without introducing the concept of limit theorems. You can't adequately represent it with a geometry construct except in the case of interior for points and multipoints. The interior of points is simply the points and the exterior is the rest of coordinate space that is not the points. In the case of lines and polygons the interior and exterior are limits approaching the boundary of the geometry and therefore not representable by themselves. So in short, the model of a geometry is a mathematical trick. In the case of linestrings and polygons we can't quantify what an interior is or an exterior is, but we can say there exists an object called "a geometry" composed of an infinite number of points that has an interior, an exterior and a boundary where the interior and exterior approach the boundary.

The result of an intersection of these 9 pairs can be non-dimensional (no intersection), 0 dimension (finite points), 1 dimensional (lineal), 2 dimensional (areal polygons) or a combination thereof in which the dimension is the dimension of the highest dimension element in the collection.

These intersects classes of functions should only be used with valid geometries. The reason for that is that when there are self-intersections at the boundaries the concepts of interior, exterior, and boundary are not well-defined. This is a common mistake people make.

The official PostGIS manual has diagrams of these relationships. We will try to focus on the corner cases where people have a hard time comprehending the relationships.

We shall use the following example for explanation of specific intersection relationships because it is a simple corner case that exercises the subtleties of all these relationships.

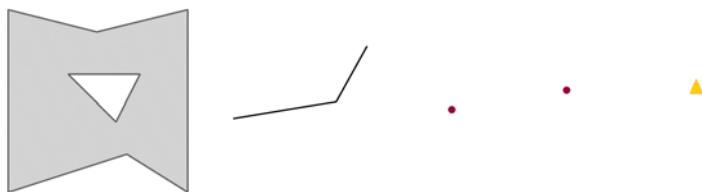


Figure 5.3 A shaded polygon with hole (a house with a courtyard), a linestring (a walkway), a multipoint represented as 2 dots (two greeters – front door and courtyard greeter), a point as a triangle (a door) generated from code in listing 5.3

When the above geometries are viewed together on the same grid, the overlay looks like the below figure.

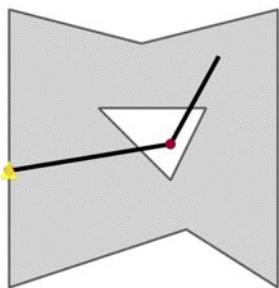


Figure 5.4 The polygon with hole (a house with a courtyard), the linestring (a walkway), the multipoint (two greeters) and the point as a triangle (a door) seen together. These are generated from code in listing 5.3

In the above example, we have 4 geometries which we can envision as any set of objects

- A polygon with a triangular hole – This is our house with a courtyard. The courtyard we represent as a hole because we don't consider it as part of the house. The courtyard is thus geometrically a part of the house's exterior.
- A line string – this can be a red carpet for visitors to walk into the house.
- A point represented by a triangle icon - This is the front door to our house
- A multipoint represented by two circles – This can be two greeters who are stationed at the front door to greet incoming visitors and at the courtyard to sit down guests.
- At one moment in time we see all these objects at these particular positions.

We create this particular dataset with the following piece of code:

### **Listing 5.3 Create sample geometries to exercise intersect relationships**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

CREATE TABLE example_set(ex_name varchar(150) PRIMARY KEY,
    the_geom geometry);
INSERT INTO example_set(ex_name, the_geom)
VALUES
('A polygon with hole', ST_GeomFromText('POLYGON ((110 180, 110 335, 184
316, 260 335, 260 180, 209 212.51, 110 180),
(160 280, 200 240, 220 280, 160 280))' ), ,
('A point',ST_GeomFromText('POINT(110 245)' )) ,
('A linestring',ST_GeomFromText('LINESTRING(110 245,200 260, 227 309)' )) ,
('A multipoint',ST_GeomFromText('MULTIPOINT(110 245,200 260)' )) ;

```

### 5.3.2 Contains and Within

Contains and Within are companion relationships. If geometry A is within geometry B then geometry B contains geometry A. The Within and Contains relationships are supported by the PostGIS ST\_Within and ST\_Contains functions. Both of these functions are OGC/SQL-MM defined functions so can be found in other spatial databases. These mean more or less the same thing in all spatial databases.

One of the confusing but necessary conditions for geometry A to contain geometry B is that the intersection of the boundary of A with B can not be B. In otherwords B can not sit entirely in the boundary of A. A geometry does not contain its boundary, but a geometry always contains itself.

With the below query we can answer such fascinating questions as are both greeters inside the house and not in the courtyard? Are they both on the walkway? Is one still at the front door?

#### **Listing 5.4 What contains what?**

```

SELECT A.ex_name As a_name, B.ex_name As b_name,
    ST_Contains(A.the_geom, B.the_geom) As a_co_b,
    ST_Intersects(A.the_geom, B.the_geom) As a_in_b
FROM example_set As A CROSS JOIN example_set As B;

```

The result of the above code is:

**Table 5.3 Result of query in listing 5.4 all intersect but not all contain**

a_name	b_name	a_co_b	a_in_b
A polygon with hole	A polygon with hole	t	t
A polygon with hole	A point	f	t
A polygon with hole	A linestring	f	t
A polygon with hole	A multipoint	f	t
A point	A polygon with hole	f	t
A point	A point	t	t
A point	A linestring	f	t

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

A point	A multipoint	f	t
A linestring	A polygon with hole	f	t
A linestring	A point	f	t
A linestring	A linestring	t	t
A linestring	A multipoint	t	t
A multipoint	A polygon with hole	f	t
A multipoint	A point	t	t
A multipoint	A linestring	f	t
A multipoint	A multipoint	t	t

The following example brings to light the following items

- All the above objects intersect each other. This tells us at least one greeter is at the front door, at least one greeter is on the walkway and at least one greeter is wholly within the confines or on the boundary of the house (but they can't both be in the courtyard since they as a whole would not intersect the house if they were both in the courtyard).
- Since the polygon (house) does not contain the multipoint (the greeters), but the greeters intersect the house, we know that one person must be in the courtyard or outside the outer boundary of the house. We know that the pathway is not completely in the house but intersects it – it must have some piece in the courtyard or that sticks out of the house.
- If geometry B sits wholly on the boundary of geometry A, geometry A does not contain geometry B. Since the point (door) intersects the house but the house does not contain the door, we know the door must be on the boundary of the house. The door intersects the linestring (walkway) but the walkway does not contain the door, therefore the door must be on the boundary of the walkway (at the start point or the end point of the walkway). The walkway contains both people therefore at most one person can be at the start or end of the walkway but not both.
- Lastly all geometries contain themselves (a multipoint (the greeters) contains itself, a linestring (walkway) contains itself etc.)

If we were to use ST\_Within, it would just be the inverse of ST\_Contains – just flip the A and B geometry columns.

### 5.3.3 Covers and Covered by

As you have observed from the Contains example, the concept of OGC/SQL-MM containment is non-intuitive at the boundaries. Most people make the mistake that a geometry should contain its boundary points and that is an often desired feature. This is why PostGIS introduced the concept of Covers and CoveredBy to satisfy this need which interestingly enough also exists in Oracle Spatial via the SDO\_RELATE mask=COVEREDBY construct. These functions are called ST\_Covers and ST\_CoveredBy in PostGIS and they are NOT OGC/SQL-MM defined functions. Other caveats:

- These functions rely on functionality introduced in GEOS 3.0, so if you happen to be running say PostGIS 1.3 compiled with GEOS < 3 you will not have these functions available.

ST\_Covers is exactly like ST\_Contains except it will return true also in the case when a geometry lies completely in the boundary of the other. ST\_CoveredBy is to ST\_Covers what ST\_Contains is to ST\_Within.

#### **Listing 5.5 How is ST\_Covers different from ST\_Contains?**

```
SELECT A.ex_name As a_name, B.ex_name As b_name,
       ST_Covers(A.the_geom, B.the_geom) As a_co_b,
       ST_Intersects(A.the_geom, B.the_geom) As a_in_b
  FROM example_set As A CROSS JOIN example_set As B
 WHERE NOT (ST_Covers(A.the_geom, B.the_geom) = ST_Contains(A.the_geom,
   B.the_geom));
```

In the above code we are only going to list those geometries where the ST\_Covers answer is different from the ST\_Contains answer. Below is the result of this query.

**Table 5.4 Result of query in listing 5.5 list all where the answer of covers is different from contains**

a_name	b_name	a_co_b	a_in_b
A polygon with hole	A point	t	t
A linestring	A point	t	t

Since we limited our result only to the case where the answer produced by ST\_Covers is different from ST\_Contains, that A is considered to cover B even in the case where a geometry sits wholly on the boundary of A. Both the walkway and the house cover the door, but they do not contain the door.

### 5.3.4 Contains properly

Contains Properly is a concept that is more stringent than the contains or covers relationships. The main benefit of it is that it is in general faster to compute than the others and if you want to exclude geometries that sit partly or wholly on the boundary of another,

its the one you want. You may want to do this for example if you want to make sure your ships are all without a doubt legally within your political boundary.

`Contains` properly will give you the same result as `Contains` except in the case where any part of geometry B sits on the boundary of A. Here we repeat the same exercise except only list the contains properly options where `ST_ContainsProperly` gives a different answer from `ST_Contains`. A couple of caveats to consider:

- It is not an OGC/SQL-MM defined function – its a PostGIS specific function.
- It was introduced in PostGIS 1.4
- It requires GEOS 3.1 or above so if you are running PostGIS 1.4 with say a GEOS 3.0.3, this function will not be available to you.

#### **Listing 5.6 How is `ST_ContainsProperly` different from `ST_Contains`**

```
SELECT A.ex_name As a_name, B.ex_name As b_name,
       ST_ContainsProperly(A.the_geom, B.the_geom) As a_co_b,
       ST_Intersects(A.the_geom, B.the_geom) As a_in_b
  FROM example_set As A CROSS JOIN example_set As B
 WHERE NOT (ST_ContainsProperly(A.the_geom, B.the_geom) =
            ST_Contains(A.the_geom, B.the_geom));
```

Below is the result of the above query. Observe that `ST_ContainsProperly` is identical to the answer you get with `ST_Contains` except in the case of an areal geometry or a line geometry compared to itself or a geometry sitting partly on the boundary of another. A geometry never properly contains itself except in the case of points and multipoints. The reason points and multipoints properly contain themselves is that they are a finite number of points and therefore have no boundary to speak of. A point can never be sitting partly on its non-existent boundary.

**Table 5.5 Result of query in listing 5.6. Geometries where the contains and contains properly are different**

a_name	b_name	a_co_b	a_in_b
A polygon with hole	A polygon with hole	f	t
A linestring	A linestring	f	t
A linestring	A multipoint	f	t

So we see from the above that the only cases where the `ST_ContainsProperly` answer is different from `ST_Contains` are in the case of a polygon against itself, a linestring against itself and a point or multipoint that partly sits on the boundary of another. This tells us one person must be on the start or end of the walkway since the walkway does not properly contain both people, but does contain both people.

### 5.3.5 Overlapping geometries

Two geometries overlap when they are the same geometric dimension (points, areal, linestring), they intersect, and one is not completely contained in another. The function that supports overlaps in PostGIS is called ST\_Overlaps. This function is an OGC/SQL-MM compliant function. If we used the same example as we did above comparing if each overlaps the other, we would find that none overlap. The reason for that is as follows:

- The linestring (walkway) as we have modeled it is not of the same dimension as the polygon (house) so it can't overlap and same holds true with point/polygon (door/house), point/linestring (door,walkway)
- The multipoint (greeters) can't overlap with the point (door), because the door is in the same position as a greeter. It is contained and covered by the greeters.
- The line/line, multipoint/multipoint, point/point, polygon/polygon don't overlap because each contains itself.

### 5.3.6 Touching geometries

Two geometries are considered to touch if they have at least one point in common but those points do not lie in the interior of both geometries. The function that supports this relationship is ST\_Touches and it is an OGC/SQL-MM defined function. Revisiting our aforementioned example of the polygon with a hole (house with a courtyard), linestring (walkway), point (door), and multipoint (front and courtyard greeters), we ask which pairs of these touch. Can you guess?

#### **Listing 5.7 Which geometries touch?**

```
SELECT A.ex_name As a_name, B.ex_name As b_name,
       ST_Touches(A.the_geom, B.the_geom) As a_tou_b,
       ST_Contains(A.the_geom, B.the_geom) As a_co_b
  FROM example_set As A CROSS JOIN example_set As B
 WHERE ST_Touches(A.the_geom, B.the_geom) ;
```

The result of the above question is listed below:

Table 5.6 Result of query in listing 5.7. Geometries that touch each other

a_name	b_name	a_tou_b	a_in_b
A polygon with hole	A point	t	f
A polygon with hole	A multipoint	t	f
A point	A polygon with hole	t	f
A point	A linestring	t	f
A linestring	A point	t	f

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

A multipoint

A polygon with hole

t

f

There are a couple of important points to glean in the above results:

- The touch relationship is symmetric (or commutative), if a touches b then b touches a.
- The house touches the door and the walkway touches the door, because the door lies on the boundary of the house and walkway (not the interior of the house or walkway) even though the shared point is in the interior of the door.
- The multipoint/point pair (people/door) is missing because one contains the other and also the shared point is interior to both geometries. In fact a point can never touch a point or a multipoint because the shared points would always be interior to both.
- We see that the multipoint (people) and the polygon (house) touch because one person of the multipoint is on the boundary of the house and the other is in the hole (courtyard). NOTE: Since we modeled the courtyard as a hole, the house the courtyard is part of the exterior of the house not the interior. So even though one greeter is basking in the courtyard and the other is at the door, they as a pair – are touching the house.

### **5.3.7 Crossing geometries**

Two geometries are said to cross each other if they have some interior points in common but not all. The function that supports crosses is ST\_Crosses and it is an OGC/SQL-MM defined function.

Revisiting our aforementioned example of the polygon with a hole (the house with a courtyard), linestring (walkway), point (door), and multipoint (front door and courtyard greeters), we ask which pairs of these cross. Can you guess?

#### **Listing 5.8 Which geometries cross?**

```
SELECT A.ex_name As a_name, B.ex_name As b_name,
       ST_Touches(A.the_geom, B.the_geom) As a_tou_b,
       ST_Contains(A.the_geom, B.the_geom) As a_co_b
  FROM example_set As A CROSS JOIN example_set As B
 WHERE ST_Touches(A.the_geom, B.the_geom) ;
```

The result of the above question is listed below:

**Table 5.7 Result of query in listing 5.8. Geometries that cross each other**

a_name	b_name	a_cr_b	a_co_b
A polygon with hole	A linestring	t	f

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

A linestring

A polygon with hole

t

f

A couple of things to glean from this example:

- We only have 1 pair of geometries that cross each other, the linestring (walkway) and the house. The reason these cross is because they do not touch or contain each other but they do intersect. The shared region contains points interior to both, but one is not completely contained by the other. Note that this touching is made possible by the hole (the courtyard). The walkway has some points that fall in the courtyard so its interior is not completely contained by the house's interior.
- None of our touch winners are crossing winners. If you touch you can not cross.
- Its okay for boundary points to be shared.

### **5.3.8 Disjoint geometries**

The disjoint relationship is the antithesis of intersects. It means the two geometries have no interiors or boundaries shared. In the case of invalid geometries it is possible for them to not intersect or be disjoint. If you see such a thing, you know your geometry is invalid.

The disjoint relationship is supported by the function ST\_Disjoint and it too is an OGC/SQL-MM defined function.

Now that we have covered all the many facets of intersects and intersection, in the next section, we will take a look at output functions that are very closely related to intersection. These are the difference family of functions.

### **5.4 The remainder: ST\_Difference and ST\_SymDifference**

There are two output relationship functions that are very closely related to intersection and intersects. These are the Difference and the Symmetric Difference. These are much less commonly used than ST\_Intersection. These both return the remainder of an intersection. The ST\_Difference is a non-commutative function while the symmetric difference as the name implies is commutative.

The symmetric difference is the dark twin of the intersection. It will return the simplest geometric representation of what is left out when two geometries form an intersection. The ST\_Difference function when given a geometry A and B → ST\_Difference(A,B) returns that portion of A that is not shared with B.

One way to think about it, though it gets fuzzy at the boundaries

$ST\_SymDifference(A,B) = Union(A,B) - Intersection(A,B)$

$ST\_Difference(A,B) = A - Intersection(A,B)$

In the below we repeat a similar exercise we did with intersection except doing a difference instead of an intersection.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### Listing 5.9 What is left of the polygon and line after clipping

```

--(1)
SELECT ST_Intersects(g1.geom1,g1.geom2) As they_intersect,
       GeometryType(ST_Difference(g1.geom1, g1.geom2)) As intersect_geom_type
FROM (SELECT ST_GeomFromText('POLYGON((2 4.5,3 2.6,3 1.8,2 0,-1.5 2.2,0.056
3.222,-1.5 4.2,2 6.5,2 4.5))') As geom1,
ST_GeomFromText('LINESTRING(-0.62 5.84,-0.8 0.59)') As geom2) AS g1;

--(2)
SELECT ST_Intersects(g1.geom1,g1.geom2) As they_intersect,
       GeometryType(ST_Difference(g1.geom2, g1.geom1)) As intersect_geom_type
FROM (SELECT ST_GeomFromText('POLYGON((2 4.5,3 2.6,3 1.8,2 0,-1.5 2.2,0.056
3.222,-1.5 4.2,2 6.5,2 4.5))') As geom1,
ST_GeomFromText('LINESTRING(-0.62 5.84,-0.8 0.59)') As geom2) AS g1;

--(3)
SELECT ST_Intersects(g1.geom1,g1.geom2) As they_intersect,
       GeometryType(ST_SymDifference(g1.geom1, g1.geom2)) As
intersect_geom_type
FROM (SELECT ST_GeomFromText('POLYGON((2 4.5,3 2.6,3 1.8,2 0,-1.5 2.2,0.056
3.222,-1.5 4.2,2 6.5,2 4.5))') As geom1,
ST_GeomFromText('LINESTRING(-0.62 5.84,-0.8 0.59)') As geom2) AS g1;

```

- (1) Results in a POLYGON which is pretty much the same polygon we started out with. This some may find surprising or would hope a linestring would split it.
- (2) Results in a MULTILINESTRING composed of 3 LINESTRINGS where the polygon cut thru
- (3) This results in a geometry collection as expected composed of a multilinestring and a polygon.

Below is a diagram of the above.

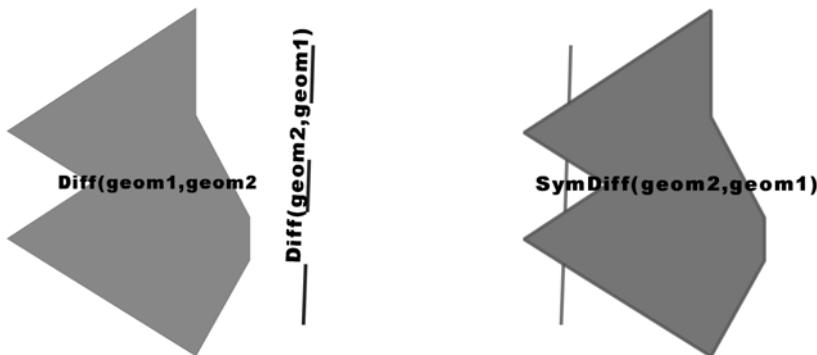


Figure 5.5 Queries from Listing 5.9 output – observe the difference of the polygon and the line is pretty much the polygon we started out with.

As you can see from above, the linestring does not bisect the polygon, though this is a common desire of many people. The reason for that is that if you think about a geometry as an infinite set of points, the difference caused by the linestring would be two polygons with partially shared boundaries. The simplest geometry to describe them is the union of these polygons which is more or less the original polygon not a geometry collection of 2 polygons or a MULTIPOLYGON. It can not be a multipolygon because a valid multipolygon can not have polygons that intersect at more than one point.

### THE DIFFERENCE OF THE POLYGON AND LINE IS NOT QUITE THE POLYGON

The result you get of the difference of the polygon from the line is not quite the polygon you started out with. It looks like it, but has one or 2 point differences at the boundaries where the line cuts thru. This is more an artifact of the differencing operation and not because of any theoretical reason.

One not so elegant hack to achieve bisection is to turn the linestring into a thin knife by buffering it ever so slightly such that the boundaries of the resulting polygons will not intersect and then possibly gluing back the left over slivers to one of the resulting polygons. This often is good enough in many cases. Below is such a demonstration minus the glue.

#### **Listing 5.10 Bisecting a polygon using the knife trick**

```
SELECT foo.path[1] As gid, ST_AsText(ST_SnapToGrid(foo.geom, 0.0000001)) As wktpoly
FROM (SELECT g1.geom2 As the_knife_cut, (ST_Dump(ST_Difference(g1.geom1,
g1.geom2))).*
FROM (SELECT ST_GeomFromText('POLYGON((2 4.5,3 2.6,3 1.8,2 0,-1.5 2.2,0.056
3.222,-1.5 4.2,2 6.5,2 4.5))') As geom1,
ST_Buffer(ST_GeomFromText('LINESTRING(-0.62 5.84,-0.8 0.59)'),0.00000001)
As geom2) AS g1
WHERE ST_Intersects(g1.geom1,g1.geom2)) As foo;
```

Table 5.8: Query result of Listing 5.10 – the polygon is cut into 3 parts

gid	wktpoly
1	POLYGON((2 4.5,3 2.6,3 1.8,2 0,-0.760732 1.7353172,...-0.6572407 4.7538133,2 6.5,2 4.5))
2	POLYGON((-0.760732 1.7353173,-1.5 2.2,-0.7274017 2.7074521,-0.760732 1.7353173))
3	POLYGON((-0.6936062 3.6931535,-1.5 4.2,-0.6572407 4.7538133,-0.6936062 3.6931535))

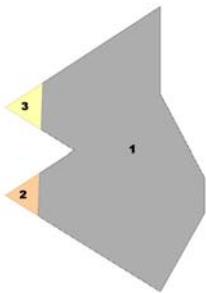


Figure 5.6 Result of the knife trick in Listing 5.10

As you can see in the above example, we buffer the line to make it a very thin polygon we can cut with. This bisects the polygon into 3 – resulting in a MULTIPOLYGON which we then break apart into 3 polygons using the ST\_Dump function we learned about in the prior chapter. In later sections we'll delve into more precise but advanced options of cutting geometries.

### **POSTGIS 2.0 ENHANCEMENTS**

In PostGIS 2.0, a new function was introduced called ST\_Split which allows for splitting a line by a point, line by line or polygon by line with a single statement. All the above can be done much simpler and more exact using ST\_Split.

In the coming sections of this chapter we will explore the ST\_DWithin relationship function which we have seen in prior examples. It is very closely related in functionality to ST\_Intersects.

## **5.5 Nearest Neighbor**

We briefly covered the ST\_DWithin function in Chapter 1. In this section, we will cover some more examples of its usage.

ST\_DWithin as mentioned earlier is used most commonly to find geometries that are close to another or determine if a geometry is within X units of another. This process is often called a nearest neighbor search. In versions of PostGIS prior to 1.3.5, this was simply short-hand for `ST_Expand(A,X) && B AND ST_Expand(B,X) && A AND ST_Distance(A,B) < X`.

### **Why 2 expands in ST\_DWithin**

The above expand calls may seem redundant as if one is true then both are true and if one is false then both are false. The reason both are needed is that depending on which you expand you may end up using an index or not and SQL planners do not process

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

statements in order. They first try to process statements in order of cost when computing costs is not more costly than sequential. At runtime one of those checks is generally far more costly than another as one is generally a constant geometry (that is the one you would want to expand) and the other is a table with a spatial index on the geometry which if you expand it will no longer use the spatial index. The planner will choose the least costly to process first so only in the case of positives will the second call be made. Also keep in mind that ST\_Expand expands the bounding box, not the geometry so returns a box. One can think of ST\_Expand as the lazy sibling of ST\_Buffer.

From PostGIS 1.3.5 on, this has additional short-circuit logic is built in.

The ST\_DWithin is not an MM/SQL function, but it is pseudo standards compliant in that the many Web Feature Servers (WFS) services such as GeoServer, Mapserver, MapInfo WFS, and Deegree have a DWithin filter operator that does the same thing. Oracle also has a function that does the same thing called SDO\_GEOM.WITHIN\_DISTANCE except the Oracle version accepts a unit of measure as argument while the PostGIS one always assumes the units are the units defined for the spatial reference system of the geometries.

### **5.5.1 Intersects with tolerance**

There is another side use of this function, and that is as a substitute for the ST\_Intersects function. This usage we term as doing an **intersects with tolerance**.

Below are a couple of reasons why you may choose to use this function over the more obvious ST\_Intersects:

- Because of rounding errors, your geometries are really close but they don't intersect. If for example you take the point on surface of a line, often that point will no longer intersect the line it came from. In these cases you can treat ST\_DWithin as a more forgiving intersects or as we like to call it ST\_Intersects with tolerance by providing a distance that is very small.
- ST\_DWithin doesn't choke on invalid geometries like ST\_Intersects often does. With ST\_Intersects you will often get a topology intersects exception error if your geometry has some self-intersecting regions. ST\_Dwithin does not care since it doesn't rely on an intersection matrix.
- Although ST\_DWithin does not work with Curved Geometries as of PostGIS 1.5, you can expect it to in later versions; possibly even in a minor release. This is because ST\_DWithin is a native function of PostGIS and ST\_Intersects is a GEOS function.
- Though it is not absolute – it is likely you will see ST\_DWithin utilizing the Z coordinate when dealing with 3D geometries sooner than you will see that happen with ST\_Intersects.
- For geodetic support – again this function will most likely support geodetic distance searches before you see it appear in GEOS functions such as

ST\_Intersects.

When used in this manner, your distance parameter is set to very small to represent the maximum distance error to consider two geometries as intersecting. So for example, you may consider the line string and a point to be intersecting if they are within 0.01 units of each other:

```
SELECT
  ST_DWithin(
    ST_GeomFromText('LINESTRING(1 2, 3 4)'),
    ST_Point(3.00001,4.000001),
    0.0001
  );
```

### **5.5.2 Finding N closest objects**

There are many ways of doing a nearest neighbor. The classic example is finding say the nearest 5 roads to a particular location that are within 10 kilometers of the location.

#### **Listing 5.11 Find closest 5 roads to a point search up to 10 kilometers out**

```
SELECT r.road_name, ST_Distance(r.the_geom, loc.the_geom)/1000 As dist_km
  FROM roads As r INNER JOIN
    (SELECT ST_Transform(ST_SetSRID(ST_Point(-118.42494, 37.31942),
  4326),2163) As the_geom) As loc
  ON ST_DWithin(r.the_geom, loc.the_geom, 10000)
  ORDER BY ST_Distance(r.the_geom, loc.the_geom)
  LIMIT 5;
```

In the above example we are only concerning ourselves with finding roads to a single point. The location need not be a point; it can be just as well be a lake, a river or a building that is represented as a linestring or polygon or even a geometry collection. In the above we are assuming our table geometries are stored in US National Atlas Equal Area meter projects (2163) and our point of interest is in long lat (4326) to we transform to (2163) to do a meter based search. The ST\_DWithin check as described in earlier chapters returns true if any point on geometry A is within X distance of any point on geometry B. The X is always in the units of the spatial reference system of those geometries and the SRID of the two geometries must be the same.

#### **Why do you need to specify a distance for 5 closest?**

As you will note – with ST\_DWithin you need to provide a limit distance. If you wanted to find the 5 closest doing a simple order by ST\_Distance(...) limit 5 without the ST\_DWithin , it would be very slow as the query would resort to what is called a table scan – calculating the distance of loc to every geometry in the table and returning the 5

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

closest. The ST\_Dwithin reduces the set of false positives significantly and that is the point. As a result it requires you know something about your data – that you know all the top 5 geometries are within 10 kilometers. The less you know about your data, the larger you should make your X and in return the slower your query will become as it would grab more false positives to inspect. Some of this will change in PostgreSQL 9.1 and PostGIS 2.0+ because k nearest neighbors (kNN) scanning will be added to GIST indexes in PostgreSQL 9.1 that may make this exercise much easier and efficient.

Another common use case is to find the closest geometry to another. For this kind of query the DISTINCT ON custom SQL addition of PostgreSQL in conjunction with ST\_DWithin comes in very handy. Below is a classic example of its use. DISTINCT ON is guaranteed to return at most one record for any DISTINCT set defined in the ON part. In this example we do the same as our prior, except we do it for all locations of interest and return the closest road to each location of interest.

#### **Listing 5.12 Find the closest road to each location search 10 kilometers out**

```
SELECT DISTINCT ON(loc.loc_name, loc.loc_city) loc.loc_name,
               loc.loc_city, r.road_name,
               ST_Distance(r.the_geom, loc.the_geom)/1000 As dist_km
  FROM loc LEFT JOIN
       roads As r
  ON ST_DWithin(r.the_geom, loc.the_geom, 10000)
 ORDER BY loc.loc_name, loc.loc_city, ST_Distance(r.the_geom, loc.the_geom)
;
```

There are three import things about using DISTINCT ON.

- the ON(...) can have as many fields that uniquely identify the record you want to be distincted, but these must appear in the ORDER BY and be the first fields to appear in the ORDER BY
- The next set of ORDER BYs are what you want to use for your preferred, the first record with the combo loc.loc\_name, loc.loc\_city is what gets returned and the closest (our last ORDER BY) is the record in that set that will be returned.
- Observe we changed our query to be a LEFT JOIN – this is so that all loc\_name,loc\_city combos are returned even if there is no road within 10 kilometers of it. If you didn't care or were assured all would have roads that close, you get slightly better performance by making this an INNER JOIN

#### **5.5.3 Using SQL Window Functions to number results**

Sometimes you want n nearest neighbors for each of your location records and sadly DISTINCT ON nor the earlier will help you pull this off in one query. Luckily PostgreSQL 8.4 has what are called Window functions. Windowing functions are an ANSI SQL 2003 standard piece you will find in all the high end enterprise class commercial databases (Oracle, IBM ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

DB2, and SQL Server 2005/2008). The windowing support in PostgreSQL is more feature rich than SQL Server 2005/2008, but not quite as feature rich as what you will find in Oracle or IBM DB2. Read the SQL Primer appendix for more details of what ANSI windowing functionality is supported and what is not.

The below example requires PostgreSQL 8.4. It will return the top 5 closest roads for each location of interest that are within 20 kilometers.

### **Listing 5.13 Find the closest 2 roads to each station search 1 kilometer out**

```

SELECT pid, land_type, row_num, road_name, round(CAST(dist_km As
numeric),2) As dist_km
FROM (SELECT
--(1)
ROW_NUMBER() OVER (PARTITION BY loc.pid
                    ORDER BY ST_Distance(r.the_geom, loc.the_geom)) As row_num,
loc.pid,loc.land_type,r.road_name, ST_Distance(r.the_geom,
loc.the_geom)/1000 As dist_km
                FROM land As loc
--(2)
LEFT JOIN
      road As r
ON ST_DWithin(r.the_geom, loc.the_geom, 1000)
WHERE loc.land_type = 'police station') As foo
--(3)
WHERE foo.row_num < 3
ORDER BY pid, row_num;

```

The above query is one against the fictitious poorly designed town we demonstrated at PostgreSQL Conference 2009 (PGCON2009). The code is available for download to build the town

from

[http://www.bostongis.com/PrinterFriendly.aspx?content\\_name=pgcon2009\\_postgis\\_spatial](http://www.bostongis.com/PrinterFriendly.aspx?content_name=pgcon2009_postgis_spatial).

Below are a sampling of the results. There are 3 important things to take away from this example. (1) The windowing function ROW\_NUMBER() will create sequentially numbered rows for each partition defined in the PARTITION BY of the OVER clause and restart numbering at the next partition (here we are partitioning by the parcel id) and the rows will be numbered by what we specify in the ORDER BY of the OVER clause – in this case Distance to a road. (2) We are using a left join which guarantees that all records in our where will be included – in this case all Police stations. However only roads within 1 kilometer of each police station will be considered. (3) We then want to order the final result by parcel id and then the proximity of road which is our row\_num (the result of our ROW\_NUMBER() windowing call).

**Table 5.9 Results of query in listing 5.13**

Pid	land_type	row_num	road_name	dist_km

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

000001038	police station	1		
000001202	police station	1		
000002997	police station	1	Main Rd	0.25
000003927	police station	1	Main Rd	0.07
000006442	police station	1		
000010131	police station	1	Main Rd	0.23
000010131	police station	2	Curvy St	0.34
000013872	police station	1		
000015423	police station	1	Elephantine Rd	0.45

:

The table above lists partial results from our query. Observe that for police stations that are further than 1 kilometer from a road, we still get back a record – but the street slots are filled in with nulls. Some police stations have more than one road within 1 kilometer (e.g. 000010131) and for that one we get two records back with the first including the closest road and the second the next closest.

Now that we have covered the basics of proximity analysis, we will move on to bounding boxes and basic geometry comparators. Although we didn't stress it in this section, what makes proximity queries fast, are the short-circuit comparisons we can do with the box caricature that surrounds them. It is this bounding box and the geometry operators that work on these that are used by spatial indexes to reduce the set of records that need to be more closely scrutinized. We will cover this in the next section.

## 5.6 Bounding box and geometry comparators

Recall from the previous chapter that every geometry has a bounding box, defined as the smallest rectangular box that completely encloses the geometry. Bounding boxes play a critical role when two geometries interact. In PostGIS and other spatial databases, comparisons between two or more geometries almost always start with bounding box comparisons in an attempt to shortcut having to compare the geometries themselves. Since geometries can take on many shapes and sizes, avoiding having to actually enumerate the details of the geometry themselves can vastly improve query time.

### 5.6.1 The bounding box

Let us demonstrate with a quick example. Suppose we have two multi-polygons, one representing the state of Washington and one representing the state of Florida. If the bounding box of Washinton is strictly above and to the left of (northwest of) Florida, then we

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

know for sure that the actual geometries share the same relationship as well. Of course if the bounding boxes check is false we can't be sure. Remember now that both states have numerous islands hanging off their coasts. In order to answer the questions unequivocally, you would have to pretty much visit every point in Washington and compare it to every point in Florida. Only after this exhaustive point-by-point checking can you conclude that all of Washington is northwest of Florida. The bounding box methodology shortcuts the point-by-point checking by first drawing rectangular boxes around each state and then asks if the box enclosing Washington is above and to the left of the box enclosing Florida. We obtain an answer almost instantly. Furthermore, because a rectangular box is completely specified by the coordinate of two opposing corners, we can pre-calculate all the bounding boxes for geometries in our table and store their coordinates in indexes. Once we have the bounding box of every geometry indexed, comparing any two geometries becomes a simple task of comparing two pairs of numbers.

Bounding boxes are so fundamental to the spatial queries that PostGIS will always assume that you would take advantage of them, freeing you from having to worry about explicitly calculating bounding boxes and creating indexes out of them.\* [The only exception is when you use deprecated functions.] Certainly, we can foresee many instances where bounding boxes will not do us much good in the end. Suppose we want to know if the centroid of Washington state is to the left of the centroid of Oregon, we cannot shortcut ourselves to an answer by simply looking at the bounding box of the two states. Below are examples of geometries with their bounding boxes wrapped around them.

#### **Listing 5.14 ST\_Box2D and geometry**

```
SELECT ex_name, ST_Box2D(the_geom) As bbox2d , the_geom
FROM (
VALUES
  ('A line', ST_GeomFromEWKT('LINESTRING (3 5, 3.4 4.5, 4 5)'), ,
   ('A multipoint',ST_GeomFromText('MULTIPOINT (4.4 4.75, 5 5)'), ,
   ('A triangle', ST_GeomFromText('POLYGON ((2 5, 1.5 4.5, 2.5 4.5, 2 5))') )
)
AS foo(ex_name, the_geom);
```

Below is the output of the above query showing the geometries encased in their bounding boxes.



**Figure 5.7 Various geometries and their bounding boxes from geometries in listing 5.14**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

## BOUNDING BOX IS NOT THE SMALLEST BOX

For brevity we stated the bounding box is the smallest box you can wrap around a geometry. Strictly speaking it is not the smallest. PostGIS will often expand a bounding box to avoid having to store too many decimal points. For example, if the smallest box is defined by the two corners of (-3.14159265,0) and (0,2.71828182), PostGIS may round this off to (-3.15,0) and (0,2.73).

### 5.6.2 Bounding box and geometry operators

PostGIS offers a number of geometry bounding box comparators that work exclusively with box2d objects and one comparator that works against the actual geometry. Some, but not all, of these operators have functional counterparts that apply to the entire geometry. As a convenient shorthand, PostGIS use various operators to symbolize comparators. For example, A && B returns true if bounding box of geometry A intersects bounding box of geometry B or vice versa where the double ampersand operator (&&) is the intersection comparator. The && operator is the one most commonly used as a pre-check for spatial relationships.

Below is a quick table of the operators, what they do and what kind of index they use. Keep in mind that in general – you put GIST indexes on geometries and btree indexes are only possible if geometry objects are relatively small such as points and small polygons and lines. If you have no btree index then an operator that only works with btree will not use an index. You can have both a gist and a btree index on the same geometry field; btree indexes on geometries is rare and of minimal utility.

Table 5.10 PostGIS operators that can be applied to geometries

Operator	What it checks	index
&&	Returns TRUE if A's bounding box intersects B's	gist
&<	Returns TRUE if A's bounding box overlaps or is to the left of B's.	gist
&<	Returns TRUE if A's bounding box overlaps or is below B's.	gist
&>	Returns TRUE if A's bounding box overlaps or is to the right of B's	gist
<<	Returns TRUE if A's bounding box is strictly to the left of B's	gist
<<	Returns TRUE if A's bounding box is strictly below B's	gist
=	Returns TRUE if A's bounding box is the same as B's	btree
>>	Returns TRUE if A's bounding box is strictly to the right of B's	gist
@	Returns TRUE if A's bounding box is contained by B's	gist
&>	Returns TRUE if A's bounding box overlaps or is above B's	gist

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

>>	Returns TRUE if A's bounding box is strictly above B's	gist
~ -	Returns TRUE if A's bounding box contains B's	gist
~=	Obsolete superceded by ST_OrderingEquals	gist

### **BOUNDING BOX OR GEOMETRY COMPARATORS?**

While all the operators are used to compare only bounding boxes (box2d objects), except for the ~= sameness operator which works against true geometries, all these operators can be called for both plain bounding boxes box objects (box2d) as well as actual geometries. They are generally used with geometries and look at the bounding box wrapper around the geometry.

Now that we have covered the basics of bounding boxes which are used extensively as a pre-check for other relationships, we will explore the most fundamental relationship – equality and its multi-faceted meeting when talking about geometries.

## **5.7 The many faces of equality**

In conventional databases, you probably never gave the equality comparator (=) a second thought before using it. When it comes to numeric and text data, equality means equality, not worth a second thought. This unambiguity does not carry over to spatial databases. When we compare two geometries, “equality” is a multi-faceted question. We can ask if the geometries occupy the same space. We can ask if they are represented by the same points. We can ask if their bounding boxes are the same.

There are 3 basic kinds of equality specific to geometries in PostGIS:

- Spatial equality (occupying the same space)
- Geometric equality (same space, more or less same points and same point order with subtleties)
- Bounding box equality - the geometries have the same smallest box that can enclose them

### **5.7.1 Spatial equality**

We consider two geometries as spatially equal if they occupy exactly the same underlying space. PostGIS uses ST\_Equals to test for spatial equality. ST\_Equals is also an OGC/SQL-MM defined function you will find in many spatial databases and all are based on the same intersection matrix model we shall cover in more detail.

ST\_Equals is most commonly used to determine if two geometries that are described by potentially different points or polygon rings in a different orientation represent the same geometry. It will also equate a polygon with a multipolygon or geometry collection that only has that single polygon in its list. Its an equality that does not care about the vector direction of the line segments or point ordering in the geometry or the fluff encasing the geometry.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### 5.7.2 Geometric equality

Geometric equality is even more strict than spatial equality as far as valid geometries are concerned. When two valid geometries are geometrically equal, they must not only share the same space, but the underlying geometric representation must also be more or less the same. For example, take any interstate highway road in the United States. Depending on which side of the road you are traveling on, the interstate is signed as north versus south or east versus west. Though it is the same interstate highway, the direction of travel matters. Sometimes it matters greatly when you get lost. In the same vein, LINESTRING(0 0,1 1) is spatially equal to LINESTRING (1 1,0 0), but are not geometrically equal.

PostGIS offers ST\_OrderingEquals function for geometric equality. In versions Pre PostGIS 1.4, the `~=` meant the same thing, but in newer installs is just bounding box equality that uses a spatial index. So to be safe -- stay away from using `~=` and stick with ST\_OrderingEquals.

#### Invalid Geometries

In the case of invalid geometries, ST\_Equals (spatial equality) may be false when two invalid geometries are exactly the same. Only in this case will you run across the paradox of geometries being geometrically equivalent ST\_OrderingEquals but not being spatially equivalent. The reason for this is that the space occupied by an invalid geometry is often ambiguous. We'll explore the concept of ambiguity when we talk about intersection matrix. `~=` from PostGIS 1.4 on is a deprecated form of ST\_OrderingEquals so stay away from it.

#### Listing 5.15 ST\_OrderingEquals (`~=`) equality vs. Spatial Equality

```
SELECT ex_name, ST_OrderingEquals(the_geom, the_geom) AS g_oeq_g,
       ST_OrderingEquals(the_geom, ST_Reverse(the_geom)) AS g_oeq_rev,
       ST_OrderingEquals(the_geom, ST_Multi(the_geom)) AS g_oeq_m,
       ST_Equals(the_geom, the_geom) AS g_seq_g,
       ST_Equals(the_geom, ST_Multi(the_geom)) AS g_seq_m
  FROM (
    VALUES
      ('A 2d line', ST_GeomFromText('LINESTRING(3 5, 2 4, 2 5)'), ),
      ('A point', ST_GeomFromText('POINT(2 5)'), ),
      ('A triangle', ST_GeomFromText('POLYGON((3 5, 2.5 4.5, 2 5, 3 5))'), ),
      ('poly with self-inter', ST_GeomFromText('POLYGON((2 0,0 0,1 1,1 -1, 2 0))'))
    )
   AS foo(ex_name, the_geom);
```

Table 5.11 Results of Listing 5.15. Even an invalid polygon is ordering equal to itself in PostGIS

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

<b>ex_name</b>	<b>g_oeq_g</b>	<b>g_oeq_rev</b>	<b>g_oeq_m</b>	<b>g_seq_g</b>	<b>g_seq_m</b>
A 2d line	t	f	f	t	t
A point	t	t	f	t	t
A triangle	t	f	f	t	t
poly with self-inter	t	f	f	f	f

Observe also in the above query that we include the result of `~=`. Recall we said the `~=` is the only true geometric operator and in fact the answer you get from it is the same as what you get the `ST_OrderingEquals` function in all cases. `ST_OrderingEquals` is just a wrapper function name for the `~=` operator. Also observe that even our polygon with self-intersections is equal to itself as far as the `~=` operator and `ST_OrderingEquals` is concerned.

Observe also that the multigeometry variant is not geometrically equal to the singular version, but in the case of spatial equality – since they occupy the same space, they are equal.

### **ST\_OrderingEquals vs. ST\_AsEWKB ...= ST\_AsEWKB check**

The geometric equality is not quite the same as what you get when you compare the point by point structure of a geometry. It doesn't even really follow the OGC `ST_OrderingEquals` standard which considers geometries equal if they are spatially equal and the ordering of the points is the same.

Another caveat is that `ST_OrderingEquals` does not work with Curved Geometries in PostGIS 1.4 and below, though it works fine with 3D geometries. If you have curved geometries, do a binary compare with `ST_AsEWKB(A) = ST_AsEWKB(B)` and if you don't care about SRID do a `ST_AsBinary(A) = ST_AsBinary(B)`.

In versions of PostGIS 1.3 and below, `~=` returned the same answer as `ST_OrderingEquals`. This may or may not be true in PostGIS 1.4 versions and above and all depends if you did a soft upgrade or hard upgrade. For people who soft-upgraded from PostGIS 1.3 and /PostGIS 1.4.1 and below, it still behaves as `ST_OrderingEquals`, but for PostGIS 1.4.0 and PostGIS 1.5+, it behaves like an spatial indexable `=`. (We suggest you do not rely on `~=` anymore and use `ST_OrderingEquals` instead.)

#### **5.7.4 Bounding box equality**

In PostGIS, the one-and-only equality comparator (`=`) is reserved for bounding box equality. If we ask if geometry A = geometry B, the result will return true if the bounding boxes of A and B are spatially equal. Because bounding box equality usurped the ubiquitous equal sign,

many people mistake bounding box equality for geometric equality. A = B does not mean that A is B. Below is an example illustrating the difference.

#### **Listing 5.16 Bounding box equality (=) verses geometric equality (~=)**

```
SELECT ST_GeomFromText('LINESTRING (3 5, 3.4 4.5, 4 5)') =
ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))') As op_eq ;
--Result
t
```

The comparison of a polygon and a linestring returns true. One may ask how is this possible? It is because their bounding boxes are equal though the geometries are quite different. However if you use the geometric equality operator, you get the expected false answer:

```
SELECT ST_OrderingEquals(ST_GeomFromText('LINESTRING (3 5, 3.4 4.5, 4 5)'),
, ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))') ) As op_same;
--Result
f
```

Important! Bounding box equality is what PostGIS uses for equality comparison which produces often unexpected results when UNIONING without ALL, using DISTINCT or doing a GROUP BY on a geometry. The following examples will demonstrate this anomaly.

#### **Listing 5.17 DISTINCT is not always DISTINCT**

```
--(1)
SELECT ST_AsText(the_geom)
  FROM (SELECT ST_GeomFromEWKT('LINESTRING (3 5, 3.4 4.5, 4 5)')
UNION ALL
SELECT ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))')
) As foo(the_geom);

--(2)
SELECT ST_AsText(the_geom)
  FROM (SELECT ST_GeomFromEWKT('LINESTRING (3 5, 3.4 4.5, 4 5)')
UNION
SELECT ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))')
) As foo(the_geom);

--(3)
SELECT DISTINCT the_geom
  FROM (SELECT ST_GeomFromEWKT('LINESTRING (3 5, 3.4 4.5, 4 5)')
UNION ALL
SELECT ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))')
) As foo(the_geom);

--(4)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

SELECT DISTINCT ST_AsText(the_geom)
FROM (SELECT ST_GeomFromEWKT('LINESTRING (3 5, 3.4 4.5, 4 5)')
UNION ALL
SELECT ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))')

) As foo(the_geom);
(1) 2 records
(2) 1 record
(3) 1 record
(4) 2 records

```

The above examples demonstrate the oddity that is the bounding box = operator. Since = for geometries is mapped to = of the bounding box and SQL uses = for DISTINCT checks, you end up with somewhat strange situations as demonstrated above. In (1) we get two records back because we are doing a UNION ALL and a UNION ALL by definition returns all records in the union. In (2) We get one record back which is the first one that is hit (the line string), because our subquery has a UNION and UNION without ALL puts in an implicit DISTINCT. Since the bounding boxes of the geometries are = they are seen as =. In (3) We get 1 record back for the same reason as (2). In (4) we get 2 records back because the output of ST\_AsText is not a geometry, but text and text = text means the text has to match exactly.

This operator also comes into play when you group by geometries and this is probably the case where most people get bitten the most. Below is a demonstration of this tragedy.

### **Listing 5.18 A count DISTINCT is not always a DISTINCT count**

```

-- (1)
SELECT COUNT(DISTINCT the_geom)
FROM (SELECT ST_GeomFromEWKT('LINESTRING (3 5, 3.4 4.5, 4 5)')
UNION ALL
SELECT ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))'

) As foo(the_geom);

-- (2)
SELECT COUNT(DISTINCT the_geom)
FROM (SELECT ST_GeomFromEWKT('LINESTRING (3 6, 3.4 4.5, 4 5)')
UNION ALL
SELECT ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))'

) As foo(the_geom);

-- (3)
SELECT the_geom
FROM (SELECT ST_GeomFromEWKT('LINESTRING (3 5, 3.4 4.5, 4 5)')
UNION ALL
SELECT ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))'

) As foo(the_geom)
GROUP BY the_geom;

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
-- (4)
SELECT the_geom
FROM (SELECT ST_GeomFromEWKT('LINESTRING (3 6, 3.4 4.5, 4 5)'))
UNION ALL
SELECT ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))')
) AS foo(the_geom)
GROUP BY the_geom;
```

- (1) Gives 1 as answer
- (2) Gives 2 as answer
- (3) Returns 1 geometry
- (4) Returns 2 geometries

As you see in (1) the DISTINCT count, you get an answer of 1, because both geometries share the same bounding box and therefore there is only 1 distinct bounding box. In (2) you get answer of 2 because we changed the Linestring slightly so the bounding box is different from the polygon. In (3) we get only one geometry back, because the group by sees the geometries as the same though they are different. In (4) we get both geometries back because they have different bounding boxes.

If this behavior causes so much confusion and pain, why do we have it? Not sure. One theory is its efficient and is not that much of an issue. It is efficient because when doing a group by or a union, the query planner need only consider the bounding box caricature that surrounds the geometry which is a lot less painful than considering a huge complex geometry. In most cases, it is rare that you are only doing a DISTINCT or GROUP BY on a geometry and that your geometries have exactly the same bounding boxes, so the uniqueness of the other fields and the rareness of non-dupes have exactly the same bounding box counter balances this behavior.

#### **STEPS YOU CAN TAKE TO AVOID THE = TRAP**

- When doing a GROUP BY or a UNION, make sure you have some other meaningful field in the GROUP BY or UNION clause. For example use a primary key of a table or something of that sort.
- If you are using GROUP BY, UNION to dedupe your geometries, GROUP or UNION BY ST\_AsEWKB (or similar) or CAST the geometry to bytea or text. GROUP BY CAST(the\_geom As text) as illustrated below.
- If you want to test spatial equality use ST\_Equals. If you want to test true geometric equality use the ~ = operator instead of =

#### **Listing 5.19 Guaranteeing unique geometries**

```
CREATE TABLE mygeom_unique(the_geom geometry);
INSERT INTO mygeom_unique(the_geom)
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
SELECT CAST(the_geom As text)
FROM (SELECT ST_GeomFromEWKT('LINESTRING (3 5, 3.4 4.5, 4 5)'))
UNION ALL
SELECT ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))')
UNION ALL
SELECT ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))')
) As foo(the_geom)
GROUP BY CAST(the_geom As text);
```

Note that the above example we are stuffing the text representation of the geometry into a geometry field. The text representation happens to be the HEXEWB normally displayed when you do a SELECT of a geometry. When you insert the above into the table, PostgreSQL silently casts it to a geometry for you.

### Support for curved geometries and 3D geometries in operators

All the bounding box operators work for curved geometries except for `~` as of PostGIS 1.4. For 3D geometries (geometries with a Z-coordinate), the Z coordinate is simply ignored for the bounding box operators, but considered with `~`.

## 5.8 Underpinnings of relationship functions

The intersection relationship we covered earlier might have given you the impression that `ST_Intersect` is the most generic relationship between two geometries. In actuality, we can generalize one step further. The underpinning of most relationship functions in PostGIS and in fact most spatial databases is based on the Dimensionally Extended 9 Intersection matrix (DE-9IM) which we shall loosely refer to as the intersection matrix. The PostGIS function that can work directly with an intersection matrix is the `ST_Relate` function.

### 5.8.1 The Intersection Matrix

The intersection matrix is the foundation of most geometric relationships supported by the OpenGIS OGC/SQL-MM standards. It is a mathematical approach that defines the pair-wise intersection and geometric dimension of the resulting intersection of the 3 regions of a geometry. It is a 3x3 matrix consisting of Interior, Boundary, Exterior on each axis, with one axis defining geometry A and the other defining geometry B. This matrix is used to both define a requirement for an arbitrary relationship as well as define the most encompassing relationship between two geometries. When used to define a custom relationship – it can have (T,F, \*, 0,1 or 2) in each of the 9 cell slots. When used to output the most constraining relationship between predefined geometries A and B – it can only contain F,0,1, or 2 in the cell slots. The reason for that is if there is an intersection there must be a corresponding dimensionality to that intersection resulting geometry and F (no intersection)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

there is no dimensionality. Not only do there exist quite a number of possible matrix, but you can construct more complex statements by chaining intersection matrixes together with boolean and/or operations.

The DE-9IM matrix concept derives from the work by M.J Egenhofer, J.R. Herring et. al <http://www.spatial.maine.edu/~max/9intReport.pdf>.

PostGIS has two variants of the ST\_Relate function. The first variant returns a boolean true or false that states if geometry A and B satisfy the specified relationship matrix. The second variant denotes what the most constraining relationship matrix is satisfied by the two geometries.

The 3 quadrants of the intersection matrix are listed below as well as what is meant by them:

- Interior – that portion of a geometry that is inside the geometry and not on the boundary
- Exterior – The coordinate space outside of a geometry but not including the boundary
- Boundary – The space neither interior nor exterior of the geometry – it is the space that separates the interior from the exterior

For each cell in the matrix – one of the following values can be put in the cell.

**Table 5.12 intersection matrix cell possible values**

Value	Description
T	An intersection must exist; the resultant geometry can be 0,1 or 2 dimensions (point,line, area)
F	An intersection must not exist;
*	It does not matter if an intersection exists or not
0	An intersection must exist and intersection must be at finite points (dim = 0 )
1	An intersection must exist and the intersection's dimension must be 1 (finite lines)
2	An intersection must exist and the intersection's dimension must be 2 (areal)

In the below exercise we will demonstrate ST\_Disjoint in intersection notation. ST\_Intersects is the opposite of ST\_Disjoint. If you were to write out ST\_Intersects in DE-9IM notation it would require 3 matrix statements. In DE-9IM notation – its easier to use proof by contradiction (assuming you are dealing with valid geometries) -- state that geometry A intersects Geometry B if they are not Disjoint thus reducing the 3 matrix statement to a NOT 1 matrix.

		B		
		Interior	Boundary	Exterior
A	Interior	F	F	*
	Boundary	F	F	*
	Exterior	*	*	*

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Figure 5.8 Disjoint relationship expressed in intersection matrix (FF\*FF\*\*\*\*)

### 5.8.2 Equality and the intersection matrix

The intersection matrix idea of equality means you can represent 2 geometries with totally different points or reversed points and as long as the resulting geometry occupies the same space, they are equal. This is what we earlier referred to as "spatial equality" (space equal). This kind of equality is determined using the OGC SQL-MM function ST\_Equals, which can be written as the below in DE-9IM notation:

		B		
		Interior	Boundary	Exterior
A	Interior	T	*	F
	Boundary	*	*	F
	Exterior	F	F	*

Figure 5.9 Equality Relationship expressed in intersection matrix (T\*F\*\*FFF\*)

Observe in the chart that interiors must intersect, exterior/interiors and exterior/boundary never intersect. The reason for that is that for a given geometry there is a demarcation between exterior/interior so the exterior should never intersect with the interior for a valid geometry. Points however have no boundary so you can't say two points that are equal have intersecting boundaries or anything about the intersection relation of the boundary with its interior.

Below is a simple example for various geometries and the accompanying ST\_Relate matrix.

#### Listing 5.20 ST\_Equals testing – a self-intersecting polygon is not equal to itself

```

SELECT ex_name, ST_Equals(the_geom, ST_Reverse(the_geom)) AS g_eq_rev,
       ST_Equals(the_geom, the_geom) AS g_eq_g,
       ST_AsText(ST_Reverse(the_geom)) AS g_rev,
       ST_Relate(the_geom, ST_Reverse(the_geom)) AS g_rel_rev,
       ST_Equals(the_geom, ST_Multi(the_geom)) AS g_eq_m
  FROM (
  VALUES
    ('A 2d line', ST_GeomFromText('LINESTRING(3 5, 2 4, 2 5)'), ),
    ('A point', ST_GeomFromText('POINT(2 5)'), ),
    ('A triangle', ST_GeomFromText('POLYGON((3 5, 2.5 4.5, 2 5, 3 5))'), ),
    ('poly with self-inter', ST_GeomFromText('POLYGON((2 0,0 0,1 1,1 -1, 2 0))'))
  )
 AS foo(ex_name, the_geom);
  
```

As you can see in the results of the above query, a given geometry is generally equal to itself, its reverse (same geometry with coordinate points reversed), and it is also equal to its ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

multigeometry counterpart. This model of a geometry is equal to itself can break down if we have an invalid geometry. The DEM-9IM relation matrix of all satisfy the T\*F\*\*FFF\* rule except for our bow-tie self intersecting polygon from chapter 2. It fails the DEM-9IM test because its interior intersects with its exterior. In other words – the area it defines is ambiguous.

**Table 5.12 Results of query 5.20**

ex_name	g_eq_rev	g_eq_g	g_rev	g_rel_rev	g_eq_m
A 2d line	t	t	LINESTRING(2 5,2 4,3 5)	1FFF0FFF2	t
A point	t	t	POINT(2 5)	0FFFFFF2	t
A triangle	t	t	POLYGON((3 5,2 5,...))	2FFF1FFF2	t
poly with self-inter	f	f	POLYGON((2 0,1 -1,1 1,0 0,2 0))	212111212	f

### 5.8.3 Using the intersection matrix with ST\_Relate

The most generic of all relationship function is ST\_Relate. There are two variants. One takes two geometries as argument and returns the relationship matrix between the two. The other function accepts any two geometries and intersection matrix as an input argument and returns true or false whether the geometries satisfy the constraints defined by the matrix.

In theory most of the intersect type relationships can be constructed using one or more ST\_Relate calls. In practice they are not because the core relationship functions have numerous short-cuts imbedded in them that take advantage what kind of geometric types each geometry is and how many geometries and so forth. Most of the other relationship functions such as ST\_Contains, ST\_Touches also take advantage of spatial indexes since their bounding boxes are required to intersect. ST\_Relate does not take advantage of spatial indexes automatically since it can be used to express both intersects relation types as well as non-intersecting relationships.

In some cases, the various permutations that can be allowed by the intersection matrix is more than can be achieved with the functions we have described. Although ST\_Relate is rarely used, its still good to understand it to get a better grasp of what the other relationship functions mean as many can be unequivocally expressed in DE-9IM geeky notation.

Below is an example that exercises both ST\_Relate functions. The example below uses CTEs introduced in 8.4 to create our virtual table that we use twice in our query. It also uses table row constructors syntax (VALUES ..) introduced in PostgreSQL 8.2. These are both SQL features defined in the ANSI SQL specs.

#### **Listing 5.21 ST\_Relate In action**

```
--[1 BEG]
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

WITH example_set(ex_name, the_geom) AS
(
  SELECT ex_name, the_geom
    FROM (
VALUES
  ('A 2d line', ST_GeomFromText('LINESTRING(3 5, 2.5 4.25, 1.6 5)') ),
  ('A point', ST_GeomFromText('POINT(1.6 5)'), ,
  ('A triangle', ST_GeomFromText('POLYGON((3 5, 2.5 4.25, 1.9 4.9, 3 5))') )
)
  AS foo(ex_name, the_geom)
)
--[1 END]
--[2 BEG]
SELECT A.ex_name As a_name, B.ex_name As b_name, ST_Relate(A.the_geom,
B.the_geom) As DE9IM,
      ST_Intersects(A.the_geom, B.the_geom) As inter,
      ST_Relate(A.the_geom, B.the_geom, 'FF*FF****') As relate_disjoint,
      NOT ST_Relate(A.the_geom, B.the_geom, 'FF*FF****') As relate_intersect
FROM example_set As A
  CROSS JOIN example_set As B;
--[2 END]
[1] CTE example_set
[2] Relate sample set

```

In [1] we use a CTE and row constructors (VALUES) to construct an inline table of sample geometries called example\_set. We then use example\_set in [2] and for each row relate to each other row in the set.

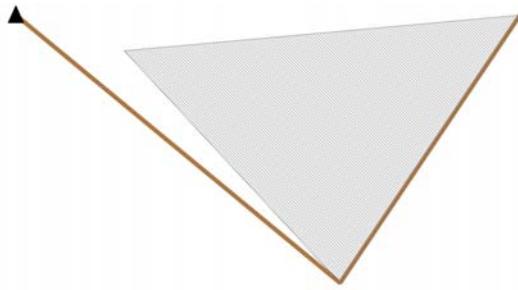


Figure 5.10 The geometries from query in Listing 5.21

Table 5.13 Result from query in listing 5.21

a_name	b_name	de9im	inter	rel_disj	not_rel_disj
A 2d line	A 2d line	1FFF0FFF2	t	f	t
A 2d line	A point	FF1FF00F2	t	f	t

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

A 2d line	A triangle	F11F00212	t	f	t
A point	A 2d line	FF0FFF102	t	f	t
A point	A point	0FFFFFF2	t	f	t
A point	A triangle	FF0FFF212	f	t	f
A triangle	A 2d line	FF2101102	t	f	t
A triangle	A point	FF2FFF10F2	f	t	f
A triangle	A triangle	2FFF1FFF2	t	f	t

Observe in the above example that the result of the not disjoint DE-9-IM statement is equivalent to the answer we get with intersects. Reading some samplings of the above relationship outputs. The relationship of the triangle with the 2d line is FF2101102 and the 2d line with the triangle is F11F00212.

		2d line			triangle			
		Interior	Boundary	Exterior	Interior	Boundary	Exterior	
triangle	Interior	F	F	2	Interior	F	1	1
	Boundary	1	0	1	Boundary	F	0	0
	Exterior	1	0	2	Exterior	2	1	2

Figure 5.11 ST\_Relate(triangle,2dline) = FF2101102, ST\_Relate(2dline,triangle) = F11F00212

Observe how if you take FF2101102 and flip the rows and columns you end up with F11F00212. Other important observations:

- The triangle and the lines interiors do not intersect as you can sort of see pictorially from the image of the geometries
- The interior of the triangle does not intersect with the boundary of the line (recall the boundary of a line is the start and end point), but the interior of the line does intersect with the boundary of the triangle and the dimension of that is a line (dimension of 1).
- Only at the intersection of exteriors of the line and interior/exterior of the polygons do we get an areal intersection (dimension of 2). This is because the exterior of the line represents all 2-D coordinate space that is not the line (so its areal).

For a more in-depth explanation of the DE-9-IM model refer to:

<http://docs.codehaus.org/display/GEOOTDOC/Point+Set+Theory+and+the+DE-9IM+Matrix#PointSetTheoryandtheDE-9IMMatrix-9IntersectionMatrix>

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Below is the intersection matrix for ST\_Within.

		B		
		Interior	Boundary	Exterior
A	Interior	T	*	F
	Boundary	*	*	F
	Exterior	*	*	*

Figure 5.12 Intersection matrix of ST\_Within (T\*F\*\*F\*\*\*)

From the ST\_Within example you can see that for a geometry to be within another the interiors of both must intersect, the interior of A can not fall outside of B (it can not intersect with exterior of B), and the boundary can not fall outside of B (boundary can not intersect with exterior of B). The boundaries however are free to intersect or not intersect.

## 5.9 Summary

In this chapter we covered a fair amount of territory involving spatial relationships. Hopefully we provided insight into the subtleties of these relationships that are not completely obvious. Now that we understand the basic foundations of spatial databases, we will look at their application in more real world examples. We will learn how to load data from various formats, dealing with spatial references and more detail about what they are, and how to do more concrete things with spatial functions. Some of the spatial functions we expose, may be ones we have not covered; many will be ones we have already explored, but that we shall combine with other functions in thought-provoking ways.

One bit of territory we have not yet delved into too deeply is the use of spatial aggregates as well as many of the geometric processing functions PostGIS has under its toolbelt. In the coming chapters we will demonstrate these.

# 6

## *Spatial reference system considerations*

This chapter covers

- What are spatial reference systems?
- Determining and selecting spatial reference systems for your data

Up to this point we have been working mostly with fictitious data with some glimpses at real world data. Using sample data to learn the basics of PostGIS is an excellent beginning. You are immediately rewarded with being able to do something without facing the distractions and the obstacles of real world data. From this chapter forward, we shield you no more.

We intend to start this chapter with a coverage of spatial reference systems. We will then follow-up with exercises on determining the spatial reference of source data and selecting suitable ones for storage.

The art and science of modeling our bulbous earth and being able to get a 2D representation on paper has been around since the antiquities. Geodetics is the science of measuring and modeling the earth. Cartography is the science of representing the earth on maps. The intricacies of these two venerated sciences are far beyond the scope of this book. After all these mathematical gyrations, we end up with something that is of utmost importance to GIS - spatial reference systems (SRS).

In this chapter, we are not going to take the easy way out by accepting SRS without understanding it. We will also avoid the path of arcane mathematics necessary to study the science in all its glory. We choose a middle ground so that you can at least have more than a one-sentence explanation of SRS when your kids finally get around to asking you about it. Our journey into the real world begins.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

## 6.1 Spatial Reference System: What is it?

The topic of spatial reference systems is one of the more abstruse in GIS to understand. This is mainly due to the loose way in which people use the term spatial reference system and secondly due to its unglamorous nature compared to other areas of GIS. If GIS is Disneyland, think of SRS as the book keeping necessary to keep the Disneyland operation afloat.

Take any two paper maps from your collection having one point in common and overlay one atop the other using the point in common. Both maps represent some or all portion of earth, but unless you are extremely lucky, the two maps have no relation to each other. Travel five centimeters right on one map and you can end up on another street. Five centimeters on the other map could put you in another continent. Your two maps do not overlay well together because they do not have the same spatial reference system. To the consumer of GIS data, our main need to acquaint ourselves with SRS is so that we can bring in data from disparate sources in different SRS and be able to overlay one atop another. Many standards exist to make this task easy without delving into the nuances of SRS. The most common is the European Petroleum Survey Group (EPSG) numbering system. Take any two sources of data with the same EPSG number and they will overlay perfectly. EPSG is a fairly recent SRS numbering system. If you uncover data from just a few decades ago, you'll be unlikely to find an EPSG number. You would have no choice but to delve into the constituent pieces that come together to form a spatial reference system. So what goes into a spatial reference system?

### 6.1.1 The geoid

From outer space, our good earth appears spherical, often described as a blue marble. To anyone living on its surface, nothing can be further from the truth. The slick glossy surface seen from outer space is replaced with mountain ranges, deep canyons, and ocean trenches. The surface of the earth with its nooks and crannies resembles a slightly charred English muffin much more than a lustrous marble. Even the idea of the earth being spherical is not quite accurate since the equator bulges out, making a trip around the equator about 42.72 km longer than a trip on one of the meridians.

In light of the fact that we have a deeply pitted somewhat squashed orange on our hands, what are we going to do? With our new GPS toys we could conceivably satellite map every square meter on earth and assign it a 3D coordinate and be done with it. This is the approach taken by many digital elevation models. Though this brute force computation methodology could certainly become the standard one day, we still need a simpler and more computationally cost effective model for most use cases.

A model, by definition, is a simplified representation of reality. All models are inherently flawed in some way or other. In exchange for their shortcomings, they provide us with a more cost effective way of looking at things. A key factor in selecting a model is finding one that balances cost of computation (in speed and complexity) with observed failure. Some models may fail in ways you don't care about in your workload since you will never exercise

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

their points of failure. Until the time when we can afford to carry around portable holograms of earth, we need several cheap models.

A starting point for any 3D model would be to actually come up with a standard definition for the surface of the earth. Do we use the mean sea level? An average of the peaks and valleys? Quite a few options are available but they all suffer from a common problem; you cannot really go out and set up a standard of measurement that is applicable around the entire world. Take the notion of sea level for instance, someone in Cardiff, England can say that his house is 50 meters above the sea during low tide. He can use this as a reference against his neighbor's house. Suppose a fellow in Pago Pago has a small house and he measures his house also to be 50 meters above sea level. What can we say about the relative elevation of the two houses to each other? Not much. Sea level varies from place to place relative to the center of the earth (provided there is a true center).

Along comes Gauss, who in the early 19<sup>th</sup> century suggests that the surface of the earth should be defined using gravitational measurements. Though he lacked a digital gravity-meter, we can picture the idea of going around the surface of the globe with such a device and measuring out a surface where gravity was constant; an equipotential surface. This is the basic idea behind the geoid. We take gravity readings of various sea levels to come up with a consensus, then use this constant gravitational force to map out an equi-gravitational surface around the globe. Many consider the geoid to be the true figure of the earth.

Surprisingly, the geoid is far from spherical. See Figure 6.1. We must not forget that the core of the earth is not homogenous. Mass is distributed unevenly giving rise to bulges and craters that far rival those found on the lunar surface. The advent of the geoid did not simplify matters. On the contrary, it created even more headaches. The true surface of the earth is now even less marble-like, even a slightly squashed orange is no longer a faithful representation.

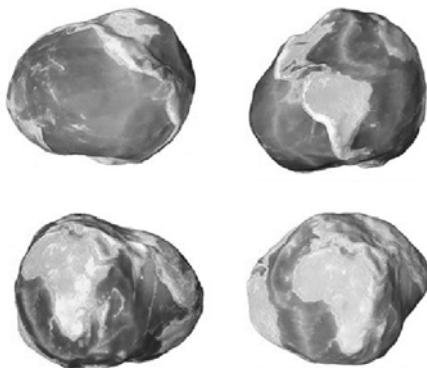


Figure 6.1 The geoid seen from different angles

Although the geoid is rarely talked about in GIS, it is one of the foundations of both planar and geodetic models. In the next section, we shall discuss the more commonly discussed ellipsoids, which are simplifications of the geoid and are a generally good enough one for most geographic modeling needs.

### **6.1.2 Ellipsoids**

Since ancient times, a good starting point for modeling the earth has always been an ellipsoid of some sort. An ellipsoid is merely a 3D ellipse.

#### **ELLIPSOIDS**

An ellipsoid is composed of 3 radii – 2 equatorial radii ( $a,b$ ) (along the x and y axes) and  $c$  is the polar radius (along the z-axis). Spheroids are a subclass of ellipsoids where  $a = b$ . A spheroid where  $c > a$  is called an oblate spheroid. By the way, if  $a = b = c$ , you have a perfect sphere.

By varying the x/y and polar axis on the ellipsoid, we can model the equatorial bulge. At some point in the history of cartography, people must have postulated one ellipsoid that could be used all around the world - a reference ellipsoid. Everyone can locate each other by finding their placement on the reference ellipsoid. The unveiling of the geoid shattered the idea of using a single ellipsoid. One look at the geoid will show why. The geoid paints a picture where the local curvature varies from place to place. An ellipsoid that fits the curvature for one spot may be awfully inaccurate for another. See Figure 6.2.

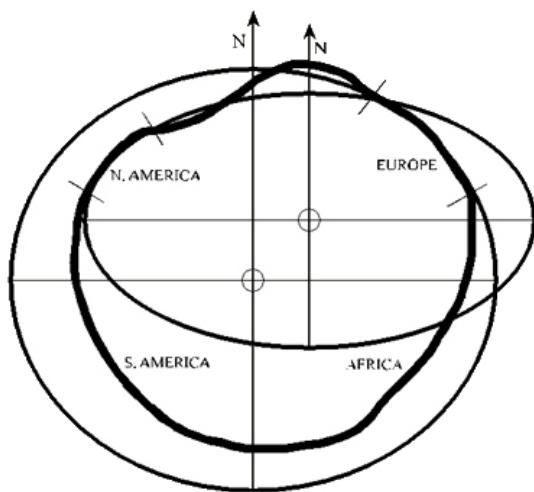


Figure 6.2 The geoid and the ellipsoid seen together

Now instead of one ellipsoid to rule us all, people on different continents want their own to better reflect the regional curvature of the earth. This gave rise to the multitude of ellipsoids we have seen today. This was all well and good when we didn't care about people far away from us. This disparate use of different systems became more of an issue with time because of the demand of scientists and governments for collaboration and the rise of oil surveying and aviation. Fortunately today, most of the world is settling on World Geodetic System (WGS84) and GRS80 ellipsoids with WGS84 becoming the standard of choice. WGS84 is what most GPS systems are based on. To call WGS84 simply an ellipsoid, is not quite accurate. The WGS84 GPS systems we use has a geoid component as well. The present WGS 84 system uses the 1996 Earth Gravitational Model (EGM96) geoid and is basically the best fitting ellipsoid to the geoid model for the selected survey points in the set.

Common ones used today

- GRS80
- WGS84 (more common nowadays) and the standard for GPS data.

The 80 and 84 are for 1980 and 1984 when those came out and are very similar.

Many ellipsoids have been used over the years and some continue to be because of their better fit for a particular region. Much of historical data is still referenced against other ellipsoids. Below is a sampling of some common ellipsoids and their various ellipsoidal parameters

**Table 6.1 Common Ellipsoids**

Ellipsoid	Equatorial radius (m)	Polar radius (m)	Inverse Flattening	Where Used
Clarke 1866	6,378,206.4	6,356,583.8	294.9786982	North America
NAD 27	6,378,206.4	6,356,583.8	294.978698208	North America
Australian 1966	6,378,160	6,356,774.719	298.25	Australia
GRS 80	6,378,137	6,356,752.3141	298.257222101	North America
WGS 84	6,378,137	6,356,752.3142	298.257223563	GPS (World)
IERS 1989	6,378,136	6,356,751.302	298.257	

One common old ellipsoid is the Clarke 1866 (this is so close to what is called the NAD27 ellipsoid that they are more or less synonymous for most purposes). So even though these old data points are measured in longitude and latitude, they are not the same longitude and latitude we use today and they also use different grounding points. They are shifted.

### LAT LON WHICH ELLIPSOID?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

This is why it is important to not just call things long lat. You can have NAD 27 Long Lat, NAD 80 Long Lat, WGS 84 Long Lat and each will be subtlety different. Though when people loosely refer to long lat – they mean WGS 84 Datum, WGS 84 spheroid in long lat units. NAD 27 is the most different because it was done a long time ago.

In the next section we'll discuss the concept of datums and how they fit into the overall picture of the spatial reference system.

### **6.1.3 Datum**

The ellipsoid alone only models the overall shape of the earth. After picking out an ellipsoid we need to anchor it down should we ever need to use it for real-world navigation. Every ellipsoid that is not a perfect sphere has at least two poles. This is where the axis arrives at the surface. These ellipsoid poles must permanently be tagged to actual points on earth. This is where datum comes into play. Even if two reference systems use the same ellipsoid, they could still have different anchors, or datum, on earth.

The simplest example of a datum is to look at the “tilt” between the geographic pole and the magnetic pole. In both models, the earth has the same spherical shape, but one is anchored at the North Pole and the other is in Canada somewhere.

To anchor an ellipsoid to a point on earth, we need two types of datums. A horizontal datum to specify where on the plane of the earth to pin down the ellipsoid and a vertical datum to specify the height. For example, the North American Datum of 1927 (NAD27) is anchored down at Meades Ranch in Kansas because it is close to the geographical centroid of the United States at the time. NAD27 is both a horizontal and a vertical datum. Below are some commonly used datums:

NAD 83 (North American 1983 which is often accompanied with GRS 80 spheroid) and NAD 27 (North American 1927 which is generally accompanied with the Clarke 1866/NAD27 ellipsoid), European Datum 1950, Australian Geodetic System 1984 and so forth.

### **6.1.4 Coordinate reference system (CRS)**

Many people confuse coordinate reference systems with spatial reference systems. A CRS is only a necessary ingredient that goes into the making of a SRS and not the SRS itself. To identify a point on our reference ellipsoid, we need a coordinate system. For use on a reference ellipsoid, the most popular CRS is the geographical coordinate system (also known as geodetic coordinate system or simply as lon-lat). You are already intimately familiar with this coordinate system. You find the two poles on an ellipsoid and draw longitude (meridian) lines from pole to pole. You then find the equator of your ellipsoid and start drawing latitude lines. Keep in mind that even though you have only seen geographical coordinate systems used on a globe, it really applies to any reference ellipsoid. For that matter to anything resembling an ellipsoid. For instance, a watermelon has nice longitudinal bands on its surface.

### **6.1.5 Projection**

Let us summarize what we discussed thus far in terms of the elements that come together to form a spatial reference system.

- We started by modeling the earth using some variant of a reference ellipsoid which hopefully is the ellipsoid that deviates least in regions on earth we care about from our calculated geoid.
- We use a datum to pin the ellipsoid down to an actual place on earth, and assign a coordinate reference system to the ellipsoid so we can identify every point on the surface. For example, the zero mile stone in Washington DC is W -77.03655 and N 38.8951 (in spatial x: -77.03655, y: 38.8951) on a WGS 84 ellipsoid using WGS 84 datum, but on a NAD27 Datum, Clarke 1866 ellipsoid, this would be W -77.03685, N 38.8950.

We can quit at this point and declare that we have all the elements necessary to tag every spot on earth. We could even develop transformation algorithms that convert coordinates based on one ellipsoid to another. In fact, many sources of geographic data do stop right at this point and do not go on to the next step topic of projections. We term this kind of data as unprojected data. All data served up in the form of latitudes and longitudes are unprojected. You can do quite a bit with unprojected data, using the great circle distance formula, you can get distances between any two points. You can use it to navigate to and from any points on earth.

Projection has distortion built-in. The concept of projection generally refers to taking an ellipsoidal earth and squashing it on a flat surface. Since geodetic and 3D globes are ellipsoidal they do not refer to a flat surface and are referred to as unprojected. In the next section, we'll briefly go over the different kinds of projections and why we have them.

### **6.1.6 Different kinds of projections**

So why do we have 2D projections of our ellipsoid or geoid? The obvious reason is the aforementioned one of portability. We cannot be carrying a huge globe everywhere we go. The less obvious reason but the one that is more relevant to us is the mathematical and visual simplicity that comes with planar (Euclidean) geometry.

As we have iterated many times, PostGIS works for the most part on a Cartesian plane and most of the powerful functions assume a cartesian model. Our brain and the brain of PostGIS can perform area and distance calculation quickly on a cartesian plane. On a plane, the area of a square is simple its side squared. Distance is nothing more than applying the pythagorean theorem. A planar model fits on a piece of paper nicely. Calculating the area of a square on the surface of an ellipsoid without a projection becomes quite a challenge, not the least of which is deciding what constitutes a square on an ellipsoid in the first place.

#### **PostGIS 1.5 supports geodetic data**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In PostGIS 1.5 support for geodetic data was introduced using the new datatype "geography" similar in concept to SQL Server 2008 geometry vs. geography types. Most of the functions still work against geometry data with only a few functions such geography operators and distance functions. We shall cover the new geography format in a later chapter.

How exactly you squash an ellipsoidal earth on a flat surface is controlled by several classes of rules we will loosely refer to as the "classes of cartesian coordinate systems". Each class of rules tries to optimize for a set of features and each specific instance of a coordinate system is bounded to a particular region on earth and uses a particular unit (usually meters or feet).

Needless to say there are four balancing features you try to optimize. Which importance coefficient you place on each will dictate what type of coordinate system and eventually what spatial reference system(s) you should settle on.

- Measurement
- Shape - how accurately it represents angles
- Direction – is north really north
- Range of Area supported

The general trade off – if you want to span a large area, you have to give up measurement accuracy or deal with the pain of maintaining multiple spatial reference systems and some mechanism to shift between them as you move around the earth. The larger your area, the less accurate and potentially grossly and unusably accurate your measurement will be. If you try to optimize for shape and to cover a large range, your measurements may be off and may be way off.

There are a couple of flavors of projections (squashing) we can do to optimize for different things. These are listed below

- Cylindrical Projections -- imagine a piece of paper rolled around the globe and it getting an imprint of the globe on its surface. Then you unroll it to make it flat. The most common of these is the Mercator Projection which has the rolled cylinder bottom parallel to the equator. This results in great distortion at the polar regions and measurement accuracy is best the closer you are to the equator since the approximation of flat is most accurate at the point.
- Conic Projections – This is sort of like the cylindrical projection except you wrap a cone around the globe – take the imprint of the globe the cone and then roll it out.
- Azimuthal Projections -- Project a spherical surface unto a plane.
- Within these 3 kinds of projections there is also the factor of how you orient the paper you roll around the globe – these are

- Oblique - Neither parallel nor perpendicular to the equator – some other angle
- Equatorial - Perpendicular to the plane of the equator
- Transverse – oriented along the equator (parallel to the equator)

Combinations of these categorizations form the main classes of planar coordinate systems:

- Lambert Azimuthal Equal Area (LAEA) – reasonably good for measurement and can cover some large areas, but not great for shape. The one we like most when dealing with United States data and when we are concerned with some what decent measurement is US National Atlas (EPSG:2163). This is a meter based spatial reference system. These are in general not good at maintaining direction or angle.
- Universal Trans Mercator (UTM) – These are generally good for maintaining measurement and shape and direction, but only span 6 degree longitudinal strips. If you need to cover the whole globe and you use this, you will have to maintain about 60 spatial ref ids. You will still have major issues when crossing poles.
- Mercator – these are good for maintaining shape, direction and span the globe, but no good for measurement and make the poles look really huge. The measurement you get from it is cartoon and levels of cartoonish vary depending on where you are. The most common in use are variants of World Mercator (SRID:3395) or Spherical Mercator (AKA) Google Mercator (SRID:900913 and now it is an EPSG standard with EPSG:3785 – this is fairly new so you may not find it in your spatial\_ref\_sys table ). These are a common favorite for web map display because you only have to maintain one SRID and they look good visually to most people.
- National Grid Systems – These are generally a variant of UTM or LAEA but used to define a region such as a country. As mentioned US National Atlas (SRID: 2163 US National Atlas Equal Area) is common for US. These are generally decent for measurement (but not super accurate), don't always maintain good shape, but cover a fair amount of area and possibly the area you care about in a single spatial ref.
- State Plane – these are US spatial reference systems usually designed for a specific state, but in general all use some derivative of UTM. Generally there are 2 for a state – one measured in meters and one measured in feet. Though some larger states have 4 or more. These tend to be the best for measurement and commonly used by state/city land surveyors but can only deal with one state.
- Geodetic – PostGIS can store WGS 84 long lat (4326) in the popular geometry

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

data type – but often more than not, you will want to transform to another spatial reference system to use it. You can sometimes get away with using it as is for small distances along the same longitude and so forth and as a generally good idea of if two things intersect, but keep in mind when you use it, PostGIS is projecting it. It is squashing it on a flat surface making longitude – x and latitude – y ; so even though it is in theory unprojected – it is really projected in a mostly unusable way – the colloquial name for this kind of projection is “Plate Carrée”.

Given all these different options for spatial reference systems, determining which one your source data is in as well as choosing one for storage is often a tricky undertaking. In the next section we'll go over selecting a spatial reference system as well as some simple exercises for determining which spatial reference system your source data is in.

## **6.2 Selecting a spatial reference system to store data**

One of the most common questions people ask is what spatial reference system(s) should you store your data in. The answer is of course – it depends.

Below is a list of the most commonly used spatial reference systems and their PostGIS/EPSC SRIDs. PostGIS SRIDs try to follow the EPSG numberings so you can assume for sake of argument they are the same. This is not necessarily true for other spatial databases and you could give a spatial reference system several different ids. While EPSG is the most common authority on spatial reference systems, it is not the only one. Many people for example load up their tables with ESRI defined ones which are sometimes identical to EPSG ones but under an SRID code more ArcGIS friendly.

**Table 6.2 Common spatial reference systems and their fit for purpose**

EPSG/PostGIS SRID	Colloquial Name	Range	Measurement	Shape
4326	WGS 84 Long Lat	excellent	bad	bad
3785/900913 (old number)	Spherical Mercator	good	bad	good
	Google Mercator			
32601-32760	UTM WGS 84 zones	medium	fairly good	good
2163	US Nat Atlas EA	all US	medium	medium
State Planes	US State planes	medium	good	good

If you deal with mostly regional data say for a country or state, then its generally best to stick with one of the national grid or state planes systems. You'll get fairly good measurement accuracy and it will also look good on a map.

Be forewarned that since PostGIS 1.4 and lower really only supports Cartesian coordinate systems you may have to use several if you need to span large areas and maintain measurement accuracy. We'll cover techniques on how to do this in a later chapter.

### **6.2.1 Pros and cons of using EPSG:4326**

The most common spatial reference system people use is WGS 84 Long Lat EPSG:4326. Aside from the common reason, that people just don't know any better, the reasons why knowledgeable people do this is as follows:

- It covers the whole globe and is the most common transport spatial reference system. For example all GPS data is stored in this. If you need to cover the world, dish out to lots of people and also deal with lots of GPS data, this is not a bad one.
- Most commercial mapping toolkits although they use some variant of Mercator for display, expect the data to be fed in WGS 84 long lat. ST\_Transform also introduces some rounding errors as you retransform data, so it is best to transform only once from source format. ST\_Transform is a fairly cheap process so is okay to run for each geometry if you keep functional indexes on the transformations you use for distance checking.

There are reasons not to use it.

- Bad for measurement. If measurement is something you do often, especially when you are only concerned about small regions such as a country or state, you will spend a lot of time transforming back and forth if you use 4326. There are hacks for avoiding this with point data using a combination of ST\_Distance\_Spheroid/Sphere and ST\_DWithin and in PostGIS 1.5+ you can just use the geography data type instead (in exchange for much fewer functions). For non-point geometries where you need minimum distance rather than distance from centroid, the ST\_Distance\_Sphere\* hack does not work for PostGIS 1.4 and below.
- Things like intersects, intersection, and union generally work fine for small geometries, but fall apart as geometries get long such as continents or long fault lines
- Bad for shape. It also doesn't look good on a map. Its all squashed because recall – we are making the latitude which is meant to be measured around an ellipsoid and showing it on a planar axis we call X. We are doing the same for longitude.

### **6.2.2 Geography data type for EPSG:4326**

If you will be storing your data in WGS 84 spatial reference system and are using PostGIS 1.5, you should consider using the new geography data type that was introduced in PostGIS

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

1.5. The key benefit it provides over the geometry EPSG:4326 is that it is good for measurement since it really is not projected and measurements are always in meters. Pros are as follows:

- If you are new to GIS and have no clue and don't have time to learn the ins and outs of spatial reference systems. This will more or less work out of the box for you.
- Distance and area measurements are as good or better than UTM so if your data covers the globe and you just need fast distance, area, length, measurements this is probably the best
- Most web mapping layers such as google, virtual earth etc. expect data to be fed to them in WGS 84 so geography will work fine out of the box.

So geography is great, why should you use geometry instead:

- Processing functions for geography are limited - As of PostGIS 1.5 -- you can do an ST\_Intersection and a ST\_Buffer. However these are just wrappers around the geometry implementation that do a behind the scenes transformation to a suitable planar projection, so its not too hard to roll your own functions. These will probably be laid out in the PostGIS wiki somewhere for cut and paste.
- While you can piggy-back on the geometry functions for processing by casting and transforming to geometry and casting back, the ST\_Transform operation is not a lossless operation. ST\_Transform introduces some floating point aberrations which can quickly accumulate if you do a fair amount of geometry processing.
- If you are dealing with regional data -- WGS 84 is generally not quite as accurate for measurement as what regional spatial refs offer.
- If you are building your own mapping app -- you will still need to learn about transforming your data to other spatial reference systems to look good on a map and while the transformation process is a fairly cheap process, it can quickly become taxing the more data you pull or more users hitting your database.
- Not as many tools support geography. In theory any tool that just uses the ST\_AsBinary and other output functions of PostGIS to render geometries, will work fine with geography without any change.

### ***6.2.3 Mapping just for presentation***

While the basic mercator projections are quite horrible for measurement calculations, in most cases, except if you sit on the equator, they are a favorite for web mappers because they

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

look good on a map. The advantage of things like Google mercator is that it is one spatial ref that covers the whole globe.

So if your primary concern is just looking good on a map and overlaying on google maps with something like OpenLayers, its not a bad option for native storage of your data. If you do have some concern for some distances and areas, it depends how accurate you need it. Below is a quick table -- generated from code in Chapter 8 that lists city pairs measured using various spatial reference systems.

**Table 6.3 Results of distance calculations in kilometers**

<b>city1</b>	<b>city2</b>	<b>sp</b>	<b>spwgs84 wm</b>	
Beijing	Jerusalem	7119	7135	9104
Beijing	Melbourne	9128	9095	9938
Beijing	Philadelphia	11060	11085	21330
Beijing	Sao Paulo	17600	17601	19656
Beijing	Shanghai	1066	1065	1315
Cairo	Jerusalem	423	424	494
Cairo	Melbourne	13977	13973	15024
Cairo	Philadelphia	9154	9173	11928
Cairo	Sao Paulo	10224	10216	10667
Cairo	Shanghai	8351	8367	10045
Rio de Janeiro	Jerusalem	10323	10315	10808
Rio de Janeiro	Melbourne	13221	13240	21078
Rio de Janeiro	Philadelphia	7706	7680	8250
Rio de Janeiro	Sao Paulo	338	338	368
Rio de Janeiro	Shanghai	18249	18256	19399
Sydney	Jerusalem	14114	14111	15040
Sydney	Melbourne	694	694	858
Sydney	Philadelphia	15895	15895	26702
Sydney	Sao Paulo	13357	13377	22041
Sydney	Shanghai	7878	7849	8354

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In the above table, we see various city point pairs and their distances measured in WGS 84 sphere (sp), WGS 84 spheroid (spwgs84), and web mercator. As we can see from the above listing Web mercator distance precision is much worse than the others and gets worse the farther away two cities are from each other or for particular regions (e.g. further away from equator). For example Beijing to Philadelphia is really really bad for Mercator. The sphere calculations are pretty good for long range/short range rule of thumb calculations.

The above table covers distance, but what about area of geometries. How bad is the story there. Again depends where you are on the globe, but in general is bad. Below is a table of areas of 10 meter buffers around the globe generated from code in Chapter 8.

**Table 6.4 Listing of different areas in different regions of the world**

city	utm_sm	geog_sm	wm_sm	diff_utm_wm	diff_utm_g
Honolulu	312	312	362	0.13	49.48
San Francisco	312	312	500	0.22	188.03
Boston	312	312	572	0.02	260.22
Paris	312	312	722	0.24	409.54
Oslo	312	312	1240	0.18	927.74
Saint Petersburg	312	312	1241	0.09	929.03
Helsinki	312	312	1260	0.15	947.76
Bergen	312	312	1272	0.11	959.40
Arkhangelsk	312	312	1681	0.20	1368.54
Murmansk	312	312	2412	0.25	2100.22

#### Why is a 10-meter buffer of a point 312 sq m?

It isn't. If you do your calculation, a perfect 10 meter buffer will give you an area of  $10^2 \pi$  which is around 314 sqm. The default buffer in PostGIS is a 32-sided polygon (8 points approximate a quarter segment of a circle). You can make this more accurate by using the overloaded version of the ST\_Buffer function that allows you to pass in the number of points to approximate a quarter segment.

#### 6.2.4 Covering the globe when distance is a concern

If you have the unfortunate predicament of needing to cover the whole globe, need good measurement and shape accuracy, then most likely one spatial reference system is not going to cut it for you. A common favorite is to use the UTM family of SRIDs. This is about 60

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

SRIDs for WGS 84 in all each covering about 6-degree longitudinal strips. There are also UTM for NAD 83, but the WGS 84 is more common.

Of course you'll want to know how to figure out the UTM WGS 84 SRID for your particular dataset. There is such a function located in the PostGIS wiki <http://trac.osgeo.org/postgis>. The below is a slight variant of that function that takes any geometry and returns the WGS 84 UTM SRID of the centroid of that geometry.

### **Listing 6.1 Determining WGS 84 UTM SRID of a geometry**

```
CREATE OR REPLACE FUNCTION upgsql_utmzone_wgs84(geometry) RETURNS integer AS
$$
DECLARE
    geomgeog geometry;
    zone int;
    pref int;
BEGIN
    --[1 Beg]
    geomgeog:=ST_Transform(ST_Centroid($1),4326);
    --[1 End]
    --[2 Beg]
    IF (y(geomgeog))>0 THEN
        pref:=32600;
    ELSE
        pref:=32700;
    END IF;
    zone:=floor((ST_X(geomgeog)+180)/6)+1;
    --[2 End]

    RETURN zone+pref;
END;
$$ LANGUAGE 'plpgsql' immutable;
[1] convert to a long lat point
[2] determine utm start and sub
```

[1] We convert our geometry to a point and then transform it to WGS 84 long lat. This function assumes the srids are named the same as the EPSG for UTMs which is the case with the default spatial\_ref\_sys that comes packaged with PostGIS. [2] We determine if latitude is positive or negative positive UTMS start in the 32600 and go up one for every 6 degrees. Negative or 0 start at 32700. So the final srid is the sum of these.

If you need to maintain multiple SRIDs, you have basically three approaches

- Store one (usually 4326) and transform on the fly as needed
- Maintain one for each region and possibly partition your data by region using table inheritance
- Maintain multiple geometries, one field for each you commonly use.

There are many philosophies on which way to go and none is really right or wrong. For our cases, we have found that keeping one SRID usually 4326 and transforming as needed, but maintaining functional indexes on transforms used for distance calculations works best.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

We also like using views as an abstraction layer and the view would contain the calculated transform. PostgreSQL not only supports functional indexes, but also partial ones as well. A partial index allows you to for example index only part of your data. So for example you would only want to apply an ST\_Transform function on a certain region of data for a given UTM, otherwise you will run into coordinate bounds issues. Generally speaking, its best to just partition your data using table inheritance and apply different transform index on each table separately. Below is an example of a functional st\_transform index and a possible view you may create to take advantage of it.

### **Listing 6.2 Using functional indexes**

```
CREATE INDEX feature_data_the_geom_utm
  ON feature_data
  USING gist
  (st_transform(the_geom, 32611));

CREATE VIEW vwfeature_data AS
  SELECT gid, f_name, the_geom,
         ST_Transform(the_geom,32611) As the_geom_utm
    FROM feature_data;
```

In the above view, we are transforming our native data to SRID 32611 which is one of the UTM SRIDs for a region of California in the US.

### **Functional indexes on ST\_Transform**

Putting functional indexes on ST\_Transform is something we do often and then build a view against our data to use the transformed version of the data. It is a grey zone, in the sense that we are exploiting a small violation. In PostGIS, the ST\_Transform is marked as immutable mostly for performance reasons, which means when you calculate it for a given geometry it can be assumed to never change and PostgreSQL kindly believes PostGIS and caches it and allows it to be used in functional indexes. Only functions marked as immutable can be used in functional indexes and in theory a function that relies on a table (except possibly for a static system table in pg\_catalog) is at best considered stable (meaning it won't change within a query given the same inputs). In actuality, it is a bit of lie that it is immutable, because it relies on entries in the spatial\_ref\_sys table. If you happened to change the entry for your transform in the table, you really need to reindex your data otherwise it will be wrong, but then again so would be case if you kept a second transformed geometry column. We tend to think a bit liberally and think of the spatial\_ref\_sys table as practically immutable. Though you may add entries, it is rare that you change the definitions of entries once created and thus the immutability argument is valid.

The other issue with functional indexes, is they get dropped when you restore your data, unless if you make sure to set the search\_path of the ST\_Transform function to include the schema the spatial\_ref\_sys resides in (only supported in PostgreSQL 8.3 and above).

Read our diatribe on this topic for more details  
<http://www.postgresonline.com/journal/index.php?archives/121-Restore-of-functional-indexes-gotcha.html>

So why do we use it even though it is a bit of a no no? Well the other alternative is to keep a geometry field for your alternative spatial references. This is annoying for 2 reasons. (1) you have to ensure it is updated when your main geometry field is updated, which means putting in a trigger. Someone may get confused and update that one instead. (2) The more annoying reason is that if you have big geometries, having a second big geometry in your table slows down updates considerably because of the MVCC nature of PostgreSQL to do an insert/delete of the whole record during an update. It probably slows down selects too as you have a fatter row to contend with. ST\_Transform on the fly is cheap, but doing an index search on this calculated call is not possible without a GIST index on this transformed data.

Often times, you will be loading spatial data into your database that you did not create. Before you even worry about what spatial reference you should use to transform your source data to for storage, you first have to figure out what spatial reference system your source data is in. If you guess wrong on that, then all your spatial transformations moving forward will be wrong. In the next section we shall cover how to determine the spatial reference system of a source of data.

### **6.3 Determine the spatial reference system of source data**

In this section, we will go thru some exercises of determining the spatial reference system of source data. This will prepare you for the next chapter, when we finally start loading real data. Before we even do that, we need to know where we can get free data to play with. Locations for free data can be found in Appendix A.

Determining the spatial reference system of your source data is sometimes a fairly easy task and sometimes is not. Sometimes a site just tells you the EPSG code for their data, and your work is done. Often times, they will give you a text representation of the spatial reference system either in WKT SRS notation or some sort of free text. In these cases you will need to match up the description with a record in the spatial\_ref\_sys table.

Often with ESRI Shapefiles there is a file with a .prj extension that gives the spatial reference system information in WKT SRS notation. This is also used by third party tools to infer projection for cases when different layers need to be transformed to the same spatial reference system to be overlaid together on a map. In the following exercises, we'll demonstrate some SRS text descriptions and demonstrate how you can match these with a srid in the spatial\_ref\_sys table. In some cases your task may be harder, as the record you are looking for may not exist and you will need to add it. We shall go over that too.

More shockingly some data comes with no spatial reference information or the wrong information. The easiest way to determine this is to overlay this on top of a layer for the same region that you know the spatial reference system for and reproject to that projection.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Common errors are for example setting NAD 27 data to a NAD 83 spatial reference system. In these cases you will see Doppler like shifts when you overlay the two on a map. If things are way off, one of your layers won't even show when you transform it to the same SRS as your known layer.

### **6.3.1 Guessing at a spatial reference system**

We shall go over some very simple but common exercises for determining the spatial reference system of source data. In these examples we'll cover picking out key elements in SRS text representations.

#### **EXERCISE 1: THE US STATES DATA**

Earlier in this chapter, we downloaded the file

<http://edcftp.cr.usgs.gov/pub/data/nationalatlas/statesp020.tar.gz>

However for this particular set, they gave us a states020.txt which gives us spatial reference information as well as lots of details about how the dataset was made and licensing.

If you scroll far enough in the file you will see this:

```
Spatial_Reference_Information:  
    Horizontal_Coordinate_System_Definition:  
        Geographic:  
            Latitude_Resolution: 0.000278  
            Longitude_Resolution: 0.000278  
            Geographic_Coordinate_Units: Decimal degrees  
        Geodetic_Model:  
            Horizontal_Datum_Name: North American Datum of 1983  
            Ellipsoid_Name: GRS1980  
            Semi-major_Axis: 6378137  
            Denominator_of_Flattening_Ratio: 298.257222
```

The above is a very important piece of information. It tells us that the data is in decimal degrees, uses GRS1980 ellipsoid, and datum: North American Datum 1983. These are 3 ingredients you want to know about every data source you have:

- unit: degrees
- ellipsoid: grs1980
- datum: nad1983

If you are dealing with projected data (non-degree data), there are some other fuzzy pieces you will want to know. One of them being projection and depending on the projection, there are additional magical parameters each has.

- projection : (degree is longlat), eaea, utm, tmmerc, lcc, stere

After you have figured out these pieces, the next thing you want to do is match your source to a spatial reference system defined in the spatial\_ref\_sys table and then record the srid number for it. Sometimes the record you are seeking may not be in that table and you will need to add it or just live without one. Living without one is only an option if you know

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

your data is planar, you know the units, and all data you will be getting is from the same source where they made sure to use the same spatial reference system.

The two fields of information most helpful in the spatial\_ref\_sys table to help you guess at a projection.

For the above data, we do a simple SELECT query to determine the srid and use the PostgreSQL ILIKE predicate to do a case insensitive search.

```
SELECT srid, srtext,proj4text
FROM spatial_ref_sys
WHERE proj4text ILIKE '%nad83%'
    AND proj4text ILIKE '%grs80%' AND proj4text ILIKE '%longlat%';
```

The select query we ran will return one record with srid 4269. Its generally easier to query the proj4text field for matches since the proj4text is much shorter and more consistent than the srtext.

#### **EXERCISE 2: SAN FRANCISO DATA (READING FROM .PRJ FILES)**

For this second exercise we grabbed a zip file containing bay area bridges. The file includes a .prj file which has projection information.

[http://gispub02.sfgov.org/website/sfshare/catalog/bayarea\\_bridges.zip](http://gispub02.sfgov.org/website/sfshare/catalog/bayarea_bridges.zip)

The .prj contents look as shown

```
PROJCS[ "NAD_1983_StatePlane_California_III_FIPS_0403_Feet",
GEOGCS[ "GCS_North_American_1983",
DATUM[ "D_North_American_1983", SPHEROID[ "GRS_1980", 6378137.0, 298.257222101]],
, PRIMEM[ "Greenwich", 0.0],
UNIT[ "Degree", 0.0174532925199433]],
PROJECTION[ "Lambert_Conformal_Conic"], PARAMETER[ "False_Easting", 6561666.66666666],
PARAMETER[ "False_Northing", 1640416.6666666667], PARAMETER[ "Central_Meridian",
-120.5],
PARAMETER[ "Standard_Parallel_1", 37.06666666666667],
PARAMETER[ "Standard_Parallel_2", 38.43333333333333],
PARAMETER[ "Latitude_Of-Origin", 36.5],
UNIT[ "Foot_US", 0.3048006096012192]]
```

We can surmise from this file based on the PROJCS that the units are measured in feet, its NAD83 datum and some California State Plane. So now we guess by doing a query.

```
SELECT srid, srtext,proj4text
FROM spatial_ref_sys
WHERE srtext ILIKE '%california%' AND proj4text ILIKE '%nad83%'
    AND proj4text ILIKE '%ft%';
```

The above query yields 6 records. When we look at the srtext of each, each has something of the form NAD83 / California zone 1 (ftUS). Where the 1 goes from 1-6. Remembering our roman numeral lessons from grade school, we recall that III is roman numeral for 3. So our answer must be **SRID 2227** which has an srtext field that looks like the below.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
"PROJCS[ "NAD83 / California zone 3 (ftUS)" ,
GEOGCS[ "NAD83" ,DATUM[ "North_American_Datum_1983" ,
SPHEROID[ "GRS 1980" ,6378137,298.257222101,AUTHORITY[ "EPSG" , "7019" ]],
AUTHORITY[ "EPSG" , "6269" ]],
PRIMEM[ "Greenwich" ,0,AUTHORITY[ "EPSG" , "8901" ]],
UNIT[ "degree" ,0.01745329251994328,AUTHORITY[ "EPSG" , "9122" ]],
AUTHORITY[ "EPSG" , "4269" ]],
UNIT[ "US survey foot" ,0.3048006096012192,AUTHORITY[ "EPSG" , "9003" ]],
PROJECTION[ "Lambert_Conformal_Conic_2SP" ],
PARAMETER[ "standard_parallel_1" ,38.43333333333333],PARAMETER[ "standard_parallel_2" ,37.06666666666667],PARAMETER[ "latitude_of_origin" ,36.5],
PARAMETER[ "central_meridian" ,-
120.5],PARAMETER[ "false_easting" ,6561666.667],
PARAMETER[ "false_northing" ,1640416.667],AUTHORITY[ "EPSG" , "2227" ],AXIS[ "X" ,EAST],AXIS[ "Y" ,NORTH]]"
```

Now that we have a bit of a grasp of how to match an SRS to one in our table, what do we do if there isn't one in the table?

#### **EXAMPLE 3: IF YOU GUESS WRONG**

Lets say you guessed wrong at the SRID of your data and you've already loaded in all your data. What do you do now? Luckily there is a maintenance function in PostGIS to help you out in this situation. Its called UpdateGeometrySRID which will correct the mistake.

```
SELECT UpdateGeometrySRID('sf', 'bridges', 'the_geom', 2227);
```

In the below exercise lets say we brought our San Francisco data in wrong with -1 SRID or some other wrong spatial reference. This would become quite apparent if we tried to transform our data. If we did and the data is wrong, we would get errors such as NaN when doing distance checks on the transformed data or a Transform error when doing the transformation. In the next section we'll talk a bit about what to do when you have concluded your spatial\_ref\_sys does not have the spatial reference you are looking for.

#### **6.3.2 When the spatial reference system is missing**

Sometimes you may come up short and no record in the spatial reference system matches what you are looking at. The best place to go at that point is <http://spatialreference.org>

The spatial reference org site contains thousands of user contributed spatial reference systems in addition to the standard ones. Best of all, if the record you are looking for can not be found and you happen to have .prj file, you can submit the contents of that via the "Upload your own" link, and the site will magically determine the insert statement you need to use to insert the new item in your spatial\_ref\_sys table.

**SpatialReference.org use the auth\_srid instead of srid**

The spatial reference site by default assigns an SRID starting with 9 to denote it was grabbed from the spatial reference org site. For sake of consistency, we replace this SRID number with what is listed in the auth\_srid field. By using this convention, you will not accidentally insert a record in the spatial\_ref\_sys that is already in the table.

While it is possible to create your own custom spatial reference system to suit your specific needs, such a topic is beyond the scope of this book. The underpinning of the spatial reference system transformation support in PostGIS uses the proj.4 library. For those interested in how to do this links to articles in Appendix A on spatial reference systems proj.4 syntax may be of use.

## 6.4 Summary

In this chapter we explained the details of a spatial reference system and what makes up one. We hope from our discussions, that you understand the importance of them as well as a general rule of thumb for selecting one and determining which ones your source data is using.

In the next chapter we shall continue our journey into the real world by loading real geographic data. We will cover some of the more popular free and open source tools both packaged and not packaged with PostGIS that are useful for importing and exporting data. We will go over the pros and cons of each as well as provide examples on how to use them.

## 7

# *Working with real data*

This chapter covers

- Tools for importing/exporting spatial data
- Importing data from various file/types

In the prior chapters we explored many of the functions provided in PostGIS by creating our own test data and also discussed spatial reference system considerations for data. In this chapter we shall cover how to load real data and export data. Real data for PostGIS is mostly of an earthly geographic flavor and there are numerous locations where you can find free geographic data to load your database with. Geographic data that covers large areas around a spheroidal earth needs a little special care. You'll need to understand, at least on a rudimentary level, ellipsoids, datums, and projections to be able to understand the pros and cons of each spatial reference system and which ones are suitable for your use case. We hope the basic fundamentals of these we provided in the last chapter are sufficient to help you handle working with real data.

Before we begin our exercises, we need to know where we can get free data to play with. Locations for free data can be found in Appendix A. In the next section, we will briefly cover some of the more popular free and open source tools both packaged and not packaged with PostGIS that are useful for importing and exporting data.

## **7.1 Tools for importing/exporting data**

There are many tools for getting data in and out of PostgreSQL/PostGIS. For this chapter, we will only be focusing on the more common free and open source ways of doing this.

### **7.1.1 PostgreSQL built-in tools**

PostgreSQL has some built-in command line tools useful for getting data in and out of PostgreSQL.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- **psql** - PostgreSQL psql is a command line tool that has both a non-interactive as well as an interactive interface.
  - In the interactive connection – you can use the COPY command as well as various similar tools for loading in delimited text such as comma delimited or tab delimited data and doing interactive queries as well as recording your session. The built-in copy in psql copies from/to the client's file system (machine psql is being launched from). The SQL COPY command always assumes files are copied from and to the PostgreSQL server.
  - The non-interactive mode allows you to batch load data and also run .sql scripts. It is also needed by the PostGIS shp2pgsql tool we shall cover shortly – for running the generated sql lines. Again the file location is assumed to be a location on the client that is running the psql executable.
- **PgAdmin III** – this is a graphical interface tool packaged with PostgreSQL and also available as a separate install via the <http://www.pgadmin.org> site. It can only run on machines with a graphical interface e.g. Mac OSX, Windows, Linux/BSD Unix with GNome or KDE etc. It has similar functionality to psql, but does not support a client side COPY command. You can only use the sQL built-in COPY. Via the PgAdmin plugin's interface, you can launch a psql session and it will automatically fill in the credentials of the database you are connected to in PgAdminIII in the psql terminal.
- **pg\_dump/pg\_dumpall/pg\_restore** – If ever you want to distribute large amounts of data to other people using PostgreSQL or just for simple backup and restore to other databases, then these are the tools you want to use. They will dump out data even of a spatial nature and can dump out in compressed format to save space. pg\_restore can then be used to restore these tables, functions etc. to another PostgreSQL database.

We have various PostgreSQL cheat sheets for psql and pg\_dump/pg\_dumpall/pg\_restore on our Postgres OnLine Journal site. <http://www.postgresonline.com/specials.php>

### **7.1.2 PostGIS packaged tools**

PostGIS comes packaged with 3 useful tools for loading and outputting spatial and DBF dbase data. If you have PostGIS installed, these tools will be available in the PostgreSQL bin folder.

- **shp2pgsql** – this is a command-line tool that is used to import both plain Dbase files as well as ESRI Shape files. It has good support for converting dbase datatypes to PostgreSQL ones, but lacks transformation capabilities. It can also output to an intermediary .sql file for later processing. It gives you the option of maintaining case of field names or lowercasing them to keep with PostgreSQL standard. By default all fields are lower-cased. You also have the option of just importing DBF files or DBF file part of a esri shp/dbf

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

combination.

- **shp2pgsql-gui – (this is GTK based and not always packaged in)** this is a graphical wizard version of the shp2pgsql tool that was introduced in PostGIS 1.4 and for windows users is available as part of Stackbuilder installer as of PostGIS 1.5 or available as a separate download from postgis.org site. It is used to import both plain Dbase files as well as ESRI Shape files. It is designed for one-off imports or for people afraid of command lines. The downside is that it is not scriptable. It is friendlier to use for people new to PostGIS or if you just need to do a quick import and don't want to figure out data paths and so forth needed by command line. You can also install it as a plugin in PgAdmin III by editing the plugin.ini file. Check our instructions for details on installing as a PgAdmin III plugin <http://www.postgresonline.com/journal/index.php?archives/145-PgAdminShapefilePlugin.html>
- **pgsql2shp –** this is a command line tool that is useful for outputting PostGIS spatial data to ESRI Shapefile format. Also outputs the .prj (spatial reference system file) as of PostGIS 1.3.6. It can output both an ad-hoc query as well as a view or table. The ad-hoc queries can be of any complexity. While it is designed for spatial data, it can also be used to output non-spatial data to Dbase DBF if no geometry field is included. Is fairly light-weight so is handy for creating web apps that can output queries to ESRI shapefile format. We'll cover this use in a later chapter.

You can find a cheat sheet for getting up to speed with these tools on our BostonGIS site [http://www.bostongis.com/pgsql2shp\\_shp2pgsql\\_quickguide.bgg](http://www.bostongis.com/pgsql2shp_shp2pgsql_quickguide.bgg)

### **7.1.3 OGR2OGR: All-Purpose vector data loader**

OGR2OGR is the swiss army knife for vector data. Its companion is Geospatial Data Abstraction Library (GDAL) which is used for loading and outputting raster data. OGR is also often referred to as GDAL/OGR since OGR is now packaged as a subset of GDAL. It can be used to import and export countless vector and non-spatial data formats into PostgreSQL/PostGIS. Its strengths and weaknesses are as follows:

Strengths:

- Supported on both Windows and Linux/Unix/MacOSX
- Supports a myriad of formats (almost anything under the sun including non-spatial data sources) and in many cases it can both read and write to these. The most common are: PostgreSQL/PostGIS, Dbase, ESRI ShapeFile, ESRI Personal Geodatabase, MapInfo, MySQL (both spatial and non-spatial data), Any ODBC data source via ODBC driver, SQLite (newer versions can support SpatiaLite which is a spatial extender for SQLite), GML, KML, GPX, and GeoRSS, (read from ArcGIS SDE, FME, or Oracle Spatial if you compile with the proprietary dlls from these companies).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- Fairly light-weight though not as light as shp2pgsql or pgsql2shp. While there is an install, you can for the most part run by copying the needed binaries etc. into a folder and running from the folder which allows it to be callable from web apps and anywhere without install.
- Supports transformation – if you are not content with the native spatial reference of a data source, you can transform it to a different one as part of the import process. Note this is not the case with shp2pgsql for versions PostGIS 1.5 and below.
- Can do ad-hoc SQL queries, though when doing ad-hoc SQL, the field format is even more impoverished – often not respecting the data field lengths. So again for outputting to ESRI Shapefile format, pgsql2shp is usually a better option.
- Can do batch importing by specifying a folder or database
- Can often guess at spatial reference based on .prj or for mapinfo the in-built projection information and so forth

Weaknesses:

- Not packaged with PostGIS so not always available, though you can download the source or the pre-compiled binaries. There is a nice install for Windows users. The linux pre-compiled binaries tend to be a bit out of date, but windows ones are most always up to date.
- Its not as light-weight as pgsql2shp or shp2pgsql so if you just need to output ESRI Shapefiles in a web app, then pgsql2shp is generally a better option.
- Somewhat impoverished with datatypes. For example shp2pgsql is generally a better tool for loading dbf and Shapefiles because it maintains string length and data type better than does OGR2OGR as of this writing. OGR2OGR seems to like to bring in dbase fields as character instead of varchar (which technically they are but not ideal for most use cases).
- Even for non-spatial data, OGR2OGR seems to insist on registering the table in geometry\_columns.
- The options it offers are overwhelming and sometimes hard to figure out.
- Kind of weak as far as dealing with text encodings, though this varies from driver to driver

### **GDAL EXPORT POSTGIS WKT RASTER**

The latest version of GDAL and the windows FWTools 2..4.6+ binaries come packaged with a driver called PostGIS WKT Raster. This allows you to export data from the PostGIS raster type into other raster formats using the gdal\_translate executable. We will be covering this in the PostGIS WKT Raster chapter.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

One tool that looks promising is ogr2gui [http://www.inventis.ca/ogr2gui/index\\_en.htm](http://www.inventis.ca/ogr2gui/index_en.htm) . This tool doesn't support too many formats yet, but what it does is provide a simple ogr builder interface that asks for input, output, projection, select etc, and provides you with the magical ogr2ogr command to use to accomplish what you are trying to do. We won't cover it here since it currently doesn't support PostgreSQL/PostGIS as a wizard option, though it may in the future.

### **7.1.4 QuantumGIS - Shapefile to PostGIS import tool (SPIT)**

QuantumGIS is a common free desktop tool used for viewing and editing geometry and raster data. It has lots of python scripting capabilities and integration with other tools such as Geographic Resources Analysis Support (GRASS). It works on Windows/Linux/Unix and Mac OSX. We will cover it in our later section on desktop tools. In addition to its other features, it has an easy to use GUI shapefile import plugin just for PostGIS called Shapefile to PostGIS Import Tool (SPIT). Below is a screen shot and is fairly self-explanatory. It is similar in concept to the shp2pgsql-gui tool and probably a good one to use if you are a heavy QuantumGIS user.

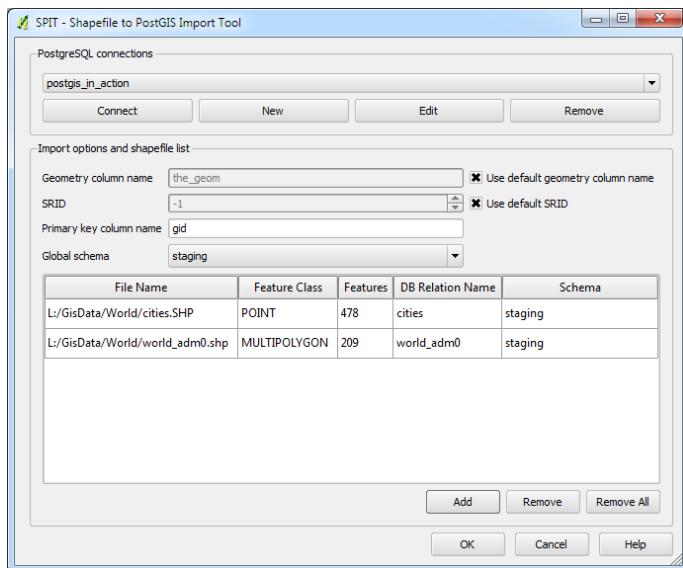


Figure 7.1 Quantum GIS Shapefile to PostGIS Import Tool (SPIT)

In the above snapshot we demonstrate what importing 2 files looks like. This is using QuantumGIS 1.5.0 version.

Strengths:

- Supported on Windows,Linux/Unix/MacOSX

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- You can add multiple files at once by clicking the add button, though all the files you add must be of the same spatial ref
- Gives a brief summary of each file before load – like count of geometries and geometry type.

Cons:

- Not packaged with PostGIS so not always available, need to install QuantumGIS
- No support for transformation
- As of this writing, it brings all the field names in as upper case (basically the field casing of the dbf file) and there doesn't seem to be a way to disable this. This is terribly annoying since PostgreSQL non-lowercase field names need to be quoted when used in SQL statements. This most likely will change in future versions as many have complained about it.

One workaround for this is outlined here though be forewarned the solution involves updating system tables which is generally a bit of a no-no since you can't be guaranteed tables won't change from version to version and if you screw up you could destroy your database - though its much shorter to write <http://workshops.opengeo.org/stack-intro/postgis.html#postgis>

### **Listing 7.1 Generate ddl to rename columns**

```
SELECT array_to_string(ARRAY(SELECT 'ALTER TABLE ' || 
quote_ident(c.table_schema) || '.' ||
    || quote_ident(c.table_name) || ' RENAME "' || c.column_name || 
'" TO ' || quote_ident(lower(c.column_name))
FROM information_schema.columns As c
WHERE c.table_schema NOT IN('information_schema', 'pg_catalog')
    AND c.column_name <> lower(c.column_name)
ORDER BY c.table_schema, c.table_name, c.column_name
),
' ; ' || E'\r') As ddlsql;
```

The ddl example will generate a line for each column that looks like below that you can quickly inspect and run.

```
ALTER TABLE staging.statesp020 RENAME "AREA" TO area;
ALTER TABLE staging.statesp020 RENAME "PERIMETER" TO perimeter;
```

- Only supports ESRI shape files, and can't guess at srid from reading the .prj like ogr2ogr can.

Now that we have covered the more common free options available for loading data, in the next section we will test drive these tools.

In many cases, you will load data in its native form, but often times depending on your use case, you will want to store your data in a different spatial reference system from its

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

native form and do some additional spatial massaging. In either case you will want to know the spatial reference system of your input data and what are the pros and cons of keeping it in this native form. For that discussion you will want to refer to the section on spatial reference system and how to figure out what the SRID is of your input data.

### **7.1.5 Osm2PgSQL: OpenStreetMap to PostGIS loader**

OpenStreetMap is an exciting project that not only makes spatial data available free of charge via mapping web services -- similar to Google Maps and MS Virtual Earth, but also has tools for importing this data into a PostGIS spatial format using the OSM2PGSQL tool. Having the data in your own local PostGIS database is useful for more advanced querying or if you want to manage your own services.

As of this writing the data provided by OpenStreetMap is licensed under an open source license called Creative Commons Attribution-ShareAlike 2.0. This license is expected to change soon to the in-development license known as Open Database License. This new license is more specifically geared toward the sharing of data. In the next section of this chapter, we'll cover how to carve out specific areas of Open Street Map data and download it in OSM XML format and then how to import this file into your PostGIS enabled database.

## **7.2 Loading Data**

In this section we shall go over some real use-cases with the aforementioned tools and focus on loading data. We will start off with the built-in PostgreSQL/PostGIS tools and use that to load an ESRI shapefile.

Before we start we will create some schemas to hold our data. In real case scenarios, you may want to create several schemas to logically partition your data. The PostGIS tables such as geometry\_columns and all the PostGIS functions, you can keep in the public schema. These will be shared across all schemas your data resides in.

For starters, we shall create some schemas we will use later on. The following exercises will assume you have created a spatially enabled database called postgis\_in\_action and we will use that to store all our data.

### **Listing 7.2 Create custom schemas**

```
CREATE SCHEMA us; -- for holding US centric data
CREATE SCHEMA canada; -- for holding canada centric data
CREATE SCHEMA staging; -- this is a schema to hold temporary data
ALTER DATABASE postgis_in_action SET search_path=public,"$user",us,canada;
```

### **Use non-public schemas for your data and custom functions**

It is a good idea to get into the habit of using your own custom schemas for your data instead of throwing everything in public. If you build a lot of your own custom functions, you may also want to store them in your custom schemas and even set aside a schema

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

you create just to hold functions. By using named schemas for your own custom functions and data, it will make upgrading to newer PostGIS installs easier and also just restoring selective data and functions to another database much less painful.

We think logically partitioning your data and functions is a good idea, however we don't think it necessary to schema qualify commonly used things when querying. In fact you may not want to just for ease of use and portability. To get around having to schema qualify, make sure to add your commonly used schemas to your database search path as we covered earlier. Less used schemas or schemas you set aside to hold temp data as we are doing with our staging schema, you probably want to always schema qualify and leave out of your database search path.

### **7.2.1 Getting and Extracting compressed files**

#### **WGET – FOR DOWNLOADING FILES**

Wget is a tool that generally comes prepackaged with Linux/Unix systems. It is a command line tool that can grab files. It can be downloaded for free for windows as well and can be downloaded from <http://gnuwin32.sourceforge.net/packages/wget.htm>. Get the binaries and the dependencies and extract to same folder.

If you are on windows or any OS with a GUI, you can also download files using your browser, however you may find wget handy for automating the download of many files even if you do have a graphical interface to your OS.

Below are some handy tips for making good use of WGET. These apply whether you are on Unix/Linux or Windows.

```
cd /gisdata
wget http://www2.census.gov/geo/tiger/TIGER2009/72_PUERTO_RICO/ --
no-parent --relative --recursive --level=2 --accept=zip,txt --
mirror --reject=html
```

The above code snippet will download all the Puerto Rico zip files into the folder gisdata. Wget maintains the folder structure of the ftp/http site so the folder structure created on your disk will be www2.census.gov/geo/tiger/TIGER2009/72\_PUERTO\_RICO. The other nice thing about the mirror option, is that it will not redownload a file if you already have it. This is great if you lose internet connection, you can just pick up where you left off.

```
wget
http://www2.census.gov/geo/tiger/TIGER2008/tl_2008_us_zcta500.zip
```

The code to download a single file will put the file in the current directory you are in. It will not create subfolders like the aforementioned mirror example. If you are downloading from an ftp, you can also use a wildcard such as \*zcta500.\*.

**EXTRACTING FILES**

Most files you will download are often compressed in tar.gz or zip. Most Linux systems have command line tools to extract this. We'll quickly go over the basics for those new to Linux as well as some useful tips for windows users.

**WINDOWS 7-ZIP**

For Windows users, we highly recommend using the free 7-zip extract/compress tool. 7-zip is free to use for both personal and commercial and can extract all the aforementioned formats plus more. For simple .zip files, you can also use the built-in uncompress in windows. The 7-zip uncompress/compress we have found to be better than the built-in windows one because it can handle compressing/extracting bigger files over 4 gigs and gives you many compression feature options such as password protection and level of compression. You can download 7-zip from : <http://www.7-zip.org/> and after you install, you can simply right-click on a file in windows explorer and choose to extract with 7-zip.

Although most people think of 7-zip as a nice gui tool for extracting various compression formats, it also has a handy command-line interface which is very useful for automating many zip unzip processes. The command line interface is the 7z.exe file. To make this portable – you can just copy the 7z.exe and 7z.dll file to a floppy or usb or folder and use from anywhere without install. Below are some simple tips for using the 7z command line interface.

**Table 7.1 Example uses of 7z command line**

---

**Example uses**

```
# Extract single file in same directory – tar.gz (the first creates the .tar and second extracts the tar)
7z e statesp020.tar.gz
7z x statesp020.tar -o"C:\gisdata\states"

# Extract all zip files in current folder to a new folder called extracteddata – use flat folder structure
7z e C:\gisdata\*.zip -oC:\gisdata\extracteddata

# Extract all zip files in current folder to a new folder called extracteddata – keep same folder structure as
in archive
7z x *.zip -y -oC:\gisdata\extracteddata

# Extract all zip files in current folder to a new folder called extracteddata – keep same folder structure as
in archive, and recursively search for .zip files
7z x *.zip -y -oC:\gisdata\extracteddata
```

**LINUX/UNIX**

For people who happen to find themselves dealing with linux/unix servers, below are some common commands to extract the most common types of files

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

**Table 7.2 Using uncompress tools in Linux****Linux examples**

---

Unzip a single zip file

```
unzip somefile.zip
```

Unzip all zip files in folders recurse down and put in same folder  
`for z in */*.zip; do unzip -o $z; done`

Unzip a single tar file and extract its contents (we have 2 variants below)

```
tar xvzf somefile.tar.gz
```

```
gzip -d -c somefile.tar.gz | tar xvf -
```

**7.2.2 Using PostGIS and PostgreSQL tools to load data**

PostGIS comes packaged with two command line tools and one GUI tool that are useful for loading/outputting ESRI Shapefiles as well as plain Dbase DBF files. In this section we shall walk through the following approaches.

- Load single ESRI Shape file and dbf files with shp2pgsql command line
- Quick demo of the shp2pgsql GUI

**LOADING DATA WITH SHP2PGSQL**

If you launch shp2pgsql from the command line without any arguments, the help screen comes up.

The most important switches to keep in mind are

- **-s** the spatial reference system. If you are dealing with geographic data, you should always provide this especially if you are going to be loading data from various sources
- **-W** the encoding. The default is ASCII and in many cases will work but in other cases you will silently lose data when a blip screen goes by saying failed encoding incorrect something or other. If your data has spanish, italian, french, and other diacritical marks, a better choice is LATIN1. LATIN1 tends to work for most data sets we have come across, and even if the data is coded in ASCII, it is generally harmless to specify LATIN1 encoding.
- **-I** this is useful if you will not be appending future data to this table, and you want the process to go ahead and add a spatial index after loading.

For this section we shall load an ESRI shapefile data set into PostgreSQL using the command line shp2pgsql loader. For this exercise we have chosen the following data source  
 State boundaries from U.S Geological Survey Earth Science Information center.

<http://edcftp.cr.usgs.gov/pub/data/nationalatlas/statesp020.tar.gz>

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

1. Extract the above file using your uncompression tool of choice
2. You should end up with statesp020.dbf, .shp, .shx, .txt\* [ESRI shape files may also contain a .prj file that describes the projection of the data as well as an XML file instead of .txt file to convey more information about the data.]

**Figure out spatial reference system:** For this dataset we know the spatial reference system of this data is NAD 83 long lat which has a magical EPSG code and PostGIS SRID number of 4269. How we arrived at this magical conclusion and how you can determine the same things for your data, we describe in a later section of this chapter on determining the SRID for your source data.

3. Next load up your data using shp2pgsql. We will be loading the states data into a named schema called *us* we created earlier. For windows users, you will normally find the binaries located in C:\Program Files\PostgreSQL\8.4\bin. For this exercise – you can cd into the bin folder of PostgreSQL something like

```
cd "C:\Program Files\PostgreSQL\8.4\bin" (just for windows users)
```

On Linux installs, it ends up being in the path so you can simply use them without specifying the full path. The below should be run as a single line

```
shp2pgsql -s 4269 -g the_geom_4269 -I -W "latin1"  
"C:\GISData\statesp020" staging.statesp020  
| psql -h localhost -p 5432 -d postgis_in_action -U postgres
```

- In the above listing we are loading up our statesp020 data we extracted into a folder called C:\GISData folder and denoting to put it in a new table in the staging schema in the statesp020 table. We don't bother putting in the .shp or any other extension, though you can if you prefer. The shp2pgsql will use both the .shp and the .dbf files and store dbf attributes in common field types and the .shp geometry information in a field called the\_geom\_4269 that we noted with the -g switch. If you don't specify a -g switch, then the geometry is stored in a column called the\_geom. We are loading into a schema called staging, so that we can massage it into a form we prefer for our us schema. NOTE: the shp2pgsql part simply generates sql statements to create the table, add the geometry column (so it is registered in geometry\_columns), insert the data into the table, and add the index. It DOES NOT DO ANYTHING REALLY; the work is being done by the psql part. For this example we really didn't need to add an index because we'll be throwing this table out once we are done with it anyway.
- The second part beginning with | is the part that actually does the loading. The | part works equally well on most Windows OS (including Vista) as it does on Linux. What it does is pipes the sql output generated by shp2pgsql to the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

psql command to process.

- If you did not want the data loaded and wanted to just store the .sql file for future loading or to massage before loading you would do the below instead. Dumping to .sql file is convenient if you have some bad items that will fail, or your shape file is considerably large that all the data can't easily fit into memory before it is piped to psql.

```
shp2pgsql -s 4269 -g the_geom_4269 -I -W "latin1"
"C:\GISData\statesp020" staging.statesp020 >
C:\GISData\statesp020.sql

psql -h localhost -p 5432 -d postgis_in_action -f
C:\GISData\statesp020.sql
```

The above raw data format has some issues not suitable for our use case: it is in long lat so not suitable for measurement, it has over 2000 records and we would prefer one for each state, and it is sufficiently dense and more precise than we need. To solve these issues, we take our staged data and dump it into our US schema.

### **IT IS BEST TO USE POSTGIS 1.4 AND ABOVE**

We highly suggest not trying the following exercise if you are running lower than PostGIS 1.4 with GEOS 3.1+. While this should in theory work in lower versions, it will probably take hours if you don't run out of memory. In contrast in PostGIS 1.4+, because of the significantly improved speed at unioning many polygons, this will take about 26 seconds or less.

#### **Listing 7.3 Converting data from native format to more optimized format**

```
-- 1
CREATE TABLE us.states
(
    gid serial NOT NULL,
    state character varying(20),
    state_fips character varying(2),
    order_adm integer,
    month_adm character varying(18),
    day_adm integer,
    year_adm integer
);

-- 2
SELECT AddGeometryColumn ('us','states','the_geom',2163,'MULTIPOLYGON',2);

INSERT INTO us.states(state, state_fips, order_adm, month_adm, year_adm,
the_geom)
SELECT state, state_fips, order_adm, month_adm, year_adm,
-- 3
    ST_Multi(
-- 4
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
ST_SimplifyPreserveTopology(
-- 5
    ST_Union(
-- 6
        ST_Transform(the_geom_4269, 2163)
    ),
-- 7
    700
)
) As the_geom
FROM staging.statesp020
-- 8
GROUP BY state, state_fips, order_adm, month_adm, year_adm;
-- 1 create table
-- 2 create geometry column
-- 3 convert to multipolygon
-- 4 ,7 simplify every 700 meters
-- 5 dissolve boundaries
-- 6 to equal area meters
```

In code listing above, we are (1) creating a new table to store our United States records. (2) We add the geometry column after creating the table so that it gets constraints added and gets registered in `geometry_columns` table. PostGIS 1.4+ does provide additional ways of doing this. (6) We transform our initially loaded data to 2163 space (Lambert Azimuthal Equal Area US National Atlas meters). We then use the PostGIS spatial aggregate function (5) `ST_Union` in combination with our (8) sql `GROUP BY` clause to dissolve boundaries between records of the same state so that we end up with one record per state (this will make our resulting table contain only 53 records instead of the original 2895 records we had in our staging file. After we are done unioning we then simplify the new geometries (4) . (7) The value of 700 in this case is in meters (because since simplification is happening after the transform and union operations our geometry is in SRID 2163 space. We are basically telling PostGIS to remove vertexes such that our geometry has vertexes separated at least 700 meters apart. Simplification is useful for many reasons, but has the cost of making your data a little less accurate. The higher your simplification tolerance, the lighter your geometry will be at the cost of lower accuracy. Simplification is important if you later need to redistribute data via WFS or for download and want to make download speed as fast as possible and it makes processes such as `ST_DWithin` and `ST_Intersects` faster (and in some cases a lot faster) as they have fewer vertices to worry about. It is important to simplify in planar space not long lat space since the same degree change means different length which results in very choppy simplification. If you want to keep your data in long lat, you should probably transform it to planar space, simplify and then retransform back to long lat space. After we are done with all of that we then apply (3) a PostGIS `ST_Multi` operation. Recall from prior chapters – this converts a polygon to a multipolygon. We do this since some states after unioning will be polygons and some will be multipolygons so to maintain consistency we make them all multipolygon. In short, the order we apply our operations

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

above we have found to be the best way to ensure good quality data and some of the operations absolutely need to be applied in that order.

### SIMPLIFICATION IS GENERALLY CHEAP

The simplification process in PostGIS is pretty fast, so in many cases such as when servicing regions selected by a user, it is often fast enough to simplify on the fly and base your simplification tolerance on how zoomed in the user is into the data. For large datasets and for indexing reasons, you may want to keep pre-built simplified versions of the data as well as the more high-grained data.

Once you are done loading the data, it is a good idea to at lease put a spatial index on the new data and possibly other indexes you know you will commonly use in the WHERE clauses of your sql statements. The below exercise should be more or less a refresher course from past chapters.

#### **Listing 7.4 Putting in indexing and preparation for querying**

```
-- 1
CREATE INDEX idx_us_states_the_geom ON us.states USING gist (the_geom);

-- 2
ALTER TABLE us.states
ADD CONSTRAINT pkey_us_states_state_fips PRIMARY KEY(state_fips);

-- 3
CREATE UNIQUE INDEX uidx_us_states_gid
ON us.states USING btree (gid);

CREATE UNIQUE INDEX uidx_us_states_state
ON us.states USING btree (state);

-- 4
VACUUM ANALYZE us.states;

-- 5
SELECT DropGeometryTable('staging', 'statesp020');
-- 1 spatial index
-- 2 primary key
-- 3 unique keys
-- 4 purge dead rows
-- 5 drop temp table
```

(1) First we create the spatial index on our new data set. (2) Then we add a primary key. We are using state\_fips since a lot of US data comes with this code so probably the best to join with if you ever need to or alternatively you can use state etc. (3) Here we add a dummy key (which is our serial key) and make it unique. This is mostly to appease many GIS tools which have the requirement that a key be an integer. You might need to make this a primary key for some tools, which many database purists will find incredibly annoying

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

because it has no business meaning. Next we add other keys we will use in queries. We highly suggest not doing this step until you have started doing queries against this table and benchmarked how you are using this. What we are doing here in technical speak is called *premature optimization* which is in general a bad thing unless you have great insight as to how the data will be used; we know most likely a spatial index will always be needed so we always add it. We are doing the btree indexes early because it's a static table (will probably never be updated or added to) and we wanted to show how to do this. For tables that are frequently updated/added, indexes are not free because processing/IO time is spent updating them when data changes. (4) Finally we vacuum analyze our new table so the planner statistics are up to date and (5) drop our staging table – no need to keep junk around. The drop will drop the table and remove all entries of it in the geometry\_columns table.

#### QUICK DEMO OF SHP2PGSQL-GUI

Below is a screen shot of what the shp2pgsql-gui looks like when trying to load the states table as we did in the command line version.

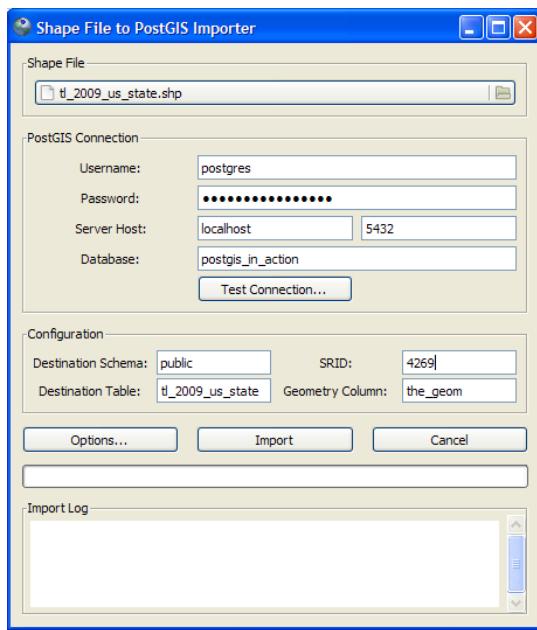


Figure 7.2 Using shp2pgsql-gui to load states table

In the above figure we have browsed to the states table using the browser icon and filled in the relevant information. We then will click the options button to verify the other settings.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

The character encoding of many shape-files is usually LATIN-1 or WINDOWS-1252 (a variant of LATIN1 with additional windows characters).

### **POSTGIS SHP2PGSQL-GUI IN PGADMIN III**

The PostGIS shp2pgsql-gui is designed so that it can be deployed as a Plugin for PgAdmin III. For instructions on doing this, check out our article on the topic - <http://www.postgresonline.com/journal/index.php?archives/145-PgAdmin-III-Plug-in-Registration-PostGIS-Shapefile-and-DBF-Loader.html>

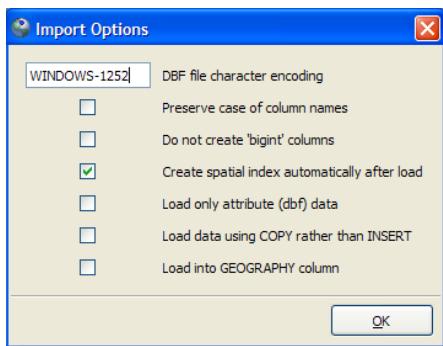


Figure 7.3 shp2pgsql-gui options button dialog showing the advanced options

### **POSTGIS 2.0 SHP2PGSQL-GUI ENHANCEMENTS**

In PostGIS 2.0, the GUI has been enhanced to allow loading of multiple files at once similar to what you can do with the QGIS SPIT plugin.

In this section we covered how to load spatial data with the pre-packaged tools provided with PostGIS. In the next section, we shall demonstrate using OGR2OGR to load in spatial data from various different kinds of spatial datasources.

#### **7.2.3 Loading data with OGR2OGR**

As mentioned in our quick survey, while shp2pgsql is nice for loading ESRI Shapefiles and Dbase files, that is all you can use it for. If you have MapInfo, GPX, ODBC, MySQL, SQL Server, ESRI Personal Geodatabase, AutoCAD files among many other formats, then OGR2OGR will do the trick for you.

OGR2OGR is supported both on Linux/Unix as well as Windows and Mac OSX. You can download for your particular OS from <http://trac.osgeo.org/gdal/wiki/DownloadingGdalBinaries>

Formats supported are listed here [http://www.gdal.org/ogr/ogr\\_formats.html](http://www.gdal.org/ogr/ogr_formats.html)

Some of the formats that require proprietary dlls such as Oracle Spatial, ESRI ArcSDE, and FME are not compiled in by default and require you compile yourself with the dependency libraries.

The most common package that contains OGR2OGR is called FWTools. There is a version for both Linux as well as windows. To get a list of formats that your install supports, launch the FWTools command line (for windows users, for Linux just add to your search path or cd to the folder) and then type

```
ogr2ogr --formats
```

The more common installed formats are listed below

#### **Listing 7.5 OGR2OGR supported formats list ogr2ogr --formats**

```
"ESRI Shapefile" (read/write), "MapInfo File" (read/write),
"UK .NTF" (read-only), "SDTS" (read-only), "TIGER" (read/write),
"S57" (read/write), "DGN" (read/write), "VRT" (read-only), "REC" (read-only),
"Memory" (read/write), "BNA" (read/write), "CSV" (read/write),
"NAS" (read-only), "GML" (read/write), "GPX" (read/write),
"KML" (read/write), "GeoJSON" (read/write),
"Interlis 1" (read/write), "Interlis 2" (read/write),
"GMT" (read/write), "SQLite" (read/write), "ODBC" (read/write),
"PGeo" (readonly), "OGDI" (readonly), "PostgreSQL" (read/write),
"MySQL" (read/write), "XPlane" (readonly),
"AVCbin" (readonly), "AVCE00" (readonly),
"Geoconcept" (read/write), "GeoRSS" (read/write)
```

In the exercises that follow we shall demonstrate loading data from GPX, ESRI Personal Geodatabase (PGeo -- MDB) , and MapInfo to PostgreSQL.

OGR2OGR given its size is an extremely rich tool. We would have to devote a whole book to it to do it justice. Hopefully the samplings we have picked are the most common use cases people will need.

For other common types feel free to check out examples on our satellite sites:

- Examples of non-spatial data loading:  
<http://www.postgresonline.com/journal/index.php?archives/31-GDAL-OGR2OGR-for-Data-Loading.html>
- Additional spatial data loading and install for windows:  
[http://www.bostongis.com/PrinterFriendly.aspx?content\\_name=ogr\\_cheatsheet](http://www.bostongis.com/PrinterFriendly.aspx?content_name=ogr_cheatsheet)

Before we begin our OGR journey, we have outlined some options that are specific to working with the OGR PostgreSQL driver that you will find useful regardless of what data source you are importing from.

#### **POSTGRESQL LAYER CREATION OPTIONS**

The -dsco and -lco options are specific to the driver in use. PostgreSQL has the following layer creation options. Most of the below is copied from the official OGR2OGR documentation with some additional comments and some rarely used options left out. Full details here  
[http://gdal.org/ogr/drv\\_pg.html](http://gdal.org/ogr/drv_pg.html)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- **GEOM\_TYPE:** The GEOM\_TYPE layer creation option can be set to one of "Geometry", "BYTEA" or "OID" to force the type of geometry used for a table. In general there is no need to set this.
- **LAUNDER:** This may be "YES" to force new fields created on this layer to have their field names "laundered" into a form more compatible with PostgreSQL. This converts to lower case and converts some special characters like "-" and "#" to "\_". If "NO" exact names are preserved. The default value is "YES". If enabled the table (layer) name will also be laundered.
- **PRECISION::** This may be "YES" to force new fields created on this layer to try and represent the width and precision information, if available using NUMERIC(width, precision) or CHAR(width) types. If "NO" then the types FLOAT8, INTEGER and VARCHAR will be used instead. The default is "YES".
- **GEOMETRY\_NAME:** Set name of geometry column in new table. If omitted it defaults to wkb\_geometry. If you are not happy with the default name of wkb\_geometry and for example want your fields to be named to the\_geom to be in line with the shp2pgsql driver, then you would do something like this  
-lco GEOMETRY\_NAME=the\_geom
- **SCHEMA ::** Set name of schema for new table. Using the same layer name in different schemas is supported, but not in the public schema and others.

#### **POSTGRESQL/OGR2OGR ENVIRONMENT VARIABLES**

These are variables that you can't pass as part of the command line but can be controlled with environment variables. On Linux/Unix, you can set these by using the export command

```
export PGCLIENTENCODING=latin1
export PG_USE_COPY=yes
```

On windows you can set this by going into control panel -> System ->Settings ->Advanced and clicking the Environment Variables.

- **PGCLIENTENCODING** – this is really a PostgreSQL environment variable but it overrides OGR's default of UTF-8. So all data you import will be assumed to be in this encoding if specified.
- **PGSQL\_OGR\_FID** – this controls the name of the dummy primary key OGR sets up. By default ogr calls it ogc\_fid. This for some reason never works for us.
- **PG\_USE\_COPY** - This may be "YES" for using COPY for inserting data to PostgreSQL. The docs say COPY is less robust than INSERT, but significantly faster. That may have been true in older versions of PostgreSQL. We like to set this to YES as well. In many cases we have found it not only faster, but more robust than INSERT.

### **EXERCISE 1: LOADING A GPS EXCHANGE FORMAT (GPX) FILE**

GPX files are the standard transport format for GPS generated data. GPX data is an XML format, so you can also use the built-in XML functionality in PostgreSQL if you need much finer grain control or want to do everything in the database. We cover that in <http://www.postgresonline.com/journal/index.php?archives/116>Loading-and-Processing-GPX-XML-files--using-PostgreSQL.html>.

GPX data is always in WGS 84 long lat which has a magical PostGIS SRID/EPSG number of **4326**. OGR2OGR is smart enough to know that so it puts in the correct SRID for you. For more details about command line switches specific to the OGR GPX driver check out - [http://www.gdal.org/ogr/drv\\_gpx.html](http://www.gdal.org/ogr/drv_gpx.html).

OpenStreetMap is full of user contributed GPX files that are uploaded by users every minute. You can find these at <http://www.openstreetmap.org/traces>. We have randomly selected one from Australia by going here <http://www.openstreetmap.org/traces/tag/australia> entitled "A bike trip around Narangba" which we downloaded from <http://www.openstreetmap.org/user/Ash%20Kyd/traces/468761>.

OGR2OGR comes with a utility called ogrinfo which gives you a summary about a file or set of files. Below is what we get when we do ogrinfo 468761.gpx

#### **Listing 7.6 Displaying ogrinfo about GPX file**

```
--1
ogrinfo 468761.gpx

--2
Had to open data source read-only.
INFO: Open of `468761.gpx'
      using driver `GPX' successful.
1: waypoints (Point)
2: routes (Line String)
3: tracks (Multi Line String)
4: route_points (Point)
5: track_points (Point)
--1 command
--2 results
```

We'll now load this up into our staging schema with the following simple OGR2OGR commands:

#### **Listing 7.7 Loading data from GPX**

```
--1
ogr2ogr -f "PostgreSQL" PG:"host=localhost user=postgres port=5432
dbname=postgis_in_action password=mypassword" 468761.gpx -overwrite -lco
GEOMETRY_NAME=the_geom -nln "staging.aus_biketrip_narangba"

--2
ogr2ogr -f "PostgreSQL" PG:"host=localhost user=postgres port=5432
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
dbname=postgis_in_action password=mypassword" 468761.gpx -overwrite -lco
GEOMETRY_NAME=the_geom -lco SCHEMA=staging tracks track_points
-- 1 single table load
-- 2 multi table load
```

(1) Just does a simple load into a newly named table we call staging.aus\_biketrip\_narangba. This table contains all the layers so has a lot of blank fields to accommodate the attributes of the various layer types. We also are specifying the geometry\_name column field. If you leave this out, geometries are stored in a field called wkb\_geometry. (2) In the second approach, we are taking the same gpx file, but breaking it out into separate tables by feature type. We are also pulling only a subset of the layer types available. Many of the others for this GPX file are empty.

In the next exercise we shall import a layer from an ESRI Personal Geodatabase file and will also demonstrate the power of OGR2OGR to reproject data.

#### **EXERCISE 2: LOADING AN ESRI PERSONAL GEODATABASE**

The ESRI Personal Geodatabase format is really a Microsoft Access database with geometries stuffed in blob fields and some meta data tables added to maintain information about these geometry tables. The personal geodatabase is nice in the sense that you can hold a number of layers in one file but is limited to 4GB in size. It is reaching obsolescence however and is slowly being replaced by ESRI's File Database format which can handle larger file sizes, but is more proprietary (non-published standard) and few tools aside from ESRI made ones know how to deal with it. OGR2OGR currently supports reading of ESRI's Personal Geodatabase, but not the new file database format. For this exercise we will download the personal geodatabase of world administrative boundaries from here - [http://biogeo.berkeley.edu/gadm/data/gadm\\_v0dot9\\_mdb.zip](http://biogeo.berkeley.edu/gadm/data/gadm_v0dot9_mdb.zip) (this is a 389 MB zip file and 800 mb extracted as mdb). To get a catalog of what is in our treasure chest of a personal mdb, we use the trusty ogrinfo tool:

#### **Listing 7.8 Use ogrinfo to list out layers in personal geodatabase**

```
ogrinfo gadm_v0dot9.mdb
# result below
INFO: Open of `gadm_v0dot9.mdb'
      using driver `PGeo' successful.
1: gadm
```

A personal geodatabase can have many layers/features/tables, but this one happens to have only one layer. To find out more about this layer we can specify the layer in ogrinfo clause. For the below since this is a large database, it is not instantaneous for this to return back. We are using the -so switch to let OGR know we only want summary data we don't want to actually see the data in the records. The -geom=YES is generally not needed, but in some cases you may not care about the hefty geometry if you want to list out the data.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

**Listing 7.9 Use ogrinfo to list of fields for a personal geodatabase layer**

```
-- 1
ogrinfo gadm_v0dot9.mdb -so -geom=YES gadm
-- 2
INFO: Open of `gadm_v0dot9.mdb'
      using driver `PGeo' successful.

Layer name: gadm
Geometry: Unknown (any)
Feature Count: 116996
Extent: (-180.000015, -90.000000) - (179.999999, 83.627419)
Layer SRS WKT:
GEOGCS["GCS_WGS_1984",
  DATUM["WGS_1984",
    SPHEROID["WGS_1984",6378137.0,298.257223563]],
  PRIMEM["Greenwich",0.0],
  UNIT["Degree",0.0174532925199433]]
OBJECTID: Integer (10.0)
ISO: String (255.0)
NAME_0: String (255.0)
NAME_1: String (255.0)
VARNAME_1: String (255.0)
NL_NAME_1: String (255.0)
:
:
ENGTYPE_5: String (255.0)
VALIDFR_5: String (255.0)
VALIDTO_5: String (255.0)
Shape_Length: Real (0.0)
Shape_Area: Real (0.0)
-- 1 command
-- 2 results
```

From the above (1) we can see the results give us the names of the fields, the sizes of them and also the spatial reference system of the data (WGS 84 longlat – which is magical SRID 4326)). It also tells us the geometry is of a mixed type. Its not all polygons, linestrings etc.

We will now take this data, select just the USA portion of it and bring it into our database transformed to US National Atlas Equal Area.

```
ogr2ogr -f "PostgreSQL" PG:"host=localhost user=postgres port=5432
dbname=postgis_in_action password=mypassword" gadm_v0dot9.mdb -lco
GEOMETRY_NAME=the_geom -where "ISO='USA'" -t_srs "EPSG:2163" -nln
"us.admin_boundaries" gadm
```

In the above we are performing a couple of things: We are selecting just USA boundaries with the ISO='USA' where clause, and we are transforming from the native spatial ref of the data EPSG:4326 to our preferred EPSG:2163 for this subset. We are then bringing this subset into a new table called us.admin\_boundaries that resides in the us schema. In this particular case, OGR2OGR has enough to guess at the source spatial reference system, so

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

we did not have to provide it. In many cases you may need to provide a `-s_srs "EPSG:4326"` or whatever the native is so OGR transforms correctly.

If we tried to load the full dataset above we may run into errors because of the various languages the text are in. We need to set the client encoding of the data to LATIN1 to prevent this. Unfortunately, OGR doesn't have as direct a way of doing this as shp2pgsql does so we set the environment variable as mentioned earlier. So we have to fiddle with environment variables or set our database to client\_encoding latin1 or some other relevant encoding while we are loading.

#### **EXERCISE 3: LOADING A MAPINFO TAB FILE AND FOLDER OF MAPINFO FILES**

Another popular format used is the MapInfo tab file format. MapInfo format has spatial reference info built into the file format, and unlike ESRI shape format, you can have multiple different kinds of geometry types in the same file and field names can be upper/lower/mixed and are not limited in length to 10 characters as are DBF files. It also encodes in it a lot of style formatting (which is ignored by OGR2OGR). For this exercise we shall pull a file from geostatistics Canada <http://www.statcan.gc.ca/mgeo/boundary-limite-eng.htm>

and pull their Population Ecumene Census Division Cartographic Boundary and choose MapInfo tab format. This zip file is composed of several different tab files. For this exercise we shall demonstrate loading a whole folder of files, which is one of the beautiful features with ogr2ogr.

```
ogr2ogr -f "PostgreSQL" PG:"host=localhost user=postgres port=5432
dbname=postgis_in_action password=mypassword" "C:\gisdata\canada"
-lco GEOMETRY_NAME=the_geom -lco SCHEMA=canada -a_srs "EPSG:4269"
```

In the above example we explicitly specified the source spatial reference. If we didn't for this particular file, then OGR2OGR creates a new entry with srid = 32768 and proj4text= "+proj=longlat +ellps=GRS80 +datum=NAD83 +no\_defs ". Why it can't guess sometimes is puzzling and may have to do with the MapInfo driver. If you look at the proj4text – you will see it is identical to the entry for 4269 which is why we force it.

#### **7.2.4 Importing OpenStreetMap data with OSM2PGSQL**

The OpenStreetMap export format, referred to as OSM, is an XML format. You can choose to download and load the whole database, which is currently about 16 GB in size, or use the export web service tool -- <http://www.openstreetmap.org/export/> to carve out a section of space and export out just that section. There are various other command line tools for working with Osm data. One that is specific for working with road networks and the PostGIS pgRouting add-on is osm2pgrouting which can be downloaded from <http://pgrouting.postgis.org/wiki/tools/osm2pgrouting>.

In our Chapter 3 Paris example, we used this interface to export out regions of Paris to load in our database. To export a region of space.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

1. Go to <http://www.openstreetmap.org/export/> and type in the long lat block you want or draw a box on the map
2. We selected a region encompassing Arc de Triomphe.
3. BBOX we chose - 2.28568,48.87957,2.30371,48.8676
4. Then you select the Export format OpenStreetMap XML Data. We called ours arctriump.osm

Once you have a .osm formatted file, you can load that using OSM2PGSQL which can be downloaded from <http://wiki.openstreetmap.org/wiki/Osm2pgsql>

#### **LOADING OSM FORMATTED DATA WITH OSM2PGSQL**

We were installing on Windows so we downloaded and extracted the zip from [http://wiki.openstreetmap.org/wiki/Osm2pgsql#Windows\\_XP](http://wiki.openstreetmap.org/wiki/Osm2pgsql#Windows_XP)

Osm2pgsql has numerous other options we will not explore such as on the fly projection using the -E switch, -ll to bring in long lat. You can get a listing of all of these by calling

```
osm2pgsql -h
```

The rest of the steps are more or less the same regardless of which OS you are on. Note that psql is located in the bin folder of your PostgreSQL install.

Load the 900913 (more or less same as web mercator), spatial reference into your database using the following command using psql:

```
psql -f 900913.sql -d mydb -U postgres -p 5432
```

If you already had this spatial reference system in your database, you wil get an error which you can safely ignore.

If you want to use the key value store feature of PostgreSQL and be able to import the OSM key tags into this structure, you'll need to install hstore contrib.

#### **POSTGRESQL 9.0 HSTORE ENHANCEMENTS**

The hstore contrib from 9.0 on has been enhanced to now support GROUP BY and DISTINCT operations as well as allow larger lengths. Some other functions have been added to the mix that work on it as well.

To install you need to run the hstore.sql that is located in your PostgreSQL /share/contrib folder

```
psql -f hstore.sql -d mydb -U postgres -p 5432
```

Now we are ready to load our .osm formatted data into PostgreSQL. The commands for user name, postgresql port and database are pretty much the same as psql, except that for port (which is only really needed if installing in a PostgreSQL database that is not on the standard port), the switch is upper case P instead of lower case p.

```
osm2pgsql arctriump.osm -d postgis_in_action -U postgres -P 5432 -S default.style --hstore
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

When you are done, you should see a bunch of tables created in the public schema that start with planet\_osm.

### **APPENDING VS. OVERWRITING WITH OSM2PGSQL**

By default, OSM will overwrite the tables and create them fresh. If you are appending multiple OSM files and different points in time, you will want to use the --append switch to switch it to append mode. Note that you can process multiple files at once by separating the file names with spaces -- file1.osm file2.osm etc.

#### **READING HSTORE TAGS**

If you used the --hstore flag as we did above, each table should have a column called **tags** which uses the PostgreSQL key value hstore storage type. Tags can be different for each object, but if you request a tag that doesn't exist, it will return null. This is often referred to as a schemaless design. Most of the key OSM tags are already included as database columns in the osm PostgreSQL output, but querying tags, is useful to get at the more obscure ones that may be particularly useful to you. To demonstrate querying it, suppose you wanted to pull out all the cycleways from the lines table and also have a pipe delimited list of the other keys each has, then you would write a query such as:

```
SELECT name, array_to_string(akeys(tags), ' | ') As keys,
       tags -> 'cycleway' As cycleway
  FROM planet_osm_line
 WHERE (tags -> 'cycleway') IS NOT NULL ;
```

If you wanted to pull out each tag as a separate row, which is more suitable for storage in other relational databases, you could write a query like this which would create a new table called osm\_key\_values consisting of a row that has columns osm\_id, key, value. You would get a record for each key value pair. So if you had 10 entries in each tags column, you would get 10 rows for each row.

```
SELECT osm_id, (foo.e).key, (foo.e).value
  INTO osm_key_values
  FROM (SELECT osm_id, each(tags) As e
        FROM planet_osm_line ) As foo ;
```

The hstore data type can use the GIST index for added performance similar to what you create against PostGIS geometry/geography/raster columns.

As we have demonstrated, there are numerous open source freely available tools available for getting data into your PostgreSQL/PostGIS tools. Many of these tools grew up along side PostGIS and so the PostGIS free importer tools are often more tried and tested and functional than what you will find for other spatial databases. While it is easy to get data into your PostgreSQL/PostGIS database, its just as easy to export data out into various

formats. In the next section, we shall demonstrate how to export data out of your spatial database into formats consumable by various GIS desktop tools and other spatial databases.

## **7.3 Exporting data out of PostGIS**

A database is only as good as the information you can get out of it and the data you can share with others. There are many tools you can use to get data out of your database in a portable format suitable for consumption for field workers or people wanting to explore your data via various desktop GIS tools. A subset of free tools at your disposal:

- PostgreSQL has a myriad of copy commands. One that is part of its SQL offering which will dump data to a server file that the postgres process has access too. This can only be used by super admin. Another copy is part of the psql command line interactive client and will dump the file to the client workstation and doesn't require admin rights to use. psql client in addition has some handy features for making text and html reports for regular tabular data. Since this is a book about spatial data, we will not focus on this though it is useful for generating spatial tabular statistical reports.
- PostGIS comes with a command line tool called pgsql2shp which allows you to dump any data in your database (even plain attribute) as ESRI Shapefiles or plain DBF. It is fairly powerful and light-weight. This tool we shall demonstrate.
- OGR2OGR as we have discussed already, can also export PostGIS and regular PostgreSQL attribute data to various formats. To use it we will merely make PostgreSQL the from source instead of the to source.

In this section we shall cover these tools and how to use them.

### **7.3.1 Using pgsql2shp to dispense PostGIS data**

We like to think of pgsql2shp as a light weight candy dispenser. It only needs a couple of files to be functional (libpq plus libpq dependencies,pgsql2shp) and can output any spatial query you can think of to ESRI shapefile format. It is located in the bin folder of your PostgreSQL install, and when you launch it without any argument, the screen by default gives you a help menu.

As of PostGIS 1.3.6 and above, the pgsql2shp does the following

- Output the following related ESRI files (related files all prefix of -f option .dbf,.shp,.shx,.prj). The .prj file is only output if the projection is known – e.g. you did not use an SRID of -1 or 0 and all the geometries you are outputting are of the same spatial ref.
- If you output a table or query with no geometry column, it will just output a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

.dbf file.

- For field names that are too long (greater than 10 characters per dbf standard), the names will be truncated and dupes will be numbered
- Large text fields greater than 255 characters will be truncated since ESRI format doesn't support Dbase memo fields.

In the exercises that follow, we'll go over some of the common use cases.

#### **EXERCISE 1: EXPORTING A SPATIAL TABLE OR VIEW**

This is the easiest and perhaps most commonly used.

This first snippet exports a whole table in schema ca and table zips in database gisdb to a file called cazips.\* (it will create cazips.shp, cazips.dbf, cazips.shx and for PostGIS 1.3.6+ packaged pgsql2shp, will also output the cazips.prj to denote the spatial reference system of the data).

```
pgsql2shp -f /gisdata/cazips gisdb ca.zips
```

This second snippet does the same as the first accept is needed if your PostgreSQL server doesn't run on the standard port or you want to authenticate as specific user.

```
pgsql2shp -f /gisdata/cazips -h localhost -u pguser -P somepassword -p 5432
gisdb ca.zips
```

While exporting whole tables is a common need, for large tables, you may only want to export a portion of a table or even a complex query. The next example demonstrates outputting results of ad-hoc queries.

#### **EXERCISE 2: EXPORTING AN ADHOC QUERY**

##### **Listing 7.10 Export a query**

```
--1
pgsql2shp -f boszips -h localhost -u postgres gisdb "SELECT * FROM ma.zips
WHERE city = 'Boston'"
```

```
-- 2
pgsql2shp -f boszips localhost -u postgres gisdb "SELECT zip5,
ST_Transform(the_geom, 4326) As the_geom FROM ma.zips WHERE city =
'Boston'"
-- 1 plain ad-hoc
-- 2 ad-hoc with transform
```

We demonstrate in the examples above how to use pgsql2shp to output database queries. (1) is a very basic query. (2) is one that includes an ST\_Transform call. The first query will output a projection that is the native format of the data. The second query will reproject the data and output it in WGS 84 long lat which is the most common of distribution

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

spatial reference systems. From PostGIS version 1.3.6 on will provide .prj files for autoprojection in tools that support reading .prj info such as ArcGIS and MapInfo.

### **7.3.2 Using OGR2OGR to dispense PostGIS data**

If you have the need to output data in other formats without having to do much programming, then pgsql2shp will not do that for you. OGR2OGR however will in most cases be a good free light-weight tool for that purpose. It is not quite as light-weight as pgsql2shp, but it makes up for that by providing tons more formats to choose from.

OGR2OGR like pgsql2shp allows you to output both spatial queries as well as tables and views. The –sql switch seems much more finicky and not as robust as the pgsql2shp query output even with the PostgreSQL driver. It is hard to use randomly complex queries with it as you can with pgsql2shp. The –sql seems to have trouble figuring out data types so in many cases, you would be better off creating a temporary view and outputting the view. We shall show various common formats in the examples below. One unique thing about OGR2OGR which is quite nice, is that you can use it to output multiple spatial tables at once.

As far as generic options go, the most important switches for outputting data with OGR2OGR are the following

- -select – the fields you want to output. No need to include the geometry field here
- -where – the filter condition
- -sql – if you want to output a more complex query than what the –select and –where combo offer, but data types may be more impoverished.
- -t\_srs – this is the output spatial reference system that you want OGR2OGR to output to. If you have SRID encoded in your geometries, then –t\_srs is all you need. However if you have your data in unknown projection (SRID -1 or 0 or something not in the proj list of your OGR2OGR, then you will need to specify the –s\_srs which denotes what projection OGR should assume the source data is in.
- -dsco overwrite=YES most drivers that support write support this data creation option. This tells OGR to destroy the old files if they exist. Useful if you have a nightly scheduled dump where you are constantly overwriting the same files.

#### **EXERCISE 1: EXPORT TO KML**

In this example we shall demonstrate how to output both a table and query to KML format using OGR2OGR. If you want very granular control, you'll probably want to write your own export logic since KML is fairly easy to code up and has lots of styling options not available via OGR2OGR. We'll demonstrate an example of this when we get to web applications. Details about the OGR2OGR KML driver can be found at [http://gdal.org/ogr/drv\\_kml.html](http://gdal.org/ogr/drv_kml.html) .

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

The most important data creation option in the KML driver is the NameField. This is the field that determines which field in your KML output is used for the KML title for each object. It must also be noted that the spatial ref of KML format is always EPSG:4326. If your data is in a known projection, then OGR2OGR will automatically convert it to 4326 for you without having to specify it.

### **Listing 7.11 Export PostGIS table and query to KML**

```
-- 1
ogr2ogr -f "KML" /gisdata/us_adminbd.kml PG:"host=localhost user=postgres
port=5432 dbname=postgis_in_action password=mypassword" us.admin_boundaries
-dsco NameField=name_2

-- 2
ogr2ogr -f "KML" /gisdata/biketrip.kml PG:"host=localhost user=postgres
port=5432 dbname=postgis_in_action password=mypassword" -dsco
NameField=time -select "SELECT track_seg_point_id, ele, time" -where "time
BETWEEN '2009-07-18 04:33:04' AND '2009-07-18 04:34:04'"
staging.aus_biketrip_narangba

-- 3
ogr2ogr -f "KML" /gisdata/biketrail.kml PG:"host=localhost user=postgres
port=5432 dbname=postgis_in_action password=mypassword" -dsco
NameField=time staging.track_points staging.tracks

-- 1 Simple export
-- 2 Export of filtered set
-- 3 Export as multiple tables
```

In the KML exercise, we have demonstrated 3 approaches to using the OGR2OGR to export to KML format. (1) Simple table/view export, (2) filtered export that uses the –sql and –where switches of OGR2OGR, (3) and multiple table export. For the KML format, the multi table export exports all the tables into the same kml file. If you view the KML generated in (3) in Google Earth, you will see 2 layer folders underneath the biketrail.kml file. One for each table. (track\_point and tracks).

### **EXERCISE 2: EXPORT TO MAPINFO TAB**

In this example we shall demonstrate outputting to MapInfo Tab format. Unlike the KML format which is always in WGS 84 long lat, map info data can be in any spatial reference system. In many cases the spatial reference system you store the data in, may not be the one you want to use to distribute in. For the below exercise, we will demonstrate how to get OGR2OGR to transform for you. While MapInfo Tab format is not as popular as ESRI shape format, it has a couple of advantages – it is not constrained by field name lengths and also it can store more than one geometry type in a tab. ESRI shapefile only supports one type (POLYGON/MULTIPOLYGON, POINT/MULTIPOINT etc.) per file therefore you can not dump out a mixed geometry type table in ESRI. ESRI shape is also limited to 10 characters per field name dictated by the DBF standard. As such many of your field names may get truncated.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

**Listing 7.12 Export PostGIS table and query to MapInfo tab format**

```
--1
ogr2ogr -f "MapInfo file" /gisdata/us_boundaries.tab PG:"host=localhost
user=postgres port=5432 dbname=postgis_in_action password=mypassword" -
t_srs "EPSG:4326" us.admin_boundaries

--2
ogr2ogr -f "MapInfo file" /gisdata/biketrip.tab PG:"host=localhost
user=postgres port=5432 dbname=postgis_in_action password=mypassword" -
select "SELECT track_seg_point_id, ele, time" -where "time BETWEEN '2009-
07-18 04:33:04' AND '2009-07-18 04:34:04'" staging.aus_biketrip_narangba

--3
ogr2ogr -f "MapInfo file" /gisdata/tab_files PG:"host=localhost
user=postgres port=5432 dbname=postgis_in_action password=mypassword"
staging.track_points staging.tracks
--1 export with transform
--2 export with filter
--3 export multi-file
```

We performed similar exercises for MapInfo tab as we did for KML. (1) For our US admin boundaries output, we stored the data in national atlas, but want to export it to wgs 84 long lat. In (3) we are outputting 2 tables. This unlike KML, creates a set of files for each table. The files in this case are named staging.track\_points.\* , staging.tracks.\* (tab,map,dat,id) and creates the folder tab\_files to store it in. If this folder exists the command will fail if you don't add a --dsco overwrite=YES.

## 7.4 Summary

In this chapter we demonstrated the use of the included shp2pgsql, shppgsql-gui, pgsql2shp and psql tools packaged with PostgreSQL/PostGIS and explored how to deal with other spatial formats using OGR2OGR to both import and export spatial data. We also demonstrated how to take advantage of the popular OpenStreetMap project. We pointed out some caveats with these tools and how to overcome them. Hopefully these exercises will provide you with the base knowledge to load and export your own data. In the chapters that follow, we shall focus on using PostGIS to solve real world problems.

# Part 2

## *Putting PostGIS to work*

In Part 1 of PostGIS in Action, we covered the fundamentals of geometry types, geometry functions and loading spatial data. In this part, we'll demonstrate how to use PostGIS to solve real world problems.

Chapter 8 covers various common problems that you will come across in building spatial queries for applications. We'll demonstrate how to solve these problems with PostGIS spatial functions and ANSI SQL constructs as well as PostgreSQL specific enhancements to SQL. We'll dive in a bit into build PostgreSQL functions. For some problems, we may demonstrate more than one approach to solving the problem.

Chapter 9 will teach you how to improve performance of your queries and will also demonstrate various SQL Anti-Patterns to avoid when building queries. We will cover the finer points of both spatial and non-spatial indexes, PostgreSQL settings, and function costing. In addition we'll demonstrate how to simplify geometries so that they are complex enough for you rendering needs, but that can provide better performance when querying.

# 8

## *Techniques to solve spatial problems*

This chapter covers

- Using joins with spatial
- SQL Aggregates and spatial aggregate
- Proximity analysis
- Common geometric processing

In prior chapters we looked at spatial functions separately and didn't focus too much on how these functions could be combined to solve real world problems. In this chapter we shall combine several spatial and PostgreSQL functions and sql join constructs to accomplish real world objectives. No single chapter, let alone an entire book can catalog all the disparate spatial challenges faced by the GIS analyst. Instead we want you to focus on the techniques. The same technique can usually solve a whole range of problems.

We shall tackle this chapter using pre-built data combined with ad-hoc generated data sets. If SQL is new to you, you may want to read the Appendix C "SQL primer" which will provide a primer on the fundamentals of SQL. The SQL primer demonstrates SQL constructs applicable to many relational databases.

In this chapter as well as the rest of the book, we will no longer be shy about combining the strength of relational databases with spatial functions. We will not satisfy ourselves with asking if something X is related to something Y. We are going to query entire tables at a time and use the expressiveness of SQL joins with wild abandon. We will learn the fundamentals of doing proximity analysis as well as various methods for processing geometries. Being able to combine, convert simple geometries into more complex or less complex ones is useful to do for map rendering as well as for input into more analytical

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

spatial operations. We will also start to explore blackboxing long pieces of spatial logic into PostgreSQL functions so that we can easily reuse them.

## **8.1 Proximity analysis**

When it comes to GIS, the first thing that comes to mind is where something is located. Once we can locate places using a set of coordinates, the second questions always involve some kind of distance calculation: questions such as how far my house is from the nearest expressway, how many pizza joints are within a mile drive, or even what is the average distance that people have to commute to work.

Non-spatial relational databases have the ability to join tables by various common attributes. Spatial databases give you the added benefit of being able to relate things by proximity just as easily as you can relate things by numbers, dates, and strings. In this section we shall explore proximity relationships and demonstrate how we can use these to derive relationships that conventional SQL joins cannot accomplish.

### **8.1.1 Check for intersections and measuring distances**

We will begin by showing the use of the ST\_Intersects function. This ubiquitous function accepts two input geometries and determines if they intersect. What makes this function handy is that the input geometries can be most any geometry. You can throw points, linestrings, multipolygons at it and ST\_Intersects will return an answer. This function also illustrates the innate ability wielded by spatially-enabled databases. Try doing this using common datatypes of numbers and strings and you will be stuck.

#### **ST\_INTERSECTS AND GEOMETRY COLLECTIONS**

ST\_Intersects geometry function currently does not work with generic geometry collections, but ST\_DWithin does. To use with geometry collections, you need to explode them with (ST\_Dump(the\_geom)).geom or just use ST\_DWithin. Note for PostGIS 1.5 geography, the ST\_Intersects operator does work with geography collections since the geography implementation is distance based rather than intersection matrix based and also doesn't rely on GEOS.

We will start with some common examples. For our exercises, we pulled some publically available data for the San Francisco Bay area from DataSF.org. We start with abridged table of bridges and cities. Naturally, bridges are multilinestrings and cities are multipolygons.

#### **USING ST\_DISTANCE: FINDING DISTANCE OF BRIDGE TO VARIOUS CITIES**

We begin with a simple distance query.

```
SELECT c.city, b.bridge, ST_Distance(c.city, b.bridge) As dist_ft  
FROM sf.cities AS c CROSS JOIN sf.bridges AS b;
```

We chose the above example to specifically demonstrate that ST\_Distance always returns the minimum distance between two geometries. You see this come to light when both the  
©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

distance from San Francisco to the Bay Bridge and the distance from Oakland to the Bay Bridge are zero.

### **ST\_MaxDistance AND ST\_DFullyWithin In PostGIS 1.5+**

In PostGIS 1.5, the ST\_MaxDistance function along with various other distance functions such as ST\_DFullyWithin and ST\_ClosestPoint were introduced. This is all thanks to the work of Nicklas Avén. ST\_MaxDistance is useful if you wanted to know the distance between the furthest point from the city to the furthest point on the bridge.

For those of you unfamiliar with the Bay Area, San Francisco and Oakland are the two termini for the Bay Bridge, a 8.4 mile (13.5 km) span across the San Francisco Bay. ST\_Distance finds the distance between the two closest points of the input geometries - always. The distance unit for geometry is always in the units of the spatial reference system for geometries and the geometries have to have the same spatial reference. Since our SF data came in feet, our distance answers are all in feet.

### **ST\_DISTANCE FOR GEOGRAPHY DATA TYPE**

For PostGIS 1.5+, the geography data type also has an ST\_Distance function, and while geography data is stored in WGS 84 long lat, the distance function always outputs in units of meters. Note also that the new ST\_Distance\_Sphere/Spheroid functions have been upgraded in PostGIS 1.5 to work with most types of geometries (not just points as prior versions were limited to).

#### **USING ST\_INTERSECTS: WHICH BAY AREA CITIES HAVE BRIDGES?**

ST\_Distance provides the actual distance between two geometries, but frequently we just need to know if the distance is either zero or positive. For this we use the fast ST\_Intersects function. As its name implies, ST\_Intersects returns true if the geometries intersect and false otherwise.

```
SELECT c.city, b.bridge
  FROM sf.cities AS c INNER JOIN
       sf.bridges AS b ON ST_Intersects(c.the_geom, b.the_geom);
```

#### **USING ST\_DWITHIN: WHAT BRIDGES ARE WITHIN 1000 FEET OF SAN FRANCISCO?**

The intersects function, while useful and fast, simply gives a true or false answer; it only checks to see if minimum distance between two geometries is zero or positive. If you want objects that fall within a certain radius of another the ST\_DWithin is much more useful.

In the below example we will locate all bridges that are within 1000 feet of San Francisco.

Our San Francisco data is in feet so therefore our unadulterated distance check is done in feet. The ST\_Distance function is an often used companion function to the ST\_DWithin.

#### **Listing 8.1 Basic ST\_DWithin query**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

**-- 1 st\_dwithin query**

```
SELECT DISTINCT c.city, b.bridge,
ST_Distance(c.the_geom, b.the_geom) As dist_ft
FROM sf.cities AS c INNER JOIN sf.bridges AS b
    ON ST_DWithin(c.the_geom, b.the_geom,1000)
WHERE c.city = 'SAN FRANCISCO';
```

**-- 2 Buffer for visualization**

```
SELECT ST_Buffer(ST_Union(c.the_geom),1000) As sf_1000ft
FROM sf.cities As c
WHERE c.city = 'SAN FRANCISCO';
```

In (1) we have the standard query we would use for tabular reporting. In (2) we show the buffer we would create for visualization of our region. Note that in (2) we are using the aggregate ST\_Union function which will group all those San Francisco records into a single geometry record and then buffer it.

ST\_DWithin should be the function of choice when it comes to finding geometries within a desired distance because it is extremely efficient. Novice PostGIS users often resort to two common, considerably slower, alternatives to determine if two geometries are within a certain distance of each other. The first is simply pair up all the geometries and find the distance between the two, then order them from closest to farthest and then pick off the set that is with the desired distance. The second alternative is when you have one reference geometry and you try to locate all other geometries within a certain buffer distance. You create a buffer zone around the reference geometry of the desired distance and then check for all other geometries intersecting the buffered geometry. Both of these approaches are extremely slow in PostGIS.

Unlike the plain ST\_Distance alternative, ST\_DWithin can use a spatial index and thereby avoids having to calculate exact distances for every pairing. This in many cases makes it orders of magnitude faster. Buffering has the disadvantage of needing to first create a derivative geometry using buffering, which introduces inexactitudes of its own since buffers are always approximations of a true buffer. A buffer is still useful particularly for visualization of the effected area.

**ST\_DWITHIN FOR GEOGRAPHY DATA TYPE**

The ST\_DWithin for geography data type is the indexable companion to the ST\_Distance for geography. The distinction between the geometry and geography is that the ST\_DWithin for geography tolerance is always in meters while geometry is in the units of the spatial reference system of the input geometries.

**8.1.2 Convert among different units of measurement**

While feet and meters may be useful units, you may want to measure in miles and so forth. One common trick we use for that is to do a cross join with a units table. Below is a quick script to generate such a table.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

**Listing 8.2 Create a simple units conversion table**

```

CREATE TABLE utility.lu_units (
    unit character varying(50) NOT NULL PRIMARY KEY,
    unit_to_meters numeric(10,4)
);

INSERT INTO lu_units (unit, unit_to_meters) VALUES ('mile', 1609.3400);
INSERT INTO lu_units (unit, unit_to_meters) VALUES ('kilometer', 1000);
INSERT INTO lu_units (unit, unit_to_meters) VALUES ('meter', 1);
INSERT INTO lu_units (unit, unit_to_meters) VALUES ('feet', 0.3048);

```

To get our units in one of the above we now do the following.

**Listing 8.3 What bridges are within half-mile of San Francisco**

```

SELECT DISTINCT c.city, b.bridge_name, ST_Distance(c.the_geom, b.the_geom)
As dist_ft,
-- 1
ST_Distance(c.the_geom, b.the_geom)*u.convfactor As dist_miles
FROM (
-- 2
    SELECT uf.unit_to_meters/um.unit_to_meters As convfactor
        FROM lu_units As uf CROSS JOIN lu_units As um
        WHERE uf.unit = 'feet' and um.unit = 'mile') As u
CROSS JOIN sf.cities AS c

    INNER JOIN sf.bridges As b ON (
-- 3
    ST_DWithin(c.the_geom, b.the_geom, 0.5/u.convfactor))
WHERE c.city = 'SAN FRANCISCO'
ORDER BY dist_miles;
-- 1 convert feet to miles
-- 2 conversion table
-- 3 0.5 mile radius search

```

- (2) We use our conversion table twice to grab the units of measure in feet and unit of measure in miles and to generate the conversion factor between the two units. We alias this variable as convfactor. (1) We then use this in our dist\_miles calculation to convert our native feet to miles and in (3) our ST\_DWithin match to convert our 0.5 miles to feet.

Having to include the aforementioned cross join in a query everytime can become a bit long-winded. We can blackbox the conversion in an sql function. Note in this example we are storing our function in a schema called utility to keep it separate from other things. We have included our utility schema in our database search path and therefore do not need to prefix it with the schema name when using it unless we prefer to.

**Listing 8.4 Example sql function to convert between two units**

```

CREATE OR REPLACE FUNCTION utility.units_from_to(unitfrom character
varying, unitto character varying, thevalue double precision)
RETURNS double precision AS
$$
WITH u(unit, unit_to_meters) AS

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

        (VALUES ('mile', 1609.3400),
                 ('kilometer', 1000),
                 ('meter',1),
                 ('feet', 0.3048)
        )
SELECT ufrom.unit_to_meters/uto.unit_to_meters*$3
      FROM
        u As ufrom CROSS JOIN u As uto
      WHERE ufrom.unit = $1 and uto.unit = $2;
$$
LANGUAGE 'sql' IMMUTABLE STRICT
COST 10;

```

We create a function using sql that implicitly converts all measurements to meters so that we only have to keep conversion table based on meter unit of measurement. We use a CTE (PostgreSQL 8.4+) to simplify our SQL.

### Using tables in functions

The function we have here uses the PostgreSQL 8.4+ CTE functionality which allows us to inline the definition of the table and its data. We could have used the table lu\_units we created instead of inlining the table or used two identical inlined subqueries. The upside of inlining the table is that we can make the function IMMUTABLE instead of just STABLE since it doesn't rely on a table which provides for slightly better caching. It also makes the function self-standing. The upside of using a CTE here instead of a subquery is that we only need to define the table once though we use it twice, but on downside we lose backward compatibility with older versions of PostgreSQL. The downside of this approach instead of using the lu\_units table, is that its not as user-friendly because you can't have a non-programmer go in and add new records to the table to make the system knowlegable about new units.

### Listing 8.5 Using unit conversion function

```

SELECT DISTINCT c.city, b.bridge_nam,
    ST_Distance(c.the_geom, b.the_geom) As dist_ft,
    -- 1
    units_from_to('feet','mile',
                  ST_Distance(c.the_geom, b.the_geom) ) As dist_miles
FROM
    sf.cities AS c INNER JOIN sf.bridges AS b ON (
    -- 2
    ST_DWithin(c.the_geom, b.the_geom,
    units_from_to('mile','feet',0.5) ) )
WHERE c.city = 'SAN FRANCISCO'
ORDER BY dist_miles;
-- 1 feet to mile
-- 2 0.5 mile (mile to ft)

```

We now use our black boxed function to convert feet to miles and miles to fee. (2) We want to express our within distance in terms of miles, but since our geometry units are in feet, ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

this 0.5 miles needs to be converted to our geometry units of feet. (1) We want to also display our distance in miles, so we need to convert our geometry units of feet to miles.

### **8.1.3 Measure large distances**

So far, our distance calculations have presupposed a Cartesian plane. This is adequate for short distances where the curvature of the earth does not come into play. If we attempted to use functions like ST\_Distance on a global scale, you will need to take the earth's curvature into consideration. To obtain sensible results, you need to make sure that you have transformed your measurements into a spatial reference system using distance preserving projections before applying the ST\_Distance function. For instance, the popular Web Mercator spatial reference system looks great on maps because it preserves shape, but in most cases is poor for measuring actual distances and areas. If accurate distance calculation is a must, the best approach is to use spatial reference systems covering your specific region of interest. Regional datasets, such as our San Francisco one, tend to use distance-preserving/space preserving spatial reference systems and already incorporate common units of measurement such as meters or feet.

If you are unable to find a distance-preserving spatial reference system and you only need to measure distance between point geometries, PostGIS offers two functions that will take earth's curvature into consideration: ST\_Distance\_Sphere and ST\_Distance\_Spheroid. These functions were upgraded in PostGIS 1.5 to support all the other common geometries.

For users of PostGIS version 1.5 or higher, you can take advantage of the new geography data type to ease geodetic distance computations and still be able to take advantage of spatial indexes. Unlike the conventional geometry data type, the geography data type is based on a spheroidal surface—not a Cartesian plane. The geography data type also avails itself of spatial indexes based on a spheroid-based model. The reference SRID currently supported by the geography data type is 4326 (WGS 84/Datum long lat units). Even though the WGS 84 spheroid serves as the basis for all geography objects and points of objects are referenced in long lat degree units, when it comes to calculating distances, lengths, and areas, the units for geography are in meters and square meters.

#### **Geography Data Type**

Data type that parallels geometry data type but pre-supposes a spheroidal surface and a fixed SRID of 4326. Geography expects all data to be in WGS 84 long lat degrees, but returns measurements in meters. If your data is in a different spatial ref, you need to transform it by performing a geography(ST\_Transform(the\_geom,4326)) dance to convert to geography. This may change.

The distance/area calculations for geography default to using an WGS 84 earth spheroid, but also supports the faster but less accurate sphere model with radius of 6370986 meters. To use the faster but less accurate sphere model, pass in a "false" for the optional "use\_spheroid" last argument for the measurement functions. Computing

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

distances against a sphere is faster than calculating against the spheroid, but difference in speed is relative to the size and complexity of geometries. For many use cases requiring many long range calculations sphere is often sufficient. You should test both to see which works best for you.

In the below exercise, we shall look at Web Mercator distance in meters compared to distances we get using better measurement spatial reference systems designed for a region and later the distance spheroid functions. We'll compare and see just how bad or good these different approaches stand up to accuracy.

#### **Listing 8.6 Compare distance measurement accuracy of various spatial refs**

```
SELECT DISTINCT c.city, b.bridge_nam,
-- 1
CAST(units_from_to('feet','meter',
    ST_Distance(c.the_geom, b.the_geom)) AS numeric(10,2)) AS ca_m,
    CAST(ST_Distance(ST_Transform(c.the_geom,2163),
        ST_Transform(b.the_geom,2163) ) AS numeric(10,2) ) AS natea_m,
-- 2
CAST( ST_Distance( ST_Transform(c.the_geom,3785),
    ST_Transform(b.the_geom,3785)
) AS numeric(10,2) ) AS wm_m,
-- 3
CAST( ST_Distance(
    geography(ST_Transform(c.the_geom,4326)),
    geography(ST_Transform(b.the_geom,4326)), false) AS numeric(10,2))
        AS geog_spheroid_m
FROM
    sf.cities AS c
    INNER JOIN sf.bridges AS b
    ON (ST_DWithin(c.the_geom, b.the_geom, units_from_to('mile','feet',0.5)
) )
WHERE ST_Distance(c.the_geom, b.the_geom) > 0;
```

This example requires PostGIS 1.5+ because we are using the geography data type. In this example we are comparing the distance between bridges and cities. We perform quite a bit of casting to numeric with two places after the decimal point because the ST\_Distance will return a double precision number with many digits. (1) We begin by converting our native feet distance to meters since we will be using meters to compare all the final measurements. In (2) we are transforming to National Atlas Meters SRS. In (3) we are transforming to WGS84 Long Lat and then casting our geometry to geography data type. Once casted to geography data type, ST\_Distance will automatically use a spheroidal-based calculation and return all answer in meters.

**Table 8.1 Sample records of distances between cities generated by Listing 8.x**

city	bridge_nam	ca_m	natea_m	wm_m	geog_spheroid_m
SAUSALITO	Golden Gate Bridge	83.52	84.10	106.04	83.67

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

SAN FRANCISCO	Golden Gate Bridge	16.29	16.35	20.62	16.27
SAN FRANCISCO	Third Street Bridge	16.14	16.27	20.47	16.17

Table 8.1 provides sample output from the distance query. Assuming the CA State Plane represents the best distance, we can easily see that even for areas within half mile, Web Mercator significantly exaggerates distances.

Also as expected the national atlas equal area since its planar and covers a fairly large range but not as large as web mercator is more accurate than web mercator, but less accurate than the geography data type or the native California state plane. Both the natea and geography give an error of about 0.001 to 0.005% which is generally good enough for many kinds of workloads. Both web mercator and natea have the advantage of being presentable on a map where as geography is not as supported and may require transformation to look good on a map. The natea has the disadvantage unlike mercator of not being able to cover the globe, just covers North America.

In the next example we'll only do point distance checks but for a much larger range. Since we are just doing point checks and not using geography (which behaves like Distance\_Spheroid), this example will work on PostGIS 1.3+.

### **Listing 8.7 Distances between cities in kilometers using various projections**

```
SELECT w1.name As city1, w2.name As city2,
       CAST(
              ST_Distance_Sphere(w1.the_geom,w2.the_geom)/1000
              As integer) As sp,
       CAST(ST_Distance_Spheroid(
              w1.the_geom,
              w2.the_geom,
              spheroid('SPHEROID["WGS 84",6378137,298.257223563]')
              )/1000 As integer) As spwgs84,
       CAST(ST_Distance( ST_Transform(w1.the_geom, 3785),
              ST_Transform(w2.the_geom, 3785)
              )/1000 As integer) As wm
  FROM world.cities As w1 INNER JOIN
       world.cities As w2 ON (w1.name <> w2.name)
 WHERE w1.name IN('Beijing', 'Cairo', 'Rio de Janeiro', 'Sydney')
   AND w2.name IN('Jerusalem', 'Melbourne', 'Philadelphia', 'Shanghai',
 'Sao Paulo')
 ORDER BY w1.name, w2.name;
```

The result of the above query can be seen in Table listing 6.3 of Chapter 6. The mercator while much worse may be suitable for some close proximity rule of thumb calculations and has the advantage of working on older PostGIS installs, good for presentation, as well as being able to take advantage of spatial indexes and working with more geometry types than the older versions of ST\_Distance\_Sphere. Mercator is in general worse than geography in all cases except that since its a native geometry type it enjoys all the power of the GEOS

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

geometric processing functions (though care must be taken) and extensive support by third party tools.

- If your distance ranges are small say covering a country, state, county, choose good for measure planar spatial reference for your area of interest. You'll get good geometric processing, good measurement, and good display all in one package.
- If your data spans huge ranges, then geography is a good consideration and transforming on the map as needed to the most suitable spatial ref for the zoomed in area. Note for many use cases like GoogleMaps, Microsoft Bing, that require data in WGS 84 long lat, this transformation step is not necessary -- just use the standard output functions ST\_AsText, ST\_AsGML, ST\_AsKML etc. accordingly or convert back to geometry → then transform to google web Mercator/etc.

#### **8.1.4 Choose spatial reference systems when measuring area**

The consideration for area are a bit different than those for distance. In the case of area, we need local measurements to be accurate and the span of space we are calculating the area for is generally smaller than the span of space we need to measure distance. In the next exercise we compare areas calculated using a variety of spatial reference systems.

##### **Listing 8.8 Area calculations for large objects**

```
SELECT city,
       CAST(casp_m/1000 As integer ) As casp,
       CAST(geog_m/1000 As integer ) As geog,
       CAST(naea_m/1000 As integer ) As naea,
       CAST(wm_m/1000 As integer ) As wm,
       CAST((1 - c.geog_m/c.casp_m)*100 As numeric(10,2)) As pgeog,
       CAST((1 - c.naea_m/c.casp_m)*100 As numeric(10,2)) As pnaea,
       CAST((1 - c.wm_m/c.casp_m)*100 As numeric(10,2)) As pwm
  FROM (SELECT city, the_geom, ST_Area(the_geom)*POWER(0.3048,2) As casp_m,
             ST_Area(geography(ST_Transform(the_geom, 4326))) As geog_m,
             ST_Area(ST_Transform(the_geom, 2163)) As naea_m,
             ST_Area(ST_Transform(the_geom, 3785)) As wm_m
   FROM sf.distinct_cities ) As c
 WHERE ST_Area(c.the_geom) BETWEEN 13271000 AND 22751000 -- Small Sqft
   OR ST_Area(c.the_geom) > 10400000000 -- Large Sqft
 ORDER BY ST_Area(c.the_geom) ASC;
```

In the query we took the smallest and the largest in our set of data for comparison and also included the percent difference in measurement between our state plane and our competitive alternatives. The results are below.

Table 8.2 Comparing area of cities around San Francisco area in different spatial ref

city	casp	geog	naea	wm	pgeog	pnaea	pwm
SAN QUENTIN	1233	1233	1232	1986	-0.01	0.05	-61.08

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

PORT COSTA	1866	1866	1865	3014	-0.01	0.05	-61.52
DIABLO	2114	2114	2113	3395	-0.01	0.04	-60.64
LIVERMORE	967657	967796	967297	1544566	-0.01	0.04	-59.62
NAPA	1254894	1254884	1253968	2050866	0.00	0.07	-63.43
BAY AND OCEAN	2227233	2228367	2225935	3566644	-0.05	0.06	-60.14

The pgeog,pnaea,pwm are the percentage differences from the California State Plane measurements. As we can see the national atlas is about 0.06% off (NOTE: if you use the ST\_Area(geog,false) – geography measurement will be using sphere which is very close to NAEA numbers) and geography (using spheroid) is about 0.01% off or less from those numbers while the web mercator is a whopping 38% off. These are huge objects though so might not reflect the more common small object considerations spanning large regions of data. In the next exercise we will draw a small 10 meter radius patch of land around several cities using the UTM spatial reference system for that region so our units are all in square meters.

In this section we shall break one of the common best practices we noted in previous chapters. We are going to use one table to store records with different spatial reference systems. Part of this is just so we can easily compare the more preferable good for distance UTM with our geography (web sphere/spheroid model) and web mercator. For this next exercise we will create a new table of circles that have a 10 meter radius around key cities. We are doing this buffering in the UTM zone for that region because in long lat units, the circles will be lop-sided. To figure out the UTM SRID for each city point, we are going to use the utmzone contrib function in the PostGIS wiki - <http://trac.osgeo.org/postgis/wiki/UsersWikipgsqlfunctionsDistance>

We will also exercise the new geography datatype as a storage type instead of just casting to it as we have in prior exercises.

### Listing 8.9 Using multiple spatial reference ids

```
-- 1
CREATE TABLE world.city_buffers(city varchar(150) PRIMARY KEY,
                               the_geom geometry,
                               the_geog geography(POLYGON,4326));

-- 2
INSERT INTO world.city_buffers(city, the_geom)
SELECT DISTINCT ON (c.name) c.name,
              ST_Buffer(ST_Transform(the_geom, utmzone(c.the_geom)), 10) As the_geom
FROM world.cities As c;

-- 3
UPDATE world.city_buffers SET the_geog = geography(ST_Transform(the_geom,
4326));
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In the code listing we are (1) creating a table with a geometry data type field and a geography data type. Since we are using a generic geometry data type, we can stuff any kind of geometry with any kind of SRID in it. We are also creating a parallel column using the geography datatype. One of the features of the new geography data type is that it uses the PostgreSQL 8.3+ typmod enhancement that allows you to define the type and the constraints in the table creation so no need for AddGeometryColumn. (2) We now populate our table by buffering in UTM zone (comprised of about 60 SRIDs), for the the geometry type we keep in the respective UTM zone spatial ref. (3) We then update our the\_geog field with the transformed 4326 cast to geography. Note: We could have used the ST\_Buffer function of geography, but they may result in a different geometry than what the\_geom represents (particularly in places close to the poles where a North/South pole Equal area projection is more suitable than UTM).

### GEOGRAPHY ST\_BUFFER

The geography datatype also has an ST\_Buffer implementation where the units are measured in meters. However the buffer implementation is a thin wrapper around the geometry implementation -- transforming to UTM or North/South pole LAEA (most suitable spatial ref), buffering and then transforming back to wgs 84.

#### **Listing 8.10 Compare areas of 10 meter utm radius buffers around cities**

```

SELECT city, CAST(utm As integer) As utm_sm,
       CAST(geog As integer) As geog_sm,
       CAST(wm As integer) As wm_sm,
       CAST(abs(c.utm - c.geog) As numeric(10,2) ) As diff_utm_g,
       CAST(abs(c.utm - c.wm) As numeric(10,2) ) As diff_utm_wm
  FROM (
    SELECT city, ST_Area(the_geom) As utm,
           ST_Area(the_geog) As geog,
           ST_Area(ST_Transform(the_geom, 3785)) As wm
      FROM world.city_buffers) As c
 WHERE abs(c.utm - c.wm) < 0.2 or abs(c.utm - c.wm) > 900 or city
   IN('Boston', 'Honolulu', 'Paris', 'San Francisco')
 ORDER BY abs(c.utm - c.wm);

```

In the example we compare areas of our little patches of land we created and compare our utm answers to the geography and web mercator. To get a sense of how bad web mercator is for area measurement depending on where you are in the world.

The results of this query can be seen in Chapter 6 Table 6.4.

This demonstrates web mercator is great for area if you are living in anywhere along the equator. If you live way up north in Murmansk or Helsinki, then your property size will be very inflated.

## 8.2 Data tagging

Data tagging refers to a class of spatial techniques where we start with point observations and then try to situate these points within the context of another geometry. Tagging to regions and linear referencing are two common tasks faced by GIS practitioners because they are preparatory steps prior to performing any type of meaningful statistical analysis. For example, if we have rainfall data from various collection stations, unless you group the stations into regions, you wouldn't be able to arrive at any interesting conclusions.

### 8.2.1 Techniques for generating dummy data

While generating data may seem like a pointless exercise, generating random data or data that fits a certain pattern is very useful for testing to determine how your queries will stand up when you are dealing with immense amounts of data or just for testing out general theoretical models against reality. Normally when you start off, the data you collect is minuscule, only as you grow will you discover errors in your query or speed bottlenecks. This is especially true in data collected by instruments. For example, many flight tracking software takes FAA data of plane locations during flight and superimpose the position on a map. In the US, at any given moment, several thousand commercial aircrafts could be in the air. If we collect position data every 5 minutes, in just a few hours, we could have over 100,000 records. We do not want to end up in a position where our common queries take minutes to run and for web mapping -- any query that takes longer than 10 seconds to output a result is considered too long by impatient web visitors. To this end, simulated data is critical to load test our queries prior to deployment in a production environment.

The data generation requires PostgreSQL 8.4+ because of our use of array unnest. The already generated data is included in source code download.

#### **Listing 8.11 Create dummy observation data**

```
-- 1
CREATE TABLE us.observations (
    obsid serial PRIMARY KEY,
    obs_name varchar(50),
    obs_date date,
    state_fips varchar(2),
    state varchar(20));

-- 2
SELECT
AddGeometryColumn('us','observations','the_geom','4326','POINT','2');

-- 3
INSERT INTO us.observations(obs_name, obs_date, the_geom)
SELECT a.obs_name,
       DATE '2008-01-01' + CAST( CAST (random()*1000 As text) || ' days' As
interval ),
       ST_SetSRID(ST_Point(-170 + random()*200, 17 + random()*50),4326) As
the_geom
FROM unnest(ARRAY['parrot', 'parakeet', 'dove', 'pigeon', 'lizard monster',
'eagle', 'cat eater bird']) As a(obs_name)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

CROSS JOIN generate_series(1,2000, 1) As i;

--4
INSERT INTO us.observations(obs_name, obs_date, the_geom)
SELECT a.obs_name, DATE '2008-01-01' + CAST(CAST (random()*1000 As text)
|| ' days' As interval),
ST_SetSRID(ST_Point(-100 + random()*30, 30 + random()*20),4326) As
the_geom
FROM unnest(ARRAY['dinoparrot', 'platibird', 'dove', 'pigeon', 'lizard
monster', 'eagle', 'cat eater bird']) As a(obs_name)
CROSS JOIN generate_series(1,4000, 1) As i;

--5
DELETE FROM us.observations
WHERE NOT EXISTS (SELECT s.gid FROM us.states AS s
WHERE ST_Intersects(s.the_geom,
ST_Transform(us.observations.the_geom,2163)) ) ;

-- 1 create observation table
-- 2 add point field
-- 3 create random observations
--4 add more to shift weight
-- 5 remove unrealistic observations

```

The above demonstrates how you can generate observation data when no real data is at your disposal.

Note in (3) we use PostgreSQL 8.4 to quickly convert an array to a table. In our last step (4) we throw out observations that don't fall in a state boundary. In our table we created fields for state\_fips and state to hold the state that each of these observations occurs in. We shall update these fields in the next example.

### **8.2.2 Tag data to a specific region**

One of the more common things a spatial database is used for is for what we shall refer to as tagging to regions. Below are the classic steps.

- You have named regions of space broken out into polygons or multipolygons. These could be political districts within a city, sales territories, states etc.
- You have geocoded data with long lat and you need to figure out which region this geocoded data falls in.
- You want to derive a new set of data that has all the fields of your geocoded data, with an extra field holding the name of the region it falls in.

#### **EXERCISE 1: TAG POINTS WITH A REGION**

A common scenario is where you have a table of observation points and you need to associate each observation with a region for later statistical processing. For this exercise, we will use the state boundaries we created in the previous chapter as the region and some random WGS 84 long lat points we will make up for the observation data. We will then

determine which state each observation point belongs in. Keep in mind that this can be applied to any region or set of points.

#### **Listing 8.12 Region tag observation data**

```
UPDATE us.observations
    SET state = s.state, state_fips = s.state_fips
  FROM us.states As s
 WHERE ST_Intersects(s.the_geom, ST_Transform(us.
observations.the_geom,2163));
```

In the above example we have tagged each observation with a state and we are transforming our long lat to US national atlas EA since we stored our us.states in US national atlas EA.

Notice also that we denormalized our data by updating a column with information that depends on other tables. If you are wondering why we did not simply create a view that can tag the data on the fly, we offer at least two reasons. First is speed consideration. Data tagging is something that only needs to be done once. Once we place a point within a particular region, the information does not change. In the interest of not having to repeat our tagging computation, which can become quite elaborate over large datasets, we store our results. The second reason is simply one of expediency. Often we need to export our data to non-relational applications for the lay user. This requires that we flatten out our structure. For better or for worse, much of spatial work still involves spoon feeding to these rudimentary systems.

Another kind of tagging is not tagging data to a region, but tagging it to a location on a linestring such as a road. In the next section we will go over how to find the closest point on a line to a point.

#### **8.2.3 Snapping points to closest linestring**

A common task needed in linear referencing is given a set of point locations and a set of lines, what points fall on what line and where on the line do they fall. This comes into play for example if you are collecting data points with your GPS device, your GPS observation may not always line up with the road you happen to be driving on. The road itself could be imprecisely surveyed, or you could simply be swerving from side to side. Whatever the reason, you need to snap your observations back to the road.

This one is derived from Paul Ramsey's "Snapping Points in PostGIS"  
<http://blog.cleverelephant.ca/2008/04/snapping-points-in-postgis.html>.

#### **EXERCISE 1 GIVEN A TABLE OF POINTS AND A TABLE OF LINES, SNAP ALL THE POINTS WITHIN 10 UNITS OF THE LINES TO THE CLOSEST LINE.**

This approach should work with most versions of PostGIS 1.3+ and PostgreSQL 8.2+. It uses the linear referencing functions `ST_Line_Interpolate_Point` and `ST_Line_Locate_Point` functions.

The basic approach to solving this is:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

1. First find which line each point is closest to. We do that with the PostgreSQL DISTINCT ON and PostGIS ST\_Distance combo to return each point once and the line it is closest to. We also use ST\_DWithin as a fast distance filter to quickly reject combination point and line that are too far from each other to be considered the closest.
2. Next given this line and point, we interpolate the point on the line to figure out the closest point on the line it falls.

#### **Listing 8.13 Query to snap points to linestrings – version 1**

```

SELECT
pt_id,
ln_id,
ST_Line_Interpolate_Point(
    ln_geom,
    ST_Line_Locate_Point(ln_geom, pt_geom)
) As snapped_point

FROM
(
SELECT DISTINCT ON (pt.id)
    ln.the_geom AS ln_geom,
    pt.the_geom AS pt_geom,
    ln.id AS ln_id,
    pt.id AS pt_id
FROM
    point_table AS pt INNER JOIN
    line_table AS ln
ON
    ST_DWithin(pt.the_geom, ln.the_geom, 10.0)
ORDER BY
    pt.id,ST_Distance(ln.the_geom, pt.the_geom)
) AS subquery;

```

The above can be written without a subquery as shown below, but the version without the sub query tends to be slower.

#### **Listing 8.14 Query to snap points to linestring without subquery – version 2**

```

SELECT DISTINCT ON (pt.id)
ln.the_geom AS ln_geom,
pt.the_geom AS pt_geom,
ln.id AS ln_id,
pt.id AS pt_id,
ST_Line_Interpolate_Point(
    ln.the_geom,
    ST_Line_Locate_Point(ln.the_geom, pt.the_geom)
) As snapped_point

FROM
point_table AS pt INNER JOIN
line_table AS ln
ON

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
ST_DWithin(pt.the_geom, ln.the_geom, 10.0)
ORDER BY
pt.id, ST_Distance(ln.the_geom, pt.the_geom);
```

Below is a pictorial view of the original points and the snapped points.

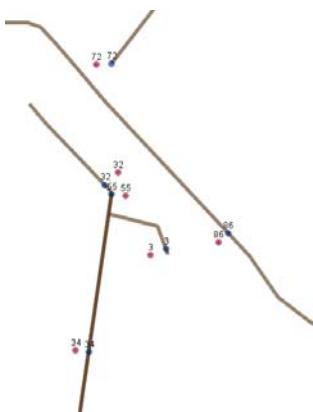


Figure 8.1 Snapping points to a line using query from Listing 8.x. The red are the original points and the blue are the snapped ones.

### **ST\_ClosestPoint in PostGIS 1.5+ a better Line\_Interpolate\_Locate Point**

If you have PostGIS 1.5+, its more efficient and shorter to write to use the ST\_ClosestPoint function. Plus you can use it for more than lines and it is also generally faster.

Then you can replace the `ST_Line_Interpolate(ST_Line_Locate_Point ...` construct with `ST_ClosestPoint(ln.the_geom, pt.the_geom)`

Once you have the points snapped to the line, you can then use the `ST_MakeLine` functions as we described earlier to form a linestring path of your data by ordering by gps time.

#### **8.2.4 Geocoding an address to a point on a street**

Often times you just get address information about a point and you have to determine from the address information, where this address is located along a road. This example uses `stclines_streets` file (Street Centerlines for San Francisco area) from <http://gispub02.sfgov.org/website/sfshare/index2.asp>. Note that these units are in planar state plane feet like the other sf sources we've been working with. We will also use a made up table of addresses. Note that this example is very simplified in the sense that our address names are very well formed and normalized. In most cases you will have to use a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

combination of soundex, prefix matching etc. to get hand input addresses into a normalized form suitable for this step of geocoding.

### BE CAREFULL WHEN USING LONG LAT (DEGREES)

For best results, its best to stick with planar units like meters or feet since these functions are designed to work on a plane, however if you are talking about relatively small streets, the error introduced by long lat is not too bad. The assumption of planar though wrong in those cases is not too far off at very small scales. This is why with the Tiger data we cover in Chapter 10, the fact the data is in long lat is okay, since the line segments are small enough that the error introduced is not significant.

#### **Listing 8.15 Geocode an address to a point**

```
-- 1
CREATE TABLE sf.test_addresses(gid SERIAL PRIMARY KEY,
    st_num integer, st_name varchar(150), zipcode char(5),st_pos char(1),
    the_geom geometry);
INSERT INTO sf.test_addresses(st_num,st_name,zipcode)
VALUES
    ( 33, 'NEW MONTGOMERY ST', '94105' ),
    ( 250, 'CALIFORNIA AVE', '94130' ),
    ( 360, 'ROOSEVELT WAY', '94114' )
;
UPDATE sf.test_addresses
SET
-- 2
    st_pos = CASE WHEN MOD(sc.lf_fadd,2) =
MOD(sf.test_addresses.st_num,2)
        THEN 'L'
        ELSE 'R'
        END,
-- 3
    the_geom = ST_Line_Interpolate_Point(
        ST_Linemerge(sc.the_geom),
        ( sf.test_addresses.st_num - least(sc.lf_fadd,
sc.rt_fadd) )
        / ( greatest (sc.lf_toadd, sc.rt_toadd) - least
(sc.lf_fadd, sc.rt_fadd) )
    )
FROM sf.stclines_streets AS sc
WHERE
-- 4
    substring(sc.zip_code,1,5) = sf.test_addresses.zipcode AND
    sc.streetname = sf.test_addresses.st_name AND
    (sf.test_addresses.st_num BETWEEN sc.lf_fadd AND sc.lf_toadd
    OR sf.test_addresses.st_num BETWEEN sc.rt_fadd AND sc.rt_toadd)
;
-- 1 create test data
-- 2 left or right
-- 3 point on street
-- 4 locate street
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In this example we first (1) create our dummy table of addresses and insert the data with no geometry. In the next part, we determine which street segment in our (4) street center lines the address is on and then we (3) locate the approximate point on the street by assuming the street addresses are evenly spaced along a street using the ST\_Line\_Interpolate\_Point PostGIS function. The second argument to the interpolate point is the percent along the line we think it is and this we determine by taking the fractional percentage of the street number along the full percentage of the line. We use a couple of built-in PostgreSQL helper functions to achieve (3) – we use the least and greatest functions in PostgreSQL that take an infinite number of arguments and return the smallest or the largest in the argument set. We are also using the PostGIS ST\_LineMerge function to convert our MULTILINESTRING to a LINESTRING. This is needed if your street centerlines are all stored as MULTILINESTRING but are contiguous so can be stitched to form a single linestring since the ST\_Line\_Interpolate\_Point function only works with LINESTRINGS.

## **8.3 Slicing and splicing linestrings**

In this section and next we will explore various techniques for breaking apart geometries and combining geometries into larger units or into higher dimensional geometries. We covered the key functions used for slicing and splicing operations in Chapter 4. Here we will show some real-world examples and also not hold back on using SQL to achieve more concise solutions.

### **8.3.1 Create linestrings from points**

In the past decade, the use of GPS devices has gone mainstream. GPS buffs spent their leisure time visiting their own points of interest, take GPS readings, and offer them to the general public via popular mapping tools. Some of the common venues have been local taverns, eateries, fishing holes, and filling stations with the lowest prices. A common followup task after gathering the raw positions of the various places is to connect them together to form a course.

In this exercise we'll use australian track points we imported in chapter 7 to create linestrings. Any GPS track points data will do for this exercise. To create lines from points, we use the spatial aggregate ST\_MakeLine function.

#### **EXERCISE: OBSERVATION LINE PATHS FROM GPS POINTS**

For this exercise we want to create a new line string for every 15 minutes of movement. You can use any grouping you want such as for example the GPS course if that is filled in, but in this case we are using the 15 minute mark for grouping since our other fields are blank and also the date time functions in PostgreSQL are pretty powerful, but not usually demonstrated. This particular exercise should work on most versions of PostgreSQL and PostGIS 1.3+.

#### **Listing 8.16 Create line path from point observations**

-- 1

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

SELECT t.track_period, MIN(time) As t_start,
       Max(time) As t_end,
       ST_MakeLine(the_geom) As the_geom
  INTO work.aussie_run
   FROM (
      SELECT p.time, p.the_geom,
         -- 2
         DATE_TRUNC('minute', p.time)
            - CAST(
               MOD(
                  CAST(DATE_PART('minute', p.time) As integer),15)
                     ' minutes' As interval) As
        track_period
       FROM staging.track_points As p
         -- 3
      ORDER BY p.track_period, p.time) As t
 GROUP BY
         -- 4
        t.track_period
   HAVING COUNT(time) > 2
  ORDER BY t.track_period;

-- 5
SELECT CAST(track_period As timestamp), CAST(t_start As timestamp) As
t_start, CAST(t_end As timestamp) As t_end,
       ST_NPoints(the_geom) As np,
       CAST(ST_Length_Spheroid(the_geom,
          CAST('SPHEROID["WGS_1984",6378137,298.257223563]' As spheroid)
             ) As integer) As dist_m, (t_end - t_start) As dur
  FROM work.aussie_run;
-- 1 return columns
-- 2 snap time to closest track period
-- 3 order by track period
-- 4 group by track period
-- 5 calculate length, time per period

```

In the above we are (2) creating a sub query with a calculated field called track\_period that starts at the 15 minute mark of each hour. This calculated field uses the DATE\_PART function in PostgreSQL as well as the interval data type and mod functions to shove every time point to the 15 minute slot that comes on or before it. (3) In order for our points to be ordered by our track\_period and time, we then order our subquery by those two fields. In this case, the order by p.track\_period is redundant since the p.time guarantees that anyway, but if you were going to group by course for example, then this will be needed. In (4) we define our grouping by our calculated track\_period so that each record will be for a 15 minute mark period and in the SELECT (1) we define our outputs to return our track\_period, min, max, and the line created when we stitch the points together and then dumping it into a new table we call work.aussie\_run. (5) Finally we do a query against our new table to see the results of our handy work. In this query, we use the ST\_Length\_Spheroid instead of ST\_Length since our gps data is in long lat; a simple ST\_Length would give us distance in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

degrees instead of meters which is not terribly useful. We also do a lot of casting to strip off timezone and floating points that ruin presentation.

Below is a sampling of our query in (5).

**Table 8.3 output of query in Listing 8.16**

track_period	t_start	t_end	np	dist_m	dur
2009-07-18 04:30:00	2009-07-18 04:30:00	2009-07-18 04:44:59	133	2705	00:14:59
2009-07-18 04:45:00	2009-07-18 04:45:05	2009-07-18 04:55:20	87	1720	00:10:15
2009-07-18 05:00:00	2009-07-18 05:02:00	2009-07-18 05:14:59	100	1530	00:12:59
:2009-07-18 15:00:00	2009-07-18 15:09:16	2009-07-18 15:14:57	45	1651	00:05:41

As we can see from the above not all of our time increments are even or 15 minutes in duration. Who knows what was happening – perhaps we took a rest or our GPS died.

### 8.3.2 Break linestrings into smaller segments

In this section we'll go thru a couple of exercises for breaking up lines. How you break the lines depends on what you are trying to do and the approach you take.

#### CREATING TWO POINT LINES FROM MANY POINT LINESTRINGS

One common need is ability to take a linestring with various points and convert it to linestrings each with 2 points. In many use cases, a line with just a start point and end point is easier to work with such as in case when you are trying to group together edges shared by polygons. Below is an example that takes generated gps tracks (those imported from GPS format with ogr2ogr) and converts them to 2 point lines.

#### Listing 8.17 Make two point lines from linestrings

```

SELECT ogc_fid, n As pt_id,
--1
    ST_MakeLine(
        ST_PointN(the_geom,n),
        ST_PointN(the_geom,n + 1)
    ) As the_geom
FROM staging.tracks
--2
    CROSS JOIN generate_series(1,10000) As n
--3
WHERE n < ST_NPoints(the_geom)
ORDER by ogc_fid, pt_id;
-- 1 non-aggregate ST_MakeLine
-- 2 generate_series as iterator
-- 3 limit iterator

```

In the above we are using the (1) non-aggregate version of the ST\_MakeLine function that takes in 2 point geometries and makes a simple 2 point line. The ST\_PointN is a point

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

iterator function that for a linestring (non-MULTI), will return the Nth point. (2) We use the powerful in-built PostgreSQL generate\_series function to do cross join to generate an iterator between 1 and 10000. This only works if each linestring has fewer than 10000 points and then we use the (3) WHERE n < number of points to limit the number of records to be the number of points for each line. This example will work in lower versions of PostgreSQL as well as versions of PostGIS 1.3+.

The above using generate\_series is a common idiom in SQL to simulate a procedural “for loop” and especially common in PostGIS since a lot of geometry processes involve iterating over geometries. As mentioned, the above only works with Linestrings, but what if you have MULTILINESTRINGS. This is one of the occasions where the ST\_Dump() function comes in handy. Watch closely. The below example is more expensive but will handle MULTILINESTRINGS as well.

### **Listing 8.18 Make two point lines from Multilinestrings or Linestrings**

```

SELECT ogc_fid, n As pt_id, sl.path[1] As nline,
--1
    ST_MakeLine(
        ST_PointN(sl.geom,n),
        ST_PointN(sl.geom,n + 1)
    ) As the_geom99
--2
FROM (SELECT ogc_fid, n, (ST_Dump(the_geom)).*
FROM staging.tracks) As sl
--3
CROSS JOIN generate_series(1,10000) As n
WHERE n < ST_NPoints(sl.geom)
ORDER by ogc_fid, nline, pt_id;

```

In the above we are using ST\_MakeLine again (1), but note we are using a field call “geom” instead of the\_geom. This is because the (2) ST\_Dump function we learned about in earlier chapters returns a set of geometry\_dump objects consisting of two fields “geom” and “path” – the

(ST\_Dump(the\_geom)).\* forces the properties of the geometry\_dump to be expanded as table columns. Since ST\_Dump is a set returning function it explodes the number of rows we have so that we will have one row for each linestring in our multilinestring/linestring. The path object is a 1-dimensional array consisting of the path position of the subgeom within the geometry. In the case of MULTILINESTRINGS, there is only one element in the array which is the position of the linestring in the multilinestring, but for a nested geometry collection, the path info becomes more interesting. Again (3) We use WHERE clause to limit expansion to just number of points.

### **POSTGIS 1.5 ST\_DUMPPOINTS**

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In PostGIS 1.5 ST\_DumpPoints function was introduced which makes the above exercise a bit easier since you can go straight to using ST\_DumpPoints and skip the generate\_series cross join and replace ST\_Dump with ST\_DumpPoints.

#### **BREAKING LINESTRINGS AT POINT JUNCTIONS**

In this example, we will demonstrate how one would go about given a table of points and table of linestrings, split the lines at the intersecting points. This comes into play for example if you need to put up roads posts, and need your roads split at these post points. While this exercise does make great use of linear referencing functions, people tend to not think of it as a linear referencing activity.

The basic steps for this exercise are to

1. Figure out which points intersect with each line
2. For each point intersection use the ST\_Line\_Locate\_Point to figure out the percentage along the line the point lies
3. Use ST\_Line\_Substring to return the respective portions of the LINESTRING
4. Use ST\_SetPoint to patch floating point errors.

For starters, these are too many steps to try to stuff in your SQL statement, so its best to wrap this logic in a stored function. Also keep in mind that for these kind of activity, its very useful to keep different variable states and there isn't much use of spatial indexes within an overall body of an SQL. This means we want to choose a language that allows us to define variables and can be seen as a black-box (rather than inlined in the SQL plan). Because of these two desired criteria, we will choose to write our function in plpgsql over sql this time around.

Our desired function will take two inputs -- a multilinestring/linestring and a multipoint/point, and a distance tolerance in units of the spatial ref of the geometries and will output a multilinestring where the individual linestrings in the multilinestring, have been cut at the point junctures defined by the multipoint/point. The tolerance is the small margin of distance error we will allow to assume treat a point as being on the line. So for example if we set our tolerance to be 1 ft, then our function will consider any point within 1 ft of the line to be on the line by snapping it to the closest point on the line.

#### **Listing 8.19 Function to cut linestring at point junctions**

```
CREATE OR REPLACE FUNCTION upgis_cutlineatpoints(param_mlgeom geometry,
                                                param_mpgeom geometry,
                                                param_tol double precision)
RETURNS geometry AS
$$
DECLARE
    var_resultgeom geometry;
    -- dump out multis into single points
    -- and lines so we can use line ref functions
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

-- 1
var_pset geometry[] := ARRAY(SELECT geom FROM ST_Dump(param_mpgeom));
var_lset geometry[] := ARRAY(SELECT geom FROM ST_Dump(param_mlgeom));
var_sline geometry;
var_eline geometry;
var_perc_line double precision;
var_refgeom geometry;

BEGIN
-- Loop thru the points
-- 2
FOR i in 1 .. array_upper(var_pset,1)
LOOP
-- Loop thru the linestrings
FOR j in 1 .. array_upper(var_lset,1)
LOOP
-- Check the distance and update if within tolerance
IF ST_DWithin(var_lset[j],var_pset[i], param_tol)
    AND NOT ST_Intersects(ST_Boundary(var_lset[j]),
var_pset[i]) THEN
    IF ST_NumGeometries(ST_Multi(var_lset[j])) = 1 THEN
        --get percent along line point is
        var_perc_line := ST_Line_Locate_Point(
            var_lset[j], var_pset[i]);

        IF var_perc_line BETWEEN 0.0001 and 0.9999 THEN
            --get first cut only cut if not too close to edge
            var_sline := ST_Line_Substring(var_lset[j],0,
var_perc_line);
            -- get secont cut
            var_eline := ST_Line_Substring(var_lset[j],var_perc_line,
1);
            --fix rounding so start line abutts second cut
            var_eline := ST_SetPoint(var_eline, 0,
ST_EndPoint(var_sline));

            -- collect the two parts together
            -- to create a multiline string cut
        -- 3
            var_lset[j] := ST_Collect(var_sline, var_eline);
        END IF;
    ELSE
--4
        -- We are trying to cut a multilinestring -- recurse
        var_lset[j] := upgis_cutlineatpoints(var_lset[j], var_pset[i]);
    END IF;
    END LOOP;
END LOOP;

RETURN ST_Union(var_lset);

END;
$$
LANGUAGE 'plpgsql' IMMUTABLE STRICT
COST 100;

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

(1) We use our favorite pattern of exploding a multigeom into single geom pieces and then collapsing it into a geometry array for easier processing. This will turn our multilinestring/linestring into an array of linestrings, and multipoints into array of points. (2) Next we step thru each point and for each line that intersects the point but is not on the boundary -- we perform the 4-step process we outlined earlier which for matches will result in a (3) Multilinestring composed of 2 linestrings we form by collecting the 2 linestrings -- a start segment and end segment and we replace our original linestring with this 2 line multilinestring. Note that if a line is cut multiple times (the multilinestring can expand to more than 2 pieces) -- we use the power of recursion (4) to repeat the whole process so that at any point in time we are always dealing with single linestrings and single points. Functional recursion has existed in PostgreSQL for quite a while so this exercise will work in versions of PostgreSQL 8.1+.

Now for a simple test to see our function in action. We will cut San Francisco streets that fall within 100 feet of our desired point and we will use ST\_Dump again to explode our multilinestring into individual linestrings.

### **POSTGIS 2.0 ST\_SPLIT**

In PostGIS 2.0 there is a function called ST\_Split which will allow you to split a line string by a point or a polygon by a line. Using that function would be more efficient.

#### **Listing 8.20 Cut streets with a point tolerance of 100 ft**

```
SELECT gid, the_geom As orig_geom,
       upgis_cutlineatpoints(the_geom, foo.the_pt, 100 ) As changed
  FROM sf.stclines_streets AS s CROSS JOIN
       (SELECT ST_SetSRID(ST_Point(6011200, 2113500),2227) As the_pt) As foo
 WHERE ST_DWithin(s.the_geom, foo.the_pt, 100);
```

The above query wil return the id of the street that was cut and the individual pieces as separate rows. We also include the original for compare. A pictorial view of this is shown below.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Figure 8.2 Using our upgis\_cutlineatpoints function - we cut Mission street with a point that is within 100 feet of the line.

In the section that follows, we talk about the other useful things you can do with PostGIS affine family of functions.

## **8.4 Slicing and Splicing Polygons**

In the aforementioned exercises, we saw how we could use basic geometries to do distance and areal analysis. Before you can even get to the point of doing interesting things with space, your data needs to be geocoded. Once you have data with a defined location you can do even more interesting things. You can use simple common attributes such as road names, region names, or even observations of activity to form complex geometries such as paths, polygon high crime areas etc. This is all made possible thru the power of aggregation.

Aggregation is the concept of rolling up several records into one by grouping by some set of attributes and aggregating them using aggregation functions. We go into a little more detail about this concept in Appendix C and cover the various parts of group by, having, and the select clauses.

In a standard relational database, the most common aggregation functions used are COUNT, SUM, MIN, MAX, and AVG. With a spatial extender such as PostGIS, you can add to the mix of functions geometry aggregation functions – ST\_MakeLine, ST\_Union, ST\_Collect, and ST\_Polygonize. The ST\_Union function is by far the most commonly used of the spatial aggregates.

### **8.4.1 Create a single multipolygon from many multipolygon records**

In our example of San Francisco, we noted that our cities table has multiple records for San Francisco. This is useful in some cases because you may want to break out your geometries by political boundaries which may or may not be adjacent or overlapping each other. Recall from prior chapters that if two polygons share an edge or overlap (intersect at non-finite points), then you can not form a valid multipolygon out of them. You must either store them as a generic geometry collection or loose the shared boundaries by unioning them. We also learned in prior chapters that generic geometry collections are pesky creatures because many functions do not work with them. For our study of proximity, we would prefer each city to be represented as a single record even with the realization that we may loose political boundaries. To do that – we shall combine them using the OGC ST\_Union function and group by city name.

#### **DISSOLVE BOUNDARIES WITH ST\_UNION**

Below we have code that tells us for each city that will be collapsed how many records we will be collapsing and what the number of polygons will be in the new record.

#### **Listing 8.21 Analysis pre-dissolving records**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

SELECT city, COUNT(city) As num_records,
       SUM(
           ST_Numgeometries(the_geom)
           ) As numpolygons_before,
       ST_Numgeometries(
           ST_Multi(ST_Union(the_geom))
           ) As num_polygons_aftre
  FROM sf.cities
 GROUP BY city
 HAVING COUNT(city) > 1;

```

From the above code we know that we will be collapsing 10 cities, but of those only Brisbane and San Fancisco will result in a dissolving of boundaries by the fact that our dissolved world will have fewer polygons per geometry than what we started out with.

In order to create a new table with distinct cities – one per record, we can now do this:

#### **Listing 8.22 Create one record per city**

```

-- 1
SELECT city, ST_Multi(ST_Union(the_geom)) As the_geom
INTO sf.distinct_cities
FROM sf.cities
GROUP BY city;

-- 2
SELECT populate_geometry_columns('sf.distinct_cities'::regclass);

-- 3
ALTER TABLE sf.distinct_cities ADD CONSTRAINT pk_distinct_cities
    PRIMARY KEY(city);
-- 4
CREATE INDEX idx_distinct_cities_the_geom ON
    sf.distinct_cities USING gist(the_geom);
-- 1 bulk create table
-- 2 register table in geometry_columns
-- 3 primary key
-- 4 spatial index

```

In (1) we automatically create and populate a new table called sf.distinct\_cities in one step. We also use the ST\_Multi function to ensure that all our resulting geometries will be MULTIPOLYGONS and not POLYGONS. In (2) we register the the\_geom geometry column and also add constraints (srid and geometry type constraints to the table). populate\_geometry\_columns is a management function introduced in PostGIS 1.4 that both registers the table for you and infers and creates the constraints that need to go on the table by inspecting it. (3,4) For good measure we put in a primary key and a spatial index.

### **8.4.2 Tessellate areas**

One thing that is commonly useful for statistics is to divide your areas by spatial area size or by population such that each region has approximately the same area or the same population.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

For these exercises, we shall explore both of those approaches. We will first experiment with breaking our data into equal areas to demonstrate one approach for doing that. We will then create what we will call an observation\_tract for the things we are tracking such that each observation tract will have approximately the same number of things in it. We will demonstrate the power of sql by bringing together several concepts together

- Revisit our dicing routine we observed in earlier chapters to divide US into areal units
- Using the new PostgreSQL 8.4 window functions in combo with some simple mathematics to group our areas into a collection of areas whose total area is close to our desired area.
- Using the powerful ST\_Union spatial aggregate function we saw earlier to union these areas into single areas.
- Using the new common table expressions feature introduced in PostgreSQL 8.4 to combine all these SQL statements into one single query to create a table.

**EXERCISE: CREATE A GRID AND SLICE YOUR TABLE GEOMETRIES WITH THE GRID**

In this exercise we will cut our states into smaller units using a grid. We saw this exercise before, but we'll add a couple of twists to it. This is useful for in a couple of scenarios

- Improve spatial searches
- Divide an area into smaller units which are more suited for heat maps
- Reallocate areas by first dividing and then putting them back together differently.

Here we divide the US data into smaller quadrants so that we can collect them up later. When we are done – we will end up with a throwaway\_grid that looks like below:



Figure 8.3 Our throw\_away grid

**Listing 8.23 Divide US into quadrants**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

WITH usext AS
(
-- 1
SELECT ST_SetSRID(
    CAST(ST_Extent(the_geom) As geometry),2163) As the_geom_ext,
    60 as x_gridcnt, 40 as y_gridcnt
FROM states As s
),
-- 2
grid_dim AS
(SELECT (
    ST_XMax(the_geom_ext) - ST_XMin(the_geom_ext)
    )/x_gridcnt As g_width,
    ST_XMin(the_geom_ext) As xmin, ST_Xmax(the_geom_ext) As xmax,
    (
        ST_YMax(the_geom_ext) - ST_YMin(the_geom_ext)
    )/y_gridcnt As g_height,
    ST_YMin(the_geom_ext) As ymin,
    ST_YMax(the_geom_ext) As ymax
    FROM usext
),
grid As (
SELECT x,y,
    ST_SetSRID(ST_MakeBox2d(
        ST_Point(xmin + (x - 1)*g_width, ymin + (y-1)*g_height),
        ST_Point(xmin + x*g_width, ymin + y*g_height)
        )
        ,2163) As grid_geom
FROM (SELECT generate_series(1,x_gridcnt) FROM usext) As x(x)
    CROSS JOIN (SELECT generate_series(1,y_gridcnt) FROM usext) As y(y)
CROSS JOIN grid_dim
)
-- 3
SELECT grid.x, grid.y, state, state_fips,
    ST_Intersection(s.the_geom, grid_geom) As the_geom
    INTO us.grid_throwaway
    FROM states As s
    INNER JOIN grid ON (ST_Intersects(s.the_geom, grid.grid_geom));
-- 4
CREATE INDEX idx_us_grid_throwaway_the_geom
    ON us.grid_throwaway USING gist(the_geom);
-- 5
vacuum analyze us.grid_throwaway;

-- 1 Define constants
-- 2 Divide extent into rectangles
-- 3 cut grid by state boundary
-- 4 index
-- 5 vacuum

```

The above exercise just uses a grid with 60 cells along x and 40 along y of the extent of US to dice up our state boundaries such that no two states fall in the same region. Note that in our above picture – because of the way the tile cuts thru the US, each tile is of various

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

shapes and sizes. This is not terribly ideal for a study area if we want all our quadrants to be more or less the same size in each state.

### 8.4.3 Create equal-area slices

Tessellation is fast and works well when we need many small dices without paying too much attention to the size of each piece. In most scenarios, we require fewer cuts, but of equal size.

#### CREATE A SINGLE LINE CUT THAT BEST BISECTS INTO EQUAL HALVES

To create equal-area slices the first strategy we will employ is basically one of convergence towards a solution. We start with a trial cut through our area, measure the area of the cut. If it is larger than what we need, we translate the cut line to get a smaller slice. We keep doing this until we obtain a cut with the desired area.

#### Listing 8.24 Bisect state of Idaho

```
WITH RECURSIVE
-- 1
ref(the_geom, env) AS (
    SELECT the_geom,
        ST_Envelope(the_geom) AS env,
        ST_Area(The_geom)/2 AS targ_area,
        1000 AS nit
    FROM us.states
    WHERE state = 'Idaho'
),
-- 2
T(n,overlap) AS (
    VALUES (CAST(0 AS Float),CAST(0 AS Float))
    UNION ALL
    SELECT n + nit, ST_Area(ST_Intersection(the_geom, ST_Translate(env,
n+nit, 0)))
    FROM T CROSS JOIN ref
    WHERE ST_Area(ST_Intersection(the_geom, ST_Translate(env, n+nit, 0))) > ref.targ_area
),
-- 3
bi(n) AS
(SELECT n
    FROM T
    ORDER BY n DESC LIMIT 1)
-- 4
SELECT bi.n,
    ST_Difference(the_geom, ST_Translate(ref.env, n,0)) AS geom_part1,
    ST_Intersection(the_geom, ST_Translate(ref.env, n,0)) AS geom_part2
FROM bi CROSS JOIN ref;
-- 1 state variables
-- 2 recursive iterator
-- 3 cte returns how far in x dir to cut
-- 4 return both parts of idaho
```

- (1) The reference geometry we want to cut in half along x, the envelop of the reference geom, and (nit) number of meters we will be moving per iteration, and our target area which
- © Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

is half the area of the state. Note the units are in meters since our `us.states` table is in national atlas equal area meter units. (2) This is our recursive iterator that keeps on moving the extent box across until it hits our target area. (3) We only care about the last record of 2 -- so we hold it in this CTE call `bi`. This represents the number of meters to move the bounding box of the reference geometry to get it to bisect. (4) Our final query which returns the 2 halves of our geometry.



Figure 8.4 United State of Idaho bisected into 2.

In our query, we use a CTE to perform the iteration. This is more to flaunt the capability of PostgreSQL more than anything else. For clarity and portability, we advise creating a function that performs the cut and then call the function successively until you reached the desired cut.

We presented the basic technique for vertically slicing areas into two equal halves, an eastern half and a western half. We hope that you recognize that more slices can be created simply by looping through our cutter multiple times. For slicing into fourths, perform the cut twice. For slices that are multiple of twos, you can use another layer of recursion to continually bisect your resultant areas until you have the number of total slices you want. You can even combine vertical cuts with horizontal cuts simply by iterating through the Y-axis simultaneously to divide an area into quadrants.

#### **CREATING EQUAL AREAS BY DISINTEGRATION AND INTEGRATION**

In this next approach, we shall use a grid similar to what we did earlier, cut with the grid and accumulate the shards recursively into buckets. When the total area of a set of shards is equal to our desired bucket area or count, we will create a new bucket of shards. We will then union them together by the bucket they are in. We will employ the following tricks:

- Gridding
- Recursive queries (requires PostgreSQL 8.4+) to bucket
- `ST_Union` to regroup our bucket into a single geometry
- PostgreSQL outparameters (introduced in PostgreSQL 8.2+) to output a typed set of rows consisting of a bucket and a geometry

Please note that while we are demonstrating this to cut in equal areas, you can use a similar trick by data tagging point data into grids and then summing up counts or other

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

features of a point to achieve other equalities such as equality of population, trees etc. So each bucket you have would say represent an equal population. This is basically what the US census does – each tract has an equality of population, not area.

When we are done, we will have a function that takes a geometry and number of sections as input that we can call like this:

```
SELECT bucket, the_geom, ST_Area(the_geom) As the_area
FROM utility.upgis_slicegeometry(
  (SELECT the_geom FROM us.states
   WHERE state = 'Oklahoma'), 4) As foo;
```

The above example would produce an Oklahoma broken into 4 equal regions that looks like the following diagram:



Figure 8.5 The state of Oklahoma broken into 4 equal quadrants

We can verify our work by taking the resulting area from our tabular which looks like:

Table 8.4 Area of Oklahoma cut into 4 equal quadrants

bucket	the_area
4	45407005343.697
2	45287294131.5032
1	45267841092.5329
3	45214837721.5997

Lets walk thru the function and how its constructed.

#### Listing 8.25 upgis\_slicegeometry – cuts a geometry into equal areas

```
CREATE OR REPLACE FUNCTION utility.upgis_slicegeometry(geom geometry
, numsections integer, OUT bucket integer, OUT the_geom geometry)
RETURNS SETOF record AS
$$
WITH RECURSIVE
--[1 Beg]
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

    ref(the_geom, the_box, targ_area, x_mov, y_mov, x_length, y_length,
xmin, ymin) AS (
    SELECT the_geom,
        ST_SetSRID(ST_MakeBox2D(ST_Point(xmin, ymin),
        ST_Point(xmin + CAST(x_length/ngrid_xy As integer),
        ymin + CAST(y_length/ngrid_xy As integer)
        )
    ),
    ST_SRID(s.the_geom) ) As the_box, ST_Area(the_geom)/$2 As
targ_area,
        CAST(x_length/ngrid_xy As integer) As x_mov,
        CAST(y_length/ngrid_xy As integer) y_mov,
        s.x_length,
        s.y_length, xmin, ymin
    FROM (SELECT $1 As the_geom,
        ST_XMin($1) As xmin,
        ST_YMin($1) As ymin,
        ST_XMax($1) - ST_XMin($1) As x_length,
        ST_YMax($1) - ST_YMin($1) As y_length, 15*$2 As ngrid_xy
        ) As s
), --[1 End]
--[2 Beg]
x(x) As (VALUES (CAST(0 As Float))
    UNION ALL
    SELECT x + ref.x_mov FROM X CROSS JOIN ref WHERE x < ref.x_length),
y(y) As (VALUES (CAST(0 As Float))
    UNION ALL
    SELECT y + ref.y_mov FROM Y CROSS JOIN ref WHERE y < ref.y_length),
--[2 End]
--[3 Beg]
diced AS
(SELECT ROW_NUMBER() OVER(ORDER BY x,y ) As row_num
, g.x, g.y, g.the_geom
FROM
(SELECT x, y,
    ST_Intersection(
        ref.the_geom,
        ST_Translate(ref.the_box, x, y)
    ) As the_geom
    FROM x CROSS JOIN y CROSS JOIN ref
WHERE ST_Intersects(ref.the_geom,
    ST_Translate(ref.the_box, x, y)
)
) As g
) --[3 End]
,
--[4 Beg]
T(bucket, row_num, the_geom, total_area, targ_area, remaining_area) AS
(
    SELECT 1 As bucket,
        row_num,
        diced.the_geom, ST_Area(diced.the_geom) As total_area,
        ref.targ_area,

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

        ST_Area(ref.the_geom) - ST_Area(diced.the_geom) As remaining_area
        FROM diced CROSS JOIN ref WHERE diced.row_num = 1
    UNION ALL
        SELECT CASE WHEN (T2.total_area + ST_Area(diced.the_geom) <
T2.targ_area OR T2.remaining_area < T2.targ_area/4)
        THEN T2.bucket
        ELSE T2.bucket + 1 END As bucket,
        diced.row_num, diced.the_geom,
        CASE WHEN ( T2.total_area + ST_Area(diced.the_geom) ) <
T2.targ_area
        THEN T2.total_area + ST_Area(diced.the_geom) ELSE
ST_Area(diced.the_geom) END As total_area,
        T2.targ_area, T2.remaining_area - ST_Area(diced.the_geom) As
remaining_area
        FROM diced INNER JOIN (SELECT * FROM T ORDER BY row_num DESC LIMIT 1)
As T2
        ON diced.row_num = T2.row_num + 1
    ) --[4 End]

--[5 Beg]
    SELECT bucket, ST_Union(the_geom) As the_geom
    FROM T
    GROUP BY T.bucket, T.targ_area
--[5 End]

$$
LANGUAGE 'sql' IMMUTABLE;
-- [1] define constants
-- [2] start positions squares
-- [3] window translate dice
-- [4] bucket shards
-- [5] union buckets

```

We are using a recursive CTE construct with several sub table expressions some of which are recursive and some of which are not. [1] We first define our reusable constants by inspecting the input geometry: ngrid\_xy defines the number of cuts we make along x and y. If we were to do more cuts our solution would be slower, but more exact. -- we are doing 15\*numsections desired cuts along x and along y. [2] We do a recursive query to return x, y starting positions for each square – we could use generate\_series instead for this, but this is slightly shorter. [3] Window / translate query to dice our geometry. The row\_num column will return sequential unique numbers ordered by our OVER(ORDER ... Note: the ROW\_NUMBER OVER(ORDER BY x,y) controls our cut. If we want our cuts going down instead of across we would order by y and then x (for esoteric cuts use a function like quadrant location like ST\_SnapToGrid and then x,y or some esoteric function). [4] Recursive query to throw our shards into buckets. The trick here is our SQL CASE statement -- we keep on adding into the existing bucket until the desired area is exceeded or what is remaining is less than 1/4th of target area. The 1/4<sup>th</sup> is arbitrary. [5] We union the shards in each bucket. Our resulting table will have fields called bucket and the\_geom because our output parameters are called that. In PostgreSQL all arguments to a function are assumed to be only input parameters unless you explicitly put in OUT or INOUT.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In this section we've already tasted a little bit of what translating geometries can do for us. In the section that follows, we will explore a few more tricks you can do with translate and other Affine functions.

## **8.5 Translating, scaling, and rotating geometries**

Should you still remember what you learned during your first linear algebra course (we don't), shifting, scaling, and rotating constitute affine transformations on a plane. PostGIS has built-in functions to perform all three: ST\_Translate, ST\_Scale, and ST\_Rotate. All three fall under an umbrella function ST\_Affine where you can explicitly specify the transformation matrix. We will not go into detail about the ST\_Affine function since it is rarely used directly.

Though you may think of shapes on a map as mostly static objects with little use for repositioning, these handy functions intrude more often than expected. We have encountered the following common uses and are sure that more creative uses abound:

- creating grids you then use to dice larger geometries into smaller pieces or to use as an overlay on maps or to produce heatmaps based on number of geometries and size of geometries in each tile.
- Simulating movement along a road
- For simulations changing positions of objects
- Correcting coordinates of a geometry when some bozo gave you shifted data
- Creating parallel road lines or edges to turn a line to a polygon
- compensating for lack of Z support in GEOS functions by rotating the axis so you can always be comparing x and y planes.

### **8.5.1 Move a geometry along X, Y, Z**

The most popular of the transformations functions is ST\_Translate, which shifts a geometry. ST\_Translate has many overloads, we will demonstrate the ST\_Translate(geom,x,y) variant, which is the most predominantly used.

One of the most common and unexpected use for ST\_Translate is to create grids by using one geometry to paint across and down a region. Once you have your artificial graticules, you can intersect it with a reference geometry to dice it into rectangles regions or other shapes that suite your fancy. We call this spatial design pattern the "Cookie cutter grid strategy". Take any paper road map and you can see this in action. The map is always divided into alpha and numeric rectangles so that you can find a street in the index and be able to find it on the map itself. Another common use with an artificial grid is to summarize geometries within each tile to produce what are called heatmaps. In the meantime, we will show you how to create artificial cells on a map. The example we are about to present will create a honeycomb and a rectangular grids somewhere in the middle of the United States. Again, we are not going to be shy about holding back on the SQL since the awesome power of PostGIS only comes to light when we make liberal use of SQL's bulk processing capabilities. In this example we shall use the PostgreSQL 8.4 Common Table Expression  
©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

feature again. If you are using 8.3 and below, you will need to repeat the CTEs we are using where we use them or create temp tables to hold the CTE expressions. Better yet, upgrade!

### **Listing 8.26 Generate a Rectangle and a Hexagonal Grid centered in USA**

```
--[1 Beg]
WITH center_point(x,y) AS
(
    SELECT -288499, -2718
),
--[1 End]
--[2 Beg]
paintbrush(the_hex, the_rect) AS
(
SELECT ST_SetSRID(ST_Translate(ST_GeomFromText('POLYGON((0 0,64 64,128,0
192,-64 128,-64 64,0 0))')),  

           x, y), 2163) As the_hex,  

ST_SetSRID(ST_Translate(CAST(ST_MakeBox2D(ST_Point(-64,0), ST_Point(64,192)
) As geometry),
           x, y), 2163) As the_rect
FROM center_point
)
--[2 End]
--[3 Beg]
SELECT xf.x, yf.y, ST_Translate(paintbrush.the_hex, xf.x_hex, yf.y_hex) As
hex_tile,
       ST_Translate(paintbrush.the_rect, xf.x_rect,yf.y_rect) As rect_tile
FROM (SELECT x, x*(ST_XMax(the_hex) - ST_XMin(the_hex)) As x_hex,
x*(ST_XMax(the_rect) - ST_XMin(the_rect)) As x_rect
FROM
generate_series(-50, 50) As x CROSS JOIN paintbrush) As xf
CROSS JOIN (SELECT y, y*(ST_YMax(the_hex) - ST_YMin(the_hex)) As y_hex,
y*(ST_YMax(the_rect) - ST_YMin(the_rect)) As y_rect
FROM
generate_series(-50, 50) As y CROSS JOIN paintbrush) As yf
CROSS JOIN paintbrush;
--[3 End]

[1] cte center_point
[2] cte hex/square dual paintbrush
[3] start from center paint
```

The example generates a hexagonal and a rectangular grid consisting of 10201 records in about 1-3 seconds using a 2 headed paintbrush. It uses the new CTE functionality introduced in PostgreSQL 8.4 to break up the steps a little more and to prevent repetition of code. [1] We define a CTE called center\_point that returns a 1 row table where we will position our paint brush. This will be where we will move the center of our paintbrush heads. [2] We create our paintbrush CTE with 2 heads the\_hex representing the hexagonal head and the\_rect representing the rectangular head. We are using the ST\_MakeBox2D function to create our rectangular head since its simpler to express squares and rectangles with

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

boxes, but since a box is not a geometry, we then convert that to a geometry with the ANSI SQL CAST function. [3] For the final output of our CTE expression, we create a grid that will output iterators x and y as well as 2 geometries – one representing each rectangular tile and one representing each hexagonal tile. Observe that our iterators we multiply by the width and height of each brush to ensure that the tiles we create do not overlap each other. We could have used the step variant of generate\_series instead generate\_series(start,end, step) as is done in the wiki example. Since we are doing a cross join – our final output will consist of 10201 records ((50 + 1 + 50) \* (50 + 1 + 50)).

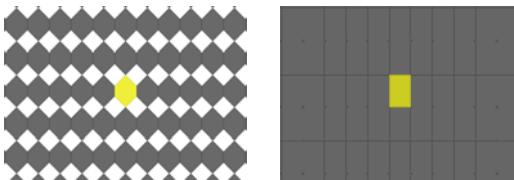


Figure 8.6 The center portions of the hexagonal and rectangular grids generated with code from Listing 8.26. The highlighted center tiles are the location of where our paintbrush CTE is situated.

### **8.5.1 Increase and decrease size of geometry**

The scaling family of functions consists of two versions ST\_Scale(geometry, xfactor, yfactor) and ST\_Scale(geometry, xfactor, yfactor, zfactor). Both preserve dimension in the sense that if you pass in a 3-D geometry, you will get back a 3-D geometry.

Scaling takes every coordinate and multiplies it by the factor parameters. If you pass in a fractional factor, then you will shrink the geometry. If you pass in negative factors, you will end up flipping the geometry in addition to any scaling.

#### **Listing 8.27 Example scaling a hexagon to different sizes**

```

SELECT xfactor, yfactor, ST_Scale(hex.the_geom, xfactor, yfactor) As
scaled_geometry
FROM
( SELECT
--[1 Beg]
ST_GeomFromText('POLYGON((0 0,64 64,64 128,0 192,-64 128,-64 64,0 0))') As
the_geom
As hex
--[1 End]
--[2 Beg]
CROSS JOIN (SELECT x*0.5 As xfactor
            FROM generate_series(1,4) As x) As xf
CROSS JOIN (SELECT y*0.5 As yfactor
            FROM generate_series(1,4) As y) As yf;
--[2 End]
[1] hex to scale
[2] scaling values

```

In this example we are starting with a [1] hexagonal polygon and shrinking and expanding the geometry in the x and y directions from 50% of its size to twice its size by using a cross join that generates numbers from 0 to 2 in x and 0 to 2 in y incrementing  $\frac{1}{2}$  for each step.

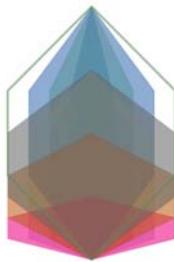


Figure 8.7 Diagram of query in Listing 8.27. The black dark is the original hexagon (xfactor: 1, yfactor: 1) and the larger thick lined one is the hexagon scaled to twice its size (xfactor: 2, yfactor:2).

The above diagram – when you scale it simply multiplies our coordinates and since our hexagon starts at the origin, all our resulting geometries have a base at the origin. Normally when you scale, you want to maintain the centroid constant so you would use a combination of scale and translation.

### **Listing 8.28 Combining Scale and Translation to maintain centroid**

```

SELECT xfactor, yfactor, ST_Translate(ST_Scale(hex.the_geom, xfactor,
yfactor),
ST_X(ST_Centroid(the_geom))*(1 - xfactor), ST_Y(ST_Centroid(the_geom))*(1 -
yfactor) )  As scaled_geometry
FROM
( SELECT ST_GeomFromText('POLYGON((0 0,64 64,64 128,0 192,-64 128,-64 64,0
0))') As the_geom
As hex
CROSS JOIN (SELECT x*0.5 As xfactor
            FROM generate_series(1,4) As x) As xf
CROSS JOIN (SELECT y*0.5 As yfactor
            FROM generate_series(1,4) As y) As yf;
    
```

This example is similar to our previous. In this we are scaling a hexagon from half its size to twice its size in x and y directions. We are then translating the resulting scaled geometry so that the new centroid is where the original hexagon centroid was.

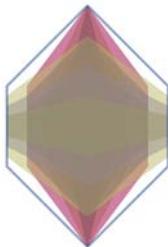


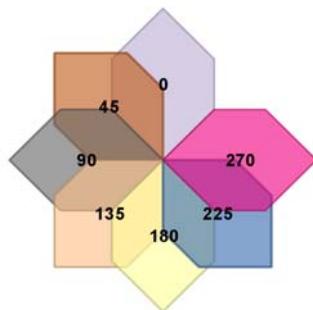
Figure 8.8 Result of query in Listing 8.28. This demonstrates scaling and then translating to maintain the original centroid position. The darkened black bordered geometry is our original hexagon and the outer most hexagon is our hexagon scaled to twice its size in all directions. The various spectrums are permutations of x and y coordinate scaling from 0.5 to 2 in 0.5 increments.

### 8.5.3 Rotate a geometry

The ST\_Rotate, ST\_RotateX, ST\_RotateY, ST\_RotateZ are used to rotate a geometry about the X, Y or Z axis in radian units. ST\_Rotate and ST\_RotateZ are exactly the same since the default axis rotation is Z for most 2D applications. These functions are rarely used in isolation because their default behavior is to rotate the geometry around an axis rather than about the centroid. Rather we almost always combine ST\_Rotate with some translation to achieve rotation about the centroid.

#### **Listing 8.29 Example of ST\_Rotate rotating a hexagon from 0 to 270 degrees**

```
SELECT rotrad/pi()*180 As deg,
       ST_Rotate(hex.the_geom,rotrad)   As rotated_geometry
  FROM
    ( SELECT ST_GeomFromText('POLYGON((0 0,64 64,64 128,0 192,-64 128,-64 64,0
0))') As the_geom)
      As hex
CROSS JOIN (SELECT 2*pi()*x*45.0/360 As rotrad
            FROM generate_series(0,6) As x) As xf;
```



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Figure 8.9 Result of query in Listing 8.29 of rotating a hexagon from 0 to 270 in 45 degree increments

As we can see in the figure our original polygon which happens to have its foot on the 0,0 axis is rotated about its foot. If we had our polygon way out in the middle of space, the rotation radius would be huge.

Most people desire a rotation around the centroid when rotating objects which requires doing a combination of translation and rotation.

There is a simple function available in the PostGIS wiki called RotateAtPoint <http://trac.osgeo.org/postgis/wiki/UsersWikipgsqlfunctions> which will allow you to rotate a geometry about any point. In the following example we use this function to rotate a geometry about its centroid.

### **Listing 8.30 Example of ST\_Rotate in combination with ST\_Translate to rotate about centroid**

```
--[1 Beg]
CREATE OR REPLACE FUNCTION RotateAtPoint(the_geom geometry,
    pt_x double precision, pt_y double precision,
    rotarads double precision)
RETURNS geometry AS
$$
SELECT ST_Translate(ST_Rotate(ST_Translate($1,-1*$2,-1*$3),$4),$2,$3)
$$
LANGUAGE 'sql';
--[1 End]

--[2 Beg]
SELECT rottrad/pi()*180 As deg,
RotateAtPoint(hex.the_geom,ST_X(ST_Centroid(hex.the_geom)),
ST_Y(ST_Centroid(hex.the_geom)), rottrad)) As rotated_geometry
FROM
( SELECT ST_GeomFromText('POLYGON((0 0,64 64,128,0 192,-64 128,-64 64,0
0))') As the_geom
    As hex
CROSS JOIN (SELECT 2*pi()*x*45.0/360 As rottrad
            FROM generate_series(0,1) As x) As xf;
--[2 End]
[1] RotateAtPoint function
[2] rotate hexagon about centroid
```

In this example [1] we have installed the RotateAtPoint function from the PostGIS wiki - <http://trac.osgeo.org/postgis/wiki/UsersWikipgsqlfunctions> which we use in our [2] query. This query takes a hexagon and returns 2 records – the original hexagon, and a new hexagon which is the original rotated 45 degrees.



Figure 8.10 Result of query in Listing 8.30. The gray black is our original hexagon and the shaded geometry is our hexagon rotated 45 degrees.

## 8.6 Summary

In this chapter we learned a lot about leveraging the power of SQL with the power of space. We didn't focus too much about making fast speedy queries and for the examples we had given the limited amount of data, speed was a non-issue. In more real scenarios, you will have lots of data, hopefully lots of people pounding on the database for that data, and you will want to get every inch of speed out of the database you can. Achieving snappy queries requires attention to things like indexes, function costing, simplifying geometries so they are only as complicated and weighty as you need them, and even what constructs you use in writing your queries. In the next chapter, we will focus more on performance, how to analyze performance, and how to improve on it.

## 9

# *Performance tuning*

This chapter covers

- Planner basics
- Reading plans
- Common query patterns
- Geometry processing for better performance
- Influencing plans

When dealing with several tables at once, especially large ones, how you write your queries and how you tune your tables and geometries becomes a major consideration. The main reason is because the same statement expressed in 2 or 3 different ways can have vastly different performance. The complexity of your geometries, your memory allocations, and even storage considerations affect performance.

The query planner has many options to choose from when joining tables. The planner can choose certain indexes over others, the order in which it navigates these indexes, and which navigation strategies (Nested Loops, Bitmap scan, index scan, Hash Joins, etc). All these play a role in the speed and efficiency of the queries.

It is partly true that SQL is a declarative language that allows you to state a request without worrying about the path that is eventually taken to satisfy it. In fact the database planner may use one approach one day and for the same query use a different approach the next day because the distribution of data has changed. In practice, how you state your question often greatly influences the path the planner takes to answer it and that path impacts speed. This is why all high-end databases provide what are called “explain plans” or “show plans” to let you get a glimpse at how the planner is strategizing. SQL only allows you to ask questions and not define explicit steps, but you should still take care in how you ask questions.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

## 9.1 The query planner

All relational databases employ a query planner to digest the raw SQL statement prior to actually executing the query. The most important thing to keep in mind when writing any query is that the planner is not perfect and has an easier time optimizing certain SQL statements over others. The query planner breaks down an SQL query into execution steps and decides which indexes, if any, and navigation strategies will be used based on various heuristics and its knowledge of the data distribution. What it knows that you often don't know is how your data is distributed at any point in time. It will not relieve you of having to write efficient queries.

The classic examples of this we can see in the PostGIS spatial world.

- Asking for the top 5 closest objects, which we covered in chapter 5. You can ask for the top 5 closest (which forces the PostgreSQL planner to do a table scan of all records and rank all by distance and pick the top 5), or you can ask for the top 5 closest within 10 miles. In the second case, the planner can use a spatial index to throw out all objects that are not within 10 miles, and then just scan the remaining. You don't care about the 10 miles and don't really want to make that a requirement, but it makes the planner's task simpler. An example of what the two different SQL statements look like is shown below:

- **The fast way -**

```
SELECT restaurant_name
FROM restaurants
WHERE ST_DWithin(ST_GeomFromText(...), restaurants.the_geom, 10)
      ORDER BY ST_Distance(ST_GeomFromText(...),
      restaurants.the_geom) LIMIT 5;
```

- **The very slow but more obvious way -**

```
SELECT restaurant_name
FROM restaurants
      ORDER BY ST_Distance(ST_GeomFromText(...), restaurants.the_geom)
LIMIT 5;
```

- Another is what we called the Left-handed trick (or LEFT JOIN trick). In this case you want to know everything that doesn't fit a particular criterion, but the straightforward way often (not always) leads to inefficient planner strategies. So instead, you ask to collect all that meet a criteria which is capable of using an index as well as what doesn't and throw out the meeting criteria. This surprisingly happens to work pretty well for most relational databases, where as the straightforward question, is not as often converted to an efficient strategy.

We'll delve into this more as well as other planner topics, and PostgreSQL settings as we examine real case scenarios.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### 9.1.1 Planner statistics

The planner uses planner statistics collected about the data as well as various server configurations such as allocated memory, shared buffers, seq costs, etc to make its decision.

Most relational databases use planner statistics as input into their planner cost strategies. Planner statistics are updated in PostgreSQL when you do a:

```
vacuum analyze verbose sometable
```

or during one of PostgreSQL's automated vacuum runs if you have autovacuum enabled.

Note that from PostgreSQL 8.3 and above autovacuum is enabled by default unless you explicitly disable it in your postgresql.conf file. In addition in PostgreSQL 8.3 on, you can selectively set the frequency of vacuum runs or turn off automated vacuuming for certain tables if you want. Selectively controlling vacuum settings for problem tables is generally a better option than completely disabling autovacuum.

A vacuum analyze will both get rid of dead rows as well as update planner statistics for a table. For bulk inserts and updates, its best to just do a vacuum analyze of the table after the load rather than waiting for PostgreSQL to do its vacuum run.

You can also do a plain:

```
analyze sometable verbose
```

If you just want to update the statistics without getting rid of dead tuples.

Planner statistics are a summary of the distinct values in a table and a simple histogram of the distribution of common values in a table. You can get a sense of what they look like by first updating statistics with

```
vacuum analyze us.states;
```

and then running a query:

```
SELECT attname AS colname, n_distinct,
       array_to_string(most_common_vals, E'\n') AS
common_vals,
       array_to_string(most_common_freqs, E'\n') AS
dist_freq
  FROM pg_stats
 WHERE schemaname = 'us' and tablename = 'states';
```

The result of the above query looks like:

**Table 9.1 Result of planner statistics query**

colname	n_distinct	common_vals	dist_freq
gid	-1		
state	-1		
state_fips	-1		
order_adm	-0.962264	0	0.0566038
month_adm	-0.226415	December	0.169811

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

: January	: 0.132075
: June	: 0.113208

:

A -1 in the n\_distinct column implies the values are pretty much unique across the table in that column. A number less than one tells you the percentage of records that are unique. If you see a number greater than one in the n\_distinct column, then that is usually the exact number of distinct records found. The common values column lists the most commonly observed values. If you look at the month\_adm – it tells us December is the most common month and since our n\_distinct number is relatively high – we can expect that about 70% of the records fall in our common values section for that column. This is very useful to the planner for devising plans because it can use this information to decide the order to navigate tables and apply indexes as well as the strategy. It can also guess at whether a nested loop is more efficient than a hash and so forth by looking at the where and join conditions of a query and estimating the number of results from each table.

### PLANNER STATISTICS SAMPLING

The planner analyzes a sample of the records when analyze is run. The number of records sampled is usually about 10% but varies depending on size of table and the default\_statistics\_target. Note you can also set planner statistics separately for each column in a table using ALTER TABLE ALTER COLUMN somecolumn SET STATISTICS somevalue. We cover this in more detail in Appendix D.

In the next section we'll look into the mind of the planner and investigate how it is thinking about the queries asked of it.

## 9.2 Using explain to diagnose problems

There are a couple of items you look for when troubleshooting query performance.

- What indexes if any are being used?
- What is the order of function evaluation?
- What order are the indexes being applied?
- What strategy is being used – Nested loop, hash join, merge join, bitmap etc.?
- What are the calculated versus actual costs?
- How many rows are being scanned?

In this section we'll go over all those considerations and demonstrate how to infer them by looking at sample query plans. PostgreSQL like most relational databases allows you to view execution and planned execution plans.

### Explain in other relational databases

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

If you are coming from another relational database such as MySQL, SQL Server, or Oracle -- you'll recognize the PostgreSQL explain plan as a parallel to the following:

**MySQL** - same as PostgreSQL -- EXPLAIN sql\_goes\_here

**Oracle** - EXPLAIN PLAN for sql\_goes\_here

**SQL Server** - has both a graphical (built into Enterprise Manager, Studio or Studio Express) as well as a text explain plan similar to PostgreSQL. The graphical explain in SQL Server is much more popular than the below text explain.

```
SET SHOWPLAN_ALL ON  
GO  
sql_goes_here
```

There are three levels of explain plans in PostgreSQL:

- EXPLAIN – which doesn't try to run the query but just provides the general approach that will be taken without extensive analysis.
- EXPLAIN ANALYZE – this actually runs the query but doesn't return an answer. It generates the true plan and timings without returning results. As a result it tends to be much slower than a simple EXPLAIN and pretty much at least the amount of time needed to run the query (minus network effects of returning the data). In addition to estimated rows it provides actual row counts and timings for each step. In 8.4+ it also provides amount of memory used. Comparing the actual against the estimate is a good way of telling if your planner statistics are out of date.
- EXPLAIN ANALYZE VERBOSE - does an in-depth plan analysis, and for PostgreSQL 8.4+ also includes more information such as columns being output.

### **POSTGRESQL 8.4 EXPLAIN AND PLANNER CHANGES**

EXPLAIN ANALYZE VERBOSE provides you with the columns being pulled in the query. This is useful for alerting you when you are using the evil SELECT \* and how costly it is. EXPLAIN ANALYZE provides memory utilization.

For the exercises to follow we'll be using some of our pre-generated as well as our loaded data.

#### **9.2.1 Text Explain vs. PgAdmin III Graphical Explain**

There are two basic kinds of plan displays you can use in PostgreSQL; textual explain plans and graphical explain plans. Each caters to a different audience or a different state of mind. We enjoy using both, but for most general use, we find the graphical explain easier to scan, more visually appealing and a good rule of thumb as to where to focus our efforts. In this

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

section we'll experiment with both. There are many PostgreSQL tools that provide a graphical explain plan and textual explain with differences in look, ability to print etc. For this study we will just focus on the PgAdmin III graphical explain plan that is packaged with PostgreSQL and the native raw text explain plan output by PostgreSQL.

#### **WHAT IS A TEXT EXPLAIN**

A textual explain is the raw format of an explain output by the database. This is a common feature that can be found in most relational databases, but PostgreSQL's explain tends to be richer than most databases. The textual explain in PostgreSQL is presented as indented text to demonstrate the ordering of operations and the nesting of sub operations. You can output it using psql or PgAdmin III. For outputting nicely formatted textual explains, the psql interface tends to be a bit better than PgAdmin III. There is also an online plan analyzer that outputs text plans nicely and highlights rows that you should be concerned about - <http://explain.depesz.com/help>. It is still a work in progress so doesn't handle all plan outputs reliably.

#### **PLANNER CHANGES IN POSTGRESQL 9.0**

One of the new changes in PostgreSQL 9.0 is the ability to output the text explain in XML and JSON format. This should provide more options of how you analyze and view explain plans. We have an example of prettifying JSON using JQuery in [http://www.postgresonline.com/journal/archives/174-pgexplain90formats\\_part2.html](http://www.postgresonline.com/journal/archives/174-pgexplain90formats_part2.html)

It in general provides more information than a graphical explain plan, which we shall cover shortly, but tends to be harder to read and also sometimes provides too much information.

#### **WHAT IS A GRAPHICAL EXPLAIN**

A graphical explain plan is a very beautiful thing. It shows a diagram of how the planner is navigating the data, what functions it is processing, what kinds of strategies its using and all this in bright beautiful glowing icons and colors. The PgAdminIII graphical explain plan is quite attractive to look at. Its got cute little icons for window aggs, hash joins, bitmap scans, CTEs and provides tool tips as you mouse-over the diagram. It is very similar in flavor to Microsoft SQL Server show plan, but sadly you can not save the plan for later analysis as you can in SQL Server.

In the next set of exercises we'll look at some sample plans of queries and describe what each is telling us. We will look at each in its raw intimidating textual explain form and its user friendly cutesy PgAdmin III graphical presentation.

#### **9.2.2 *The plan with no index***

We purposely did not index our tables so that we could demonstrate what a plan without an index looks like.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### EXAMPLE SAN FRANCISCO BRIDGES AGAIN

In this example we look at the simple intersects query we looked at in the last chapter. We will demonstrate the 3 textual plans EXPLAIN, EXPLAIN ANALYZE, and EXPLAIN ANALYZE VERBOSE to see what each further level of analysis provides. We will then follow-up with the graphical explain of it.

```
EXPLAIN SELECT c.city, b.bridge_nam
  FROM sf.cities AS c INNER JOIN
       sf.bridges AS b ON ST_Intersects(c.the_geom, b.the_geom);
```

The textual query plan of the above explain looks like

QUERY PLAN

```
-----  
Nested Loop  (cost=14.40..1185.55 rows=1 width=128)  
  Join Filter: ((c.the_geom && b.the_geom) AND  
                 _st_intersects(c.the_geom, b.the_geom))  
    ->  Seq Scan on cities c  (cost=0.00..21.15 rows=115 width=9809)  
    ->  Materialize  (cost=14.40..18.40 rows=400 width=150)  
          ->  Seq Scan on bridges b  (cost=0.00..14.00 rows=400 width=150)
```

The basic explain gives us the strategy the database would take to answer this question and also its basic estimates of the cost of each step. It doesn't actually run the query, so this explain is generally faster than the others and for more intensive queries significantly faster. From the above we see the secret sauce of the ST\_Intersects (that the planner sees it as two functions -- an && operator that does a bounding box intersect check and the \_st\_intersects that does the more intensive intersect checking. You will only see this behavior with functions written in SQL since sql functions are often in-lined in queries so are transparent to the planner. This allows the planner to reorder the function, even splitting it into two and evaluating them out of order. An in-lined function is generally a good feature, but can be bad too if it distracts the planner from more important analysis or encourages it to use an index where non-index is more efficient. In the next example we'll see the same query but using the explain analyze which actually runs the query without outputting the results.

In this next example, we repeat the same SQL but using EXPLAIN ANALYZE to inspect it.

```
EXPLAIN ANALYZE SELECT c.city, b.bridge_nam
  FROM sf.cities AS c INNER JOIN
       sf.bridges AS b ON ST_Intersects(c.the_geom, b.the_geom);
```

The result of the explain analyze looks like:

QUERY PLAN

```
-----  
Nested Loop  (cost=14.40..1185.55 rows=1 width=128)  
  (actual time=135.028..159.759 rows=8 loops=1)  
  Join Filter: ((c.the_geom && b.the_geom)  
                AND _st_intersects(c.the_geom, b.the_geom))  
    ->  Seq Scan on cities c  (cost=0.00..21.15 rows=115 width=9809)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

(actual time=31.796..32.277 rows=115 loops=1)
-> Materialize (cost=14.40..18.40 rows=400 width=150)
   (actual time=0.148..0.150 rows=4 loops=115)
-> Seq Scan on bridges b
   (cost=0.00..14.00 rows=400 width=150)
   (actual time=16.930..16.937 rows=4 loops=1)
Total runtime: 163.551 ms

```

We see the explain analyze provides a bit more information. In addition to the plan, it provides us with actual timing and total time as well as the number of rows being traversed. We can see for example, the slowest part of our query is the nested loop. Nested loops tend to be the slowest bottlenecks, but in some cases are necessary. As a general rule of thumb you want to minimize the number of rows that fall in a nested loop check.

In the next example we'll look at the same query with verbose added.

```
EXPLAIN ANALYZE VERBOSE SELECT c.city, b.bridge_name
FROM sf.cities AS c INNER JOIN
     sf.bridges AS b ON ST_Intersects(c.the_geom, b.the_geom);
```

The result with verbose:

```
QUERY PLAN
-----
Nested Loop (cost=14.40..1185.55 rows=1 width=128)
   (actual time=4.114..36.481 rows=8 loops=1)
   Output: c.city, b.bridge_name
   Join Filter: ((c.the_geom && b.the_geom)
      AND _st_intersects(c.the_geom, b.the_geom))
-> Seq Scan on cities c
   (cost=0.00..21.15 rows=115 width=9809)
   (actual time=0.007..0.099 rows=115 loops=1)
   Output: c.gid, c.city, c.area__, c.length__, c.the_geom
-> Materialize (cost=14.40..18.40 rows=400 width=150)
   (actual time=0.001..0.003 rows=4 loops=115)
   Output: b.bridge_name, b.the_geom
      -> Seq Scan on bridges b
         (cost=0.00..14.00 rows=400 width=150)
         (actual time=0.006..0.010 rows=4 loops=1)
            Output: b.bridge_name, b.the_geom
Total runtime: 40.118 ms
```

Now we see the verbose sibling. The verbose version tells us also what columns are being output. Also notice that the time on this one is much lower than our EXPLAIN ANALYZE. This is because since both an EXPLAIN ANALYZE and an EXPLAIN ANALZE VERBOSE run the query, the database already knows how to plan this query and may have the answer partly cached in short-term memory. It doesn't always cache the answer, but does if it decides the cost of caching is cheaper than recalculating the answer. Now we'll take a look at the pretty friendly sibling, the graphical explain.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

To summon the graphical explain in PgAdmin III, you highlight the SQL statement in the query window and click the Explain Query icon and also check the Analyze option if you want a more in-depth tool tip with real analysis. The graphical explain, however, does not know what to do with verbose, so checking verbose will force a plain text explain.

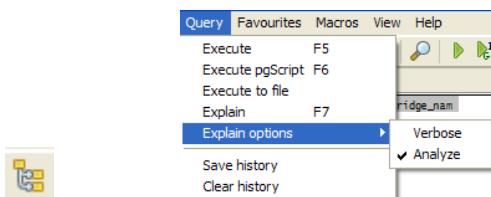


Figure 9.1 Graphical Explain controls

Our pretty sibling looks like the below:

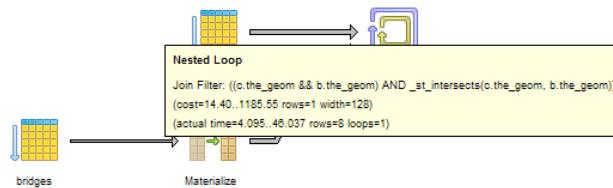


Figure 9.2 Graphical Explain Analyze of our bridge and city intersects query

It is a bit easier to see the order of operation with the graphical explain. The graphical explain always reads left to right while the textual explains the time generally goes from bottom to top. Note that the nested loop is really the last operation to happen. The other thing that is nice about the graphical explain is the way it uses the thickness of the lines to denote the cost of a step. A thicker line such as we see going to Nested loop, means a more costly step. The tool tip feature is both good and bad. Good in the sense that it allows you to focus on one area; this is especially useful if you have a large query. The downside is you can't see all the detail at once as you can with the textual version.

All the aforementioned explains tell us that we are doing sequential scans on both tables and creating a work table (a materialize is basically saying we are creating a temp table). For functions that are very costly to calculate that return few rows and are reused in the query, you really want the planner to materialize and may need to employ tricks to force that.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In the next section we'll see what a plan with indexes looks like by rerunning the same query after we have added spatial indexes and vacuum analyzed the sf.bridges and sf.cities tables.

## 9.3 Indexes and Keys

There are many kinds of indexes you can use in PostgreSQL and in many cases you will want to use more than one index. In almost all cases, you'll want to use a spatial index. In some cases you may want to use a btree or some other access method of index as well.

The main access methods of indexes used in PostgreSQL are btree, gist, and gin and certain kinds of objects such as PostGIS are designed to take advantage of a certain index access method because of the way their data is structured. PostGIS, PgSphere, and Full Text Search objects all leverage gist indexes. Full Text Search also takes advantage of another access method called a Generalized Inverted Tree (gin) index which is basically an R-Tree (implemented with gist) flipped upside down. Most everything else works best with a btree index. The hash index access method is rarely used and these days is considered deprecated because it takes longer to build, not any faster than a btree or gist and worse than a gist for apps that would use it in that fashion.

### 9.3.1 The plan with a spatial index scan

In the previous example we saw what our planner does without the help of indexes. In this exercise, we'll help the planner out a bit by adding indexes to our tables spatial indexes. Observe how the planner reacts to this change of events.

#### **Listing 9.1 Index, Vacuum, Explain**

```
-- 1 index
CREATE INDEX idx_sf_bridges_the_geom
    ON sf.bridges USING gist (the_geom)
    WITH (FILLFACTOR=90);

CREATE INDEX idx_sf_cities_the_geom
    ON sf.cities USING gist (the_geom)
    WITH (FILLFACTOR=90);

-- 2 update stats
vacuum analyze sf.bridges;
vacuum analyze sf.cities;

EXPLAIN ANALYZE VERBOSE SELECT c.city, b.bridge_name
FROM sf.cities AS c INNER JOIN
     sf.bridges AS b ON ST_Intersects(c.the_geom, b.the_geom);
```

#### -- 3 result of explain

QUERY PLAN

```
-----  
Nested Loop  (cost=0.00..22.17 rows=4 width=35)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

(actual time=0.609..30.487 rows=8 loops=1)
Output: c.city, b.bridge_nam
Join Filter: _st_intersects(c.the_geom, b.the_geom)
-> Seq Scan on bridges b  (cost=0.00..1.04 rows=4 width=401)
  (actual time=0.010..0.019 rows=4 loops=1)
    Output: b.gid, b.objectid, b.id, b.bridge_nam, b.the_geom
-> Index Scan using idx_sf_cities_the_geom on cities c
  (cost=0.00..5.27 rows=1 width=9809)
  (actual time=0.048..0.060 rows=3 loops=4)
    Output: c.gid, c.city, c.area__, c.length__, c.the_geom
    Index Cond: (c.the_geom && b.the_geom)
Total runtime: 31.613 ms

```

In the above example we have indexed the tables (1), then (2) updated the statistics by vacuum analyzing, and (3) finally we do a full verbose analyze of our query. The associated graphical explain looks like the below:

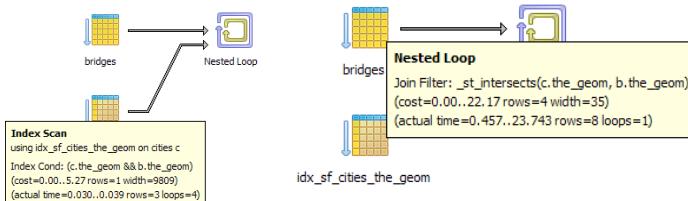


Figure 9.3 Graphical Explain after addition of index – with index tip and then with nested loop tip

In the above we see the graphical representation twice -- one with the tool tip opened to the index and one opened on the nested loop. As you can see, the plan has changed simply by adding in some indexes. The main differences in this plan are:

- The materialized table is gone
- The index on sf cities is being used and an index scan is happening instead of a table scan
- The ST\_Intersects call has been split in order so the && part which uses the index scan happens first and the \_ST\_Intersects more costly call is the only one left in the nested loop. This is a very important thing to look for in spatial queries, because sometimes this doesn't happen even when you have an index. This is one of the most common reasons for slow performance when the index scan happens too late sometimes after the \_ST\_Intersects or happens before a cheaper index scan.
- Most likely as a result of updating stats, our actual and estimated row numbers are closer. The closer the numbers are the better, since closer numbers means the planners estimates of data distribution are more accurate.
- These tables are relatively small so our speed improvement is not significant after cache.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### Planner short-circuiting

The planner employs a common programming tactic called "short-circuiting". Short-circuiting is when a program only processes one part of a compound condition if processing the second part does not change the answer. For example if one part of a logical condition (A and B) returns false or (A or B) returns true, it knows it need not evaluate the other part since the compound answer is not changed. This is a common behavior of relational databases and many programming languages. However, unlike many programming systems which implement short-circuiting, relational databases (PostgreSQL included), generally do not check A B in sequence. They apply which one they consider cheapest first. In the above example with the index, we see the planner, now considers the `&&` as a cheaper check than the `_ST_Intersects`, so processes that first, and will only process the `_ST_Intersects` for those records where `geomA && geomB` is true. For costing it looks at several things; for AND conditions, it will often look at the function cost of a function and uses that in its economic forecast as to how costly the operation is relative to others, but for "Or" compound conditions function costs are ignored. Sometimes the cost of figuring out the cost is too costly, so it will simply process the conditions in order in those cases. So even though the query planner may not process conditions in the order they are stated, it is still best for you to put the one you think is cheapest first.

This example also demonstrates the strengths and weaknesses of the plain text plan versus the graphically enhanced plan. In the plain text plan we can see in one glance how the `ST_Intersects` is broken up. In the graphical one, we need to mouse-over and click to see what is happening.

### 9.3.2 Options for defining indexes

In addition to the various index access methods we just mentioned, you also have various other options to apply to these, which we will go over.

#### PARTIAL INDEX

A partial index allows you to define criteria such as `status='active'` and only data with that condition will be indexed.

The main pros are:

- Makes for a smaller index so less storage
- Index is faster since its lighter and can better fit in memory
- Sometimes forces a more desirable strategy. For example if you have data that has 90% the same in all the rows and 10% are different, and you expect this to be the case for future data, its more efficient for the planner to only scan the index in the 10% if the where condition filters for just that set, and just do a table scan for the other 90% case. The planner has an easier time deciding whether or not to use an index.

The main cons:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- The where condition of the index has to be compatible with the queries it will be used in. This means the column your index filters by has to be used in a query for the planner to know whether the index scan is useful or not.
- Prepared statements can't always use partial indexes. This bites you with parameterized prepared queries. Queries of the form SELECT ... FROM sometable WHERE status = \$1; The reason being is that a plan planned with a parameter, can not assume anything about the value of that parameter, therefore it can't determine whether to use the index or not. This is described in Hubert Lubaczewski's Prepared statements gotcha

<http://www.depesz.com/index.php/2008/05/10/prepared-statements-gotcha/>

- You can't cluster on a partial index

Below is an example of a partial index.

```
CREATE INDEX idx_sometable_active_type
    ON sometable
    USING btree
    (type) WHERE active = true;
```

The above situation may be used if for example we only query active records normally for type and we don't care about type if we are pulling inactive records.

### **COMPOUND INDEX**

PostgreSQL, like most other databases gives you the option of having an index composed of more than one table column or calculated columns but that use the same access method. You can combine this with the aforementioned partial and the functional we will discuss shortly.

#### **What is a compound index?**

A compound index is an index that contains more than one column but the columns are indexed using the same access method (e.g. btree, gist, gin). In PostgreSQL 8.1, the bitmap index plan strategy was introduced which allowed using multiple indexes at once in a plan. The introduction of the bitmap index strategy made the compound index much less necessary since you could combine several single column indexes to achieve the same result. Still in some cases, if you always have the same set of columns in your WHERE or JOIN clause and in the same order, you might get better performance with a compound index since a simple index scan is a bit less expensive than a bitmap index scan strategy. Note that unlike some other databases such as SQL Server that can take advantage of compound indexes to satisfy a query select fields (often called a covering index), PostgreSQL always fetches the rows from disk or cache because PostgreSQL indexes are not MVCC aware. For a spatial index, it will always have to go to disk anyway since the GIST index is lossy (only indexes the bounding box). When people refer to

"covering index" they generally mean an index that contains all the columns you need to satisfy the query and does not need to go to disk to pull the real data.

Since most common data types don't have operators for gist access method except for full text search, you can't include them in a gist index, which makes combining them with spatial in a compound generally not possible.

### FUNCTIONAL INDEX

The functional index, sometimes called an "expression index" is one that indexes a calculated value. Two restrictions:

- The arguments to the function need to be fields in the same row though your function can take many columns as argument as well as constants. You can't go across rows or use aggregates etc.
- The function used must be marked immutable -- which means the same input always returns the same output regardless of query it is run in and it can be assumed to be not changing and does not rely on tables.

### WHEN IMMUTABLE FUNCTIONS CHANGE

If you find yourself needing to change the definition of an immutable function that would change its output and this function is used in an index, you should reindex your table otherwise you will run into potentially bizarre results.

Functional indexes are pretty useful things and especially useful for spatial queries. As mentioned earlier, which is a bit of a no-no, we use a ST\_Transform functional indexes of the form:

```
CREATE INDEX idx_sometable_the_geom_2163 ON sometable USING  
gist(ST_Transform(the_geom,2163) );
```

For PostGIS 1.5, an even more useful functional index to use might be a geography index against a geometry table. This would give you the option of storing data in some UTM like geometry projection for display, advanced processing and demonstration and yet be able to do long range distance filters that would span multiple UTMs and still be able to use a spatial index. When you do this, you'd probably want to use a view to simplify your queries. The utility of this is debatable and probably doesn't work well if you are using 3<sup>rd</sup> party rendering tools where you can't completely control the behavior of the generated query. This approach has not been explored and we throw it out as food for thought. Below is an example of such an approach (requires PostGIS 1.5 or above):

We can create a geography functional index by:

```
CREATE INDEX idx_sometable_the_geom_geography  
ON sometable  
USING gist  
(geography(ST_Transform(the_geom,4326))) ;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Then we can use this index, by compartmentalizing our geography calculated field version in a view.

```
CREATE VIEW vwsometable AS
    SELECT *, geography(ST_Transform(the_geom, 4326)) As the_geog
    FROM sometable AS t;
```

Only when the record is pulled will the full geography output need to be calculated. For most cases, where we are using geography as a filter in the WHERE clause such as shown below, the indexed value will be used.

```
SELECT s1.field1 As s1_field1, s2.field1 As s2_field1
    FROM vwsometable As s1 INNER JOIN
        vwsometable AS s2
    ON(s1.gid <> s2.gid AND
    ST_DWithin(s1.the_geog, s2.the_geog, 5000));
```

For the above query, the spatial index might not be used if the costs on some functions, are set too low, because in those cases the planner given bad information underestimates the cost of calculation vs. the cost of reading from the index and may avoid the index all together. We'll revisit this when we talk about function costing.

Other common uses for functional indexes in the spatial world, are indexing things such as ST\_Area, ST\_Length calculations where you don't necessarily want to store them or put a trigger on your table to keep them up to date, but you filter by them a lot and you want that filtering to be an indexed search. For geocoding, soundex is a common favorite. Below is an example of a soundex index and how it can be used.

## USING SOUNDEX

The soundex function is not installed by default in PostgreSQL. To use it, you need to run the **share\contrib\fuzzystrmatch.sql** file to load it. The fuzzy string match module contains other useful functions such as our favorite levenshtein (which returns the levenshtein distance between two strings – the least number of character edits you need to make to convert the first string to the second string).

In the next example we'll demonstrate how to create a soundex index on a table and how to use it in a query. Keep in mind that you can use any function that is marked immutable in a functional index.

```
CREATE INDEX idx_sf_stclients_streets_soundex
    ON sf.stclines_streets
    USING btree
    (soundex(street));
```

Then we update stats and clear dead tuples to make sure we get the best plan possible.

```
vacuum analyze sf.stclines_streets;
```

Finally we do our select using soundex.

```
SELECT DISTINCT street from sf.stclines_streets
WHERE soundex(street) = soundex('Devonshyer');
```

Soundex is great for the spelling challenged. The above query would return "DEVONSHIRE" whether or not you have a soundex index in place. Note that since soundex doesn't care about string casing, you can use it without upper/lowering your case to match your data. The above query finishes in 16ms with the soundex index and without the index it takes about 30ms. So in this case the index doesn't add much since the speed is already pretty good, but adds a lot of speed if you are dealing with a huge number of records.

#### **PRIMARY KEYS, UNIQUE INDEXES AND FOREIGN KEYS**

There is a lot of debate as to whether foreign keys are good. As far as Primary keys and unique keys go, I think most database specialists would consider those a must have.

The planner uses information about primary key/unique index to know when to stop scanning for matches. This is especially important if you are using inherited tables, since a primary key on the parent (though kind of meaningless), fools the planner into thinking it is unique and it can stop checking once it hits a child with the requested key. There isn't really much of a difference between primary keys and a unique index except for the following:

- Only one primary key can exist per table while you can have multiple unique indexes, though it can be a compound key (composed of multiple columns).
- Only primary keys can take part in foreign key relationships as the primary
- A primary key can not contain nulls, but a unique key can and can have many, but nulls are generally ignored when considering uniqueness

Both have the side benefit of ensuring uniqueness (except in case of nulls) so will prevent duplication of data.

What about foreign keys that enforce referential integrity? A lot of people complain they impact performance. They really only impact performance during updates/inserts and in most cases negligibly unless you are constantly updating the key fields. While foreign keys in and of themselves don't improve performance, they help in 3 indirect ways we can think of:

- They ensure you don't have orphans, which generally means fewer records for the planner to scan thru to waste time and in addition with CASCADING delete/update, you can let the database maintain this for you.
- They are self-documenting -- another database user can look at a foreign key relationship and know exactly how two tables are supposed to be joined.
- Lots of third party tools GUI query builders take advantage of this. So when an unsuspecting user drags and drops two tables in a query designer, the builder automatically joins the related fields, which prevents embarrassing mistakes such as a user joining id = id because they are named the same or for compound keys -- forgetting to join one of the related columns.

Sometimes in a query, the planner just refuses to use an index outright that you would expect it to use. There are two main reasons for this:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- It really can't use the index because you didn't set it up right. A common example of this is the btree index and how btree indexes behave has changed from version to version. We'll go more into detail about this in Appendix D.
- The second reason is that a table scan is just more efficient. If you think of an index as a thin table, scanning an index is not a completely free option. It costs additional reads, so the planner has the decision to make of if the effort of reading the index to determine location of records to pull is less costly than just scanning the raw data. For small tables or tables with much the same value as what is being searched it will often decide that scanning the table is faster.

Now that we have given you some fat to chew about deciding on indexes, we will next explore how you can write your queries to change how the planner behaves.

## **9.4 Common SQL Patterns and how they affect performance**

In the aforementioned, we were able to force a change of plan by adding in indexes. There are more complex join cases where you can control the plan simply by how you state your query. In this section, we'll demonstrate some of the common approaches for doing that.

Below are some general rules of thumb we will demonstrate in the accompanying exercises:

- Joins are powerful and effective things in PostgreSQL and many relational databases; don't shy away from them
- Try to avoid having many subselects in the SELECT part of your query. If you find yourself doing this, you may be better off with a CASE statement.
- Left joins are great things, but especially with spatial joins, they are a bit slower than an INNER JOIN. If you use one, make sure you really need it.

We will start off first by analyzing the many facets of a subselect statement and how where you position it affects speed and flexibility.

### **9.4.1 SELECT Subselects**

As mentioned in Appendix C, a subselect can appear in the SELECT part, WHERE or FROM part of a query. For large numbers of rows to be returned, you are much better off not using a subselect in the SELECT or WHERE clause because it forces a query for each row particularly if it is a correlated subquery. For small numbers of records to return it depends, but it's generally cleaner to put these in the FROM clause. Sometimes you are just better off not using a subselect at all. If speed becomes problematic, you may want to test out various ways of writing the same statement.

#### **Correlated Subquery**

A correlated subquery is a query that can not stand on its own because it uses fields from the outer query within its body. A correlated subquery always forces a query for each row so often results in slow queries.

The first exercise we will look at is the classic example of how many objects intersect with a reference object.

#### **EXERCISE: HOW MANY STREETS INTERSECT EACH CITY**

For this exercise we will ask this question with two vastly different queries. One is kind of the naive way people new to relational databases approach this problem where they put the subselect in the SELECT. In some cases, counter intuitively to most database folk, this performs better or the same as the conventional JOIN approach. The second approach doesn't even use a subselect and uses the power of joins instead. Although the strategies of these are vastly different, the timings are pretty much the same.

```
EXPLAIN ANALYZE SELECT c.city, (SELECT COUNT(*) AS cnt
    FROM sf.stclines_streets As s
    WHERE ST_Intersects(c.the_geom, s.the_geom) ) As cnt
FROM sf.distinct_cities As c
ORDER BY c.city;
```

The output of the aforementioned analyze is:

```
Sort  (cost=825.49..825.73 rows=98 width=11484)
      (actual time=3663.059..3663.103 rows=98 loops=1)
Sort Key: c.city
Sort Method: quicksort Memory: 22kB
-> Seq Scan on distinct_cities c
(cost=0.00..822.25 rows=98 width=11484)
      (actual time=1.464..3662.481 rows=98 loops=1)
      SubPlan 1
        -> Aggregate (cost=8.28..8.29 rows=1 width=0)
          (actual time=37.367..37.368 rows=1 loops=98)
            -> Index Scan using
idx_sf_stclines_streets_the_geom
      on stclines_streets s
(cost=0.00..8.27 rows=1 width=0)
      (actual time=12.567..37.226 rows=158 loops=98)
        Index Cond: ($0 && the_geom)
        Filter: _st_intersects($0, the_geom)
Total runtime: 3664.534 ms
```

Now we will ask the same query, but not using a subselect at all.

```
EXPLAIN ANALYZE SELECT c.city, COUNT(s.gid) AS cnt
FROM sf.distinct_cities As c
LEFT JOIN sf.stclines_streets As s
  ON ( ST_Intersects(c.the_geom, s.the_geom) )
GROUP BY c.city
ORDER BY c.city;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

The result:

```
GroupAggregate  (cost=649.24..651.20 rows=98 width=14)
  (actual time=3720.125..3737.232 rows=98 loops=1)
    ->  Sort  (cost=649.24..649.49 rows=98 width=14)
      (actual time=3720.102..3726.944 rows=15610 loops=1)
        Sort Key: c.city
        Sort Method: quicksort  Memory: 1350kB
        ->  Nested Loop Left Join  (cost=0.00..646.00 rows=98
width=14)
          (actual time=1.228..3689.535 rows=15610 loops=1)
            Join Filter: _st_intersects(c.the_geom, s.the_geom)
            ->  Seq Scan on distinct_cities c
              (cost=0.00..9.98 rows=98 width=11484)
              (actual time=0.008..0.079 rows=98 loops=1)
                ->  Index Scan using
idx_sf_stclines_streets_the_geom on stclines_streets s
                (cost=0.00..6.48 rows=1 width=332)
                (actual time=0.018..0.682 rows=318 loops=98)
                  Index Cond: (c.the_geom && s.the_geom)
Total runtime: 3738.086 ms
```

In terms of which performs better, using a join generally performs much better than the subselect, the more records your query outputs. When used in views however, the planner is generally smart enough not to compute the column if it is not asked for. In these cases, it is better to put the subselect in the SELECT if you don't need other fields from the subselect table and you know your subselect calculated column is rarely asked for. This is another reason to avoid the greedy SELECT \* especially with views because you have no idea what complicated formula is stuffed in a column.

#### **EXERCISE: HOW MANY CITIES HAVE STREETS, HOW MANY STREETS AND HOW MANY GREATER THAN 1000 FT**

In this exercise, we will demonstrate the danger of subselects. When you see yourself having multiple subselects in your SELECT clause ask yourself if this is really necessary. Again we will demonstrate this query using two different approaches. One is the naive subselect way and one is the join way with the CASE WHEN statement.

#### **Listing 9.2 Subselects gone too far**

```
EXPLAIN ANALYZE SELECT c.city, (SELECT COUNT(*) AS cnt
  FROM sf.stclines_streets As s
  WHERE  ST_Intersects(c.the_geom, s.the_geom) ) As cnt,
  (SELECT COUNT(*) AS cnt
  FROM sf.stclines_streets As s
  WHERE  ST_Intersects(c.the_geom, s.the_geom) AND
ST_Length(s.the_geom) > 1000) As cnt_gt_1000
  FROM sf.distinct_cities As c
  WHERE EXISTS(SELECT s.gid
    FROM sf.stclines_streets As s
    WHERE  ST_Intersects(c.the_geom, s.the_geom) )
ORDER BY c.city;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

The result of the above query looks like

### **Listing 9.3 Explain plan of subselect gone too far query**

```

Sort  (cost=662.59..662.60 rows=1 width=11484)
(actual time=8553.707..8553.709 rows=4 loops=1)
  Sort Key: c.city
  Sort Method: quicksort  Memory: 17kB
->  Nested Loop Semi Join  (cost=0.00..662.58 rows=1 width=11484)
(actual time=16.260..8553.659 rows=4 loops=1)
  Join Filter: _st_intersects(c.the_geom, s.the_geom)
    ->  Seq Scan on distinct_cities c
        (cost=0.00..9.98 rows=98 width=11484)
        (actual time=0.005..0.078 rows=98 loops=1)
    ->  Index Scan using idx_sf_stclines_streets_the_geom on
stclines_streets s
        (cost=0.00..6.48 rows=1 width=328)
        (actual time=0.018..0.166 rows=68 loops=98)
          Index Cond: (c.the_geom && s.the_geom)
SubPlan 1
  ->  Aggregate  (cost=8.28..8.29 rows=1 width=0)
  (actual time=924.351..924.352 rows=1 loops=4)
    ->  Index Scan using idx_sf_stclines_streets_the_geom on
stclin
es_streets s  (cost=0.00..8.27 rows=1 width=0)
  (actual time=312.990..921.197 rows=3879 loops=4)
    Index Cond: ($0 && the_geom)
    Filter: _st_intersects($0, the_geom)
SubPlan 2
  ->  Aggregate  (cost=8.28..8.29 rows=1 width=0)
  (actual time=909.088..909.089 rows=1 loops=4)
    ->  Index Scan using idx_sf_stclines_streets_the_geom on
stclines_streets s
        (cost=0.00..8.28 rows=1 width=0)
        (actual time=305.862..908.947 rows=124 loops=4)
          Index Cond: ($0 && the_geom)
          Filter: (_st_intersects($0, the_geom) AND
(st_length(the_geom) > 1000::double precision))
Total runtime: 8555.406 ms

```

To the untrained eye the above query looks impressive because we have used complex constructs such as subselects and exists and aggregates all in one query and the planner is making full use of index scans and more than one at that. To the trained eye, this is a recipe for writing a really slow and long-winded query. The graphical explain plan of it looks particularly beautiful I think.

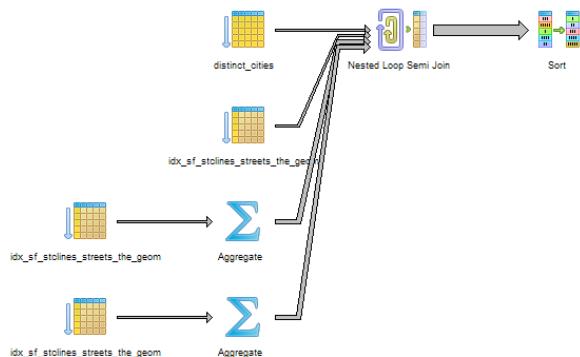


Figure 9.4 Explain analyze graphical plan for many subselect query

While this does look convoluted, it does have its place. It is a slow strategy, but for building things like summary reports where your count columns are totally unrelated to each other except for the date ranges they represent, it is not a bad way to go. It is a very expandable model for building query builders for end users where flexibility is more important than speed and where no penalty is paid if the column is not asked for.

### POSTGRESQL 9.0 JOIN REMOVAL OPTIMIZATION

In PostgreSQL 9.0 there is an enhancement to the planner that allows it to remove unnecessary joins. This feature will make queries that use views with lots of joins but output few fields comparable in speed (or faster) than the subselect approach.

The following is the same exercise satisfied without using a subselect, but using a CASE statement. A CASE statement is particularly useful for writing cross tab reports where you use the same table over and over again and aggregate the values slightly differently.

```
EXPLAIN ANALYZE SELECT c.city, COUNT(s.gid) AS cnt,
    COUNT(CASE WHEN ST_Length(s.the_geom) > 1000 THEN 1 ELSE NULL
END) As cnt_gt_1000
FROM sf.distinct_cities As c
    INNER JOIN sf.stclines_streets As s
        ON ( ST_Intersects(c.the_geom, s.the_geom) )
GROUP BY c.city
ORDER BY c.city;
```

The query plan of the above looks like

#### **Listing 9.4 Query plan of count of streets and min length with no subselects**

```
Sort  (cost=728.48..728.73 rows=98 width=342)
(actual time=3751.973..3751.975 rows=4 loops=1)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

Sort Key: c.city
Sort Method: quicksort Memory: 17kB
-> HashAggregate (cost=723.28..725.24 rows=98 width=342)
(actual time=3751.932..3751.936 rows=4 loops=1)
-> Nested Loop (cost=0.00..646.00 rows=10304 width=342)
(actual time=1.356..3700.814 rows=15516 loops=1)
    Join Filter: _st_intersects(c.the_geom, s.the_geom)
    -> Seq Scan on distinct_cities c
(cost=0.00..9.98 rows=98 width=11484)
(actual time=0.005..0.074 rows=98 loops=1)
-> Index Scan using idx_sf_stclines_streets_the_geom on
stclines_streets s
(cost=0.00..6.48 rows=1 width=332)
(actual time=0.018..0.667 rows=318 loops=98)
Index Cond: (c.the_geom && s.the_geom)
Total runtime: 3752.642 ms

```

As one can see, this query is not only shorter, but also faster. Also observe that in this case we are doing an INNER JOIN instead of a LEFT JOIN as we had done much earlier. This is because we only care about cities with streets.

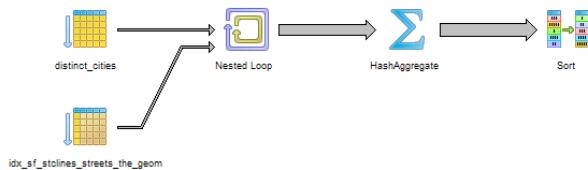


Figure 9.5 Cities with streets and count with min length using no subselects

As we can see from the diagram, the graphical explain plan is simpler, but not as much fun to look at. We do have an exciting HashAggregate in our midst that combines both aggs into a single call as a result of our CASE statement.

#### 9.4.2 FROM Subselects and basic common table expression

A FROM subselect is a favorite of SQLers old and new. It allows you to compartmentalize all these complex calculations as columns into an alias you can use elsewhere in your statement. In PostgreSQL 8.4 ANSI standard common table expression, there is a little twist added to the subselect we know and love. The benefit of the common table expression is that you can reuse the same subselect in as many places as you want in your SQL statement without repeating its definition.

There are a couple of things about subselects used in FROM and in CTEs that are not entirely obvious, even to those with extensive SQL backgrounds

- Though you write a subselect in a FROM as if it was a distinct entity, it is often not. It often gets collapsed in (rewritten if you will). It is not always materialized and the order of its processing isn't even guaranteed.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- For PostgreSQL CTE incarnation, though the ANSI specs don't require it, a CTE always seems to result in a materialization of the work table, though you can't really tell this from the plan since it just shows a CTE strategy.

### **CTE GOTCHA**

Be careful with CTEs. Try to avoid table expressions within your overall CTE that return a lot of records unless the calculations you have in it are really costly, you reuse them in subsequent or the final expression or in several columns and you really need all the records in that expression. The final result of your CTE can return a big table though since the final has to be returned and behaves much like any other query.

For small subselect tables with complex function calculations such as spatial function calculations, you generally want the subselect to be materialized and for large data sets, you don't generally want subselects to be materialized. You can't directly tell PostgreSQL this, but of course you can write your queries in such a fashion to sway it in one direction or another. For PostgreSQL 8.4+ you would just write those expressions as CTE subexpressions to force a materialization. For prior PostgreSQL you can throw in an OFFSET 0 which tricks the planner into thinking there is a costly sort and more likely to materialize or preprocess the costly subselect function calls. Example of using OFFSET is shown below. Note however this is not guaranteed to cache. We recommend this kludge only if you are observing you are having significant performance issues and in those cases doesn't hurt to compare the timings to see which gives you better. For most queries it doesn't make much of a difference, but for some, it is fairly significant.

Here is an example that uses Offset to encourage materialization:

```
SELECT a_gid, b_gid,
       dist/1000 As dist_km, dist As dist_m
  FROM (SELECT a.gid As a_gid, b.gid As b_gid,
              ST_Distance(a.the_geom, b.the_geom) As dist
         FROM poly As a INNER JOIN poly As b
        ON (ST_Dwithin(a.the_geom, b.the_geom, 1000) AND a.gid != b.gid)
      OFFSET 0
     ) As foo;
```

The example encourages a caching of distance calculation by making the subselect look more expensive. Since distance is a fairly costly calculation, if you will use it in multiple locations, you would prefer it to be materialized.

Examples where this situation arises we describe at  
<http://www.postgresonline.com/journal/index.php?archives/127-PostgreSQL-8.4-Common-Table-Expressions-CTE.-performance-improvement.-precalculated-functions-revisited.html>  
and where Andrew Dustan also demonstrates at  
[http://people.planetpostgresql.org/andrew/index.php?archives/49-Well-use-the-old-offset-0-trick.-99\\_.html](http://people.planetpostgresql.org/andrew/index.php?archives/49-Well-use-the-old-offset-0-trick.-99_.html)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### 9.4.3 Windows and Self Joins

The new window function support of PostgreSQL 8.4 is very closely related to the practice of using self joins. In prior versions of PostgreSQL, you could use a self join to simulate the behavior of a window frame. In PostgreSQL 8.4, there are still many cases where a self join comes into play and still can not be mimicked by a window in PostgreSQL. For cases where you can use a window, and you are not concerned about backward compatibility with prior versions of PostgreSQL, then using a window frame approach is much more efficient and results in shorter code as well. Below demonstrates the same spatial query -- one with a window and one with a self-join (pre-PostgreSQL 8.4 way)

#### **Listing 9.5 Rank number results -- using the self join approach (pre-PostgreSQL 8.4)**

```

SELECT count(p3.gid) As rank, main.p2_gid As gid, main.city_2, main.dist
FROM
-- 1
(SELECT p1.city As city_1, p2.city As city_2, p1.the_geom As p1_the_geom,
p2.the_geom As p2_the_geom, p2.gid As p2_gid,
ST_Distance(p1.the_geom, p2.the_geom) As dist, p1.gid As p1_gid
FROM (SELECT city, gid, the_geom FROM sf.cities WHERE city = 'ALBANY') As p1
INNER JOIN sf.cities AS p2 ON (p1.gid <> p2.gid AND
ST_DWithin(p1.the_geom, p2.the_geom, 500) )
-- 2
OFFSET 0
) As main
-- 3
INNER JOIN sf.cities As p3 ON ( ST_DWithin(main.p1_the_geom,
p3.the_geom, 500) )
WHERE (main.p2_gid = p3.gid OR ST_Distance(main.p1_the_geom, p3.the_geom)
< main.dist ) AND main.p1_gid <> p3.gid
GROUP BY main.p2_gid, main.city_2, main.dist
ORDER BY rank, main.city_2;
-- 1 subselect
-- 2 offset hack
-- 3 self-join

```

In the above example, we are employing a lot of techniques in unison. We are (1) using a subselect to define a virtual worktable we will use extensively which determines what cities are within 500 feet of ALBANY. (2) We are using the OFFSET hack described to encourage caching. Without the OFFSET our query takes 2 seconds and with the OFFSET the timing is reduced to 719ms (fairly significant improvement). This is because the costly distance check is not recalculated. (3) We are doing a self-join to collect and count all the cities that are closer to ALBANY than our reference p2 in main. Note the or main.p2\_gid = p3.gid -- this is so our RANK will count at least our reference geom even if there is no closer object.

The above is more efficiently done with a window statement which is a feature supported in many enterprise relational databases and PostgreSQL 8.4+. The next example is the above written using the RANK() window agg function.

**Listing 9.6 Using window aggs to number results - PostgreSQL 8.4+**

```

SELECT RANK() OVER w_dist AS rank,
       p2.city AS city_2, ST_Distance(pl.the_geom, p2.the_geom) AS dist
  FROM sf.cities AS pl INNER JOIN sf.cities AS p2
    ON (pl.gid <> p2.gid AND ST_DWithin(pl.the_geom, p2.the_geom, 500))
   WHERE pl.city = 'ALBANY'
-- 1
WINDOW w_dist AS (PARTITION BY pl.gid
                   ORDER BY ST_Distance(pl.the_geom, p2.the_geom))
                   ORDER BY RANK() OVER w_dist, p2.city;

```

The equivalent window aggregate implementation using the RANK function is a bit cleaner looking and also runs much faster. This runs in about 215ms and the larger the geometries the more significant the speed differences between the past RANK hack and the new one. In (1) we see the declaration of WINDOW -- WINDOW naming is more of a PostgreSQL specific feature that doesn't exist in most other databases supporting windowing constructs. It allows us to default our partition and order by frame and reuse it across the query instead of repeating it where we need it.

Now that we have covered the various ways you can write the same queries and how each affects performance, we'll examine what system changes you can make to affect performance.

## **9.5 System and Function Settings**

Most system variables that affect plan strategy can be set at the server level, session level, or the database level. To set it at the server level simply edit the postgresql.conf file and restart/or reload the PostgreSQL daemon service.

As of PostgreSQL 8.3, many of these can also be set at the function level.

Many of these settings can be set at the session level as well with a  
`SET somevariable TO somevalue;`  
 To set at the database level  
`ALTER DATABASE somedatabase SET somevariable=somevalue;`

To set at the function level - requires PostgreSQL 8.3+  
`ALTER DATABASE somedatabase SET somefunction(argtype1,argtype2,arg...) = somevalue;`

To see the current value of a parameter use:  
`show somevariable;`

Now lets take a look at specific system variables that impact query performance.

### **9.5.1 Key system variables that affect plan strategy**

In this section, we'll cover the key system variables that seem to affect query speed and efficiency the most. For many of these particularly the memory ones, there is no specific right or wrong answer. A lot of the optimal settings depend on if your server is dedicated to  
 ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

PostgreSQL work, the amount of motherboard ram you have and the amount of CPU you have, and even whether your loads are more connection intensive vs. more query intensive. Do you have more people hitting your database asking for simple queries, or is your database a workhorse dedicated to just generating data feeds to other systems for consumption. Many of these settings you may want to set for specific queries and not across the board. We encourage you to do your own tests to determine which settings work best for your loads.

#### **CONSTRAINT\_EXCLUSION**

In order to take advantage of inheritance partitioning effects, this should be set to "on" for PostgreSQL versions prior to 8.4 and set to "partition" or "on" for PostgreSQL 8.4+. This can be set at the server or database level as well as function or statement level. It is generally best to set it at the server level so you don't need to remember it for each db you create. The difference between the older "on" value and the new "partition" value, is that with "partition" the planner doesn't check for constraint exclusion conditions unless it is looking at a table that has children. This saves a bit of planner cycles over the previous "on". The on is still however useful with union queries.

#### **MAINTENANCE\_WORK\_MEM**

This is the amount of memory to allocate for indexing and vacuum analyze processes. When you are doing lots of loads, you may want to temporarily set this to a higher number for a session and keep it lower at the server or db level.

```
SET maintenance_work_mem TO 512000;
```

#### **SHARED\_BUFFERS**

shared\_buffers is the amount of memory the database server uses for shared memory. This is defaulted to 32MB, but you generally want this to be set a bit higher and be as much as 10% of available on-board ram for a dedicated PostgreSQL box. This setting can only be set in the postgresql.conf file and requires a restart after setting.

#### **WORK\_MEM**

work\_mem is the max memory used for sort operations and is set as the amount of memory in kb for each internal sort operation. If you have a lot of on board ram and do a lot of intensive geometry processing and have few users at a time doing intensive things, you want this to be fairly high. This is also a setting you may want to conditionally set at the function level so you can keep it low for general careless users and high for specific functions.

```
ALTER DATABASE postgis_in_action SET work_mem=120000;
ALTER FUNCTION somefunction(text, text) SET work_mem=10000;
```

#### **ENABLE (VARIOUS PLAN STRATEGIES)**

{XE "<\$starstrange>planner strategies" }The enable options are listed below and are all defaulted to true/on. You never want to change these settings at the server or database level, but you may find it useful to set it per session or at the function level if you want to discourage a certain plan strategy that is causing your query problems. It is rare you ever need to turn these off and we personally have never had to. Some PostGIS users have  
©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

experienced great performance improvements by fiddling with these settings on a case by case basis.

```
enable_bitmapscan,           enable_hashagg,           enable_hashjoin,
enable_indexscan,  enable_mergejoin,  enable_nestloop,  enable_seqscan,
enable_sort,      enable_tidscan
```

The enable\_seqscan is one that is useful to turn off to try to force the planner to use an index that it seems it could use but is refusing to. It's a good way of knowing if the planner's costs are wrong in some way or if a tablescan is truly better for your particular case.

There are some of these that even if you set them to off, the setting will not be abided by. This is because in some cases, the planner has no other choice of valid options. Setting these off will discourage the planner from using them but will not guarantee it. These are:

```
enable_sort, enable_seqscan, enable_nestloop
```

To play around with these set them before you run a query. For example turning off hashagg

```
set enable_hashagg = off;
```

And then rerunning our earlier CASE query that used a hashagg will change it to use a GroupAggregate as shown below:

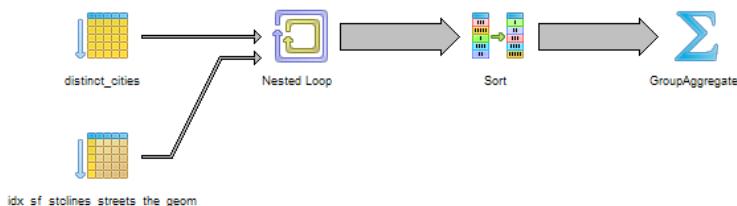


Figure 9.6 Cities streets and count with min length using no subselects after disabling hashagg strategy

Disabling specific planner strategies is useful to do for certain critical queries where you know a certain planner strategy yields slower results. By compartmentalizing these queries in functions, you can control the strategies using function settings. Functions also have specific settings only relevant for functions. We'll go over these in the next section.

```
{XE "<$endrange>planner strategies" }
```

### 9.5.2 Function specific settings

Cost and row settings were introduced in PostgreSQL 8.3. The estimated cost and rows settings are only available to functions. They are part of the definition of the function, so not set separately like the other parameters.

The form is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
CREATE OR REPLACE FUNCTION somefunction(arg1,arg2 ... )
RETURNS type1 AS
...
LANGUAGE 'c' IMMUTABLE STRICT
COST 100 ROWS 2;
```

**COST**

The cost setting is a measure of how costly you think a function is. It is mostly relevant to cost relative to other functions. Versions of PostGIS prior to 1.5 did not have these cost settings set so under certain situations such as big geometries, functions such as ST\_DWithin and ST\_Intersects behave badly and sometimes the more costly process runs before the less costly && operations. To fix this, you can set these costs in your install. You want to set the costs high on the no-public side of the functions \_ST\_DWithin, \_ST\_Intersects, \_ST\_Within and other relationship functions. A cost of 100 for the aforementioned seems to work generally well, though no extensive benchmarking has been done on these functions yet to determine the optimal settings for these.

**PostGIS costing**

Later versions of PostGIS after 1.4 will have these cost settings set. However please note that when you are upgrading your install, your custom cost settings you put in for these functions will be wiped out, so you'll need to reset them after an upgrade.

**ROWS**

This setting is only relevant for set returning functions. It is an estimate of the number of rows you expect the function to return.

**IMMUTABLE, STABLE, VOLATILE**

As shown in our above where we have IMMUTABLE, when you write a function, you can state what kind of behavior is expected of the output. If you don't, then the function is assumed to be VOLATILE. These settings have both a speed as well as a behavior effect.

An immutable function is one whose output is constant over time given the same set of arguments. If a function is immutable, then the planner knows it can cache the result and if it sees the same arguments passed in, it can reuse the cached output. Since caching generally improves speed especially for pricey calculations, marking such functions as immutable is useful.

A stable function is one whose output is expected to be constant across the life of a query given the same inputs. These are functions that can generally be assumed to produce the same result, but can't be treated as immutable because they have external dependencies such as dependencies on other tables that could change. It as a result performs worse than an IMMUTABLE all else being equal, but faster than a volatile.

A volatile function is one that can give you a different output with each call even with same inputs. Functions that depend on time or some other randomly changing or that

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

change data fit into this category since they change state. If you mark a volatile function such as random() not volatile, then it will be faster, but not behave correctly since it would be returning the same value with each subsequent call.

Now that we have covered the various system settings we can employ to impact speed, we'll take a closer look at the geometries themselves. Can we change a geometry so it is still accurate enough for our needs, but performance of applying spatial predicates and operations on it is faster?

## 9.6 Optimizing geometries

Generally speaking, spatial processes and spatial relationship checks take longer the more vertices and holes you have and also are either not doable with invalid geometries or are much slower. In this section we'll go over some of the more common techniques for making valid, optimizing and simplifying your geometries.

### 9.6.1 Fixing invalid geometries

The main reason to fix invalid geometries is

- You really can't use GEOS relationships checks and many processing functions that rely on the intersection matrix with invalid geometries. Things like ST\_Intersects, ST\_Equals etc. return false or throw a topology error for certain kinds of invalidity regardless of the true intersects nature.
- Same holds true with Union, Intersection and the powerful GEOS geometry process functions. Many will not work with invalid geometries.

Most of the cases of invalid geometries are with polygons. The PostGIS wiki provides a good resource for fixing invalid geometries. There is a contrib function called cleanGeometry.sql that does a fairly good job of this.

<http://trac.osgeo.org/postgis/wiki/UsersWikiCleanPolygons>

### POSTGIS 2.0 FIXING INVALID GEOMETRIES

In PostGIS 2.0, a function called ST\_MakeValid was introduced which can be used to fix invalid polygons, multipolygons, multilinestrings and linestrings.

### 9.6.2 Reducing number of vertices with simplification

Reducing the number of vertices you have by simplifying the geometries, has two speed improvement effects.

#### PROS

- It makes your geometries lighter weight which becomes increasingly important the more zoomed out you are on a map
- It makes relationship, distance checks, and geometry processing a bit faster since these functions are generally slower the more vertices you have.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

**CONS**

- Your geometries are less accurate. You are giving up precision for speed.
- You often loose colinearity -- things that used to share edges no longer do etc.

**NEVER SIMPLIFY IN WGS 84 LONG LAT OR OTHER LONG LAT SRIDS**

Simplification assumes a planar model and as such applying it to something to be designed to work with measurement around a spheroid will produce often unpredictable results. The best approach is to transform to a planar coordinate -- preferably one that maintains measurement accuracy, and then retransform back to long lat after the simplification process.

Below is a quick example of simplification where we simplify our state boundaries and then compare performance before and after.

**Listing 9.7 Simplified state vs. non-simplified**

```
-- 1
SELECT a.state AS st_a, b.state AS st_b
FROM us.states AS a INNER JOIN us.states AS b
  ON (a.state != b.state AND ST_DWithin(a.the_geom, b.the_geom,1000) ) ;

-- 2
SELECT state, ST_SimplifyPreserveTopology(the_geom,1500) AS the_geom
INTO us.states_simp1500
FROM us.states;

CREATE INDEX idx_us_states_simp1500_the_geom
  ON us.states_simp1500 USING gist(the_geom);

vacuum analyze us.states_simp1500;

-- 3
SELECT a.state AS st_a, b.state AS st_b
FROM us.states_simp1500 AS a INNER JOIN us.states_simp1500 AS b
  ON (a.state != b.state AND ST_DWithin(a.the_geom, b.the_geom,1000) ) ;
-- 1 21964 ms - 222 rows
-- 2 prepare simplified
-- 3 simplified - 9376 ms - 222 rows
```

In the above we run the usual (1) distance check on our full resolution data. This takes 21,964 ms and returns 222 rows. In (2) we create a new table called us.states\_simp1500 which is our original data simplified to a tolerance of 1500 meters (basically treat points within 1500 meters as being equal). The units are in meters since our data in National Atlas meters. We then run the same query again (3) against this new dataset. It completes in 9,376 ms and returns 222 rows. This was done using PostGIS 1.4.

**Distance algorithm improved in Postgis 1.5**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In PostGIS 1.5, the ST\_DWithin and ST\_Distance functions were improved to better handle geometries with more vertices. As a result, the above simplification is not as stark in PostGIS 1.5 as it is in prior versions.

In many cases you can get away with on the fly simplification and still achieve much of the performance benefit as you get with a stored simplification. The only thing you need to be careful of is not to lose using the spatial index on the table in the process. To achieve this effect, we will create a new ST\_DWithin function that works against the simplified but uses the original geometries for the index check operation so that the index is used.

### **Listing 9.8 Simplify on the fly and still use an index**

```
--[1 Beg]
CREATE FUNCTION upgis_DWithin_Simplify(geom1 geometry, geom2 geometry, dist
double precision,
simplify_tolerance double precision)
RETURNS boolean
AS
$$ SELECT ST_Expand($1, $3) && $2 AND ST_Expand($2, $3) && $1
AND _ST_DWithin(ST_SimplifyPreserveTopology($1,$4),
    ST_SimplifyPreserveTopology($2,$4), $3)
$$
language 'sql' IMMUTABLE;
--[1 End]

--[2 Beg]
SELECT a.state AS st_a, b.state AS st_b
FROM us.states AS a INNER JOIN us.states AS b
    ON (a.state != b.state AND upgis_DWithin_Simplify(a.the_geom,
b.the_geom,1000,1500) ) ;
--[2 End]
--[3 Beg]
SELECT a.state AS st_a, b.state AS st_b
FROM us.states AS a INNER JOIN us.states AS b
    ON (a.state != b.state AND
upgis_DWithin_Simplify(a.the_geom, b.the_geom,1000,4000) ) ;
--[3 End]
-- [1] simplification function
-- [2] 1500 tolerance (14,727 ms 222 rows)
-- [3] 4000 tolerance (8,408 ms 222 rows)
```

In the above example we (1) create a new function that behaves like the built in PostGIS ST\_DWithin function except that it applies a simplification tolerance before doing the distance within check. Note that the index check `&&` is done on the original geometries to utilize the spatial index on the tables. This new function takes in an additional arg than the standard PostGIS ST\_DWithin, and that is the simplification tolerance. We see in (2) we don't get quite as much performance improvement (14.7 seconds) as we did with our similar stored states\_simp1500 (9.8 seconds). However (3) by upping our level of simplification, we get even faster performance (8.5 seconds). So the trick is to find the sweet spot of what to set your simplification to with acceptable loss in accuracy.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

In the next section, we'll demonstrate another kind of simplification, and that is removing unnecessarily small features from our geometries.

### 9.6.3 Removing holes

Depending on what you are doing you may not need to show holes. Holes generally add more processing time to things like distance checks and intersection. To remove them, we would employ something like the below code.

```
SELECT s.gid, s.city, ST_Collect(ST_MakePolygon(s.the_geom)) As
the_geom
FROM (SELECT gid, city, ST_ExteriorRing((ST_Dump(the_geom)).geom)
As the_geom
FROM sf.cities ) As s
GROUP BY gid, city;
```

We use ST\_Dump to dump out the polygons from multipolygons. This is needed because the ST\_ExteriorRing function only works with polygons. We then convert the exterior ring to a polygon since the exterior ring is the linestring that forms the polygon. So we use the common spatial design pattern of “explode, process, collapse”. The “explode, process, collapse” spatial design pattern is probably one of the most ubiquitous of all especially for geometry massaging, similar to a baker preparing dough by kneading to remove the gas pockets.

You may not want to remove all holes, but possibly really small ones that don't add much visible or information quality to your geometry, but that make other checks and processes slower. Below is a simple method for removing holes of a particular size excerpted from this article [http://www.spatialdbadvisor.com/postgis\\_tips\\_tricks/92/filtering-rings-in-polygon-postgis/](http://www.spatialdbadvisor.com/postgis_tips_tricks/92/filtering-rings-in-polygon-postgis/)

#### **Listing 9.9 Filter Rings function and its application**

```
CREATE OR REPLACE FUNCTION filter_rings(geometry, double precision)
RETURNS geometry AS
$$
SELECT ST_BuildArea(ST_Collect(b.final_geom)) as filtered_geom
FROM (SELECT ST_MakePolygon(--[1 Beg]
/* Get outer ring of polygon */
SELECT ST_ExteriorRing(a.the_geom) as outer_ring
--[1 End]
),
ARRAY(--[2 Beg]/* Get all inner rings > a particular area */
SELECT ST_ExteriorRing(b.geom) as inner_ring
FROM (SELECT (ST_DumpRings(a.the_geom)).*) b
WHERE b.path[1] > 0 /* ie not the outer ring */
AND ST_Area(b.geom) > $2
)--[2 End]
) as final_geom
FROM (SELECT ST_GeometryN(ST_Multi($1),
--[3 Beg] /*ST_Multi converts any Single Polygons to MultiPolygons */
generate_series(1,ST_NumGeometries(ST_Multi($1)))
) as the_geom
)
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

        ) a --[3 End]
    ) b
$$
LANGUAGE 'sql' IMMUTABLE;
--[1] outer ring
--[2] big inner rings
--[3] single to multi

```

To put the above function to use with our San Francisco cities we do this:

```

SELECT s.city, filter_rings(the_geom, 51000) As
newgeomnohole_lt5000
FROM sf.cities;

```

which returns each city with a new set of geometries with keeping only holes that are greater than 51000 sqft.

This next query will tell us which records would be changed by the above query.

```

SELECT city, filter_rings(the_geom, 51000) As newgeom
FROM
  (SELECT city, the_geom, (SELECT SUM(ST_NumInteriorRings(geom))
  FROM ST_Dump(the_geom) ) As NumHoles
FROM sf.cities) As c
WHERE c.NumHoles > 0;

```

Note the use of `ST_Dump` in this query. This is needed because the `ST_NumInteriorRings` only returns the number of holes in the first polygon, so if we are dealing with a multi-polygon, we need to expand to polygons and then count the rings. If this is a common construct you use, you can encapsulate it into an SQL function. So again this is the “explode, process, collapse” spatial design pattern at work again. Note this behavior may change in later versions of PostGIS after PostGIS 1.5.

### **9.6.4 Clustering**

In this section, we will talk about two totally different optimization tricks that sound similar and even use the same terminology, but mean different things. We will refer to one as index clustering and the other as spatial clustering (bunching). The term bunching is more colloquial than industry standard. The index clustering concept is one that is fairly common and similarly named in other databases.

- Index Clustering -- by clustering we are referring to the PostgreSQL concept of clustering on an index. This means you maintain the same number of rows, but you physically order your table by an index (in PostGIS usually the spatial one). This guarantees that your matches will be in close proximity to each other on the disk and thus easy to pick. Your index seeks will be faster, since reading the data pages -- each page will have more matches.
- Spatial Clustering (bunching) -- this is usually done with point geometries and reduces the number of rows. This is when you take a set of points usually close to one another or related by similar attributes and collect them into multipoints and aggregate the data. You can imagine in this case you would be talking about 100,000 rows of multipoints vs. 1,000,000 rows of points which can be both a great space

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

savings as well as a speed enhancer since you have fewer index checks.

#### **CLUSTER ON AN INDEX**

We've talked about this before in other chapters, but is worthwhile to revisit. First how do you physically sort your table on an index:

```
ALTER TABLE sf.distinct_cities CLUSTER ON idx_sf_distinct_cities_the_geom;
CLUSTER verbose sf.distinct_cities;
```

Versions of PostgreSQL prior to 8.3 do not allow clustering on a gist (spatial) index that contains nulls in the indexed field. Clustering is also most effective for read only or rarely updated data.

Currently PostgreSQL does not recluster a table, so to maintain order, you need to rerun the CLUSTER ... step. If you run CLUSTER without a table name, then all tables in the database that have been clustered, will be reclustered.

Another important setting, which is specific to tables is the FILLFACTOR. Those coming from SQL Server, will recognize this term. It is basically the target fullness of a database page. During cluster runs, the fill factor tries to be reestablished. For new inserts, the database will keep on adding to a page until it is that percentage full.

For very static tables, you want the FILLFACTOR to be really high like 99 or 100. A higher fill factor generally performs better in queries, since the PostgreSQL can pull more data into memory with fewer pages. For data that you update frequently, you want this number to be the default (90) or less. This is because when you are doing updates, PostgreSQL will try to maintain the order of existing records, by inserting the newly updated row around the same location as where it was before. If there is no space on the page, then it will need to create a new page thus more likely ruining your cluster until you recluster.

FILLFACTOR is set for both tables and indexes. Yes indexes have pages too. To set the FILLFACTOR of a table you do something of the form:

```
ALTER TABLE sf.bridges SET (FILLFACTOR=80);
```

#### **USING MULTIPOLYgons INSTEAD OF POINTS**

For small geometries such as points that share more or less the same attributes, you may want to reduce the number of records by storing them clustered into proximity groups. One example is the location of trees where your proximity checks are just as good if you can talk about certain trees in a family. Intersects checks on these are often faster if you are comparing fewer multipoints to multipoints vs. more records. Below is a query that clusters points together grouping by some key features and proximity. For clustering, ST\_SnapToGrid comes in quite handy.

#### **Listing 9.10 Collecting points into multipoint bunches**

```
SELECT max(obsid) As obsid, ST_Multi(ST_Collect(the_geom)) As the_geom,
       obs_name, max(obs_date) as max_date,
       min(obs_date) As min_date
  -- [1]
 INTO work.observations_bunched
    FROM us.observations
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
-- [2]
GROUP BY ST_SnapToGrid(ST_Transform(the_geom,2163),50000,50000), obs_name;
[1] bulk insert
[2] transform and snap
```

In this example we are taking our observation points we created in WGS 84 longlat and clustering them into 1000x1000 meter grids. [2] We transform to national atlas meters so that our snap to grid measurements will be in meters. [1] We then take the last id as our new id, the collected points that snap to the same grid and share the same name, use the name, and store the min and max observation dates of our collected points. Our new dataset has 8163 records compared to our original 27316.

## 9.7 Summary

In this chapter we covered the various ways of improving performance of spatial queries. We discussed various approaches for writing spatial queries, how to troubleshoot query performance, how to optimize your geometries, and what the common settings in PostgreSQL you can change to impact performance. While many of these techniques were focused on spatial queries, many can be applied to non-spatial queries as well.

PostGIS and PostgreSQL are not islands. They intermingle with various applications and software. The power of PostGIS can only be fully appreciated when you combine it with other tools to build applications or to view outputs. In the chapters that follow, we'll take a closer look at how PostGIS interacts with other tools for viewing and building applications. We will learn how to not only view PostGIS output, but how to make attractive end user applications that leverage its power.

# Part 3

## *Using PostGIS with other tools*

In Part 2 we covered the basics of solving problems with spatial queries and performance tips for getting the most speed out of your spatial queries. PostGIS is a versatile piece of glue that interoperates very nicely with both commercial and other open source tools. In Part 3, we'll cover some of the more common open source tools that are used to complement the functionality of PostGIS.

Chapter 10 covers SQL add-on such as PostgreSQL procedural languages PL/R and PL/Python that are common favorites in GIS for leveraging the wealth of statistical functions and plotting capabilities of R and the numerous packages for Python. We'll learn how to write stored functions in these languages and call them from SQL queries. We will also cover Tiger Geocoder which is a package of scripts, sql functions, and PostgreSQL types that utilizes US Census data to build geocoders and reverse geocoders. In addition we shall cover PgRouting which is another package of SQL functions used to build routing applications and do various kinds of traveling problems.

Chapter 11 we cover the most common server side mapping servers and client side mapping frameworks that are commonly used to display PostGIS data on the web. We'll learn how to display PostGIS data alongside third-party mapping layers such as OpenStreetMap, Google Maps, and Microsoft Bing. We'll also learn the basics of setting up Geoserver and Mapservers and configuring them as WMS/WFS services.

Chapter 12 We'll learn about the more common open source GIS desktop tools used to display PostGIS and the basics of using them. Items covered will be OpenJump, uDig, QuantumGIS, and GvSig.

Chapter 13 - We will be exposed to Raster data, which is another kind of spatial data we haven't delved into in early chapters. We'll learn how you can use raster data along side vector data using the PostGIS raster type. Note that while this is currently packaged as a separate type from the rest of PostGIS, in PostGIS 2.0, it will be fully integrated as another spatial type. This provides up just a glimpse of what is in store in PostGIS 2.0.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

# 10

## *Enhancing SQL with add-ons*

This chapter covers

- Tiger Geocoder
- PgRouting
- PL/R
- PL/Python

In this chapter, we will cover common open source add-on tools that are often used to enhance the functionality of PostgreSQL. What makes these tools special is that you can unleash their power with SQL. This allows for writing more powerful queries than what you can do with PostGIS and PostgreSQL alone. It also allows for greater reusability of logic because you can reuse these same functions across all your application queries as well as your database triggers.

The tools we will be covering are as follows:

- Tiger Geocoder - a geocoding tool kit that includes scripts for loading US Census tiger street data and approximating address locations with this data. It also contains geocoding functions that build on top of PostGIS functions to do its address matching. The benefit of using it instead of a geocoding web service is that you can customize it however you like and you incur no service charges per batch of addresses you geocode.
- PgRouting - a library and set of scripts that build on top of PostGIS functions and various algorithms to perform tasks such as shortest path along a road network, driving directions, and geographic constrained resource allocation problems (aka travelling salesman).
- PL/R - a procedural language handler for PostgreSQL that allows you to write database stored functions using the R statistical language and graphing environment. With this

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

you can generate elegant graphs and leverage a breath of statistical functions to build aggregate and other functions within your PostgreSQL database. This allows you to inject the power of R in your queries.

- PL/Python - a procedural language handler for PostgreSQL that allows you to write PostgreSQL stored functions in Python. This allows you to leverage the breath of python functions for connecting with the network, importing data, geocoding, and other GIS tasks. You can similarly write aggregate functions in Python.

We expect that once you have finished this chapter, you will get a better appreciation of the benefit of integrating this kind of logic right in the database instead of pulling your data out to be processed externally.

## **10.1 Georeferencing with the tiger geocoder**

The Tiger Geocoder is a suite of SQL functions that utilize US TIGER census data. The tiger geocoder makes it easy to batch geocode data with SQL update statements as well as provide geocoding functionality to applications via simple SQL select statements. Although the Tiger Geocoder is specific to Tiger US data structure, the concepts employed by it can be used to create your own custom geocoder for specialized data sets.

### **What is a geocoder?**

A geocoder is a utility that will take a textual representation of an address such as a street address and calculate its geographic position using data such as street centerline geometries. The position returned is usually a long lat point location, though it need not be. The Tiger Geocoder returns a normalized address representation as well as a PostGIS point geometry and a ranking of the match. It utilizes PostGIS linear referencing functions and fuzzy text match functions to accomplish this.

The tiger geocoder packaged with PostGIS 1.5 and below doesn't handle the new US census data ESRI shapefile format. As such, we are using a newer version currently under development by Stephen Frost, which handles the new ESRI shapefile format. You can download this version from the PostGIS in Action book site <http://www.postgis.us> . More details on getting Steve's latest code can be found in Appendix A.

There is another geocoder built on OpenStreetMap that utilizes PostgreSQL functions and a C library. This one may be of more interest to non-US people or OSM data users. This one we didn't have time to investigate and cover. This one is called Nominatim and can be accessed from <http://wiki.openstreetmap.org/wiki/Nominatim> .

For our exercises we have taken Steve Frost's newer version, and made some minor corrections to support the Tiger Census 2009 data. We have also changed the lookup tables script to create skeleton tables we inherit from. We have introduced an additional script file called tiger\_loader.sql that generates a loader batch script for Windows or Linux. We'll

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

briefly describe how to use this custom loader and how to test drive the geocoder functions in this section.

### **10.1.1 Installing the tiger geocoder**

In order to use the Tiger geocoder functionality, you need to do the following things:

- Install the tiger geocoder .sql files
- Have wget and unzip (for Linux) or 7Zip for Windows handy. These we described in Chapter 7 on loading data.
- Use our loader\_generate\_script() SQL function that generates a windows shell or Linux Bash script. This generated script you can then call from the command line to download the specified states data, unzip it, and load it into your postgis enabled database.

The details of all of this can be found in the Readme.txt packaged with the tiger geocoder code on the PostGIS in Action book site. Now that we've outlined the basic steps, we will go into specifics about using this loader\_generate\_script function to load in tiger data.

### **10.1.2 Loading tiger data**

In this exercise we test out our tiger loader. It uses a set of configuration tables to denote differences in OS platforms, tables that need to be downloaded, how they need to be installed, pre and post process steps. The script to populate these configuration tables and generation function "loader\_generate\_script", are in the file tiger\_loader.sql. The key tables are:

- **loader\_platform** - this is a table that lists OS specific settings and locations of binaries. We prepopulated it with a generic linux and windows records. You will probably want to edit this table to make sure path settings for your OS are right.
- **loader\_variables** - these are variables not specific to OS, such as where to download the TIGER files, which folder to put them, temp directory to extract them, and year of data.
- **loader\_lookuptables** - this is a table of instructions on how to process each kind of table and what tables to load in the database. You can set the load bit to false if you don't want to load a table. For the most part, you shouldn't need to change this.
- **loader\_generate\_script** - this is the function that will generate the command line shell script to download, extract and load the data. For our example, we downloaded just Washington DC data since it's the smallest and only has one county. We did that by running the statement:

```
SELECT loader_generate_script(ARRAY[ 'DC' ], 'windows');
```

If you needed more than one state and for a different OS, you would list out the states as follows:

```
SELECT loader_generate_script(ARRAY[ 'DC', 'RI' ], 'linux');
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

The script will generate a separate script record for each state.

You can then copy and paste the result into a .bat file (remove start and end quotes if copying from pgadmin) and then run the shell script.

The generated scripts use WGet, 7zip (or unzip), and the PostGIS packaged shp2pgsql loader to download the files from US Census, Unzip, and load the data into a PostGIS enabled database. For windows users we highly suggest using 7zip and installing the wget for windows as we described in Chapter 7.

For Linux/Unix/MacOSX users, wget and unzip are generally in path, so probably don't need to do anything aside from cding into the folder where you saved your generated script. We use the states\_lookup table in tiger schema to fine tune which specific states to download data for and generate the download path based on new Tiger path convention. Then we have a single sql statement that combines all these to generate either a windows commandline or linux bash script for the selected states.

The state/county specific data is defaulted in the loader\_variables table to store in a schema called tiger\_data. All of these will inherit from shell tables defined in the tiger schema. One set of tables for each state will be generated. Each states set of tables are prefixed with the state abbreviation.

We use inheritance because it is more efficient for large data sets since it allows for piecemeal loading or reloading of data. It has the side benefit that the tiger geocoder doesn't really need to know about these tables to use them transparently via the skeleton parent tables we have setup. It also gives us the option to easily break these tables in to other schemas later however we care.

For all this to work seamlessly, we need to make sure that the tiger schema is in our database search path.

We won't be showing the code here since its too long to include, but you can download the code from the PostGIS in Action book site <http://www.postgis.us> chapter code download file the chapter 10/tiger\_geocoder\_2009 folder and read the ReadMe.txt for more details on how to install and get going with it.

### **10.1.3 Geocoding and address normalization**

In this section, we'll go over the key functions of the Tiger Geocoder package.

#### **GEOCODER**

The main function in the geocoder is called "geocode" and it calls on many helper functions. This function is specific to the way tiger data is organized. If you have non-US data or have more granular data such as city land parcel data, you'll need to write your own geocode function or tweak this one a bit. An example of its use is shown below.

#### **Listing 10.1 Example of geocode function**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

SELECT
    -- 1
    g.rating,
    -- 2
    ST_X(g.geomout) AS lon,
    ST_Y(g.geomout) AS lat,
    -- 3
    (g.addy).*
FROM geocode('1731 New Hampshire Avenue Northwest, Washington, DC 20010')
AS g;
-- 1 get rating
-- 2 lon and lat
-- 3 explode addy into elements

```

The geocode function takes an address and returns a set of records that are possible matches for the address. One of the fields in the geocode function is a **rating** field. For perfect matches, rating will be 0. The greater the number the less accurate the match. One of the objects is a complex type called norm\_addy. A norm\_addy object is output as a field called addy in the returned records. The norm\_addy represents a perfectly normalized address where abbreviations are standardized based on the various \*lookup tables in the tiger schema. In (3) we are exploding addy out into its constituent properties so they appear as individual columns. The addy object has a property for each component of the address, and is a normalized version of the closest match address. (2) The result also includes a field called geomout which is a PostGIS longlat point geometry interpolated along the street segments. We display the long,lat components of this point.

If we want just individual elements of the addy object and not all, then we would write a query something like below.

### **Listing 10.2 Listing specific elements of addy in geocode results**

```

SELECT g.rating,
    -- 1
    round(ST_X(g.geomout)::numeric,5) AS lon,
    round(ST_Y(g.geomout)::numeric,5) AS lat,
    -- 2
    (g.addy).address AS snum,
    (g.addy).streetname || ' ' || (g.addy).streettypeabbrev AS street,
    (g.addy).zip
FROM geocode('1021 New Hampshire Avenue, Washington, DC 20010') AS g;
-- 1 convert to long lat cords
-- 2 extract addy elements

```

In this example we are also testing the power of the soundex/levenshtein fuzzy string matching functionality by feeding invalid and misspelled addresses. In this example we get multiple results back because we fed in an example that has the wrong zipcode and misspelled street. In this case, we get 3 possible results all with slightly different rating. (1) We also want to trim down the number of digits of the long lat, so we are rounding the digits. We first cast to numeric since the round function expects a numeric number and PostGIS returns double precision. This casting may or may not be needed depending on

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

which autocasts you have in place. Instead of round, we could have also used the PostGIS ST\_X(ST\_SnapToGrid(g.geomout, 0.00001)) to truncate the coordinates. (2) We select specific elements out of addy and glue them together for more appropriate output.

The result of the above query produces a table like below.

**Table 10.1 Results of geocoding address in Listing 10.2**

rating	lon	lat	snum	street	zip
10	-77.04961	38.90309	1021	New Hampshire Ave	20037
12	-77.04960	38.90310		New Hampshire Ave	20036
19	-77.02634	38.93359		New Hampshire Ave	20010

Recall we had mentioned before that you shouldn't use long lat data with PostGIS linear referencing and other Cartesian specific functions. In this case it is more or less safe to do so because the street segments are generally short enough that the approximation of the sphere projected to a flat surface (Platte Carre projection), does not distort the results too much. The longer the street segments, the more erroneous your interpolations will be. It is also rare that street numbers are equally spaced along a street, so the interpolation is still a best guess assuming that perfect distribution.

From the above table listing, we can quickly tell that the first one with rating of 10 is most likely the best match. Now if we are batch geocoding a table, we may just want the first and best match and also keep what the rating was. Below is an update statement that will do that.

### **Listing 10.3 Batch geocoding with geocode function**

```
--[1 Beg]
CREATE TABLE ch10.addr_to_geocode(addrid serial NOT NULL PRIMARY KEY,
    rating integer,
    address text,
    norm_address text, pt geometry);
INSERT INTO ch10.addr_to_geocode(address)
    VALUES ('1000 Huntington Street, DC'),
            ('4758 Reno Road, DC 20017'),
            ('1021 New Hampshire Avenue, Washington, DC 20010');
--[1 End]
--[2 Beg]
UPDATE ch10.addr_to_geocode
--[3 Beg]
    SET (rating, norm_address, pt)
        = (g.rating,
           COALESCE ((g.addy).address::text, '')
           || COALESCE(' ' || (g.addy).predirabbrev, '')
           || COALESCE(' ' || (g.addy).streetname,'')
           || ' ' || (g.addy).streettypeabbrev
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

    || COALESCE(' ' || (g.addy).location || ', ', '')
    || COALESCE(' ' || (g.addy).stateabbrev, '')
|| COALESCE(' ' || (g.addy).zip, '')

                                , ST_SnapToGrid(g.geomout, 0.000001)
) --[3 End]
--[4 Beg]
FROM (SELECT DISTINCT ON (addid) addid,
      (geocode(address)).*
FROM ch10.addr_to_geocode As ag
      WHERE ag.rating IS NULL
--[5 Beg]
ORDER BY addid, rating
--[5 End]
--[4 End]
) As g
WHERE g.addid = addr_to_geocode.addid; --[2 End]

-- [1] create test data
-- [2] Batch geocoding
-- [3] multicolumn update
-- [4] just one rec per addid
-- [5] pick lowest rating

```

We [1] first create some dummy addresses to geocode. [2] Then we proceed to batch geocode these. We are using the [3] ANSI SQL multi-column update syntax so we can simultaneously update the rating, norm\_address, pt with a single set. This can be broken out as well. [4] We create a subselect using PostgreSQL's unique DISTINCT ON feature and only including those records that we have not already geocodes (ag.rating is NULL). Using DISTINCT ON (addid) will guarantee we only get one record back for each address. We order by addid and then rating to ensure we get the address back with the lowest rating number.

#### NORMALIZE\_ADDRESS

The normalize\_address function is probably the most reusable of the tiger geocoder functions. It uses the various \*lookup tables in tiger schema to formulate a standardized address object that has each part of the address broken out into a separate field. This standardized address is then packaged as a norm\_addy datatype object and fed to the various geocode functions. In fact the geocode function first does a normalize\_address of the address, and then feeds it to geocode\_address. Below is a demonstration of normalize\_address.

```

SELECT address As orig_addr, (normalize_address(address)).*
   FROM addr_to_geocode;

```

Note here again we have the strange looking (object).\* syntax to explode out the returned norm\_addy object type into its individual properties.

The result of the above query looks as follows.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Table 10.2 Result of normalizing our test addresses

<b>orig_addr</b>	<b>address</b>	<b>predirabbrev</b>	<b>streetname</b>	<b>streettypeabbrev</b>	<b>.....</b>
1000 Huntington Street, DC	1000		Huntington	St	...
4758 Reno Road, DC 20017	4758		Reno	Rd	..
:					

### 10.1.4 Summary

In this section we learned how to load tiger data and use the PostGIS tiger geocoder. We hope we have provided enough detail here that you can put it to work for your own needs.

Another common activity associated with addresses is figuring out the best route from address A to address B taking into consideration road networks. In the next section, we'll explore using another popular tool called PgRouting. PgRouting utilizes heuristic weights you define on road networks to determine feasible routes and best routes to take. Weights are arbitrary costs you assign each road element based on things like does the road require paying a toll, is the road congested, what is its capacity for traffic, what is its length and so forth.

## 10.2 Solving network routing problems with pgRouting

Once you have all your data in PostGIS, what better way to show it off than to find solutions to common routing problems such as the shortest path from one address to another and traveling sales man (TSP). PgRouting lets you do just that. All you have to do is add a few extra columns to your existing table to store input parameters and the solution, and then execute one of the many functions packaged with pgRouting. PgRouting makes it possible to get instant answers to seemingly intractable problems. These problems are often solved with fairly expensive desktop tools such as ArcGIS Network Analyst or with webservices. pgRouting allows you to solve these problems right in the database and to share them across many applications.

### 10.2.1 Installation

In order to get started with pgRouting, you must first install the library and then run a few SQL scripts in a PostGIS enabled database. Linux users will most likely need to compile their own. For windows and Mac users, there are binaries currently available for PostgreSQL 8.3 and 8.4. You can download the source and binaries from <http://pgrouting.postgis.org/wiki/pgRoutingDownload>. At the time of this writing, pgRouting 1.03 is the latest version and you should end up with three additional library files in the lib folder of your PostGreSQL installation: librouting, librouting\_dd, and librouting\_tsp.

Regardless of how you obtain the files and perform the installation, you must execute a series of scripts which wraps the base functions as PostgreSQL SQL functions. For convenience, we have collected them on our companion website at <http://www.postgis.us>.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

## PGRouting and PostGIS Roadmap

Given the wide popularity of PgRouting, it is on the roadmap to merge the PgRouting package into the core PostGIS package around version PostGIS 2.1. As such in later versions of PostGIS 2.1+, you can expect to have PgRouting as part of the install process of PostGIS instead of as a separate install.

### 10.2.2 Shortest route

The most common use of routing is to find the shortest route among a network of interconnected roads. Anyone who has ever sought driving directions from a GPS unit should be intimately familiar with this operation. For our example, we picked the North American cities of Minneapolis and St. Paul. This pair is perhaps the most well-known of the twin cities in the United States. We imagine ourselves as a truck driver who needs to find the shortest route through the Twin Cities. As with most cities in the world, highways usually bifurcate at the boundary of a metropolis, offering a perimeter route that encircles the city and multiple radial routes that extend into the city center to form a spokes-and-wheel pattern. The Twin Cities has one of the most convoluted patterns we could find of all the major cities in the US. A truck driver trying to pass through the city via the shortest route has quite a few choices to make; furthermore, the shortest choice is not apparent from just looking at the map. See Figure 10.1. A driver entering the metro area from the south and wishing to leave via the northwest has quite a few options. We will use pgRouting to point the driver to the shortest route.

To prepare your table for pgRouting, you need to add three additional columns – source, target and length.

```
ALTER TABLE twin_cities ADD COLUMN source integer;
ALTER TABLE twin_cities ADD COLUMN target integer;
ALTER TABLE twin_cities ADD COLUMN length double precision;
```

To populate the first two of these columns we run the assign\_vertex\_id function installed with pgRouting.

```
SELECT assign_vertex_id('twin_cities', .001, 'the_geom', 'gid');
```

This function loops through all the records, assigns the line string two integer identifiers; one for the starting point, one for the ending point. The function makes sure that identical points receive the same identifier even if shared by multiple linestrings. In routing lingo, this process builds the "network."

We next need to assign each linestring a cost. Since we are looking at distance, we will take the length of each linestring and fill in the length column.

```
UPDATE twin_cities SET length = ST_length(the_geom)
```

Though we are not doing it in our example, you can really expand the applicability of the shortest route by using different cost factors to weigh the linestrings. For example, we could weigh highways by a speed limit so that slower highways have a higher cost. We could even

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

get live feeds of traffic conditions so that routes with major traffic congestions would receive a high cost and provide real-time guidance to a driver.

With our network prepared and our cost assigned, all it takes is the execution of a pgRouting function to return the answer.

```
SELECT the_geom FROM dijkstra_sp('twin_cities',134,82);
```

Node 134 is on Interstate 35 south of the city and node 82 is Interstate 94 northwest of the city.

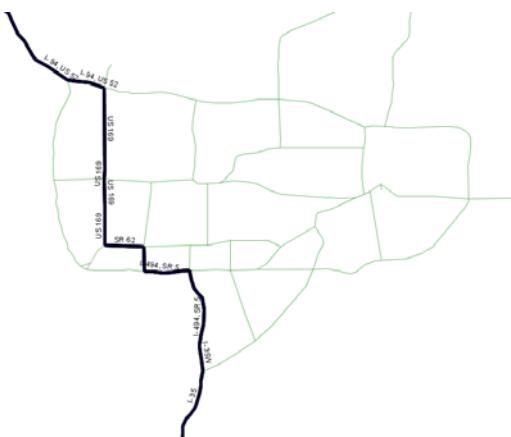


Figure 10.1 plots the shortest route through the Twin Cities.

The Dijkstra algorithm is one approach to arrive at an exact solution. For small networks, like ours, exact solutions are possible in real-time. For large networks, approximate solutions are often acceptable in the interest of computation time. pgRouting offers an A-Star algorithm to get faster but less accurate answers. To see the ever growing list of algorithms available, (or to contribute your own), visit the main pgRouting site at <http://pgrouting.postlbs.org>. For large networks, we also advise that you add spatial indexes to your table prior to executing any algorithms.

The shortest route problem is a general class of problems where one is trying to minimize the cost of achieving an objective by selecting the cheapest solution to the problem. The concept of cost is something that is user defined. Do not limit yourself to solving problems involving time and distance. For example, you can easily download a table of calories from your local McDonald, group the food items into sandwiches, drinks, and sides, and ask the question of what is the least fattening meal you can consume provided that you must order something from each group; the McRouting problem.

### 10.2.3 Traveling salesperson problem

Many times in our programming ventures, we have come across the need to find solutions to TSP related problems. Many times we have given up because nothing was readily available for us to quickly accomplish the task. Though algorithms in many languages are available, setting up a network and pairing the algorithm with whichever database we were using at the time was too much tedium. We often resorted to sub-optimal SQL based solutions. How we had hoped for something like pgRouting would come along.

The classic description of a TSP problem involves a salesman having to visit a wide array of cities selling widgets. Given that the salesman only has to visit each city once, how should he plan his itinerary to minimize total distance travelled?

To demonstrate TSP using pgRouting, we will pretend that we are a team of inspectors from IAEA (UN's nuclear energy watchdog) with the tasks of inspecting all nuclear plants in Spain. A quick search on Wikipedia shows that seven plants are currently operational on the entire Iberian Peninsula. We populate a new table as follows:

```
CREATE TABLE spain_nuclear_plants
(id serial, plant_name character varying,
lat double precision, lon double precision)
```

For TSP, we need our table to have point geometries. Each row would represent a node that the nuclear inspector must visit. Another requirement of the TSP function is that each node must be identified using an integer identifier. For this reason, we included an id column and assigned each plant a number from 1 to 7. With all the pieces in place, we execute the TSP function:

```
SELECT vertex_id
FROM tsp('SELECT id as source_id, lon AS x, lat AS y FROM
spain_nuclear_plants','1,2,3,4,5,6,7',3);
```

This tsp function is a little unusual in that the first parameter is actually an SQL string. This string must return a set of records with the columns source\_id, x, and y in order for tsp to continue. The second parameter lists the nodes to be visited and the final parameter is the starting node. The tsp function returns the nodes in order of the sequence of travel.



**Figure 10.2.** Optimized shortest distance travel path for visiting all nuclear power plants in Spain starting  
©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

from Almaraz

Like all algorithms in pgRouting, TSP only works on a Cartesian plane. In fact, SRID does not even come into play for TSP since TSP does not even require a geometry column. In the interest of computational speed, routing problems rarely demand exact answers. Just think of the number of times your GPS guidance unit took you down an awkward path. Because of this tolerance for errors, the inexactitudes generated by not accounting for earth curvature can usually be ignored, even for large areas. Do not apply the algorithms where your distances cover more than a hemisphere, otherwise, you would be finding yourself trying to sail to China by crossing the Atlantic and rediscovering the new world but without any fanfare.

#### **10.2.4 Summary**

What we wanted to show in this section is the convenience brought forth by the marriage of a problem solving algorithm with a database. Imagine that you had to solve the shortest route or TSP problem on some set of data using just a conventional programming language. Without PostGIS or pgRouting, you would have to define your own data structure, code the algorithm, and find a nice way to present the solution. Should the nature of your data change, you would have to repeat the process over again. In the next sections, we shall explore PL languages. PL languages and SQL is another kind of marriage that combines expressiveness of an all-purpose or domain specific language well suited for expressing certain classes of problems with the power of SQL to create a system that is greater than the sum of its parts.

### **10.3 Extending PostgreSQL power with PLs**

One thing that makes PostgreSQL unique among the various relational databases is its pluggable procedural language architecture. Several people have created procedural handlers for PostgreSQL that allows writing stored functions in a language more suited for a particular task. This allows you to write database stored functions in languages like Perl, Python, Java, TCL, R, and Sh (shell script) in addition to the built-in C, plpgsql, and SQL. Stored functions are directly callable from SQL statements. This means you can do certain tasks much easier than you would if you had to pull the data out directly in these language environments and push them back into the database. You can write aggregate functions, triggers, and leverage a breath of outside functions developed for these languages right in your database. When you do so, these are often referred to as PL -- PL/Perl, PL/Python, PL/Proxy, PL/Sh, PL/Java etc. The code you write is pretty much the same as you would write in the language except with additional hooks into the PostgreSQL database.

#### **10.3.1 Basic installation of PLs**

In order to use these non-built-in PL languages in your database, you need three basic things:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- The language environment installed on your PostgreSQL server
- The PL handler .dll/.so installed in your PostgreSQL instance
- The language handler installed in the databases you will use them in - usually by running a CREATE LANGUAGE statement

The functionality of the PL is usually packaged as a .so/.dll file starting with pl\*. It negotiates the interaction between PostgreSQL and the language environment by converting PostgreSQL datasets and data types into the most appropriate data structure for that language environment. It also handles the conversion back to a PostgreSQL data type when the function returns with a data set or scalar value.

### **10.3.2 What can you do with a non-native PL**

Each of the PL languages has various degrees of integration with the PostgreSQL environment. PL/Perl is perhaps the oldest and probably the most common and most tested you will find. PLs are registered in 2 flavors - trusted and untrusted. PL/Perl can be registered as both a trusted and an untrusted. Most of the other PLs just offer the untrusted variant.

#### **What is the difference between trusted and untrusted?**

A trusted PL is a sandboxed PL, meaning provisions have been made to prevent it from doing dangerous things or accessing other parts of the OS outside the database cluster. A trusted language function can be run under the context of a non-superuser, but certain features of a language are barred so behaves less like the regular language environment than an untrusted language function.

An untrusted language is one that can potentially wreak havoc on the server, so great attention must be taken. It can delete files, execute processes and other things that the Postgres daemon/service account has the power to do. Untrusted language functions must run under the context of a super user -- which means you need to create them as a super user and mark them as SECURITY DEFINER if you want non-superusers to use them. It also means you must take extra care to validate input to prevent malicious use.

In the sections that follow, we'll be demonstrating PL/Python and PL/R. We have chosen these particular languages because they have the largest offerings of spatial packages of all the languages that PostgreSQL offers. We also think they are pretty cool languages. They are favorites among Geostatisticians and GIS programmers. Both languages have only an untrusted flavor.

Python is a dynamically typed all purpose procedural language. It has fairly elegant approaches for creating and navigating objects, supports functional programming, object oriented programming, building of classes, meta programming, reflection, map reduce and all those modern programming paradigms you've heard of. R, on the other hand, is more of a domain language. R is a language specifically designed for statistics, graphing, and data

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

mining. It has a fairly large cult following among research institutions. It has many built-in statistical functions or functions you can download with a type of a line. Most of the functionality it offers you will not find anywhere else except possibly in pricey tools such as SAS and MatLab. You will find tasks such as applying functions to all items in a list, matrix algebra, and dealing with sparse matrices, fairly short and sweet to do in R once you get into the R mindset. In addition to manipulating data, R has a fairly extensive graphical engine that allows you to generate elegant looking graphs with a few lines of code. You can even do 3d plots.

You can write functions in PL/Python and PL/R that pull data from the PostgreSQL environment and can return simple scalars or more complex sets. You can even return binary objects such as image files. In addition you can write database triggers in PL/Python and PL/R that leverage the power of these environments to run tasks in response to changes of data in the database. For example you can geocode data when an address changes, or have a database trigger to regenerate a map tile on change of data in the database without ever touching the application edit code. This is a feature that is next to impossible to do with just the languages and a database connection driver. In addition, you can write aggregate functions with these languages that will allow you to feed the sets of rows to aggregate and use functions only available in these languages to summarize the data. Imagine an aggregation function that returns a graph for each grouping of data. Some examples of this you can find listed in Appendix A.

In the sections that follow, we'll be writing some stored functions in these languages. These examples will have a slight GIS bent. Our intent in these sections is to show you how to get started integrating these in your PostgreSQL database and give you a general feel of what is possible with these languages. Only your imagination limits the possibilities you can achieve with this kind of intimate integration. We'll also show you how you can find and install more libraries that you can then leverage from within a stored function.

## **10.4 Graphing and Accessing Spatial Analysis Libraries with PL/R**

PL/R is a stored procedural language supported by PostgreSQL that allows you to write PostgreSQL stored functions using the R-statistical language and graphical environment. You can call out to the R environment to do neat things like generate graphs, leverage a large body of statistical packages that R provides and do mystical things with matrices. R is a favorite among statisticians and researchers because it makes flipping matrices, aggregation, function apply across rows and columns and other data structures almost trivial. It also has a breath of options for importing data from various formats and has many packages available contributed by geospatial analysts. We'll just touch the surface of what PL/R and R provide. In order to get deeper into the R trenches, we suggest reading the Manning book "R in Action" or the "Applied Spatial Data Analysis with R" book. Check out appendix A for other useful R sites and examples.

For the exercises that follow, we will be using R 2.10. Most of these should work on lower versions of R.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### 10.4.1 Getting started with PL/R

In order to write PostgreSQL procedural functions in R, you must do the following

- Install the R environment on the box your PostgreSQL service/daemon runs on. R is available for Unix, Linux, Mac OSX as well as Windows. Unix/Linux users may need to compile plr. For Windows and Mac OSX users, there are already pre-compiled binaries for R. Any R from version R 2.5 thru R 2.10 should work fine. We have tested it against R 2.5, R 2.6 and R 2.10. You can download source and precompiled binaries of R from <http://www.r-project.org/>. Note after install, check your environment variables to make sure R\_HOME is specified and specified correctly. This is what PL/R uses to determine where R is installed and should be called from.
- Compile/Install the plr.so/dll files by copying this file into the lib directory of your PostgreSQL install. If you are using an installer, this is probably already done for you. NOTE: you must use the version compiled for your version of PostgreSQL. You can download the binaries and source from <http://www.joeconway.com/plr/>. Most likely you will need to restart the PostgreSQL service before you can install plr in a database.
- Run the packaged plr.sql file in the PostgreSQL database you will be writing R stored functions in. This step you need to repeat for each database you want to R-enable.

If any of the above is confusing or you get stuck, check out PL/R wiki installation tips guides <http://www.joeconway.com/web/guest/pl/r/-/wiki/Main/Installation+Tips>

#### R\_HOME AND PATH environment variables

PL/R relies on an environment variable called R\_HOME to denote the location of R install. It also assumes that the R libraries are in the path setting of the server install. The R\_HOME variable must be accessible by the postgres daemon service account and R binaries in default search path. These steps are already done for you if you are using an installer. For Linux/Unix you can set this with an export R\_HOME = ... and include as part of your postgresql init script. You may need to restart your postgres services for the new settings to take effect.

After you have installed PL/R in a database, run the following command to verify your R\_HOME is set right: `SELECT * FROM plr_environ();`

### 10.4.2 Saving Datasets and Plotting

Now we will take PL/R for a test drive. For these exercises, we will be using R-2.10.1, but any of these should work in lower versions of R.

#### SAVING POSTGRESQL DATA TO RDATA FORMAT

For our first example, we'll be pulling data out of PostgreSQL and saving it to R's custom binary format (RData). There are two common reasons we do this:

- It makes it easy to interactively test different plotting styles and other R functions in
- ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

R's interactive environment against real data before you package them in a PL/R function.

- If you are teaching a course and are using R as a tool for analysis, you may want to provide your datasets in a quick format that can be easily loaded in R by students. As such, you may want to create PL/R functions that dump data out in self-contained problem set nuggets.

For this example we will be using PostgreSQL pg.spi.exec function and R's save function. The pg.spi.exec function is a PL function that allows you to convert any PostgreSQL dataset into a form that can be consumed by the language environment. In the case of PL/R this is usually an R data.frame.

The save command in R allows you to save many objects to a single binary file. These objects can be data frames (including spatial data frames), lists, matrices, vectors, scalars and all of the various object types supported by R. When you want to load these in an R session, then you run the command: `load("filepath")`

#### **Listing 10.4 Saving PostgreSQL data in R data format with PL/R**

```
CREATE OR REPLACE FUNCTION ch10.save_dc_rdata() RETURNS text AS
$$ #[1 Beg]

dccounties <- pg.spi.exec("SELECT cntyidfp, name, intptlat, intptlon
                           FROM county WHERE statefp = '11'")

dczips <- pg.spi.exec("SELECT z.zcta5ce, z.intptlat, z.intptlon
                       FROM zcta500 AS z
                       INNER JOIN state AS s
                       ON ST_Intersects(z.the_geom, s.the_geom)
                       WHERE statefp = '11'")

#[1 End]
#[2 Beg]
  save(dccounties,dczips, file="C:/Temp/dc.RData")
#[2 End]
  return("done")
$$
language 'plr';
#[1] store results in R variables
#[2] save variables to R data file
```

In the above example, we [1] create two datasets that contain Washington DC counties and zips. We then [2] save this to a file called dc.RData. RData is the standard suffix for the binary R Data format and in most desktop installs when you launch it, it will open R with the data loaded.

To run the above example so it saves, we run `SELECT ch10.save_dc_rdata();`

We can load this data in R, by clicking on the file or opening R and running a `load` call in R. Below are a couple of quick commands to try in R environment. Launch R

To load a file in R:

```
load("C:/Temp/dc.RData")
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

To list contents of file in R:  
`ls()`

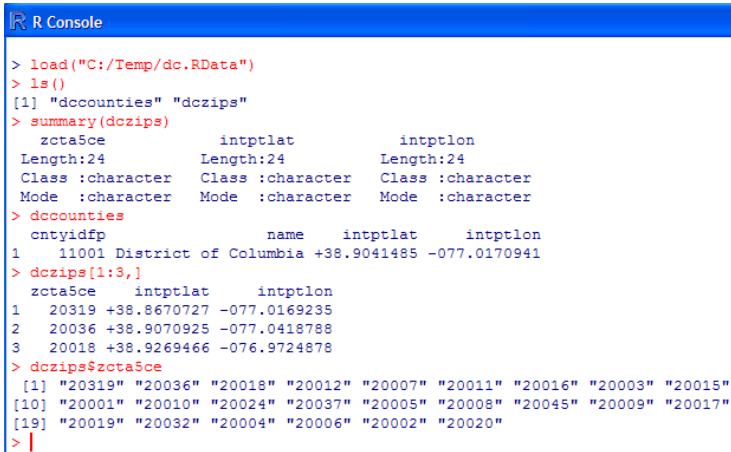
To view a data structure in R:  
`summary(dczip)`

To view data in R type the name of the data variable:  
`dcaccounts`

To view a set of rows in a variable in R:  
`dczips[1:3, ]`

To view columns of data in R variable  
`dczips$zcta5ce`

The above R commands demonstrates some commonly used constructs in R. Though it is not demonstrated, if you use the variable `<-` syntax, data gets pushed to an R variable instead of to screen. Below is a snapshot of our described commands:



```
R Console
> load("C:/Temp/dc.RData")
> ls()
[1] "dcaccounts" "dczips"
> summary(dczip)
  zcta5ce      intptlat      intptlon
Length:24      Length:24      Length:24
Class :character Class :character Class :character
Mode  :character Mode  :character Mode  :character
> dcaccounts
  cntyidfp      name      intptlat      intptlon
1 11001 District of Columbia +38.9041485 -077.0170941
> dczips[1:3,]
  zcta5ce      intptlat      intptlon
1 20319 +38.8670727 -077.0169235
2 20036 +38.9070925 -077.0418788
3 20018 +38.9269466 -076.9724878
> dczips$zcta5ce
 [1] "20319" "20036" "20018" "20012" "20007" "20011" "20016" "20003" "20015"
[10] "20001" "20010" "20024" "20037" "20005" "20008" "20045" "20009" "20017"
[19] "20019" "20032" "20004" "20006" "20002" "20020"
> |
```

Figure 10.3 Demonstration output of running the above statements in R

One task that R excels in is drawing plots. In fact many people even if they don't care about statistics are attracted to R because of its sophisticated scriptable plotting and graphing environment. In the next exercise, we demonstrate a bit of this by generating a random data set in PostgreSQL and plotting it using PL/R.

### **Listing 10.5 Plotting PostgreSQL data with R**

```
CREATE OR REPLACE FUNCTION ch10.graph_income_house() RETURNS text AS
$$
#1
randdata <- pg.spi.exec("SELECT x As income ,
Avg(x*(1 + random()*y) ) As avgprice
FROM generate_series(2000,100000, 10000) As x
CROSS JOIN generate_series(1,5) As y
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

        GROUP BY x ORDER BY x")
# 2
png('C:/temp/housepercap.png', width=500, height=400)
opar <- par(bg = "white") #set background color
# 3
plot(x=randdata$income,y=randdata$avgprice, ann = FALSE, type = "n")
yrange = range(randdata$avgprice)
#create 10 horizontal grid lines
# 4
abline(h=seq(yrange[1],yrange[2],
(yrange[2] - yrange[1])/10), lty=1, col="grey")
# 5
lines(x=randdata$income,y=randdata$avgprice, col = "green4",
lty = "dotted")
# 6
points(x=randdata$income,y=randdata$avgprice, bg = "limegreen", pch = 23)
title(main = "Random plot of house price vs. per capita income",
xlab = "Per cap income", ylab = "Average House Price",
col.main = "blue", col.lab = "red1",
font.main = 4, font.lab = 3)

# 7
dev.off()
print("done")

$$
LANGUAGE 'plr';
# 1 save result in R
# 2 plot redirect to new png
# 3 draw plot
# 4 grid lines
# 5 dotted line plot
# 6 plot points
# 7 close file

```

In this code snippet we are creating a stored function written in PL/R that when run will create a file called housepercap.png on the c:/temp folder of our PostgreSQL server. (1) We first create random data by running an sql statement using PostgreSQL generate\_series function and dump this in randdata R variable. (2) We then create a png file (note other functions such as pdf, jpeg etc. can be used to create other formats) which all the plotting will be redirected to. (3) We then draw our plot. The n type means no plot -- so just prepares the plot space so that we can then draw (4) grid lines (5) lines and (6) points on the same grid. (7) We close writing to the file with dev.off() and then return text saying "done".

### **UNABLE TO START DEVICE DEVWINDOWS**

It is a common occurrence to get a can't start device error even though the same command runs perfectly fine in the R gui environment. This is because plr runs under the context of the postgres service account. Any folder you wish to write to from plr must have read/write enabled for the postgres service/daemon account.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

You run the above function with an SQL statement  
`SELECT ch10.graph_income_house();`  
 and when it is done, it will return "done". Running this command generates a png file that looks like the below figure.



Figure 10.4 Result of `SELECT ch10.graph_income_house()`

### 10.4.3 Using R packages in PL/R

The R environment has a rich wealth of functions, data, and data types you can download and install. All the functions and the definitions of the data structures they use are distributed in packages which are often referred to as libraries. When a package is installed, it becomes a library folder you can see in the library folder of your R install. R makes finding, downloading and installing additional libraries fairly easy using the comprehensive R Archive Network (CRAN). Once a package is installed in R, you can then use it in PL/R functions.

Table 10.3 Commands for installing and navigating packages

Command	Description
<code>library()</code>	Gives a list of packages already installed
<code>library(packagename)</code>	loads a package into memory
<code>update.packages()</code>	upgrades all packages to latest version
<code>install.packages("packagename")</code>	installs a new package
<code>available.packages()</code>	lists packages available in default CRAN
<code>chooseCRANmirror()</code>	allows you to switch to a different CRAN

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

demo()	shows list of demos in loaded packages
demo(package = .packages(all.available = TRUE))	lists all demos in installed packages
demo(nameofdemo)	launches a demo (note must load lib first)
help(package=somepackagename)	gives summary help about a package
help(package=packagename, functionname)	gives detailed help about an item in package
vignette()	Lists tutorials in packages
vignette("nameofvignette")	This launches pdf of exercises

What is particularly nice about the R system is that many packages come with demos that show off the features of the package. Commands to view these demos are shown above. They also often come with things called vignettes. Vignettes are quick tutorials on using a package. Demos and vignettes make R a fun interactive environment to learn in. In order to use a vignette or demo, you first have to use the library command to load the library.

In order to test out the CRAN install process, we will be installing a package called rgdal. rgdal is a packaging of the Geospatial Data Abstraction Layer (GDAL) for R. We saw GDAL in our chapter 7 on data loading. rgdal relies on another package called sp. The R install process automatically downloads and installs dependent packages as well, so no need to install sp if you install rgdal. In addition to supporting various forms of vector data using OGR commands, GDAL also supports raster data.

To install packages in R, we use the R command-line or R-GUI.

- At a command line type R (or R-GUI)
- Once in R environment. Type the following.

To Launch R:

R

To load the rgdal library:

```
library(rgdal)
```

If prior command failed -- use this next commands to install rgdal and then load:

```
install.packages("rgdal")
library(rgdal)
```

To get help about rgdal library:

```
help(package=rgdal)
```

To quit out of R console:

```
q()
```

## Complex R packages

We were installing rgdal on windows. For this particular install and some more complex R packages such as RGTk2, you may need to exit R environment first and then come back into it to be able to use the libraries. To use these libraries from PL/R, you need to restart the postgres service after the library install. In general, these steps are not necessary for most R packages.

For other OSes besides windows, there might not be a pre-compiled binary available for rgdal, in which case you need to compile from scratch. Details can be found at: <http://cran.r-project.org/web/packages/rgdal/index.html>

Next we will test drive our RGDAL installation with a couple of exercises.

### **10.4.4 Quick primer on RGDAL**

In order to test out our rgdal install, we will run the following commands in R and then wrap some of this functionality in a PL/R function.

```
Load rgdal library:  
library(rgdal)  
Get list of raster drivers:  
gdalDrivers()  
Get list of vector geometry drivers:  
ogrDrivers()  
Return structure details about gdalDrivers list  
str(gdalDrivers())
```

str() is an R base function that provides structure details about an R object. In the case of data.frames which are similar in concept to relational tables, it outputs the fieldnames and lengths in addition to some other summary details.

Now we will make the gdalDrivers list queryable from within PostgreSQL by creating a short PL/R function as shown below.

#### **Listing 10.6 Function to get list of supported RGDAL raster formats**

```
CREATE OR REPLACE FUNCTION r_getgdaldrivers() RETURNS SETOF text AS  
$$  
library(rgdal)  
#1  
return(gdalDrivers()['long_name'])  
$$  
language 'plr';  
  
-- 2  
SELECT driver  
FROM r_getgdaldrivers() As driver  
WHERE driver ILIKE '%arc%'  
ORDER BY lower(driver);  
-- 1 load gdal lib  
-- 2 list of drivers
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

We have created a function called `r_getgdaldrivers()` which returns (1) just the long name field of the data.frame returned by `gdalDrivers` function. Since we are returning this as a set of text, we can use the function as we would any other one column table. In (2) we do just that. The output of (2) is shown below:

**Table 10.4 Result of `r_getgdaldrivers()` function**

#### **Driver**

---

ARC Digitized Raster Graphics  
 Arc/Info ASCII Grid  
 Arc/Info Binary Grid  
 EUMETSAT Archive native (.nat)

In the above example we output a whole column of the data.frame. For large data.frames or other kinds of R data objects, you may want to output just a subset of rows. For the next example, we'll create a function that allows us to query if a format is updatable or copyable. In this next example we will demonstrate the use of the built in R function `subset`, which behaves much like a SELECT SQL statement. We will also demonstrate passing arguments to PL/R functions.

#### **Listing 10.7 Demonstrating `subset` and `c` functions in R**

```

CREATE OR REPLACE FUNCTION r_getgdaldrivers(
    param_create boolean, param_copy boolean
) RETURNS SETOF text AS
$$
library(rgdal)
# gdalDrivers() returns only those that match our create,
# copy desired setting and only return long_name field
-- 1
return (subset(
            gdalDrivers(),
            create==param_create & copy==param_copy,
select=c(long_name)))
$$
language 'plr';

-- 2
SELECT driver
    FROM r_getgdaldrivers(true,true) As driver
    ORDER BY lower(driver);
-- 1 R select where
-- 2 use r function in regular SQL

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

This example demonstrates both the (1) subset function and c() in R. We are creating an overloaded function that will allow us to select just those drivers that fit our create and copy desired behavior. The subset function is similar in concept to an SQL where. The first argument is the data set we would like to select from (parallel a table in SQL), the next defines the WHERE condition with & used for AND, and the select designates the columns of data to select. The c() function in R returns a vector. In this case we want a vector composed of just one column (the long\_name). Also note that the arguments we declared to the function are used by name without any massaging. PL/R does the automatic conversion from a PostgreSQL boolean to an R boolean for us. (2) In the second code snippet, we test our new overloaded function version of r\_getgdaldrivers to list just those drivers that support both copy and create.

In the next example, we will start to use the drivers. We will create a PL/R function which will read the meta data of a RASTER file and return a summary. One of the nice things about GDAL is that in most cases it can determine which driver to use by the extension of the file:

### **Listing 10.8 Reading meta data from Image files with rgdal**

```
-- 1
CREATE TYPE gdalinfo_values AS(key text, value text);

CREATE OR REPLACE FUNCTION r_getimageinfo(param_file text)
RETURNS SETOF gdalinfo_values AS
$$
-- 2
library(rgdal)
tile_summary <- GDALinfo(param_file)
result_labels <- labels(tile_summary)
result_values <- tile_summary[result_labels]
-- 3
df <- data.frame(key = result_labels, value = result_values);
return(df)
$$
language 'plr';
-- 1 custom datatype
-- 2 load lib and store variables
-- 3 coerce into data frame
```

In this example we create a plr function that will return a set of key value pairs which represent our meta data. (1) We first create a data type to hold our results. Ideally we would have preferred using OUT parameters for something like this, but pl/r and OUT parameters don't work well together when dealing with sets. (2) To use rgdal we first load the library. The GDAL\_info call reads all the meta data from our image and loads it into a GDAL object which is an atomic vector. We then run the labels to grab all the labels from the vector and then use this to index and pickout the elements. (3) To return the data as a set of properties we can read like a table, we coerce our data into a data.frame making the labels

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

the key column and elements the values column. Note the naming of the columns of our data frame mirrors our gdalinfo\_values data type.

To run the function we do:

```
SELECT *
  FROM r_getimageinfo('C:/Winter.jpg') As vi;
```

And the result is :

**Table 10.4 Result of gdal**

Key	Value
rows	600
columns	800
bands	3
:	

#### 10.4.5 Getting PostGIS geometries into R Spatial Objects

The sp package contains various classes that represent geometries as R objects. It has lines, polygons, points and spatial polygon, line and point data frames. Spatial data frames are similar in concept to PostgreSQL tables with geometry fields. Pushing PostGIS data into these spatial data frames and spatial objects is not unfortunately not trivial as of yet.

There are two approaches you can use to push geometry data into these SpatialPolygons, SpatialLines and so forth constructs.

- Use a version of rgdal compiled with support for the PostGIS ogr driver and then use the readOGR method of rgdal as documented here <http://wiki.intamap.org/index.php/PostGIS>

There are two obvious problems with this approach. Most pre-compiled versions of rgdal are not compiled with the PostGIS driver. The other problem is that it requires you to pass in the connection string to the database. Having to specify the connection to the database that a PL/R function is running in somewhat defeats the purpose of using PL/R. If you were to use it to build an aggregate function, it would be next to impossible to manage.

- The second approach, is to reconstitute a R spatial object from the points of a PostGIS geometry. There are a couple of ways of doing this -- such as parsing WKT/WKB or just exploding the geometries in points. We have chosen the explode approach using the new function ST\_DumpPoints introduced in PostGIS 1.5 and will demonstrate this. These exercises will only work in PostGIS 1.5+, but if you need to use on a lower PostGIS, you can implement your own ST\_DumpPoints or copy one available In PostGIS 1.5.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

For our first example, we will convert our twin cities pgrouting results into R spatial objects so that we can plot them in R.

### **Listing 10.9 Plotting linestrings with R**

```

CREATE OR REPLACE FUNCTION ch10.plot_routing_results()
RETURNS text AS
$$
library(sp)
#explode the points
#[1 Beg]
geodata <- pg.spi.exec("SELECT gid, ST_X(geom) As x, ST_Y(geom) As y,
path[1] As ptn
FROM
(SELECT gid, route, (ST_DumpPoints(the_geom)).*
FROM ch10.twin_cities
) As c ORDER BY gid, path[1]")
#[1 End]

#[2 Beg]

georesult <- pg.spi.exec("SELECT ST_X(geom) As x, ST_Y(geom)As y
FROM
(SELECT (ST_DumpPoints(
ST_LineMerge(ST_Collect(the_geom)) )).*"
FROM ch10.dijsktra_result ) As r
ORDER BY path[1]")
#[2 End]

geo факт <- factor(geodata$gid)
geo.id <- levels(geo факт) #get unique list of linestring ids
ngeom <- length(geo.id)

#regroup points by line they belong to
#[3 Beg]
geo.xy <- split(geodata[2:3], geo факт)
geo.geoms <- list()
for(k in 1:ngeom ){
  geo.geoms[k] = Lines(list(Line(geo.xy[k])), ID = geo.id[k])
}
#[3 End]

# add the result
geo.result <- SpatialLines( list (
  Lines( list(Line(cbind(georesult$x,georesult$y))) ,
  ID='result' ) )
geo.sp <- SpatialLines(geo.geoms)

sdf <- SpatialLinesDataFrame(geo.sp,
  data = data.frame(geo.id, row.names="geo.id" ),
  match.ID = TRUE)
sdf_result <- SpatialLinesDataFrame(geo.result, data =
data.frame(c("result") ), match.ID=FALSE )

png('C:/temp/twin_bestpath.png', width=500, height=400)

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

#show axes, limit axes with xlim, ylim range
plot(sdf,xlim=c(-94, -93),ylim=c(44.5,45.5),axes=TRUE);

lines(sdf_result, col = "green4", lty = "dashed", type="o")

title(main= "Travel options to Twin Cities", font.main=4,
col.main="red", xlab="Longitude", ylab="Latitude")
dev.off()
return("done")
$$
LANGUAGE 'plr' VOLATILE
#[1] sql query dump points
#[2] djikstra
#[3] regroup points into splines
#[4] lines to SpatialLineDataFrame

```

For this example we [1] run a PostgreSQL query that explodes our street segments into points. We run another query [2] which gives us the points for our pgRouting djikstra result. You use # to denote comments in R similar to Python. [3] regroup points into sp lines and then [4] lines to LinesDataFrame.

We then load the R data.frame resulted in SpatialLinesDataFrames that we can then use to plot on the same axis. Result of running the query

```
SELECT ch10.plot_routing_results();
```

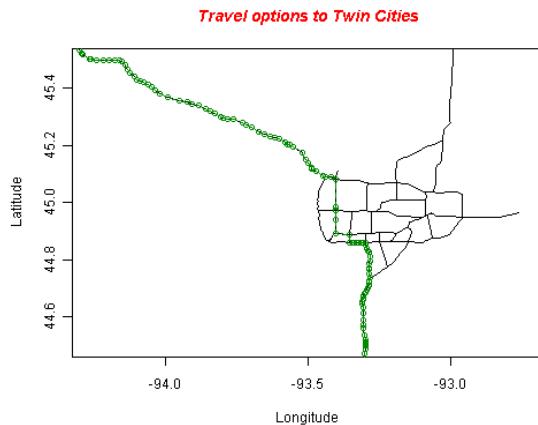


Figure 10.5 pgRouting Twin cities results plotted with PL/R

## SAVING TO OTHER VECTOR TYPES

Although we didn't demonstrate it, once you have data loaded in a Spatial Data frame, you can then use the saveOGR functions of rgdal to save it as various supported vector formats returned by the ogrDrivers() list.

The sp package has its own plot function as well called spplot. Spplot does additional things the regular R plot doesn't. spplot is specifically targeted for spatial data. We encourage you to explore it. You can see some of the fancy plots you can create with spatial data by running the following commands from R gui console

```
library(sp)
demo(gallery)
```

#### **10.4.6 Outputting plots as binaries**

In all the examples we have demonstrated with plotting, we have saved the plots to a folder on the server. This approach is fine if you are using PL/R to generate canned reports for later distribution. If however you need to output to a client such as a web browser, you need to be able to output the file directly from the query. There are a couple of approaches of doing this we have come across:

- Using RGTK2 and Cairo device. This approach is documented on the PL/R wiki and requires installing both RGTK2 and Cairo libraries. In this approach you output graph as a bytea. The problem we have found with this approach is that those two libraries are fairly hefty and call out to GTK - yet another dependency. The other issue is as of this writing, it seems to crash in PL/R during library load process when PL/R PostgreSQL is running under windows. The benefit of this approach however is that it produces nicer looking graphics and also prevents temp file clutter - since it doesn't ever need to save to disk. It is also a one step process.
- The other approach is to save the file to disk and have PostgreSQL read the file from disk. There is a function in PostgreSQL called pg\_read\_file(..) which is a super user function. It however is limited to only reading files from the PostgreSQL data cluster. One simple way to do this is to create a dummy tablespace - we'll call it r\_files and save all r generate files to there and then use the pg\_read\_file to return these files.
- Another approach is to use a PL language with more generic access to the file system such as PL/Python or PL/Perl. Doing so requires you wrap your PL/R function in another PL function.

In the next section, we'll show off another PL called PL/Python. Python is another GIS analysts/programmer language favorite. In fact these days most popular GIS toolkits have Python bindings. You'll see its use in Open source GIS desktop and web suites such as Quantum, OpenJump, GeoDjango and even in commercial GIS such as Safe FME and ArcGIS.

## 10.5 PL/Python

PL/Python is the Procedural Language handler in PostgreSQL that allows you to call Python libraries and embed Python classes and functions right in a PostgreSQL database. As a plpython stored function, you can call it in any SQL statement. You can even create aggregate functions and database triggers with Python. In this section we'll show off some of the beauty of PL/Python. For more details on using Python and PL/Python, refer to Appendix A.

### 10.5.1 Installing PL/Python

For the most part, you can use any feature of Python from within PL/Python. This is because the PostgreSQL PL/Python handler is a thin wrapper that negotiates the messaging between PostgreSQL and the native Python environment. This means any Python package you install can be accessed from your PL/Python stored functions. Unfortunately not all database to PL/Python object mappings are supported. This means you can't for example return some weird complex Python object back to PostgreSQL unless it can be easily coerced into a custom PostgreSQL data type.

#### PL/PYTHON CAVEATS

PL/Python as of PostgreSQL 8.4 does not support arrays and SETS as input arguments.

PL/Python doesn't currently support the generic RECORD type as output either. This means for composite row types, you need to declare a record type before hand. Some of this will change in 9.0.

In order to use PL/Python, you must have Python installed on your PostgreSQL server machine. Since PL/Python runs within the server, any client connecting to it such as a web app or a client pc, need not have Python installed to be able to use PostgreSQL stored functions written in PL/Python. The pre-compiled PostgreSQL 8.3 plpython libraries packaged with most distros Windows/Mac/Linux are compiled against Python 2.4, 2.5 or 2.6. These only work with the Python minor version they were compiled against.

#### WINDOWS ONE-CLICK INSTALLER

For windows users the one-click installers 8.3 version of PostgreSQL is compiled against Python 2.5 and for PostgreSQL 8.4 its against Python 2.6. The PLpython.dll is already packaged with the one-click installer. In order to use it, you just need to spend the 5 minutes to install the respective python version on your server and run enable the language in your database.

If you are using the PostgreSQL Yum repository for PostgreSQL installation, you can get PLPython by doing a:

```
yum install postgresql-plpython
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

On most Linux/Unix machines, you can determine which version of Python plpython is compiled against by doing a:

```
cd /
locate plpython.so
ldd path/to/plpython.so
```

### **POSTGRESQL 9.0 SUPPORT FOR PYTHON 3.0**

PostgreSQL 9.0 will have an additional offering of allowing PL/Python against Python 3.0. This most likely will not be a standard package and will require custom compilation to use. Whether it is part of the standard package or separate is still in debate.

Once you have Python and the PL/Python.so/.dll installed on your server, simply run the below statement to enable the language in your database.

```
CREATE PROCEDURAL LANGUAGE
'plpythonu' HANDLER plpython_call_handler;
```

If you run into problems enabling PLPython, please refer to our PLPython help guide links in Appendix A. The common issue most people face is that they don't have the required version of python installed on the server or they don't have the plpython.so/.dll file present.

#### **10.5.2 First PL/Python Function**

In order to test our PL/Python install, we'll write a simple function that uses nothing but built-in Python constructs.

##### **Listing 10.10 Compute sum of range of numbers**

```
--1
CREATE OR REPLACE FUNCTION python_addreduce(param_start integer,
                                             param_end integer)
RETURNS integer AS
$$
--2
# define add function
def add(x,y): return x+y

# return sum from range param_start to param_end
--3
return reduce(add, range(param_start, param_end + 1));
$$
LANGUAGE 'plpythonu' IMMUTABLE;
--1 take start and end range
--2 inner function to add
--3 apply inner to range
```

In this example, we define a plpython function that (1) takes a start and end number to define a range. (2) The # construct is a python comment. We first define a simple function

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

that adds two numbers, and then (3) we use the built-in python reduce construct and range construct to construct an array of numbers between the start and end values and then apply our add function to the resulting sequence.

To test the above example, we can do this: `SELECT python_addreduce(1, 4);`  
Which gives us an answer of: 10

### **10.5.3 Using Python Packages**

The python install comes with the very basics. Much of what makes python useful, is the large breath of free packages to do things like matrix manipulation, integrating with webservices and importing of data. A good place to find these extra packages is at the Python Cheeseshop package repository.

In order to use these packages, we'll be using Python Cheeseshop package repository and the setup tool called Easy Install to install them.

Easy install is a tool for installing Python packages. You can download the version for your OS and version of Python from: <http://pypi.python.org/pypi/setuptools#downloads> or use your Linux update tool to install it.

#### **EASY INSTALL ON WINDOWS**

Once installed, the `easy_install.exe` is located in the `C:\Python26\scripts` folder for Windows users.

#### **IMPORTING AN EXCEL FILE WITH PL/PYTHON**

For this example, we'll be using the `xlrd` package which you can grab from the Python Cheeseshop - <http://pypi.python.org/pypi/xlrd>

This package will allow you to read excel files using any OS. This doesn't have any additional dependencies, uses the standard `setup.py` install process, and for windows users, it has a `setup.exe` as well. For this exercise, we will install it with the command which should work for most any OS if you have installed `easy_install`: `easy_install xlrd`

We'll test our install by importing a `test.xls` file that has a header row and three columns of data. Unfortunately, PL/Python doesn't support returning `SETOF` rows like `plpgsql`, so we'll need to create a type to store our data.

First we create a PostgreSQL data type to store our returned result so we can get back more than one value from the function.

```
CREATE TYPE ch10.place_lon_lat AS (
    place text, lon float, lat float);
```

Then we create a table to store our returned results:

```
CREATE TABLE imported_places(place_id serial PRIMARY KEY,
    place text, geom geometry);
```

#### **Listing 10.11 PL/Python function to import Excel data**

```
CREATE OR REPLACE FUNCTION ch10.fngetxlspts(param_filename text)
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

RETURNS SETOF ch10.place_lon_lat AS
$$
--1
    import xlrd
    book = xlrd.open_workbook(param_filename)
    sh = book.sheet_by_index(0)
    # we will assume that the first row contains
    # the header columns so skip it
    for rx in range(1,sh.nrows):
--2
        yield(sh.cell_value(rowx=rx, colx=0),
              sh.cell_value(rowx=rx, colx=1),
              sh.cell_value(rowx=rx, colx=2))
    )
$$
LANGUAGE 'plpythonu' VOLATILE;
--1 import package
--2 append to final result

```

(1) The first thing we do is import the xlrd package so we can use it. We will assume there is only data in the first spreadsheet. The next step (2) is to loop thru the rows of the spreadsheet skipping the first row and using the python yield function to append to our result set. In the final yield, the function will return with all the data. Now we can use this data by inserting into a table and making point geometries.

```

INSERT INTO imported_places(place, geom)
SELECT f.place, ST_SetSRID(ST_Point(f.lon,f.lat),4326)
FROM ch10.fngetxlspts('C:/temp/Test.xls') AS f;

```

We are doing an insert using the excel file as the from source. Since the server is running the python code and running under the context of the postgres daemon account, the path we give to the function has to be a path accessible by the postgres daemon account.

#### **IMPORTING SEVERAL EXCEL FILES WITH SQL**

Now lets imagine you have several Excel files you get from some vendor that are all structured the same. You have them all in one folder and you want to import them all at once. Sometimes they even dear to give you duplicated rows across files. Here is where the real beauty of a PL married with SQL shows its colors.

First we'll create a python function that lists all the files in a directory, we'll write another query to treat this list like a table to filter and then we'll write one SQL function to insert all the data using this list.

We create a function that lists the files in a passed in directory path and returns a rows of text.

```

CREATE FUNCTION ch10.list_files(param_filepath text)
RETURNS SETOF text AS
$$
import os
return os.listdir(param_filepath)
$$
LANGUAGE 'plpythonu' VOLATILE;
the import os allows us to use all the operating system specific functions.

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

PL/Python takes care of converting the python list object to a PostgreSQL set of text object.

Then to use the function we created, we can use it in a SELECT statement much like we can any table applying LIKE to the output to further reduce the records we get back.

```
SELECT file
  FROM ch10.list_files('C:/temp') As file
 WHERE file LIKE '%.xls';
```

In our next example, we will use this list to pass to our excel import function to get a distinct set of records. For this example we have to use a small hack to allow us to use the set returning excel export function in the SELECT part of our query. For PostgreSQL 8.4+, this hack is no longer needed for PLPgSQL and Perl, but still seems to be needed for PL/Python. The hack is to wrap the Python function in an SQL function wrapper.

```
CREATE OR REPLACE FUNCTION ch10.fnsqlgetxlspts(param_filename text)
RETURNS SETOF ch10.place_lon_lat AS
$$
  SELECT * FROM ch10.fngetxlspts($1);
$$
LANGUAGE 'sql' VOLATILE;
```

The whole purpose of the above hack is to allow us to use a non-SQL/C set returning function in the SELECT clause instead of the FROM clause of an SQL statement. This allows for row expansion. We document the technique at <http://www.postgresonline.com/journal/index.php/archives/16-Trojan-SQL-Function-Hack-A-PL-Lemma-in-Disguise.html>

Now for the real work.

```
INSERT INTO ch10.imported_places(place, geom)
SELECT place, ST_SetSRID(ST_Point(lon,lat),4326)
FROM (
  SELECT DISTINCT (ch10.fnsqlgetxlspts('C:/Temp/' || file)).*
    FROM ch10.list_files('C:/Temp') AS file
   WHERE file LIKE 'Test%.xls'
 ) As d;
```

This example is similar to our last except that we are doing 3 interesting things. For each file in our Temp directory that starts with Test and ends with xls, we are importing the data into our places table, but we are only importing distinct values across all the files using the DISTINCT SQL predicate.

#### **10.5.4 Geocoding with PL/Python**

If per chance you have the need to geocode, but don't want to manage all that data, PL/Python is a great tool for enabling geocoding within your database using a third-party service such as Google Maps, Map Quest, Yahoo Maps or Bing. There are numerous Python packages you can find at the Cheese shop to do just that. One example is the googlemaps package which you can download from: <http://pypi.python.org/pypi/googlemaps/1.0.2> and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

either run the windows setup if you are on windows or compile your own on Linux. This particular package contains geocoder, driving directions and reverse geocoder functionality. Once you have the package installed, you can run the below exercises.

### **GEOCODING WEBSERVICES CAVEATS**

Although you can call webservices with Python easily, geocoding services tend to cost money or have limits on use. As a result, you may be better off downloading the data and building your own geocoder with the Tiger geocoder kit we discussed earlier.

For this example, we'll create a geocode function and geocode the same test addresses we had done earlier.

First we create a PostgresQL datatype to return our results in:

```
CREATE TYPE ch10.google_lon_lat AS (lon numeric, lat numeric);
```

Then we define a function using the google maps library that takes a text address and returns the long lat location using the defined type:

```
CREATE FUNCTION ch10.google_geocode(param_address text) RETURNS
ch10.google_lon_lat
AS
$$
from googlemaps import GoogleMaps
gmaps = GoogleMaps()
arg_lat, arg_long = gmaps.address_to_latlng(param_address)
return (arg_long, arg_lat)
$$
language 'plpythonu';
```

Now we can use that function in an SQL statement similar to how we used our tiger geocoder function:

```
SELECT address, (foo.g).lon, (foo.g).lat
FROM (
    SELECT address,
           ch10.google_geocode(address) As g
    FROM ch10.addr_to_geocode) AS foo;
```

The result of our query is shown below.

**Table 10.5 Results of our google map geocoder**

address	lon	lat
1000 Huntington Street, DC	-77.075906	38.957687

:

If we were to use this googlemaps class in Python outside of PostgreSQL, we would have to:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- Establish a connection to our PostgreSQL database with a few lines of python code and a connection string
- Pull the data out of our database
- Loop thru each record stuffing the new value, and update statement to push the data back to db.

By having the power to package our python as a stored function, we can reuse this same function easily against any query we have by writing a simple select or update statement. We can even use it in reporting tools that don't have access to python. We can also include it in a trigger to geocode as an address changes in the database.

## **10.6 Summary**

In this chapter, we introduced various tools you can use to enhance the functionality of PostGIS and PostgreSQL without ever leaving the database. We demonstrated loading tiger data for geocoding and using the geocoder functions provided by the Tiger geocoder scripts to geocode data with SQL. We then went on to demonstrate how you can solve routing problems with just SQL using PgRouting. We showed off a small bit of what you can accomplish with PL/R and PL/Python. We also demonstrated how to tap into the extensive network of prepackaged functions that R and Python offer and use them directly from PostgreSQL. We hope we piqued your curiosity enough that you will further explore these tools and discover what other treats they hold in store.

In the next section, we'll talk about another set of server side tools. These tools are for displaying GIS data to the world and allowing the world to edit your data via a web interface or desktop tool. In the next chapter, we will leave the safe confines of our database and expose more of our data to the world to see and enjoy.

# 11

## *Using PostGIS in web applications*

This chapter covers

- Shortcomings of conventional web solutions
- MapServer and GeoServer
- OpenLayers

In a short span of fifteen years, the World Wide Web has emerged as the leading method of information delivery, threatening to replace printed media altogether. For GIS, this has been a godsend; not only did the web introduce GIS to the popular imagination, it also affords a delivery mechanism for GIS data that would not have been possible via traditional printed media. Only twenty years ago, a GIS practitioner wishing to share his data would have had to print out large maps on over-sized printers–then came the web.

To deliver textual data and image data, conventional web technologies suffice, but for the ultimate GIS web surfing experience, we need additional tools, both on the delivery end (the server), and the receiving end (the client).

In this chapter we will cover web tools that work with PostGIS. We start with two server tools – MapServer, GeoServer – that can read data from PostGIS and serve images or data according to OGC standards. We will then move onto the client-side of the equation where we look at OpenLayers, a JavaScript-based tool that greatly enrich the viewing experience for the user. Along with OpenLayers, we will check out the new GeoExt extension to OpenLayers based on the new ExtJS framework.

### **11.1 GIS and the Web**

The first question on many readers' minds could well be why we need anything above and beyond the technologies widely available to serve up web pages. After all, we can render our textual GIS data with HTML and our maps using many of the supported image formats and send them off to the browsers upon request. In this section we start by pointing out the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

limitations of conventional web technologies in serving up dynamically generated maps, both from the server and the client perspective. We then introduce the current de-facto standard for serving up GIS data—OGC web services.

### **11.1.1 Limitations of conventional web technologies**

Conventional web technologies work well for static data and images, but suppose that we need a website where users can come and extract our map in various zoom levels. Using conventional web server technology, we would have to limit the user to a fixed set of zoom levels, generate the images beforehand and serve them as requested. Now what if the user would like to see subsections of the map only, we would have to predicatively slice up our maps and only allow users to pick from one of our prepared slices. There are two big problems here: First, users would abandon the website quickly out of frustration since we cannot possibly predict what portions the users would like to see. Second, even if we were to generate thousands of subsections for the user to pick from, our server will most likely run out of storage space after just a few maps. Add back the zoom levels and we have an intractable problem.

The client side of the picture is not any rosier. To add in zoom level selectors, we could use standard HTML combo boxes, but we would have to vary the drop down list from map to map. If a map has three zoom levels, we would have to pre-populate our combo box with three values. If the next map has thirty zoom levels, we would have to have thirty rows in the combo box for the user to pick from. Using various programming technologies now available such as Python, PHP, ASP.Net, we can dynamically generate our HTML combo box, but this requires that the mapping person also be a web programmer and not just in a single language. The demands become even more challenging if we were to offer users the capability to draw rectangles around subsections of the map that they would like to see blown up, or features where they can add their own markers to the map or see description balloons popup when they hover over certain points of interest. These interface-centric features would all require expensive programming on the client-side. If server-side programming did not discourage the GIS specialist, the client-side programming surely will. What we need is a suite of client tools already programmed for us with nice controls useful for map viewing and editing. Sure, the suite will dictate the overall appearance and functionality, but this still beats having to architect our own solution from the ground up. After all, our goal is to disseminate our maps, not in web programming.

### **11.1.2 Mapping servers**

Mapping servers have one central purpose—render images for delivery to a client on the fly. As we mentioned previously, conventional web servers cannot serve up images unless they already exist, but to generate and store all possible subsections and zoom levels associated with a map is impractical. Mapping servers solve this problem by quickly generating the static images only when requested by the client.

As of the time of this writing, four major open source server products dominate: MapServer, GeoServer, FeatureServer, and SharpMap.NET. Since mapping servers are rarely the starting point of a GIS project, people generally find themselves needing to spatially extend existing web applications or they have data already that they want to disseminate through the web. To decide which server product to go with, we recommend that you start with how they fit into your current infrastructure and data landscape. You should consider the following:

- Will the selected product require a major change in existing platform?
- Which OGC web services, if any, do you need to provide?
- How well will it connect to data sources you already have whether that be PostGIS, Oracle Spatial/Locator, SQL Server 2008, SpatiaLite, MySQL, Shape files, Raster, or something else?

### **WHAT ARE WEB SERVICES?**

Loosely speaking, a web service is non-proprietary standard for function call across the internet. The service accepts requests from clients usually using http protocols and standard messaging streams (raw get, posts, XML, JSON, SOAP etc) and returns the output. To adhere to standards set by the W3C, a web service must make known the requests that it can fulfill. In the case of OGC web services, the services available are published via what is called a GetCapabilities request using XML. To consume web services, the requestor application generally creates stub classes to make the web service call appear no different than a local function call. Many tools are available to auto-generate stub classes to spare you the pain of having to write it yourself. A stub class contains methods to pass data from the client to the service for each kind of capability the service offers and handles the serialization/de-serialization of objects into XML, or some other format, so that they can traverse the internet.

### **PLATFORM CONSIDERATIONS**

One of the most important deciding factors of a tool are the platform requirements. If you are on a shared web host, you may not want anything that requires installation. Even if you have a server you control, you may shy away from technologies that require additional installation. Below is an outline of what each of the prerequisites associated with each mapping server.

**Table 11.1 Mapping server prerequisites**

Service	MapServer	GeoServer	FeatureServer	SharpMap.Net
Java SDK	No	Yes	No	No
Python	No*	No	Yes	No
.NET or Mono.Net	No	No	No	Yes

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

CGI/Fast-CGI	Yes	No	No
--------------	-----	----	----

Mapserver is one of the most popular of these tools because it packs a lot of functionality and can run under practically any web server without requiring installation. Just drop the compiled .so/.dlls/ .exe in the cgi or some other web server executable folder, and you have a completely functional web mapping service. MapServer also offers an API called mapscript, which has many flavors PHP Mapscript, Python Mapscript, and C# Mapscript being the most common. This allows for more granular control in conjunction with a web scripting language such as PHP by allowing you to create layers and other map objects from PHP, C#, Python server side code. The downside of the mapscript interface is that it also requires writing more code in general than using the cgi executable interface.

GeoServer is built on Java Servlets and some binaries come prepackaged with their own mini web server called Jetty. It requires a Java 1.5+ SDK be installed before it will function. If you need to run it under the context of an existing web server service, you will need a servlet plugin for your web server such as Tomcat and install the WAR version. Unlike the other tools, it comes packaged with a user-friendly web-based administrative interface, making GeoServer a popular option for those who prefer GUIs and wizards over configurations scripts.

SharpMap.net is a popular option for .NET programmers. It comes packaged as a .NET dll so all you have to do is drop it into the bin folder of your .NET application. You can therefore run it on a shared host environment where you do not have your own web server. On the down side, the SharpMap.net does a lot less out-of-the-box than MapServer or GeoServer, and you will need to make up for any shortcomings by adding additional coding yourself.

FeatureServer is a REST-based web feature server written in Python and was designed to work with OpenLayers. In order to feed PostGIS layers in the various formats, you need to have psychopg or psychopg2 installed. It can run as its own standalone python server, as a CGI or in Apache via mod\_python. It can also run under IIS if you have python bindings enabled. It's a bit trickier to setup than Mapserver or Geoserver, so we won't be covering it. The features that make it stand out from other web mapping servers is that it supports both the OGC WFS as well as a simpler REST interface. It supports some fairly esoteric data sources such as Twitter, SQLite, DBM, OSM, all OGR if you have OGR python plugin configured. It can output in KML, GeoRSS, GeoJSON, GML, HTML, and OSM.

#### **OGC WEB SERVICE SUPPORT**

You may recall from earlier chapters that OGC is short for the Open Geospatial Consortium. It is the accepted standards organization in the world of GIS. OGC has outlined a series of web services that mapping servers should provide. By adhering to these standard OGC web services, mapping servers free the end users from having to use a particular web client or desktop client implementation that will only work with their server counterparts. All open source web mapping clients and desktop tools such as the ones we will cover in Chapter 12 consume OGC web services. Even proprietary desk top applications such as Manifold, ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

CadCorp, and MapInfo have come around to offer decent support for OGC web mapping services.

The most common of the web services defined by OGC are as follows:

- Web Mapping Service (WMS) - for rendering vector and raster data as map images in Jpeg, PNG, Tiff, or some other raster format. This is most suitable if you want to show a map of an area, but downloading the vector data in its textual format and then rendering it would be too processor intensive. For example, if you want to display maps on a mobile device with limited processing power, subscribing to a WMS to pull ready-made Jpeg images makes more sense than pulling the raw vector data and then provide visual rendering on the fly.
- Web Feature Service (WFS) - for outputting vector data generally using some XML standard such as GML or KML. As of this writing Geography Javascript Object Notation (GeoJSON) is often an option and is more processor friendly for consumption in Javascript since its a native Javascript format. This includes both the geometry represented as JSON encoded as well as the standard database column attributes like dates, numbers and strings encoded as JSON. This service is most suitable if you want a user to be able to highlight regions of a map and have that geometry highlighted and the attribute info display as well as allow the user styling options client side without making roundtrips to the server. It is often used in conjunction with WMS. WMS would be used to show aerials or a large zoomed out region of a map. WFS would be used to overlay key features or allow more granular control when zoomed in.
- Web Feature Service (WFS-T) - for editing vector data in a transactional mode. This is necessary if you expect end users such as web users or desktop applications to edit geometry data in the database without giving them direct rights to the database.

There are other web services as well such as Web Tiling Service and Web coverage service. Here is a brief summary of the key OGC webservices and which tools support what.

The REST architecture is a lighter weight interface than WFS and relies on concepts of GET, PUTS and DELETEs to update data and output XML streams. A WFS that supports GET requests you can consider for all intents and purposes as a REST service.

**Table 11.2 Webservices Support**

Service	Mapserver	Geoserver	FeatureServer	SharpMap.Net
WMS	Yes	Yes	No	Yes
WFS	Yes	Yes	Yes	No
WFS-T	No	Yes	No	No
Custom REST	Yes	Yes	Yes	Yes*

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

### SUPPORTED DATA SOURCES

All maps come from some kind of data. The WMS/WFS/WFS-T protocols are there to allow various data sources to be accessed via one web interface. They provide an abstract interface for GIS data similar to what ODBC and JDBC drivers do for databases. Each of these web mapping server tools, supports various data formats. Below are the ones that we think are key and a key deciding factor when choosing one of these. All these tools support PostGIS geometry types and ESRI shape files directly out of the box so we have left that out of the list.

The \* means it supports via an extra downloadable plugin or library and not prepackaged in the generic binary download or lacks a native driver.

**Table 11.3 Data source formats supported**

Service	Mapserver	Geoserver	FeatureServer	SharpMap.Net
Oracle Spatial/Locator	Yes*	Yes*	No	Yes
SQL Server 2008	Yes*	Yes*	No	Yes
DB2	No	Yes*	No	No
PostGIS geography	Yes*	No	No	No
PostGIS WKT Raster	Yes	No	No	No
Basic Raster	Yes	Yes	No	Yes
MrSid	Yes	Yes	No	No
SpatiaLite	Yes*	No	No	Yes
MySQL	Yes*	Yes*	No	Yes*

### 11.1.3 Mapping clients

Once you have web mapping services set up, you need client applications to consume them. Client applications come in two basic flavors: desktop and web. Web applications are often implemented using AJAX and a mix of web scripting languages.

Many desktop mapping toolkits are also capable of consuming standard OGC web mapping services. A desktop client can either be an open source desktop tool such as QuantumGIS, uDig, gvSig, OpenJump which we shall cover in the next chapter, and countless other open source desktop mapping environments, or a proprietary desktop tool such as Manifold, MapInfo, Cadcorp SIS, and ArcGIS desktop to name a few.

As far as web mapping clients go, OpenLayers tends to be the most popular particularly in the Open Source GIS arena. The main reason for its popularity is that it gives you the ability to overlay proprietary non-OGC compliant mapping layers with OGC WMS, WFS, and WFS-T ones.

OpenLayers is often extended to create more advanced or specific tool kits. Two common ones that build on top of OpenLayers are GeoExt and MapFish.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

GeoExt is a web mapping javascript framework that combines OpenLayers with ExtJS to provide a web client interface with more of a desktop feel.

MapFish combines OpenLayers, GeoExt/ExtJS for the client-side and Python/Pylons on the server-side to create a complete solution for client and server. It offers printing to PDF, and authentication among other advanced features.

#### **11.1.4 Proprietary services**

We would be amiss if we fail to mention that the most popular web mapping solutions around are still proprietary services such as GoogleMaps, Bing, and MapQuest. These services package server, client, and data together in a slick, easy-to-use interface and made mapping accessible to the general public. Though these packages are easy to use, each has their own proprietary javascript api with very limited control as to how you can overlay your data. You will not be able to write SQL queries let alone represent anything more complex than points, and line segments without extensive effort.

One serious drawback is their proprietary and static nature, even to the data level. You really cannot remove one piece of their offering without discarding the whole thing. For example, if you wanted to display foliage density over a region instead of the usual streets and places, you cannot do so with these popular packages. You also cannot ignore the commercial licensing clause of these packages. For recreational use, these packages are mostly free, but once you start to use these for-profit or for non-public websites, you will find yourself needing to cough up a rather exorbitant licensing fee.

Since each has their own custom API that is fairly incompatible with each other, if you use their API directly and decide to swap services, you will find yourself rewriting much of your custom data overlay logic.

Despite their commercial bent, we must pay homage to these popular services for planting the seeds of GIS into the popular imagination. They were first to show to the world the power of dynamic mapping on the internet and continue to lead the way in terms of leading edge display technologies. Since this book is devoted to open source solutions, we will not cover proprietary javascript APIs, but we advise our readers to not lose sight of the important role they play on the world wide web today.

#### **11.1.5 Summary**

Each of these web GIS server tools provides a lot of functionality out of the box. In so doing, they constrain the way you interact with your database and other spatial data by requiring you follow certain protocols. For many solutions that only need light support for maps but heavier support of data, you may want to forgo web mapping services altogether and build the logic you need to display PostGIS data right in your application.

In the sections that follow we go into detail on the basics of setting up and using Mapserver and Geoserver as well as creating solutions that don't require you host your own web mapping services.

## 11.2 Using MapServer

If we wanted to do some heavy lifting by showing thousands of hefty features, then outputting as vector text would be slow and cumbersome. In this case, we will probably want to output image tiles using a Web Mapping Service or Tile service. As a user zooms in, we might want to complement this with either a vector output we rolled our own, or with a WFS service. For this next example, we'll demonstrate using Mapserver's WMS features.

### 11.2.1 Installing Mapserver

Mapserver is a very mature product and as such, there isn't much need to compile from scratch anymore unless you want to. There are already compiled binaries for most any OS you can think of. These are detailed at <http://mapserver.org/download.html#binaries>

#### WINDOWS INSTALL

For windows installs there are several options. The OSGeo4W and MS4W are heftier installs since they package in an Apache server and various other tools and often lack the c# .net binaries.

We like using the FWTools version since it is a bit lighter weight, tends to be very up to date and also often contains the latest dev version. It also includes the csharp interop extensions to allow using Mapscript from an ASP.NET (VB or C#) environment. To deploy on a Windows IIS server as CGI we usually do the following:

1. Extract the FWTools..exe installer (yes you can treat it as if it were a zip file)
2. Copy the contents of the \$HWNPARENT/bin folder somewhere on the webserver that is marked as allowing executables (this can be a cgi-bin or some folder you create that you mark as allow executable)
3. Copy the proj\_lib folder somewhere on the web server. You'll need to reference the path in your mapserver map file later, but doesn't need to be web accessible.
4. If you want to use .NET mapscript in VB.NET or CSharp or some other .NET language, then copy the csharp folder files into the bin folder of your .NET application. There are more thread issues especially in .NET mapscript so many people prefer to use SharpMap.NET for mapping that needs to intimately integrate with a .NET app.

#### SECURITY CONSIDERATIONS

If you are going to have PostGIS layers then you may need to put the password in the map file or a file included in the map file. You don't want this information readable and may not want your map files readable at all for intellectual property reasons.

There are a couple of ways commonly done to safeguard against this. You should probably do at least one of these

- Don't put your map file in a folder that is web accessible. Admittedly we tend to break this rule just out of convenience of having everything related together.
- Use the msencrypt executable packaged with mapserver to encrypt the password and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

use only the encrypted password.

- Use an INCLUDE clause in your map file and make sure the INCLUDE file is of an extension type that is not dished out by a webserver and use that. For example we use .config extension in IIS since ASP.NET will never dish out a file with this extension. Using an include for the PostGIS connection string is pretty convenient anyway if all your PostGIS layers use the same PostGIS database. Saves you having to repeat it over and over again.
- If you have control of your own webserver. Block dishing out .map files by editing your httpd.conf or in IIS mapping to like 404.dll or some other thing.

### **11.2.2 Creating WMS and WFS services**

Mapserver has both its own specific non OGC ways of rendering maps using .map and template files. We are just going to focus on its OGC WMS and WFS functionality. For the OGC WMS/WFS features, you don't need template files. A correctly configured mapfile with WFS/WMS metadata sections, set of fonts, a symbol set, and proj\_lib will do.

For our map files, we like to use INCLUDES for sections that we may reuse repeatedly within the map or reuse across several maps such as the PostGIS connection string or general configurations like location of proj library.

Below is what such a map looks like.

#### **Listing 11.1 Map with includes**

```

MAP
-- 1
INCLUDE "config.inc.map"
NAME "POSTGIS_IN_ACTION" #name to give your map service
EXTENT 221238 881125 246486 910582
-- 2
PROJECTION
  "init=epsg:26986"
END
WEB
  MINSCALEDENOM 100
  MAXSCALEDENOM 100000
-- 3
METADATA
  "ows_title" "PostGIS in Action Chapter 11"
  "ows_onlineresource" "http://mydomain/mapserv?map=postgis_in_action&"
  "wms_version" "1.1.1"
  "wms_srs"    "EPSG:2249 EPSG:4326 EPSG:26986 EPSG:3785 EPSG:900913"
  "wfs_version" "1.0.0"
  "wfs_srs"    "EPSG:900913"
END
END #End Web
INCLUDE "layers.inc.map"
END # Map File
-- 1 paths to proj and plugins

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

-- 2 default projection of map  
-- 3 wms/wfs metadata

- (1) We include a file called config.inc.map that will contain paths to our projection library, symbolset and fontset. All includes are relative to the location of the file they are included in.
- (2) This defines the default projection of the map if none is given. Each layer can be in a different projection, but they will be reprojected to the map projection when the map is called. This projection is often overridden in WMS calls with the SRS parameter.
- (3) The meta data is particularly important because it is what makes the map file behave like a true WMS/WFS. The ows\_\* elements are short-hand for saying wfs and wms so that you don't have to specify properties that are the same for both. The wfs in 1.0.0 (which mapserv 5.6 supports) can only have one SRS. The wms standard allows for many SRS and the ones listed are the ones the WMS service will allow as parameter for passed in SRS. The online resource gets displayed in the wms capabilities as the url to call to access the service.

The config.inc.map defines the location of the symbolset, proj library and fonts. It is shown in the below snippet:

```
CONFIG PROJ_LIB "c:/mapserv/proj_lib/"
SYMBOLSET "symbols/postgis_in_action.sym"
FONTSET "c:/mapserv/fonts/fonts.list"
```

The proj library is always an absolute physical path, but the symbolset and fontset can be absolute or relative to the location of the map file. If you are on windows you can copy the fonts you will use from your windows/fonts folder into your mapserv fonts folder and then list them in the fonts.list file as shown in <http://mapserver.org/mapfile/fontset.html>.

For the symbol set you can use map symbolset codes or images. A sample of both is packaged in the mapserver source download file.

In the next example, we show how what one of the layers in our layers.inc.map file looks like. Note you can include layers directly in the main mapfile.

### **Listing 11.2 Sample Layer from layers.inc.map**

```
LAYER
-- 1
NAME major_roads
TYPE LINE
STATUS ON
DUMP TRUE
-- 2
INCLUDE "postgis.config"
DATA "geom from ch11.ma_eotmajroads using unique gid using srid=26986"
PROJECTION
  "init=epsg:26986"
END
LABELITEM "rt_number"
METADATA
  ows_title "Massachusetts Major Roads"
  gml_include_items "all"
  ows_featureid "gid"
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```
END
CLASS
COLOR 255 0 0
LABEL
TYPE truetype
FONT arial
MINDISTANCE 50
POSITION AUTO
-- 3
ANGLE AUTO
SIZE 6
COLOR 0 0 0
END
END
```

Every map layer starts with (1) LAYER and has a NAME and TYPE. Type for postgis layers is usually LINE, POINT, POLYGON, or ANNOTATION. (2) We include a file called postgis.config and will have this include for each of our postgis layers to define the connection string to our postgis database. The postgis.config file looks something like below:

```
CONNECTIONTYPE POSTGIS
CONNECTION "host=localhost dbname=somedb user=someuser port=5432
password=something"
PROCESSING "CLOSE_CONNECTION=DEFER"
```

The CLOSE\_CONNECTION=DEFER ensures that if multiple PostGIS layers are asked for, the connection will be reused instead of creating a new connection. This results in faster performance.

So we have a map file now, how do we turn this map file into a WMS/WFS service? We simply call the mapserver CGI with the map file as argument. Below is a snippet of code that calls the GetCapabilities to show what layers and functionality is provided.  
`http://yourserver/cgi-bin/mapserv.exe?map=c:/mapserver/maps/postgis_in_action.map&REQUEST=GetCapabilities&SERVICE=WMS&VERSION=1.1.1`

### **11.2.3 Calling a mapping service using a reverse proxy**

Passing in a map file for each call is often a bit long. Many people prefer to set up either a CGI script or a reverse proxy so that the need for calling the map file is not necessary. You can do a bit more with a reverse proxy than you can with a CGI script.

#### **WHAT IS A REVERSE PROXY**

A reverse proxy is a server that behaves as a client and may have access to other services such as web mapping servers that a requesting client does not have direct access to. It is often used for load balancing by accepting requests from a web browser on the outside and funneling them thru to the least busy mapping server. In addition you can use it to call services on the same machine that may be running on other ports.

If we use a reverse proxy or a cgi-bin script, our long map url example can be reduced down to:

```
http://yourserver/GetPAMap.ashx?REQUEST=GetCapabilities&SERVICE=WMS
&VERSION=1.1.1
```

In this next snippet of code, we demonstrate what a simple reverse proxy written in C# looks like. This is just a snippet. We have equivalent code in the source download packaged for VB.NET. If you are using PHP, you can implement similar logic using curl. This kind of script can be used to also dish out GeoServer web mapping services, if for example you decide you want GeoServer running on its own Apache or Jetty webserver on a local port or even on a separate server in your internal network and you want your regular port 80 to run in Apache or IIS. The example below only deals with GET requests which is generally what most WMS use. For POST you can do a check on the Request method and loop thru REQUEST or posted variables as well.

### **Listing 11.3 Snippet of a Reverse Proxy in C#**

```
string mapURLStub = "http://yourserver/cgi-bin/mapserv.exe?map=";
string mapfile = "c:/mapserver/maps/postgis_in_action.map";
System.Net.HttpWebRequest WebRequestObject;
System.IO.StreamReader sr;
System.Net.HttpWebResponse WebResponseObject;
System.Text.StringBuilder sb = new System.Text.StringBuilder();
System.Text.StringBuilder sb = new System.Text.StringBuilder();
sb.Append(mapURLStub + mapfile);
--1
foreach (var key in context.Request.QueryString.AllKeys) {
    sb.Append("&" + key + "=" + context.Request.QueryString[key]);
}
WebRequestObject = (System.Net.HttpWebRequest)
System.Net.WebRequest.Create(sb.ToString());
WebRequestObject.Method = "GET";
WebResponseObject = (System.Net.HttpWebResponse)
WebRequestObject.GetResponse();
if (context.Request["REQUEST"].ToLower() == "getcapabilities" ||
context.Request["REQUEST"].ToLower() == "getfeatureinfo") {
    --2
    sr = new System.IO.StreamReader(WebResponseObject.GetResponseStream());
    context.Response.ContentType = "application/xml";
    context.Response.Write(sr.ReadToEnd());
}
else {
    --3
    context.Response.ContentType = context.Request["format"].ToString();
    System.IO.Stream outs = WebRequestObject.GetResponse().GetResponseStream();
    byte[] buffer = new byte[0x1000];
    int read;
    while ((read = outs.Read(buffer, 0, buffer.Length)) > 0){
        context.Response.OutputStream.Write(buffer, 0, read);
    }
}
--1 loop request variables
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```
-- 2 xml request to mapserver  
-- 3 image request to mapserver
```

We first (1) loop thru all the arguments passed in via the client query string. (2) We then check to see if the OGC request is a GetCapabilities or GetFeatureInfo, if it is, we are going to assume the result returned by our internal server is going to be XML. If it is not, we will assume it will return an image and (3) process as image.

In order to overlap our PostGIS mapserver layers using our reverse proxy, we would use code similar to below.

#### **Listing 11.4 PostGIS Mapserver Layers using Proxy**

```
var postgiswmsurl = "http://www.postgis.us/demos/chapter_11/GetPAMap.ashx?"  
map.addLayer(new OpenLayers.Layer.WMS("My PostGIS Layers", postgiswmsurl,  
{ 'layers': "hospitals,major_roads,openspace",  
'transparent': "true", 'FORMAT': "image/gif"},  
{ 'isBaseLayer': false, 'visibility': true, 'buffer': 1,  
'singleTile':false, 'tileSize': new OpenLayers.Size(200,200),  
'attribution': 'Data downloaded from <a  
href="http://www.mass.gov/mgis/">MassGIS</a>' })  
);
```

In the next section, we discuss setting up GeoServer and configuring it for WMS and WFS services. GeoServer as mentioned earlier is another mapserver similar to Mapserver that has an administrative interface, also support WFS-T, but on the downside relies on Java SDK and a servlet engine. This makes it a bit trickier to configure to run in an existing web server than Mapserver.

## **11.3 Using GeoServer**

Geoserver is similar in flavor to Mapserver except that it is a bit heftier, comes with a user-interface so not as much need for manually configuring files with a text editor, and supports WFS-T.

### **11.3.1 Installing GeoServer**

Geoserver comes with various install options. These you can download from

<http://geoserver.org/display/GEOS/Stable>

- Setup installers for Windows and Mac that guide you thru the steps. Comes with jetty mini web server.
- Java binaries for any OS that you can just extract into a folder, and manually set the environment variables yourself. Comes with jetty.
- A web application archive (WAR) for those who already have a servlet engine installed on their server and just want to run geoserver as another servlet application. This doesn't come with jetty.

For our case, we chose the Java binary geoserver-2.0.1 version and did the following

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

5. Make sure you have java jdk 1.5+ installed
6. Extract the folder into root something like C:\geoserver or /usr/local/geoserver
7. On windows set system environment variables - JAVA\_HOME would be something like C:\Program Files\Java\jdk1.6.0\_16 (or whatever latest jdk you have)
8. cd into the geoserver\bin folder and from commandline -- run startup.bat (for windows), startup.sh for Linux/Unix
9. You then should be able to get to the administrative panel by navigating to the below link on your web browser <http://localhost:8080/geoserver>

### **11.3.2 Setting up PostGIS workspaces**

In this section we'll cover setting up a geoserver workspace to house our tables and registering PostGIS tables with GeoServer.

10. From Admin menu - Data - Choose Workspaces and click to add a new one. Your screen should look something like below:

The screenshot shows a 'New Workspace' configuration dialog. It has a title bar 'New Workspace'. Below it is a sub-header 'Configure a new workspace'. There are three input fields: 'Name' containing 'postgis\_in\_action', 'Namespace URI' containing 'http://postgis.us', and 'Default workspace' with a checked checkbox. At the bottom are two buttons: 'Submit' and 'Cancel'.

Figure 11.1 Setting up a GeoServer workspace

11. From the Admin left navigation menu -> Data -> Stores
12. Click Add New Store and then choose PostGIS from the list of options:

New data source

Choose the type of data source you wish to configure

**Vector Data Sources**

- Directory of spatial files - Takes a directory of spatial data files and exposes it as a data store
- PostGIS - PostGIS Database
- PostGIS (JNDI) - PostGIS Database (JNDI)
- Properties - Allows access to Java Property files containing Feature information
- Shapefile - ESRI(tm) Shapefiles (\*.shp)
- Web Feature Server - The WFSDataStore represents a connection to a Web Feature Server. This connection provides a way to perform transactions on the server (when supported / allowed).

**Raster Data Sources**

- ArcGrid - Arc Grid Coverage Format
- GeoTIFF - Tagged Image File Format with Geographic information
- Gtopo30 - Gtopo30 Coverage Format
- ImageMosaic - Image mosaicking plugin
- WorldImage - A raster file accompanied by a spatial data file

Figure 11.2 Adding a Geoserver PostGIS data source

13. Give the data source a name -- ch11 and fill in all the credentials asked for. By default geoserver stuffs in public for schema which means it will only list layers in the public schema. If you want it to be able to list for a different schema, in our case we chose ch11, then replace public with ch11.
14. Select Layers from admin menu -- and click Add a new resource and choose the postgis\_in\_action store you had created. Your screen should look something like:

Published	Layer name	
✓	ma_eotnajroads	Publish again
	ma_hospitals	Publish
	ma_openspace	Publish
	ma_rtemarkers	Publish

Figure 11.3 Selecting PostGIS Layers

## GEOSEVER DATASTORES FROM OTHER SCHEMAS

It is possible to leave the schema setting blank in Geoserver PostGIS and the layer chooser will list them all, however we have found that in that case, Publishing layers in non-public results in an error. So be advised to create a different data store for each schema you want to publish.

15. Publish a layer you want. Make sure to click the compute bounding boxes from data.
16. Click Add new resource
17. Repeat 6 and 7 for each layer you want to publish

### 11.3.3 Accessing PostGIS Layers via GeoServer WMS/WFS

Once you publish your PostGIS layers, you can quickly see them via the Layer Preview menu link. Below is an example of what that screen looks like. Note it also shows you OpenLayers code you can use to call the layer. It also support GeoJSON as a direct WFS output format.



Figure 11.4 Layer Preview screen of Geoserver

### 11.3.4 Summary

OpenLayers is a popular web mapping client companion for Geoserver and UMN Mapserver. As you saw in the layer preview, Geoserver even autogenerated OpenLayers sample javascript code to display each of your layers.

In the next section, we'll introduce you to using OpenLayers and GeoExt. OpenLayers and GeoExt are two web mapping javascript frameworks that are designed to work together. OpenLayers provides basic mapping functionality for loading layers, editing widgets and so forth. GeoExt builds on top of OpenLayers by providing more controls like data grids and some other controls that give a web map more of a desktop feel. These two frameworks are useful for both using commercial layers, WMS services dished out with

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

Geoserver/Mapserver/SharpMap.NET, as well as creating your own solution completely with a web scripting language such as ASP.NET or PHP.

## **11.4 Basics of OpenLayers and GeoExt**

In the beginning, there were all these mapping services such as GoogleMaps, Virtual Earth, MapQuest, Yahoo with their own proprietary Javascript apis for accessing their data. This was bad, because if you decided you liked the maps of service A better than the maps of service B, or As usage and pricing became too cumbersome, then you had to rewrite everything. Worst yet, if you had your own data that you fed via an OGC WMS or ArcGIS/IMS server that had specific data for your area of interest, such as the migration of elephants, it was hard to integrate the nice base layers provided by these services, with your custom study area layers.

OpenLayers changed the landscape quite a bit by allowing layers provided by different vendors with vastly different APIs to be accessed using the same API. Better yet, to be used together in the same map. OpenLayers started life as an incubation project of MetaCarta, because they needed to create an easy to use toolkit for customers to digest their map product offerings. OpenLayers is now an incubation project of OSGEO.

What does OpenLayers give you that you can't easily get elsewhere:

- Layer classes to access most of the proprietary non-OGC compliant Tile map offerings - such as GoogleMaps, VirtualEarth (Bing), MapQuest, Yahoo, ArcGIS Rest using pretty much the same interface across all of them.
- Layer classes to access OGC compliant map servers WMS, WFS, WFS-T again using the same fairly consistent map layer creation call
- Ability to overlay all these competing proprietary services in one map
- Various controls that allow you to build custom menus, toolbars, and editing widgets to enable map editing.

All these things are wonderful and that is why OpenLayers has become as popular as it is. Most great things are not without their tradeoffs. So what are these tradeoffs?

- Its hard to get at the deep features of a proprietary offering -- such as the 3D street views provided by GoogleMaps and Bing. This may change as new OpenLayers layer classes are added to support these new offerings. Note that GeoExt does have controls to get at GoogleMaps Street View and synchronize with your map.
- Yet another API to learn with the hope that you don't have to learn any other API.

### **11.4.1 Using OpenLayers**

The official site for OpenLayers is <http://www.openlayers.com>. Class documentation is available though you will probably find the numerous code samples to be much more useful to getting started. Since OpenLayers is nothing more than a glorified JavaScript file, you can

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

download the file and use it directly from your web server. Alternatively, you can link directly to it in your code ensuring that you always have the latest version.

One thing that OpenLayers is particularly good at, is allowing you to integrate various map sources from disparate services. It has classes for accessing ArcGIS Rest, ArcIMS, Google, Bing (VEarth), OpenStreetMap (OSM), Mapserver specific API (which has for the most part been superceded by just using WMS API), as well as standard OGC compliant WMS and WFS services produced with tools like Mapserver, Geoserver, Degree, and TinyOWS.

For our first example, we are going to use a base layer offered by MetaCarta and add a WKT layer with points marking New York City, Los Angeles, Chicago, Houston, Philadelphia (the five largest cities in the US at the time of writing). What we want you to observe from the code is that an OpenLayers HTML file almost always includes the following sections:

- OpenLayers and other relevant js includes. For custom layers such as OSM, Google or Bing, you would include the script source that points to those sites. In this case we include the custom script source from OpenStreetMap site that includes the OSM class. Since OpenStreetMap builds their site on OpenLayers as well, they simply extend the OpenLayers base classes. For other layers such as Google, you will find in the OpenLayers kits classes that wrap the Google API in a stub OpenLayers.Layer.Google class and so forth.
- The map object that is defined in JavaScript and is created and initialized in an initialization method
- The call to the initialization method is either in the body onload or the end of the javascript section. We do not like to put this in the body onload since for certain languages like ASP.NET, it gets messy putting calls in the body load events.

The full map can be seen at

[http://www.postgis.us/demos/chapter\\_11/osm\\_newengland1.htm](http://www.postgis.us/demos/chapter_11/osm_newengland1.htm)

and can be downloaded as part of the chapter 11 download.

### **Listing 11.6 OpenLayers General setup**

```
--1
<script src="js/o128/OpenLayers.js"></script>
--2
<script
src="http://www.openstreetmap.org/openlayers/OpenStreetMap.js"></script>

<script type="text/javascript">
--3
    var lat=43.66596;
    var lon=-73.13868;
    var zoom=7;
    var map; //complex object of type OpenLayers.Map
    function init() {
--4
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```

map = new OpenLayers.Map ("map", {
    controls:[new OpenLayers.Control.Navigation(),
              new OpenLayers.Control.PanZoomBar(),
              new OpenLayers.Control.Attribution(),
              new OpenLayers.Control.mousePosition()],
    maxExtent: new OpenLayers.Bounds(-8879149, 4938750,-
7453286, 6017794),
    maxResolution: 156543.0399,
    numZoomLevels: 20,
    units: 'm',
    projection: new OpenLayers.Projection("EPSG:900913"),
    displayProjection: new OpenLayers.Projection("EPSG:4326")
} );
-- 5
layerMapnik = new OpenLayers.Layer.OSM.Mapnik("OSM Mapnik");
map.addLayer(layerMapnik);

if( ! map.getCenter() ){
-- 6
    var lonLat = new OpenLayers.LonLat(lon, lat).transform(new
OpenLayers.Projection("EPSG:4326"), map.getProjectionObject());
    map.setCenter (lonLat, zoom);
}
}
</script>
-- 1 reference openlayers classes
-- 2 reference openstreetmap classes
-- 3 declare center and zoom
-- 4 instantiate ol map
-- 5 add map layers
-- 6 center and zoom map

```

(1) We first include the source to OpenLayers.js. If you want to customize or fully control, you should download and have this as a local reference as shown in the above code. You can also link directly to the one on the OpenLayers website <http://openlayers.org/api/2.8/OpenLayers.js> if you want to get going quickly. (2) We include link to OpenStreetMap additional classes. In the case of GoogleMaps or Bing, you would need to include scripts from those to use those as map layers. (3) We declare our global variables. The zoom denotes the default zoom level (which will be the default for our PanZoomBar)(4) The init function is the meat of the setup. It instantiates the OpenLayers map and loads it into a div called "map". You can call the div anything you want. We are only loading a few basic controls. Note that because we set teh displayProjection to be EPSG:4326 then the mouse position shows in long lat units instead of units of map. (5) We declare our OSM tile class. There are several to choose from Mapnik, CycleMap, Osmarender each of which has slightly different look and feel and data. (6) We are centering the map on the lon lat point we declared in (3). Note the transformation step from long lat to the map units.

In our next part, we will add the body of our html page.

```

<div style="width:100%; height:100%" id="map"></div>
<script type="text/javascript" defer="true">

```

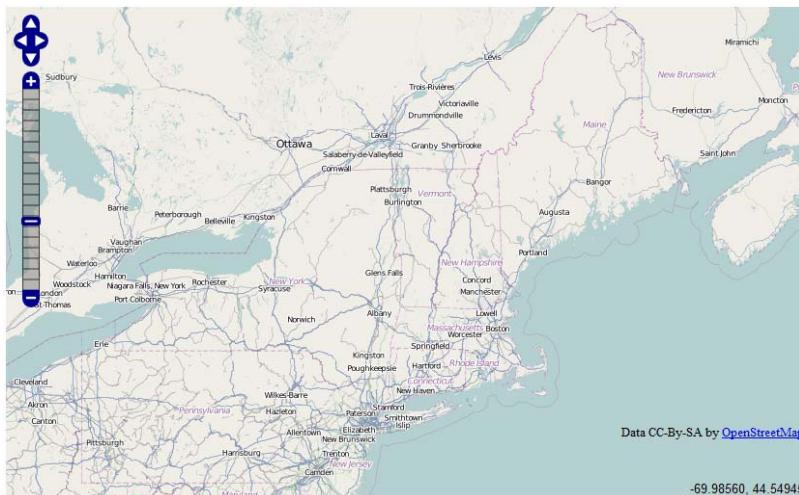
©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```
init();
</script>
```

We create the div for the map as a position to place the map. In this case we set to 100% by 100% so the map will expand dynamically to fit the page. In many cases you will want to set this a fixed pixel size width: 500px etc. We then call the init() function after and set to defer so that it doesn't get called until the rest of the page has loaded.

After we are done with all the above, our map will look as shown below:



**Figure 11.6 Result of our map in Listing 11.6**

In many cases, you will want the user to be able to pick what layers to display from a menu of layers. You may also want them to be able to toggle between several base layers. In order to allow this, we will add the OpenLayers control called LayerSwitcher as well as adding another third party layer. Some layers are always base, but for some others such as WMS, to make it a base layer you need to set the isBaseLayer property of the layer = true.

In this next example, we'll add Yahoo maps and the layer switcher control. The resulting map can be seen in [http://www.postgis.us/demos/chapter\\_11/osm\\_newengland2.htm](http://www.postgis.us/demos/chapter_11/osm_newengland2.htm)

### **Listing 11.7 Revising Map to have Bing as an option**

```
:
--1
<script
src="http://api.maps.yahoo.com/ajaxymap?v=3.0&appid=postgisus"></script>
:
function init() {
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```

:
-- 2
var lyryahoohyb = new OpenLayers.Layer.Yahoo(
    "Yahoo Hybrid",
    {'type': YAHOO_MAP_HYB, 'sphericalMercator': true}
);
map.addLayer(lyryahoohyb);

-- 3
map.addControl( new OpenLayers.Control.LayerSwitcher() );
:
}
-- 1 include yahoo map classes
-- 2 create yahoo layer and add
-- 3 add layer switcher

```

In (1) we add the script source to Yahoo which is needed for the OpenLayers Yahoo layer class. This class just acts as a proxy and translates the OL settings to Yahoo api settings. (2) We then revise our init function to create our Yahoo layer. Note the sphericalMercator setting. Without this, the Yahoo layer will not overlay with our OSM layer which is a mercator projection. (3) We add a layer switcher control so that the user can toggle back and forth between the 2 layers.

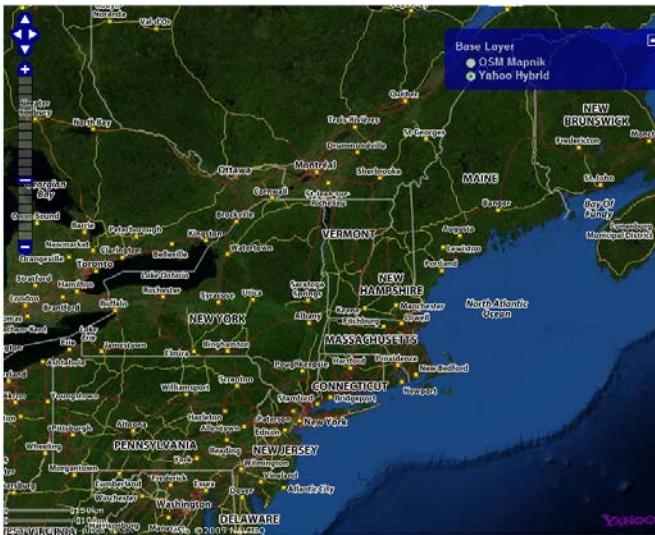


Figure 11.7 Change in map after adding change in Listing 11.7

We have just experimented with Base Layers. For any map, there can only be one base layer selected. The other type of layer is an Overlay layer which sits on top of the base. You can have as many overlays as you want selected.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

## ADDING WMS LAYERS TO A MAP

Most of the open source web mapping servers support the OGC Web mapping service standard. This means you can add layers you publish with them as well as layers you share with third parties that use this standard in more or less the same way to OpenLayers.

In this next example we will demonstrate adding MassGIS layers to our OpenLayers map. MassGIS has a very useful page describing how to use their web services as well as detailing the fundamentals of the WMS and WFS standards. You can find that information here: <http://lyceum.massgis.state.ma.us/wiki/doku.php>

For this next example, we are going to use OpenStreetMap mapnik map style as a base layer and add MassGIS WMS layers. Note that MassGIS, the primary provider of GIS data for our state, uses Geoserver. So when you setup your geoserver, the way you overlay your services will be pretty much the same as you see here. This particular example you can view at [http://www.postgis.us/demos/chapter\\_11/olmapmassfish.htm](http://www.postgis.us/demos/chapter_11/olmapmassfish.htm).

### Listing 11.8 Adding a WMS layers to OpenLayers

```
var massgisws = "http://giswebservices.massgis.state.ma.us/geoserver/wms"
    var massgiswsleg =
"http://giswebservices.massgis.state.ma.us/geoserver/wms/GetLegendGraphic?VERSION=1.0.0&FORMAT=image/png&WIDTH=20&HEIGHT=20&LAYER="
-- 1
map.addLayer(new OpenLayers.Layer.WMS("MassGIS: Seafood", massgisws,
    { 'layers':
"massgis:MORIS.QUAL_COMM_FISH_LOBSTER,massgis:MORIS.QUAL_COMM_FISH_WFLOUNDER,massgis:MORIS.QUAL_COMM_FISH_BSB",
        'styles': "",
        'transparent': "true", 'FORMAT': "image/png" },
        {'attribution': '<br /><a href="http://www.mass.gov/mgis/">MASSGIS<EOC</a> Fish',
        'isBaseLayer': false, 'visibility': true, 'buffer': 1,
        'singleTile':false, 'tileSize': new OpenLayers.Size(200,200) })
    );
-- 2
    $('legend').innerHTML = 'Layers from MassGIS (EOC)
<br />Lobster <br />' +
'Flounder: <br />' +
'Black Sea Bass: ';
-- 1 add wms layer
-- 2 pull legend images from wms
```

In this snippet of code we add (1) 3 seafood layers from MassGIS web services. These ones by default are overlays. Since MassGIS doesn't have a default attribution, we also add attribution text that will show in the attribution section whenever this layer is selected. (2) We also use another feature of WMS the GetLegendGraphic to get the graphics for each of the layers and stuff it in a div element called "legend".

The result of the above snippet is shown below.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

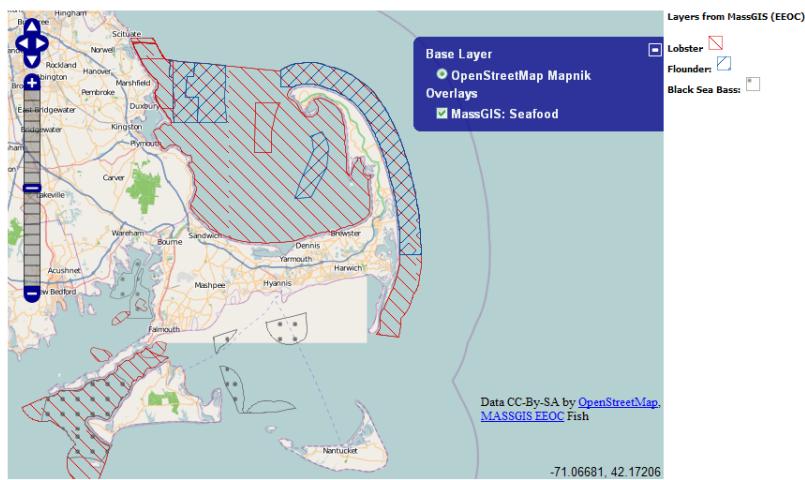


Figure 11.8 Example of overlaying WMS layers on an OpenStreetMap base using code in Listing 11.8

For our next example, we are going to enhance our OpenLayers experience with GeoExt.

#### 11.4.4 Enhancing OpenLayers with GeoExt

All those neogeographers were happy with OpenLayers. With happiness came the realization, that yes we can be happier. People wanted grids to show attribute data. They wanted people to be able to drag and drop things. They wanted to be able to sort tables and have tree menus. They wanted selections on a grid that can be selected and would reposition the map. They wanted sliding controls, date pickers, charts. In essence they wanted to build web mapping applications that felt more like desktop apps without need of messy hefty flash and SilverLight plugins. All these things could be done with OpenLayers if you were willing to do the additional custom JavaScript programming. A lot of this functionality existed in ExtJS. ExtJS is a popular JavaScript API for making rich web applications that feel more like desktop applications. What the GeoExt project did was to combine the OpenLayers mapping focus with the ExtJS general web application UI focus to create something that would be the best of both worlds, but as a compromise, would depend on both toolkits. Haiti Crisis Map <http://haiticrisismap.org/> is an example of an application built on GeoExt which uses ExtJS accordion panes to enable different features. Much of the GeoServer administrative web interfaces are now also built with GeoExt.

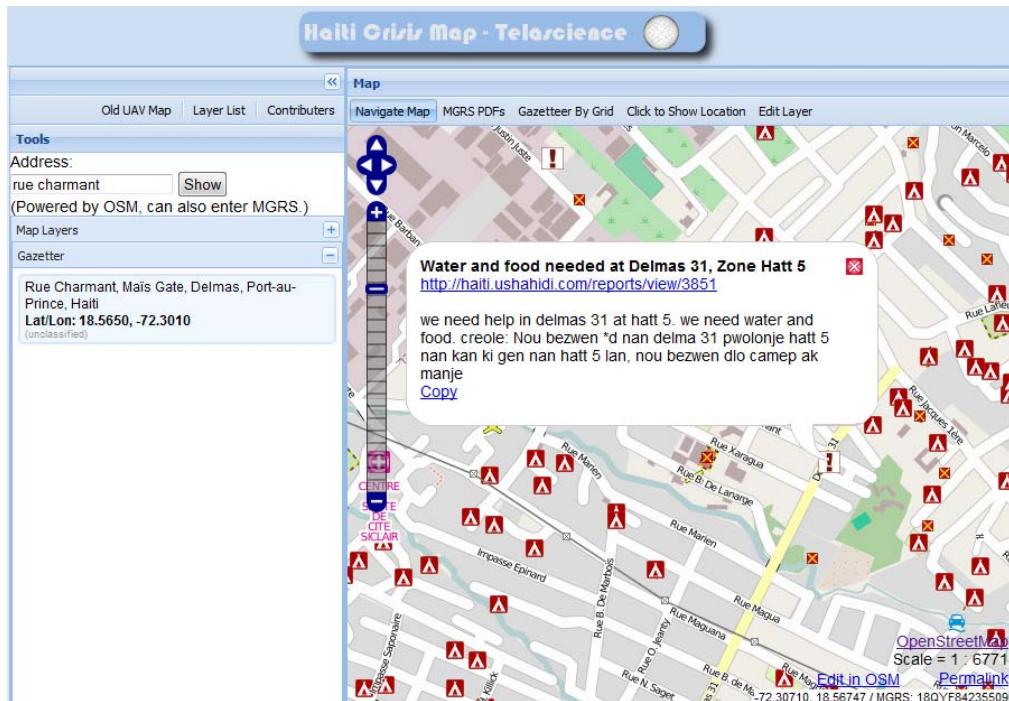


Figure 11.5 Example of GeoExt application using ExtJS collapsible panels and OpenLayers

So in short, you can use OpenLayers by itself and most people still do, or you can enrich it with GeoExt. We can't really talk about GeoExt, without first demonstrating OpenLayers. In the next section, we'll demonstrate a common activity with OpenLayers and then enhance it with GeoExt.

Both OpenLayers and GeoExt have fairly liberal commercial friendly licenses. OpenLayers is under an MIT License and GeoExt is under a BSD license. However GeoExt also relies on ExtJS which is under a dual GPLv3 and commercial license. This means if you need to extend any or modify any of the classes in ExtJS without making your source code available to the public, then you need to use the commercial ExtJS license. Details here <http://www.extjs.com/products/license-faq.php>

In order to use GeoExt, you need OpenLayers, GeoExt, and ExtJs. You can download the additional files, GeoExt.js, is part of the download file at <http://www.geoext.org/> and extjs you can download from <http://www.extjs.com/>. For our examples, we will be using extjs version 3.3.1 and GeoExt 0.6.

For this first example, we will create a page that loads an open layer map into an Ext js window using GeoExt. We divide the code into two parts the htm file and the .js file.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

**Listing 11.9 geoextnewenglandwin.htm: Basic structure of page**

```

<html>
<head>
    <title>OpenStreetMap New England States</title>
-- 1
    <script
src="http://api.maps.yahoo.com/ajaxymap?v=3.0&appid=postgisus"></script>
    <script src="js/ol28/OpenLayers.js"></script>
    <script type="text/javascript" src="js/ext-3.1.1/adapter/ext/ext-
base.js"></script>
    <script type="text/javascript" src="js/ext-3.1.1/ext-all.js"></script>
    <script src="js/GeoExt.js"></script>
    <script
src="http://www.openstreetmap.org/openlayers/OpenStreetMap.js"></script>
    <link rel="stylesheet" type="text/css" href="js/ext-
3.1.1/resources/css/ext-all.css" />
-- 2
    <script type="text/javascript" src="geoext_newenglandwin.js"></script>
</head>
<body>
<!--If we were were using panels and view ports, the divs to hold them
would go here-->
</body>
</html>
-- 1 includes to js apis
-- 2 our geoext app as a js include

```

In (1) we define all the js dependency files we need, in this case the api for yahoo, openlayers, extjs, geoext, OpenStreetMap. (2) We then declare the js for our custom app. Note for (2) you could have just included all the javascript right on the page, but it is standard practice especially if a fair amount of javascript, to include it as a separate file.

The meat of the application is in the js which is shown below.

**Listing 11.10 geoextnewenglandwin.js: Using GeoExt to display OL Map in Ext Window**

```

var lat=43.66596;
var lon=-73.13868;
var zoom=6;
var map, lyrMapnik, lyryahoo, lonlat;
var prj4326 = new OpenLayers.Projection("EPSG:4326");
var prjmerc = new OpenLayers.Projection("EPSG:900913");
lyrMapnik = new OpenLayers.Layer.OSM.Mapnik("OSM Mapnik");

lyryahoo = new OpenLayers.Layer.Yahoo(
    "Yahoo Hybrid",
    {'type': YAHOO_MAP_HYB, 'sphericalMercator': true}
);
map = new OpenLayers.Map ( {
    controls:[ new OpenLayers.Control.Navigation(),
               new OpenLayers.Control.PanZoomBar(), new
OpenLayers.Control.LayerSwitcher()
    ],

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```
maxResolution: 156543.0399,
numZoomLevels: 20,
units: 'm',
projection: prjmerc,
displayProjection: prj4326
} );
lonlat = new OpenLayers.LonLat(lon, lat).transform(prj4326, prjmerc)
--1
Ext.onReady(function() {
--2
    new Ext.Window({
        title: "New England",
        height: 600,
        width: 600,
        closable: false,
        collapsible: true,
        items: [{
            xtype: "gx_mappanel",
            map: map,
            layers: [lyrMapnik, lyryahoo],
            zoom: zoom,
            extent: [-8879149, 4938750, -7453286, 6017794],
            center: lonlat
        }]
    }).show();
});
-- 1 initialize ExtJs
-- 2 create collapsible window with map
```

The result of our page is shown below. The benefit of putting an OL map in a window is that you can move the window around on the browser screen, resize it and minimize it. There are other controls you can use in GeoExt such as view port and panel. View ports allow for auto stretching within a div. These allow you to position the map along side grids, collapsible panels and other form controls and have the map interact with those.

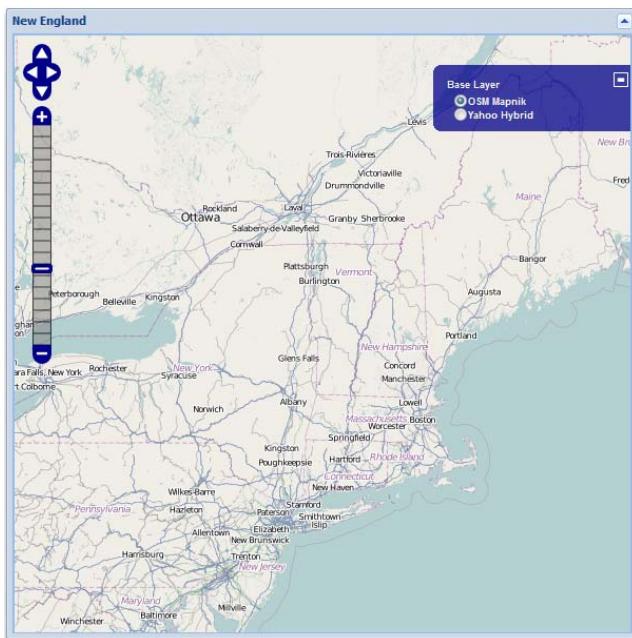


Figure 11.9 OpenLayers map in a ExtJS movable, stretchable, collapsible window using Listing 11.10

For our next example, we are going to create a PHP app that queries our database and outputs a GeoJSON layer using the PostGIS ST\_AsJSON function.

## 11.5 Displaying data with just server side web scripting

In this section, we will demonstrate three examples of using plain server-side web scripting without support of web mapping services. Although none of these use any of the OGC web mapping services, we hope it will be clear that you can mix and match these with standard web mapping services.

We shall demonstrate the following concepts

- Outputting layers with PostGIS ST\_As\* output functions
- Proximity queries with PostGIS geography

### 11.5.1 Loading custom layers with GeoExt

For this exercise, we will display a sortable grid using GeoExt where the grid data comes from a GeoJson datastream we create with PHP. This datastream can be created with any web scripting language such as Python, ASP.NET etc. When we are done, our page will look as shown:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

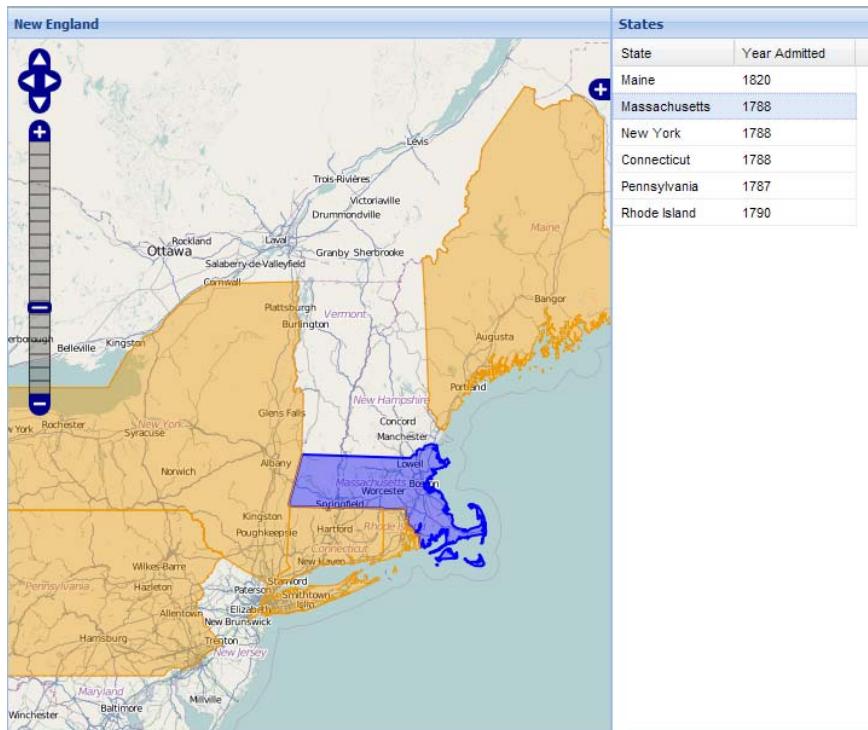


Figure 11.10 Application with feature grid in synch with map using table column layout using code in listing 11.10

To achieve the above application, we created a new .js file similar to our earlier, but with the following parts added

#### **Listing 11.10 Adding a feature grid window**

```
--1
lyrStates = new OpenLayers.Layer.Vector("States");
--2
dtstate = new GeoExt.data.FeatureStore({
    layer: lyrStates,
    fields: [
        {name: 'state_name', type: 'string'},
        {name: 'year_adm', type: 'string'}
    ],
    proxy: new GeoExt.data.ProtocolProxy({
        protocol: new OpenLayers.Protocol.HTTP({
            url: "datafeeder.php?format=json",
            format: new OpenLayers.Format.GeoJSON()
        })
    })
});
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```

        autoLoad: true
    });
-- 3
gridPanel = new Ext.grid.GridPanel({
    title: "States",
    store: dtstate,
    layout: 'fit',
    columns: [{
        header: "State",
        dataIndex: "state_name", sortable: true
    }, {
        header: "Year Admitted",
        dataIndex: "year_adm", sortable: true
    }],
    sm: new GeoExt.grid.FeatureSelectionModel()
});
-- 4
panel = new Ext.Panel({
    id:'main-panel',
    baseCls:'x-plain',
-- 5
    renderTo: Ext.getBody(),
    layout:'table',
    layoutConfig: {columns:2},
    defaults: {frame:true, height: 600},
    items:[{
        title: 'New England',
        xtype: "gx_mappanel",
        map: map,
        layers: [lyrMapnik, lyryahoo, lyrStates],
        zoom: zoom,
        extent: [-8879149, 4938750, -7453286, 6017794],
        center: lonlat,
        width: 500
    },gridPanel
    ]
});
-- 1 blank vector layer
-- 2 popular vector layer
-- 3 grid panel for attributes
-- 4 2 column mappanel and grid panel
-- 5 output to body of document

```

In (1) We define a new OpenLayers layer that will store features. (2) We create a GeoExt feature store object that pulls data from our custom defined php page and does this automatically. (3) Define a grid panel to display the attribute portion of the features, the select model determines what happens when an item is selected -- in this case, the feature gets highlighted on the map. (4) Instead of a movable window we are using a 2 column table layout with the first column holding our map (note use of gx\_mappanel) and second column holding our feature grid panel. This panel is the only panel that gets displayed because of the (5) renderTo: Ext.getBody() and contains the mappanel and gridPanel feature as child items.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

### 11.5.2 Using PostGIS output functions with PHP

For this exercise, we are going to use PHP and helper PHP libraries Smarty and PHP ADODB to build our datafeeder.php file. ADODB is a database abstraction layer used by many php applications to provide a generic interface to all dbs. Smarty is a templating engine for PHP that allows for separation of presentation from application logic. Both are free and open source with LGPL/BSD style licenses. Sites:

- PHP ADODB - <http://adodb.sourceforge.net/>
- Smarty - <http://www.smarty.net/download.php>

When we build applications with these, our general convention is to create a file called app.inc.php that includes all the includes we will need. Such as file looks something like this

```
<?php
include_once( "config.inc.php" );
include_once( "libs/adodb5/adodb.inc.php" );
include_once( "libs/smarty/Smarty.class.php" );
?>
```

We create a separate file to contain connection strings and so forth. Our config.inc.php for this app, looks like below which is a standard ADODB data url format.

```
<?php
define( "DSN" ,
'postgres://userhere:passwordhere@localhost:5432/dbhere?persist' );
?>
```

#### CREATING A DATAFEEDER IN PHP FOR OUR MAP

In the last example we introduced another file called datafeeder.php which is built using Smarty, ADODB and queries a PostGIS enabled PostgreSQL database. The format argument we pass determines the type of format. In our code we have defined a kml and a json output format that we output using Smarty templates. Using Smarty allows us to extend the number of layouts we support without cluttering our request control and data query logic.

- We use PHP ADODB for data abstraction. Note that things such as PHP PEAR are just as good. This keeps our data load logic short and also has the benefit of making it more generic. As you see, you wouldn't be able to tell this was a PostgreSQL database unless you saw the connection string and PostGIS function calls.
- The datafeeder.php acts as the controller -- reading the request and figuring out which layout and if applicable query to use to satisfy the request.
- Smarty template -- we define one for each format. Note that things like KML include styling options which makes using a generic KML handler such as OGR2OGR not always ideal. Using a template allows us to customize this and also inject database stored styles in the file.
- We are using PHP classes instead of standard PHP procedural. Since our class inherits

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

from smarty we can call all the functions built into the smarty class as if they were native.

The core pieces of the page are shown below:

### **Listing 11.11 Core pieces of datafeeder.php**

```

include_once("app.inc.php");
class _DataFeeder extends Smarty {
    private $db;
    private $supported_formats = array('json'=>'ST_AsGeoJSON', ...);
--1
    function __construct {
        $this->plugins_dir = array('plugins', 'extraplugins');
        $this->left_delimiter = '<!--(';
        $this->right_delimiter = ')-->';
        $this->db = &ADONewConnection(DSN);
        :
        $this->page_load();
    }
--2
    function page_load(){
        $data_template = 'data_json.tpl';
        if (!empty($_REQUEST['format'])) {
            $format = $_REQUEST['format'];
        }
--3
        $convertfunction = $this->supported_formats[$format];
        if ( empty($convertfunction) ){
            $convertfunction = 'ST_AsGeoJSON';
        }
        else {
            $data_template = "data_$format.tpl";
        }
--4
        $sql = "SELECT gid As id, ...
                $convertfunction(the_geom) As geom ...";
        $rsdata = $this->db->Execute($sql)->GetRows();
        $this->assign('rs', $rsdata);
        $this->display($data_template);
    }
}
--5
new _DataFeeder;
--1 constructor
--2 handle web request
--3 determine format
--4 display data in right format
--5 instantiate class

```

When a class is instantiated in PHP (5) the first function that gets called is the (1) `__construct` function which in our case sets up the database connection and also changes the smarty markup tag. Normally smarty using {..smarty code} which needs to be escaped out with a {literal}{/literal} smarty construct when you are writing javascript or JSON like templates. In (1) we also redefine the plugins to look in our extraplugins folder for  
©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

additional plugins. We replace it with above which makes it look like an html comment in html editors so we don't need to escape. (2) Then we call the page\_load() function to handle web request. You can stuff all that in the \_\_construct, but we break it out for clarity. In our page load (3) we lookup the passed in format in our associative array to pull the name of PostGIS convert functions. We then (4) build our SQL using that output function to out geom field, load this data into a PHP array using ADODB GetRows() function, assign it in our template and then display the merged data + template.

Our json smarty template looks as shown below:

#### **Listing 11.12 smarty data\_json.tpl file**

```
{
  "type": "FeatureCollection",
  "features": [
    --1
    <!--(section name=sec loop=$rs)-->
      { "type": "Feature", "properties":
        {"id":<!--($rs[sec].id|json_encode)-->
      --2
      <!--(foreach from=$rs[sec] key=prop item=val)-->
      <!--(if $prop != 'geom' && $prop != 'id')-->
        ,<!--($prop|json_encode)-->:<!--($val|json_encode)-->
      <!--(/if)-->
      <!--(/foreach)-->
        },
        "geometry": <!--($rs[sec].geom)-->
      }
    --3
    <!--(if not $smarty.section.sec.last)-->,<!--(/if)-->
    <!--(/section)-->
      ]
  }
  --1 loop rows
  --2 loop fields
  --3 , except for last record
}
```

(1) We use a smarty section tag to loop thru our php rs array. (2) We use a foreach to loop thru all the fields of each record and output the ones that are not id or geom, and then we output the geometry as a separate field. Note the json\_encode -- this is a smarty modifier we created that is just a wrapper around the php 5.2+ built in json\_encode function so it will convert our value into json safe text. (3) We need to separate each feature by a , except if its the last feature in our result set.

The smarty json\_encode smarty modifier we put in a file  
 libs/smarty/extraplugins/modifier.json\_encode.php and looks as shown below

```
<?php
function smarty_modifier_json_encode($string){
  return json_encode($string);
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```
?>
```

If we wanted our data also accessible from a KML viewer such as Google Earth, we would offer a KML format. This we would do by adding in a KML template. The data\_kml.tpl is shown below:

#### **Listing 11.13 KML template to format in KML format**

```
<?xml version='1.0' encoding='UTF-8'?>
<kml xmlns='http://earth.google.com/kml/2.1'>
<Document>
<Style id='defaultStyle'>
    <LineStyle><color>ff00ff00</color><width>3</width></LineStyle>
    <PolyStyle><color>5f00ff00</color><outline>1</outline></PolyStyle>
</Style>
<!--(section name=sec loop=$rs)-->
<Placemark>
    <name><!--($rs[sec].id|escape:html)--></name>
    <description>
        <!--(foreach from=$rs[sec] key=prop item=val)-->
        <!--(if $prop != 'geom' && $prop != 'id')-->
        <b><!--($prop|escape:html)--></b> <!--($val|escape:html)--><br />
        <!--(/if)-->
    <!--(/foreach)-->
    </description>
    <styleUrl>#defaultStyle</styleUrl>
    <!--($rs[sec].geom)-->
</Placemark>
<!--(/section)-->
</Document>
</kml>
```

In this listing, we do more or less the same as we did in our json format template except we use escape:html smarty modifier to make the fields kml friendly. The escape modifier is included with Smarty download.

Then from Google Earth, we could create a network link:

[http://www.postgis.us/demos/chapter\\_11/datafeeder.php?format=kml](http://www.postgis.us/demos/chapter_11/datafeeder.php?format=kml)

Which looks as shown:

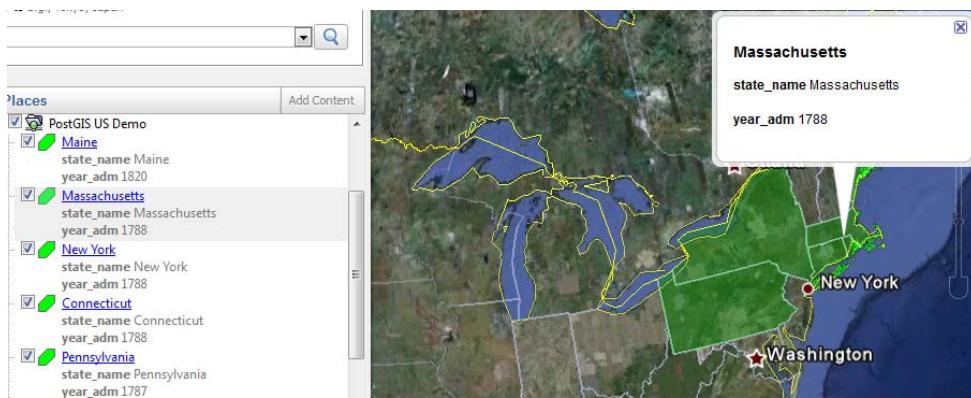


Figure 11.11 Displaying KML layer in Google Earth using template in Listing 11.13

### 11.5.3 Proximity queries with PostGIS geography

For many use cases, you don't care about maps. You just care about what data elements are of x distance from where you are or where you want to go. In this case, the PostGIS geography data type introduced in PostGIS 1.5 is probably the simplest way to achieve that goal.

Below is a simple PHP snippet of code that demonstrates the concept using ADODB or some other database abstraction layer and smarty:

#### **Listing 11.14: PHP find all roads within 1 mile of a requested lon lat**

```
-- 1
if(!empty($_REQUEST['lat']) && !empty($_REQUEST['lon'])) {
    $lat=$_REQUEST['lat'];
    $lon=$_REQUEST['lon'];
    $range = 2;//in miles
    if ( is_numeric($lat) && is_numeric($lon) ){
-- 2
        $pt = "ST_GeogFromText('SRID=4326;POINT($lon $lat)')";
-- 3
        $sql =
            "SELECT full_name, lfromadd, ltoadd
             FROM roads
            WHERE ST_DWithin(roads.geog, $pt,1609*$range)
        ORDER BY ST_Distance(roads.geog, $pt) ";
-- 4
        $rs = $this->db->Execute($sql)->GetRows();
        $this->assign('rs', $rs);
    }
-- 1 check inputs
-- 2 build point from lon lat
-- 3 build sql near neighbor
-- 4 return results
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

(1) We verify that input are passed in via GET or POST request. Note if you wanted to limit to just post variables use `$_POST[...]` To prevent SQL injection, verify the inputs are numeric. We could also validate for range to make sure it fits in spheroid range -180 to 180 etc. (2) Convert our lon lat to a PostGIS geography POINT expression (3) Build the SQL statement. Since geography is always in meters we multiple by `1609*$range` to convert our miles to meters to use in our ST\_DWithin. We also use ST\_Distance to sort the results by proximity. (4) Execute the statement and dump into a PHP array and then assign a smarty variable that will be used in a section loop similar to what we did in earlier examples.

## **11.6 Summary**

First, there is no shame to using proprietary packages such as GoogleMaps or Bing if the data you need to deliver does not exceed their limitations and your usage complies with their licensing terms. Face it; you will never be able to re-create the jaw-dropping, uber-cool features that proprietary packages now includes, nor as a GIS expert is it your job to do so.

If you cannot or choose not to take advantage of proprietary packages, you need to worry about the web experience both at the server level and the client level. On the server side, you need to inject a mapping server between your data source and your existing web server. We covered MapServer and GeoServer in detail. MapServer has a lot of power and is more portable than Geoserver, but you must learn how to compose map files in order to work with it. GeoServer comes with a nice interface, but as with most screen-driven software, you face some limitations once you start to outgrow what is offered up in the UI.

Though GIS is a data-centric pursuit, you cannot ignore the clients' browser experience when it comes to delivery of GIS data. The web consumer is no longer satisfied by the mere display of a map on their web browsers. They want zooming, panning, ability to tag, edit, layer, etc. To give users what they want, we recommend that you use OpenLayers. Its javascript base makes it more portable than anything else and it is the dominant client tool in the market today. We also suggest that you enhance OpenLayers with the new GeoExt framework for a web browsing experience that could rival many proprietary packages available today.

If sharing data via the world wide web is not your thing, in the next chapter, we cover desktop tools that connect easily with PostGIS. Many of the desktop tools can consume standard OGC web services that we described in this chapter, such as WMS or WFS. This means that even if you do not intend to share data via the web, setting up a mapping server may still be a good idea since many desktop tools can also take advantage of the data it serves up even if they don't have direct support for PostGIS.

# 12

## *Using PostGIS in a Desktop Environment*

This chapter covers

- OpenJump
- QuantumGIS
- uDig
- GvSig

In this chapter, we will cover some of the popular open source GIS desktop viewing tools that work with PostGIS. Just like proprietary software, you will find that each has its own strengths and weaknesses and caters to a certain niche of users or tasks. In this chapter we will start off by providing a brief at-a-glance summary of these tools, liberally ladling out our personal opinions. Once you have completed this chapter, we hope you will have a better understanding of which tools are best for what you are doing and your particular style of use. We will focus mostly on the use of these tools as for viewing and querying data, but will also highlight the features that each has for building custom desktop applications or extending the feature set via plug-ins and scripting.

### **12.1 At a glance**

For those of you who do not wish to unnecessarily delve into the details of each tool, we start this chapter with a quick summary of features. After reading this one section, you may be able to rule out some of the tools altogether and can thereby skip the sections that pertain to them. For those of you already invested in one of the tools, we recommend that you go through this section to at least see what you could be missing out. New features are

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

being added to the tools quicker than any book can keep up with. If you have dismissed a tool due to lack of some critical feature a year ago, you may find them now incorporated. Table 12.1 provides a quick overview of the four tools that we will be covering in this chapter.

**Table 12.1 Summary of tools based on Architecture, Language, OS, Setup**

Feature	OpenJump	QGIS	uDig	gvSig
<b>Current Version</b>	1.3.1	1.4.0	1.1.1	1.9
<b>JVM</b>	1.4+	N/A	1.5+/JAI	1.5+/JAI
<b>Plug-in</b>	jars/Jython/beans	Python/Qt	Eclipse	Jars
<b>Scripting</b>	Jython/Beanshell	Python	No	Jython (1)
<b>Download Size</b>	11MB	30MB	100MB	70MB
<b>Extract and Go</b>	Yes	No	No	No
<b>Ease of Setup (4)</b>	Easy	Moderate	Moderate	Tricky
<b>Ease of Use</b>	Easy	Easy	Moderate	Difficult
<b>Mobile (5)</b>	No	No	No	Yes (0.2)

(1) Jython is the java framework that allows you to run Python code in a JVM. (2) Java Advanced Imaging API. (3) JUMP unified mapping platform is the platform for OpenJUMP, but some other applications use it as a framework including the namesake desktop app JUMP and the more CAD focused SkyJump. (4) How easy to get up and running after performing basic configurations. (5) Does it have/claim to have a mobile companion version that can run on mobile OSes.

### **12.1.1 Capsule review**

We have used all four tools in various capacities and have communicated with other users of the various tools. In this section, we offer our opinion of each. This is of course subjective so YMMV.

#### **OPENJUMP**

This is our favorite tool because it is light weight, lets us write raw spatial SQL and immediately view the visual results. OpenJump also has nice features for correcting and analyzing geometries as well as tools to fix up faulty shapefiles. It is probably best suited to people who are not afraid of querying directly against the database and do not like cluttered workspaces. For Java, Python/Jython programmers, OpenJump easily automates common workflows. On the downside, we really wish that OpenJump would better support non-PostGIS spatial databases, such as Oracle, SQL Server 2008, and SpatiaLite (the flowering little sister of PostGIS).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

**QUANTUMGIS ("QGIS")**

New GIS users tend to gravitate towards QGIS for its user-friendly interface, GPS and raster support, Python scriptability, and stability. GIS data crowd sourcers, Python programmers, and GRASS users also tend to choose QGIS. Its speed and spatial SQL capabilities are fairly decent and is the only one of these tools besides OpenJump with support for SpatiaLite SQLite extender. The SpatiaLite support is built-in whereas for OpenJump its available via a fairly recently added plugin. QGIS provides a user-friendly simple query interface but no facility to write full SQL statements, so may be a bit of a disappointment to DB programmers.

**uDIG**

uDIG tries to do too much on the workspace and not enough on fundamental database operations. Its strength is in providing a rich suite of OGC web services and cartographic niceties. It caters to a Java audience with heavy emphasis on offering cartography niceties. Eclipse programmers might find it just what they are looking for. As of this time uDIG has no Jython/Python scripting framework though plans are in the works. As such if you are a python programmer, you will probably be disappointed with it.

**GVSIG**

GvSig has a lot of basic support for various databases, OGC services, and non-OGC products such as ESRI ArcIMS. It is also extensible via Java. If you are invested in ESRI and looking for something to tie into your legacy ESRI stack, this may be the best choice for you.

### **12.1.2 Spatial Database Support**

It goes without saying that all these 4 desktop tools are free and support PostGIS out of the box in some shape or form. For the PostGIS side of things, we'll break that out a bit into specific PostGIS features and test these products against PostGIS. For other spatial databases you may come across, we'll provide a simple Yes/No purely based on if we see said database on the menu or the documentation claims it does. We'll consider it a Yes\* if its not part of the core download, but you can download it as a separate extension.

The summary below details the depth of spatial support. Below are list of terms we use that may not be clear.

- Multi Geo Column - Can the desktop tool handle PostGIS tables that have more than one geometry/geography column or does it either randomly pick one or choke.
- gcolumns optional -- Can you view tables that aren't registered in geometry\_columns table. Note that we aren't going to consider if it uses it or not here, but will note it in the section about that tool.
- \* next to a Yes means it supports it but via a plugin you download separately or you need to download the database drivers to enable the additional support. A No\* means it sort of doesn't, but only under certain modes or there is an easy workaround.
- geography - does it support the geography data type

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

- Sql queries - means you can write fully qualified SQL queries and see the visual output of them.
- By heterogeneous column we mean the software is able to deal with rendering a table that has an unconstrained geometry type (has a mixed bag of geometry types).
- Integer unique key required -- does the software require you to have a primary or unique key that is an integer in order to render geometries in the table.
- Save PostGIS means ability to save as a new PostGIS table
- Edit PostGIS means ability to load a PostGIS layer and edit the attributes and geometry visually.

### **SQL SERVER 2008 SUPPORT**

As of this writing, none of these tools support SQL Server 2008. We expect that one or all of these tools will support SQL Server 2008 very shortly. There is some amount of support in the .NET framework GIS Open Source with such things as SharpMap, NTS Topology Suite, and in SQL Server 2008 RS Reporting Services, but these are more SDK tools than desktop tools you can use out of the box.

**Table 12.2 Spatial Database Support**

Feature	OpenJump	QGIS	uDig	gvSig
Oracle Spatial	Yes*	Yes*	Yes	Yes*
DB2	No	No	Yes	No
ArcSDE	Yes*	No	Yes	Yes
MySQL	Yes*	Yes	Yes	Yes
Multi Geo Column	Yes	Yes	No*	Yes
geography	Yes*	No	No	No
Read PostGIS	Yes	Yes	Yes	Yes
Save PostGIS	Yes*	Yes*	No	Yes
Edit PostGIS	No	Yes	Yes*	Yes
Curve support	No	No	No*	No
3D geometry	No	No	No	Yes*
Heterogeneous column	Yes	Yes	No*	No
SQL queries	Yes	No	No	No
integer unique key required	No	Yes	No	No

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

Views	Yes	Yes*	Yes	Yes*
-------	-----	------	-----	------

### 12.1.3 Format support

In this section we will cover the various vector, raster, and web service formats supported by each. Note that this list is not comprehensive but tries to cover the more common formats people look for in a desktop tool.

- Tab is default MapInfo format
- MIF/MID are MapInfo interchange formats that MapInfo can export to and maintain most of the functionality of the default Tab format
- Yes means they support it either as an Import/Export/Edit or all.
- SpatiaLite is the spatial database extender for SQLite that also uses GEOS similar to PostGIS for spatial function support. Think of SpatiaLite as a light-weight single file PostGIS.
- ESRI Personal Geodatabase is the old geodatabase format made by ESRI which is an extension of the MS Access database format. This is not to be confused with their new non-published file storage format which no open source software to our knowledge currently supports. We only know of ESRI ArcGIS and possibly Safe FME as a commercial tool that supports this newer fie storage format.

Table 12.3 Vector File Data Formats

Format	OpenJump	QGIS	uDig	gvSig
ESRI Shape	Yes	Yes	Yes	Yes
SpatiaLite	Yes*	Yes	No	No
ESRI Personal Geo (MDB)	No	Yes	No	No
GPX	Yes	Yes	Yes*	No
GML	Yes	Yes	Yes	Yes
KML	Yes*	Yes	Yes	Yes
WKT	Yes	No	No	No
DXF	Yes*	No*	No	Yes
DWG	No	No	No	Yes
MIF/MID	Yes	Yes	No	No
TAB	No*	Yes	No	No
Excel	Yes	Yes	No	No
CSV	Yes	?	No	IX

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

SVG	Yes	No	No	No
-----	-----	----	----	----

This table lists the various raster formats supported by these tools, we didn't really investigate the editing and exporting capabilities of these tools, so a Yes means it can render such format or export it.

**Table 12.4 Raster File Data Formats**

Format	OpenJump	QGIS	uDig	gvSig
JPG	Yes	Yes	Yes	Yes
TIFF	Yes	Yes	Yes	Yes
ECW	Yes*	Yes	No	No
PNG	Yes	Yes	No	Yes
MrSID	Yes*	Yes	No	No

#### **12.1.4 Web services supported**

In this section we list the common OGC web services formats and the support each has for each. Below is a brief description of what these different web services are designed for. We didn't test any of these so this is purely based on literature of each or if we saw it on the menu.

- WMS - Web Mapping Service -- this is the oldest and most common. This allows for requesting image data on layer names and bounding regions using the GetMap method. It also has a simple GetFeatureInfo call which allows for returning already formatted text information.
- WFS - Web Feature Service -- web service that generally returns vector formatted data based on web query. The standard format is Geography Markup Language (GML). There do exist some that return other formats such as KML and GeoJSON.
- WFS-T - Web Feature Service Transactional -- this is an extension of the standard WFS protocol that allows for editing geometries across the web via vector data posts such as GML or WKT.
- WPS - Web Processing Service - OGC GIS web service protocol for exposing generic work processes. Key parts DescribeProcess, GetCapabilities, Execute (execute takes a named process with arguments and executes it).
- WCS - Web Coverage Service - OGC GIS web service protocol for supporting grid coverages and the like.
- ArcIMS - is a proprietary ESRI web mapping service framework. It has been superceded by AGS, but many sites still maintain ArcIMS webservices.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

**Table 12.5 Webservices Support**

<b>Format</b>	<b>OpenJump</b>	<b>QGIS</b>	<b>uDig</b>	<b>gvSig</b>
WMS	Yes*	Yes	Yes	Yes
WFS	Yes*	Yes	Yes	Yes
WFS-T	Yes*	No	Yes	No
WPS	Yes*	No	Yes	No*
ArclIMS	Yes*	No	No	Yes
WCS	No	No	No	Yes

Now that we have a basic sense of what each supports, we will take each for a test drive.

Before we start jumping into the various desktops, we would like to note, that for all these tools, to get the full extent of a layer (in PostGIS terminology a geometry column), you right mouse click the layer and choose "Zoom to Layer". This is pretty consistent across all these.

## **12.2 OpenJump Workbench**

OpenJump is a Java-based, cross-platform open source GIS analysis and query tool. It is fairly rich in functionality for statistical analysis and geometry processing. It works well with both ESRI Shapefiles, PostGIS datastores, and many other formats. We find it to be the best open source tool for doing ad-hoc spatial queries on PostGIS-enabled databases. Its main focus is spatial analysis, geometry processing. While its cartography offering is adequate, its nothing to write home about. As such its lightning fast for geometry processing tasks such as aggregation, simplification, but you will find it to be somewhat clunky in cartography tasks such as printing.

The spatial engine driving OpenJump is the Java Topology Suite (JTS). JTS is the Java parent of Geometry Open source (GEOS), which PostGIS is based on. Since JTS is usually a few of versions ahead of what GEOS offers, you will find that much of the terminology and spatial process naming is similar to what you get in PostGIS and many new features will appear in OpenJump before they become available in PostGIS.

In the sections that follow we shall outline its strengths, explain how to set it up, detail the more useful plug-ins it has for PostGIS, and demonstrate some example uses.

### **12.2.1 Feature summary**

OpenJump Workbench is our tool of choice for basic PostGIS desktop analysis. Most of the figures of geometries in this book were rendered with OpenJump. It has the smallest download size of all the tools we cover in this chapter. Its analytical tools for processing geometries (unioning, fixing, stats) are the easiest and fastest to use. It is probably a good

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

try for those in love with Python since it does support a Jython scripting/plugin framework for injecting Python logic into the workbench.

The thing we love most about it is its Ad-hoc query tool. This allows you to write a fully formed SQL statement and render it. This is a feature that the other tools in our discussion lack. The other tools may allow you to pick tables and limit outputs with WHERE field conditions, but that is the extent of the complexity of SQL they allow you to directly write. OpenJump allows you to do both. Hopefully this functionality will appear later in other tools.

#### **INSTALLATION**

Installation is a breeze. All you have to do is extract the zip file and then launch the application executable. It unlike uDig and gvSig which also require a JVM, does not come packaged with its own JVM. As such the download is much lighter weight, but you must have a Java JVM installed already for it to work. You can get your copy at the website <http://www.openjump.org/> and find out more details about it.

#### **EASE OF USE**

We ranked it second in Ease of Use - because its add tables feature is more quirky than that of QGIS, doesn't have transform support out of the box, and the ad-hoc tool requires you to do an ST\_AsBinary on the geometry/geography column for it to render. The upside, you can use the ad-hoc tool for geography columns as well since geography also has an ST\_AsBinary function.

#### **SkyJump is cool too**

A slightly less popular Jump, called SkyJump is also actively worked on and its main focus is on CAD and Printing. You can get it from <http://sourceforge.net/projects/skyjump/> . SkyJump can do export to PDF among other things that OpenJump lacks, has integration with OGR2OGR which it comes packaged with. This means it pretty much supports all the different data types QGIS supports in addition to what Jump normally supports. It also has a slightly slicker user interface (prettier icons more right-click menus) than OpenJump. It's a bit heftier (50MB download installer file) than OpenJump mostly because the installer packages its own JVM. Some other features that SkyJump touts out of the box is a connector for ESRI SDE and export/import to DXF. It seems most focused on the windows user, since its only setup is a windows executable. The exercises we will discuss here, should pretty much work in SkyJump as well.

#### **PLUGINS**

OpenJump supports plugins, extensions, and registries. A plugin is a Java archive file (JAR) that you just drop in the lib or lib/ext directory of your OpenJump install. The plugins could be database drivers, geometry functions etc. An extension is something that manages a set of plugins to accomplish a certain workflow and manages the install and configuration of plugins. It can be packaged in the form of a jar file or can be a python or beanshell. These go in the lib/ext folder of your OpenJump install. A registry is more vague and is a dictionary of what is available that an extension reports to.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

**SCRIPTING**

In addition to Java jars, one can add functionality to OpenJump using bean shell scripting and Jython scripting. These scripts and Python classes are kept in the lib/ext folder of your OpenJump install. You will see a folder for BeanTools and for Python one called jython to throw Python scripts.

**FORMAT SUPPORT**

To load a supported vector file you use the Load Dataset option by right clicking on the workspace or using the File->Open menu. To load a Raster file, you need to use the File -> Open File menu option. To load a spatial database layer, you use the Load Data Store or Ad Hoc Query tool. OpenJump out of the box, supports loading the following vector layers: GML, JML, ESRI ShapeFile, WKT, and PostGIS. It can also save to SVG,PNG,GIF,TIFF, ESRI Shapefile, and GML out of the box. It supports loading and saving to the following Raster formats -- GIF,TIFF,JPG, PNG. With additional plugins you download separately, it can support MrSID, MIF, ArcSDE, Oracle, GPX, and XLS. With an extra plugin, it can also save workspace layers to a new PostGIS table.

The GPS layer import plugin can be found at <http://redefinedhorizons.com/resources.html> and is listed as Geotools GPX2 library and OpenJUMP plug-in.

**PostGIS SUPPORT**

- **heterogeneous column** - OpenJump is capable of rendering a heterogeneous column of geometries in Add Data Store mode as well as Ad Hoc query mode and treats it like a single layer.
- **sql queries** - OpenJump via its Layer ->Run Datastore query, allows you to type in free-hand full SQL statements and view them. This works for both geometry and geography. The only caveat is that you need to wrap an ST\_AsBinary around the geometry/geography column.

**12.2.2 Register data source**

OpenJump 1.3.1 comes packaged with a PostGIS 1.0 and PostgreSQL 8.3 JDBC driver which work fine in most cases even against a PostgreSQL 8.4 / PostGIS 1.5 database. If you insist on the latest and greatest, you can swap out the old drivers with the PostGIS 1.5 and PostgreSQL 8.4 drivers with these simple steps:

- Download PostGIS' latest JDBC .jar snapshot from <http://www.postgis.org/download>.
- Copy the PostGIS JDBC into your OpenJump/lib folder while removing postgis\_1\_0\_0.jar.
- Download the latest PostgreSQL JDBC3 driver from <http://jdbc.postgresql.org/download.html>.
- Copy the PostgreSQL JDBC driver into your OpenJump lib folder and delete the 8.3 version.

OpenJump maintains a list of data sources to which you can connect to. You must register these prior to using them. You register a data source using the OpenJump connection manager. We will demonstrate one way how to connect to a PostGIS data source. You can get to the connection manager via the Layer Menu option and then selecting Run Data Store Query. Clicking on the database icon next to the Connection drop down list (Figure 12.1) brings you to the Connection Manager pop-up (Figure 12.2).

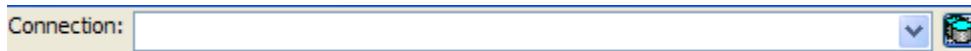


Figure 12.1 OpenJump Dropdown for Connection and Link to add a connection

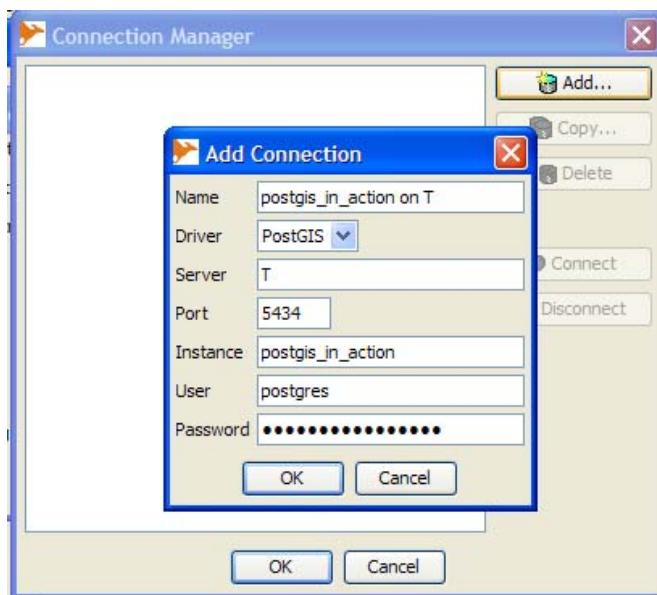


Figure 12.2 Adding a new PostGIS database connection

Watch out for the “Oracle-ish” term “instance.” This is where you enter the name of the database you wish to connect to. Once you have successfully added the connection, you should see a new item in your Connection Manager list with a green dot in the front (Figure 12.3). Should you end up with a red dot, delete the connection and try again. OpenJump does not have an edit option.

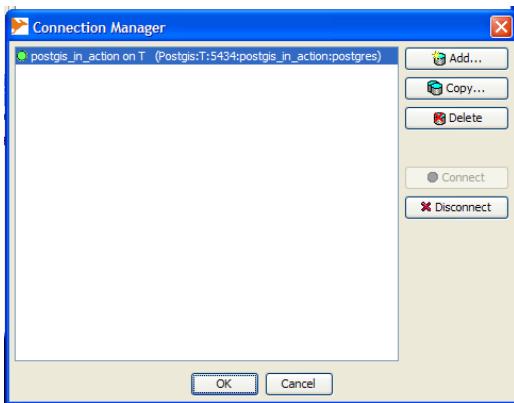


Figure 12.3 OpenJump Connection Manager with new connection

### 12.2.3 Rendering PostGIS geometry data

The Add DataStore Layer dialog is the quickest way to visually render data stored in an existing geometry column. To use it, right click Working and choose Add Datastore Layer (Figure 12.4).

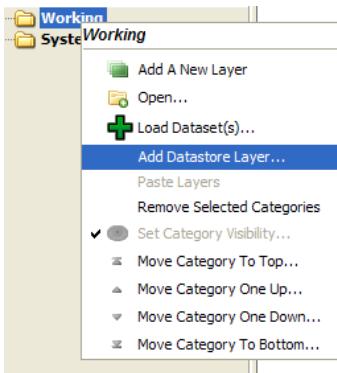


Figure 12.4 Add PostGIS table in OpenJump

Pick a connection, and then select a table and the geometry column you want to display. You can filter the data with an optional SQL where clause (Figure 12.5).

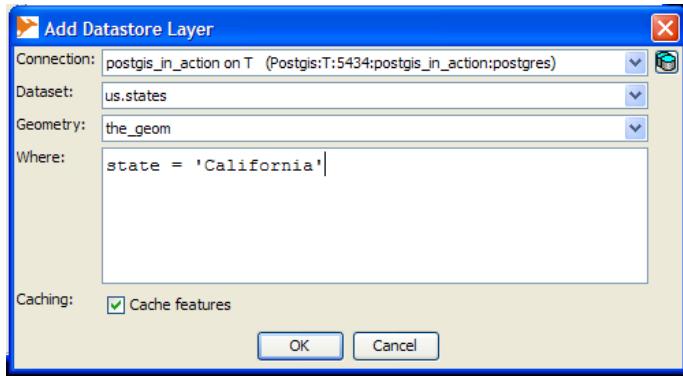


Figure 12.5 Datastore Layer setup in OpenJump

If nothing appears on the main display window after clicking the OK button, select the Layer and choose Zoom to Layer. The OpenJump 1.3 is a bit finicky in that it considers the extent of the table as the extent of the layer even when a where clause has shrunk the extent.

We spend most our time in OpenJump rendering ad hoc queries and applying theme. To display the result of an ad hoc query, follow these steps:

1. From the Layer Menu option select Run Data Source Query.
2. Type in your SQL but make sure to apply the ST\_AsBinary function for the geography or geometry column. You are free to use any SQL statement as well as accessing custom objects you have created in the database you are connected to.

Below is an artistic example to demonstrate

#### **Listing 12.1 SQL Art**

```

SELECT art.n, ST_AsBinary(art.geom) As coolbi
FROM
(SELECT n, ST_Translate(ST_Buffer(ST_MakeLine(pt), mod(n,6) + 2, 'endcap='
|| endcaps[mod(n,3) + 1] || 'join=' || joins[mod(n, array_upper(joins,1)) +
1] || 'quad_segs=' || n), n*10,n*random()*pi()) As geom
FROM
(SELECT ceiling(random()*100)::integer As n, ARRAY['square', 'round',
'flat'] As endcaps, ARRAY['round','mitre','bevel'] As joins,
ST_Point(x*random(),y*random()) As pt
FROM generate_series(1,200, 7) As x CROSS JOIN generate_series(1,500,20) As
y
) As foo
GROUP BY foo.n, foo.endcaps, foo.joins
HAVING COUNT(foo.n) > 10) As art;

```

The result of the query changes each time you run it. To apply themeing in OpenJump, Right mouse click on a layer and choose Change Styles then Enable Color Theming. Below is the output of one run of the query after applying themeing.

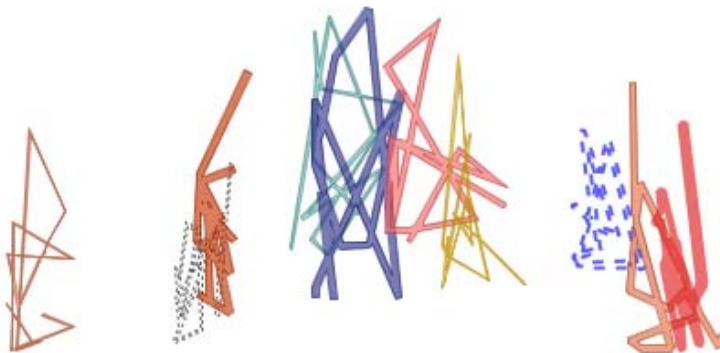


Figure 12.6 Output of SQL art query after applying custom themeing

#### **12.2.4 Exporting**

OpenJump comes packaged with basic load and export functions that allow you to save to files in ESRI, GML, WKT, raster, and SVG formats. By installing additional plugins, you can export to AutoCad DXF and print to PDF. Visit <http://sourceforge.net/projects/jump-pilot/files> for the plugins. To export a layer to ESRI, GML, WKT, right click on the layer and choose Save Dataset. To save the current view to raster or SVG, choose Save As from the File menu option.

You can also export the data within a layer to PostGIS. You may do this after using OpenJump to import data from a non-PostGIS data source or you might have edited the data you pulled from a PostGIS table. You will need to install an additional plugin called PostGISPlugin. Visit the URL referenced above. Download and copy the PostGISPlugin131.jar to the lib/ext folder of your OpenJump install. Reopen OpenJump and you should see another option called PostGIS Table when saving datasets. You may run into an invalid SRID error when saving to PostGIS. To work around this, change the SRID in OpenJump by going to the Layer menu option and choosing Change SRID. Set it to -1 or some other valid PostGIS SRID or, best yet, the SRID of the dataset. There are a couple other things you should be mindful of when saving to a PostGIS table. You should use lower case names. If you want to save to a schema other than public, prefix the table name with the schema. So for example to save to the hydro schema you would name the table hydro.rivers. In earlier versions of the plugin, saving to non-public schema was not possible.

### **12.2.5 Summary**

In this section, we have provided you a taste of what OpenJump offers. We encourage you to explore the plugins available for it to enhance your experience. You will find plugins that enable WFS, ArcSDE, JGrass, printing and export to several other formats. OpenJump also has a Jython scripting environment that allows you to make custom plugins for your specific needs. We encourage you to explore all these features. In the next section, we'll take a look at another desktop tool called QuantumGIS.

## **12.3 QuantumGIS**

QuantumGIS (“QGIS”) is a free desktop GIS viewing, editing, analysis tool. It is particularly popular among GIS novices, Python programmers, and GRASS users. Among the tools we are covering, it has the best GRASS and Python support. It is also the only one that is not based on Java. Instead, it is built upon the QT framework, a C/C++ cross-platform windowing framework.

### **12.3.1 Feature summary**

What makes it stand out the most from the other tools we are discussing is its high integration with GRASS, extensive support for RASTER analysis, integration with OGR/GDAL family suite, and its hard-core python scripting framework. Finally, what makes QGIS so appealing is its user-friendly interface. With the other tools, we have encountered places where we needed to second-guess the UI. With QGIS, everything is nicely organized and there is no need to doubt whether we are missing a key feature simply because we are unfamiliar with its navigation.

#### **INSTALLATION**

Installation was a breeze. Since QGIS does not rely on Java, you can skip the step of downloading Java Runtime. Get QGIS from this link: <http://www.qgis.org/en/download/current-software.html>. QGIS also provides a LTS (Long Term Support Edition) to appease those more apprehensive about the rapid development pace associated with open source software. The LTS version does not update as frequently as the current editions, but provides a level of comfort for those who have to contend with day-to-day support of installed software.

QGIS is also packaged into OSGEO4W. OSGEO4W is an installer that can quickly install a full suite of GIS-related tools. You can find OSGEO4W here: <http://trac.osgeo.org/osgeo4w>.

#### **EASE OF USE**

We ranked it first in Ease of Use - because its add tables feature also sports buttons for adding conditions (ala MapInfo style) and it lists the other fields in the table allowing you to pop them in. Its table viewing and attribute editing are also the nicest of the bunch. It has sort capability and zoom to location in map of the selected row. Direct editing of attribute data. One pet peeve we have is that it requires a table to have an integer unique/primary key to allow loading or editing. It does not support character primary keys. Most common

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

things are so brain-dead intuitive that you generally can get away without reading any of the 200 page manual to get the basics out of it.

### **PostGIS SUPPORT**

QGIS matured alongside PostGIS. As such its spatial database support for PostGIS has been time-tested over any of the other spatial databases supported by QGIS.

- **Heterogeneous and multiple columns** - It can also handle displaying with a heterogeneous column of geometries in one column. It presents these as separate layers in the connection list. It can also handle multiple geometry columns in a table and displays them as separate layers similar to what it does with the heterogeneous column.
- **Ad hoc queries** - It has no mechanism (or at least we couldn't find it), for writing free-wheeling sql like OpenJump sports. While the build query tool is nice and inviting, it doesn't let you do more advanced sql like aggregates and CTEs. A workaround for this is that you could create a view with the sql you want and render that assuming you expose something that looks like a serial it can latch on to.
- **PostGIS direct edit** - we found the editing of PostGIS geometries and attributes easiest to use in QGIS. However it doesn't seem to have the functionality to save as a new table as you can with other tools such as OpenJump.

#### **12.3.2 Adding a PostGIS connection**

Adding a PostGIS connection in QGIS is easy and intuitive. Most everything you need can be found under the Layers menu. The connection screen gives you the option of only searching geometry\_columns table. A geometry\_columns only search is a faster search than searching both.

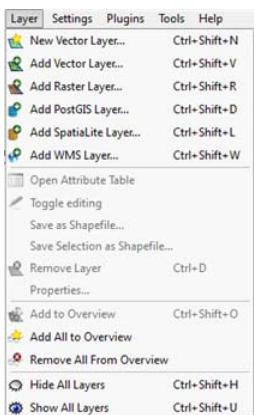


Figure 12.7 Adding a Layer and PostGIS Connection

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

The geometry need not be in geometry\_columns to be listed. For rows that contain multiple kinds of geometry types, it shows each as a separate row. Here we see QGIS PostGIS connection screen.

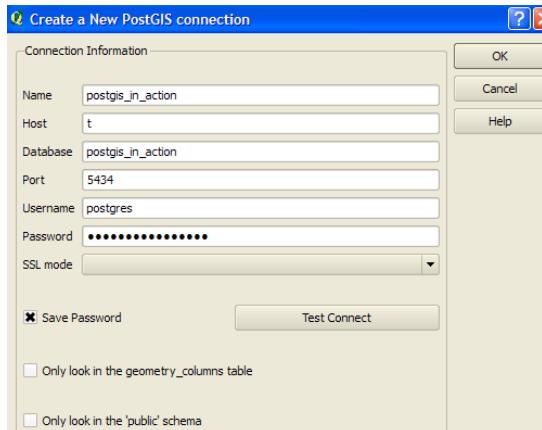
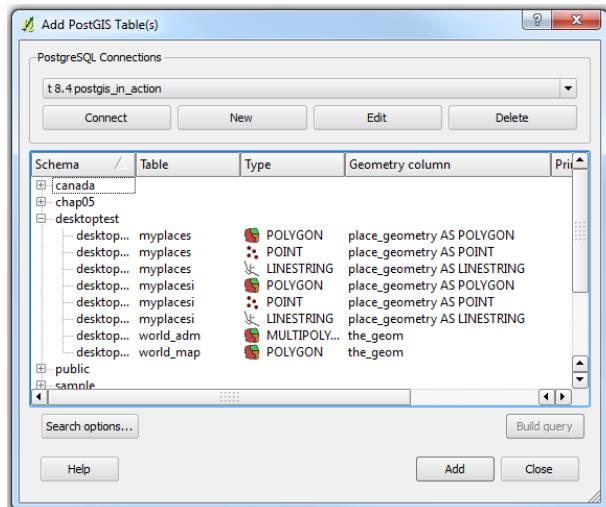


Figure 12.8 QGIS Add/Edit Connection -- allows you to specify search in geometry\_columns only

### 12.3.3 Viewing and filtering PostGIS data

QGIS equivalent of OpenJump Add Data Store Layer displays the type as an icon, unlike OpenJump's plain undecorated dropdown lists. If a geometry field is composed of multiple kinds of geometries, it lists each type as a separate layer as shown.



© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

Figure 12.9 QGIS PostGIS Connect -- tables and schemas - myplaces.place\_geometry is a bag of different kinds of geometry types. Each type shows as a separate layer option.

If you select a layer - you can filter the number of records and fields returned with the intuitive "Build Query", again ala MapInfo style layout. You can multi-select more than one layer at a time and when you click "Add", all layers with their filters will be added to the map view.

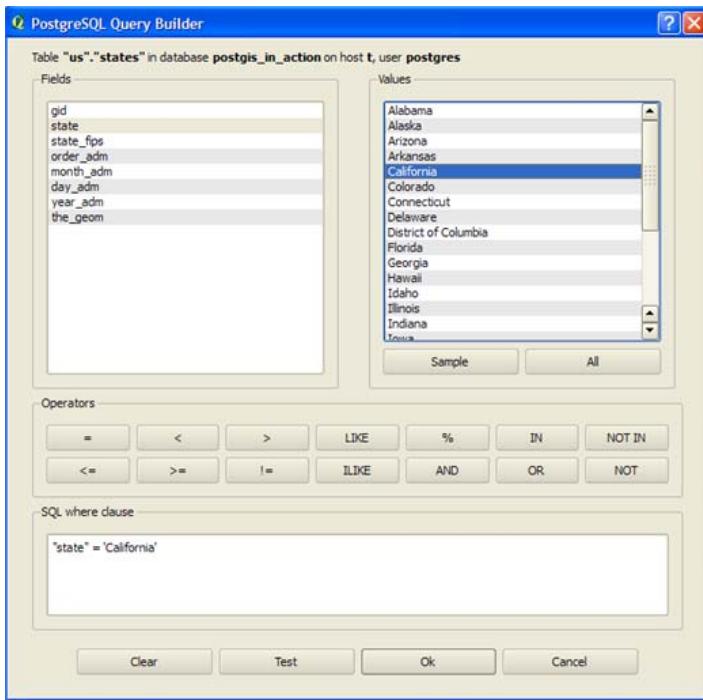


Figure 12.10 QGIS Build Query allows you to sample field data and double-click to drop in the WHERE window

The QGIS Build Query is launched when you click the "Build Query" button. In this particular case we selected the us.states from the US schema as we had done in OpenJump. QGIS lays out the fields in a list and selecting one and clicking the "Sample" or "All" allows you to get distinct values from that selected field. You can then position your cursor in the "SQL where clause" window and double-click on a value or field to plop it into the cursor position. Click OK and then Click "Add" from the layer view to render that layer.

Sadly QGIS in all its showering of love, did not allow us the one thing we most cherished "Writing a complete spatial SQL statement and seeing it rendered in bright beautiful colors

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

we could selectively theme" like OpenJump did. We also found its labeling and theming features much less intuitive than OpenJump.

#### **12.3.4 Connecting with other spatial databases**

As we saw from the menu, QGIS comes prepackaged with support for a database type called SpatiaLite. SpatiaLite is a spatial extender for the SQLite lite-weight database similar in concept to how PostGIS extends PostgreSQL. It is the first of the tools we are discussing that has support for SpatiaLite. OpenJump just recently added support via a plugin.

QGIS has support for other spatial databases. For Oracle Spatial it has support for both the OGC SFSQL for vector support as well as Oracle Spatial GeoRaster.

QGIS also has MySQL support out of the box and to load a MySQL layer choose from menu Layer -> Add vector Layer -> Database -> MySQL

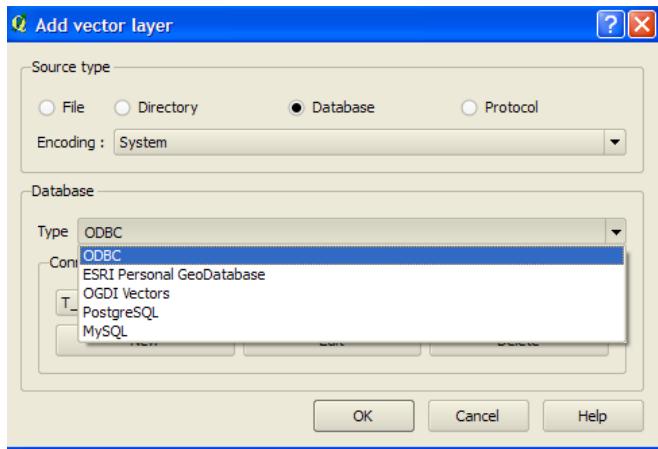


Figure 12.11 QGIS add database vector layer

#### **12.3.5 Loading other vector and raster layers**

Loading layers is probably QGIS's strongest point, the number of types available is pretty mind-boggling. To load a vector layer you choose Layer -> Add Vector Layer -> File or Directory as we had shown earlier and you will be amazed at the number of options. Most are enabled using the OGR interface.

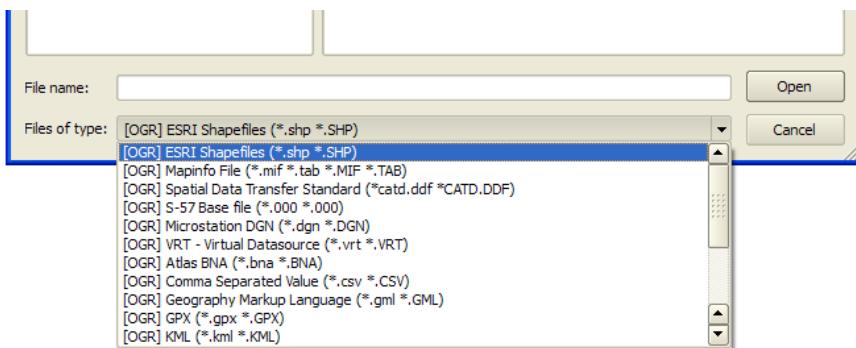


Figure 12.12 QGIS Vector File Sampling.

QGIS also has a rich set of Raster formats to the tune of about 15 different types it supports out of the box. Even more can be accessed with QGIS plugins.

### 12.3.6 Exporting data

QGIS comes packaged with a tool called SPIT, which allows you to batch load ESRI Shapefiles to PostgreSQL. We covered this in chapter 7 briefly so will not cover again. As mentioned, the most annoying thing about it even in the 1.4 incarnation, is that it brings all the field names in as Uppercase, so that you need to use hacks to lower case them so you don't spend your life quoting fields.

In terms of exporting we found exporting to ESRI Shapefile easy and available on the file save as menu. The other export options were very tucked away and difficult to use. We couldn't find an easy way to export our saved layers to other formats.

### 12.3.7 Summary

In this section we lightly touched on what QGIS has to offer. We encourage you to read the 200 some-odd page users manual packaged with it to explore its other features as well as the numerous plug-ins made available by contributors. In addition it has an autoupdate plugin module that informs you of updates to plugins. We really didn't touch at all on its RASTER features or its GPS integration features, but one of the strengths of QGIS are its RASTER analysis features and editing and various plug-ins for dealing with and collecting GPS data. In addition to that, QGIS sports some other unique options. The ones we found most alluring Mapserver Export which allows you to export your workspace as a Mapserver mapfile as well as Mapserver templates. If you are a big Mapserver developer, then this could save you some time. Similar to OpenJump, it has plugins for doing geoprocessing and analysis such as Union, Buffer, etc. These we didn't find too terribly interesting, since they can be done in general more efficiently with the raw power of PostGIS.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

In the next section, we'll cover uDig which is another popular GIS desktop tool.

## 12.4 uDig

User-Friendly Desktop GIS, more commonly known as uDig, has a rich feature set and is based on the eclipse framework. It can run as a standalone or packaged within any eclipse environment. Its main focus appears to be cartography, as a software development kit (SDK), and less on catering to casual GIS users that QGIS targets or hard-core geospatial processing and database programmers that OpenJump targets. Although it does have some database query functionality, it seems much of the effort is on making good looking presentations, good map rendering speed, and extensibility. As such it probably caters more to the high-end GIS user/Cartographer.

One thing that makes uDig stand out from the others is that it is licensed under LGPL rather than GPL. This makes it a bit more friendly licensing wise for creating proprietary applications built on top of it.

### 12.4.1 Feature summary

uDig like OpenJump, QGIS, and GvSig, grew up with PostGIS and as such the spatial database support is probably strongest and most tested for PostGIS above any of the other spatial databases supported by uDig. What makes it stand out the most from the other tools we are discussing is its strong focus on cartography, geometry editing capability and fixing routines, more vendor spatial database support out of the box, and integration with Grass via the JGrass. Note JGrass is a Grass interface built using the uDig Framework.

#### INSTALLATION

Installation was a breeze. We had a few grumblings with it. The download was a bit hefty at about ~100 MB, though that did include the JRE. On our Windows 7 desktop, it froze at the end of the install, so that we had to do end task. Because of these slight annoyances, we gave it a slightly lower score than OpenJump or QGIS. You can download an install for your OS from <http://udig.refractions.net> .

#### EASE OF USE

Loading a PostGIS layer was fairly intuitive, however we found it much more finicky to use than OpenJump, QGIS, and GvSig. First it didn't give us an option to choose which geometry column to load, so for tables with multiple geometry columns, we were extremely frustrated. For certain layers it just refused to load them or gave a yellow triangle denoting a problem. It also did not allow us to filter the layers with where conditions or write ad-hoc queries before the layer loads. This we found extremely annoying especially if you have a large table. The speed at rendering large numbers of records seems a bit better than OpenJump and QGIS or at least it felt faster to us.

On the plus side, we liked the fact that it supported all the common spatial databases out of the box and they were all right next to each other in a logical location. This was not the case with OpenJump and QGIS which were both very PostGIS centric and required acquiring separate plugins or looking in a non-intuitive location for said options after the plugin was ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

installed. We were even able to load some of our MySQL geometry layers with a click of the button without a hitch.

#### **FRAMEWORK**

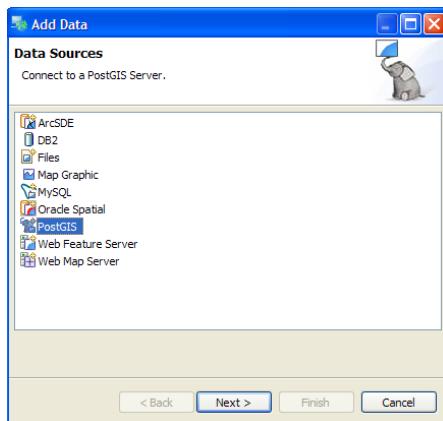
uDig is built on top of Eclipse. Eclipse is a cross platform java framework. From the examples shown on uDig site, one can tell that uDig is quite flexible and easy to morph if you are a Java programmer and is probably well suited for a java programmer building a GIS desktop suite with need for a framework to sink their teeth into. Similar to OpenJump, it is both a desktop tool as well as a platform for building desktop tools. One such interesting example is JGrass which is an interface to GRASS, built using the uDig framework as well as various others show-cased on the uDig gallery page <http://udig.refractions.net/gallery/>

#### **PostGIS SUPPORT METRICS**

- **gcolumns required** - No it will search the whole database or selected schema, but as mentioned, for tables that have multiple geometries, it will arbitrarily pick one.
- **heterogeneous column** - No - although we could select a table with a column that had different types of geometries in each row, and the table view listed the rows, we were never able to get these to display on the map.
- **sql queries** - No. uDig supports a standard called CQL (common query language), for applying filter conditions to layers. It appears to have no mechanism to write raw SQL.
- **curved geometries** - although the 1.1.1 version does not support it, there is work going on in the 1.3, so we suspect it will be the first of these desktop tools to support curved geometries.

#### **12.4.2 Connecting to PostGIS and other Spatial databases**

uDig wins as far as the ease with which you can make the connection to PostGIS and other spatial databases. Just go to Layer -> Add and you see the screen of available options



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

Figure 12.13 uDig Layer Add Connection

### 12.4.3 Viewing and filtering PostGIS data

In uDig, you can do filters using Custom Query Language (CQL) or by picking fields from the drop down list of fields and applying a single column filter. We found both approaches very cumbersome to use because it requires the full dataset loaded. It is not provided as an option when you are first loading the layer to filter as you get with OpenJump and QGIS. When you finally filter -- it highlights the records that match the condition on the screen and the map.

#### CQL FILTER

To use CQL do the following:

1. Add your PostGIS layer
2. Right click layer and zoom to layer
3. Go to table tab - pick CQL from drop down and type in your CQL statement.

CQL uses pretty much the same conventions you will find in SQL where clauses, but since it sits on the uDig client side, you can't use PostgreSQL SQL specific constructs like ILIKE etc. since these are not processed by PostgreSQL.

Below is a snapshot of CQL screen.

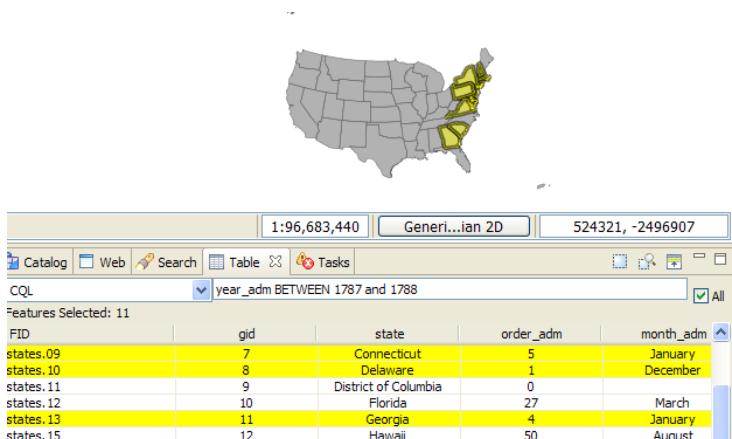


Figure 12.14 uDig CQL interface for filtering data

From within the table view of uDig, you can directly edit attribute fields and from map view edit geometry fields. uDig also supports WFS-T so can be used against a web mapping server such as Geoserver for pushing edits to various different spatial databases using

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

Geoserver or another WFS-T compliant service. This makes it a useful data agnostic spatial database editing tool.

#### **12.4.4 Exporting data**

Out of the box, uDig only supports export to ESRI shapefile format, PDF, and Image. All these you can access by right-mouse clicking on a layer and choosing export or from the menu.

#### **12.4.5 Summary**

In conclusion, we weren't too impressed with uDig's PostGIS functionality; however we were impressed with the ease with which you could load other non-PostGIS spatial databases out of the box. Though we did not cover it here, it does have some pretty good labeling and other cartography features you would expect of a professional GIS desktop tool.

Though we didn't stress test it, it seems on the surface to have the richest support for web mapping services of the bunch. In addition to that, its generous licensing model makes it enticing to build on top of commercially.

In the section that follows, we'll discuss our final tool GvSig, which like OpenJump and uDig is built on top of Java. Like uDig it also uses Java Advanced Imaging (JAI).

### **12.5 GvSig**

GvSig is an open source desktop GIS tool largely funded by the government of Spain. Its main reason for development was as a replacement for ArcGIS desktop Spanish government large install base. As a result, you may find some of the idioms used in ArcGIS are similar in GvSig, and in many cases tries to accomplish more of the basic functionality of ArcGIS desktop in a somewhat similar fashion. It also has support not just for PostGIS, but for Oracle Spatial and ESRI ArcSDE. GvSig was a project started a bit later than Jump and uDig and in looking at the other tools, the GvSig developers strived to make a more speed responsive desktop application. That is perhaps one of the strengths of GvSig is that the gui responsiveness does seem better than uDig, OpenJump and even QGIS.

The other unique feature of GvSig that makes it stand out from the other tools we have just discussed is that it has a mobile version. The mobile version is still in beta and we did not test it, so can't speak for its merits. The mobile version is geared toward users in the field such as surveyors and the like. We assume it is similar in concept to ArcGIS ArcPad.

In addition to a mobile version, it is the only one of these tools that supports ArcIMS webservices. It even has a link directly connecting to ESRI geography network.

#### **12.5.1 Feature summary**

##### **INSTALLATION**

We found GvSig the most annoying to install of the bunch, mostly because it is a Spanish born product with some Spanish screens popping up and we are predominantly English speaking.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

The first problem was that although it says it will work with your existing Java we couldn't get it to work with our already existing 1.6 install. We ended up just using the defaults, by allowing it to install its own JRE 1.5 and JAI.

The second problem was that it first defaulted to Spanish though most of the dialogs were in English or correctly detected we were running an English OS. We changed the default to English during install. When we launched the application, all the menus were in Spanish. After research -- it is in the FAQ how to switch to some other language. On the menu General tab -> Windows -> Preferences -> General -> Language (or if you are in the default Spanish mode: Ventana -> Preferencias -> General -> Idioma (after you are done Accept (Aceptar) and then reopen the application. Hopefully this will be a non-issue in later versions or is just an isolated incident for us. You can download GvSig from <http://www.gvsig.gva.es/>.

#### EASE OF USE

Although gvSig obviously has a lot of functionality under its belt, we couldn't figure out how to render our PostGIS table on a map without pulling out the manual. Because of this we ranked it lowest on ease of use. The other tools, we were at least able to get a list of PostGIS tables, and select it without opening up the manual. Once you get past that and read the manual for about 10 minutes, it all becomes clear.

The manual for GvSig is packaged as a PDF, is very extensive, and available in Italian, Spanish and English.

#### FRAMEWORK

gvSig is built on top of Java and uses the JAI framework similar to uDig for advanced imaging. It does not use Eclipse but has its own Eclipse like framework for extending it. It uses the concepts of projects that have 3 document types: View, Table, and Map.

For PostGIS quick querying and layer viewing, the View type is probably the best to use.

Just like the other tools discussed, GvSig is both a desktop tool and a mapping platform that you can extend by building your own extensions in Java. All these extensions are loaded from the bin/gvSIG/extensions folder of your GvSig install. Each extension gets its own folder and consists of a jar file and various language configuration files and an xml config file. In addition to supporting extensions via java programming, it supports a Jython scripting interface similar to OpenJump.

#### PostGIS SUPPORT METRICS

- **gcolumns required** - Yes. GvSig lists all tables when you browse your PostGIS db, and you can select any table, however views and tables not registered in geometry\_columns, the geometry field drop down is empty and you can't type into it as you can with OpenJump.
- **heterogeneous column** - No - although we could select a table with a column that had different types of geometries in each row, and the table view listed the rows, we were never able to get these to display on the map.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

- **sql queries** - No. GvSig has a query builder tool very similar to QGIS. It allows you to build the where clause filter, but that's pretty much it. It does have an SQL filter which is also a free for all WHERE you can type in, but that appears to be not functional in this particular release. Scanning the newsgroups suggests this is a known issue and will be considerably improved on in the 2.0 version.
- **curved geometries** - No - When we tried to load curved geometries, it said unsupported and just kept on popping up the error so we had to exit the application to get out of the error -OK - error cycle.
- **views** - GvSig will support views only if you manually register them in geometry\_columns or for PostGIS 1.4, you can use the populate\_geometry\_columns function.
- **3d geometries** - GvSig is the only one of these tools to support 3D geometries via the 3D pilot extension downloadable from <http://www.gvsig.gva.es/eng/gvsig0/gvsig-desktop/desk-extensiones/3d-pilot/>

### **12.5.2 Adding a PostGIS layer to a view**

As mentioned, the easiest way to view PostGIS layers and others is to use the "view" document type of GvSig. Within a view, you can add as many PostGIS layers (tables) as you want to the view. To start:

Create a new view from the Project Manager window and rename it to PostGIS Test as shown.

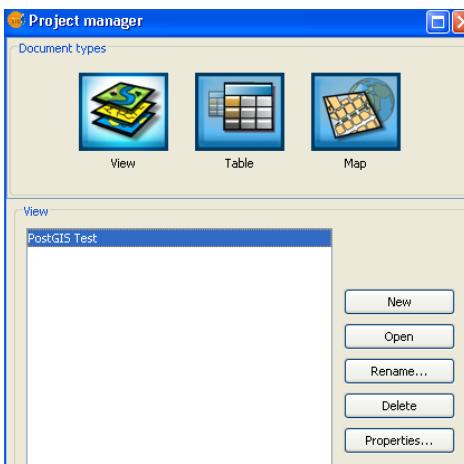


Figure 12.15 GvSig Project Manager window

Select the Open button and under the View menu choose -> Add Layer and then switch to the GeoDb tab and click on the connect icon to create a new PostGIS connection.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

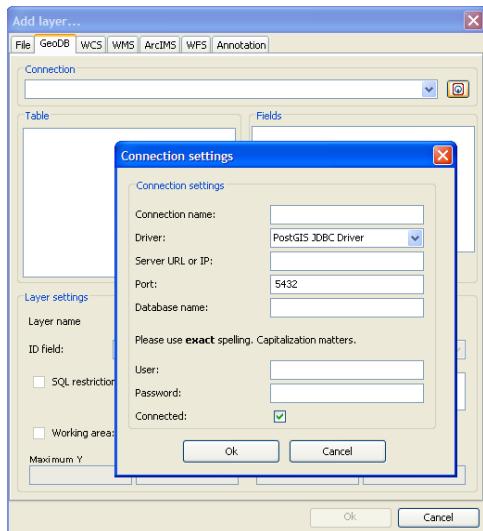


Figure 12.16 GvSig Adding a new PostGIS connection

Fill in all the information to connect to your PostgreSQL/PostGIS db. Then select the connection and the tables you want to add to the screen, filling in the parameter for each. The approach taken for this, as you can see, is very similar to that of QGIS.

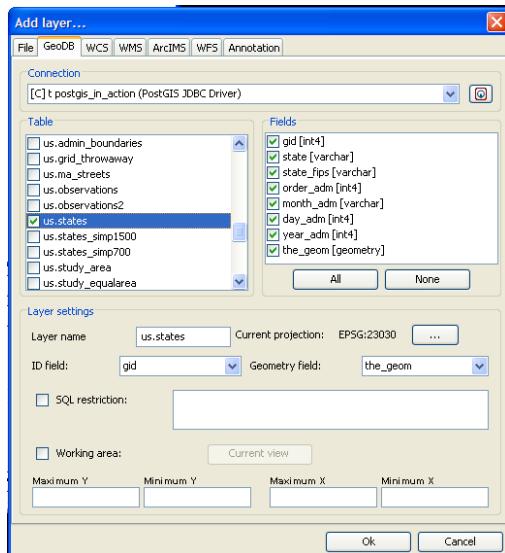


Figure 12.17 GvSig pick PostGIS layers

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

GvSig was not smart enough to read the spatial ref sys for our table, but since we were not going to be doing any reprojecting, we didn't bother changing the setting.

### **SQL RESTRICTION NOT USABLE**

It seems the SQL Restriction option is broken in 1.9. This will be fixed in 2.0 and 2.0 is also planned to support a full SQL statement similar to OpenJump as far as we can tell from scanning the newsgroup.

GvSig has some nice theming features you can access by right-clicking on the layer when once its created.

Although we didn't attempt it, GvSig does appear to support direct editing of both PostgreSQL attribute data and PostGIS geometry data.

#### **12.5.3 Exporting data**

Exporting data to other formats is pretty easy and straightforward in GvSig. To do so,

1. Select the layer
2. On Layer menu pick Export

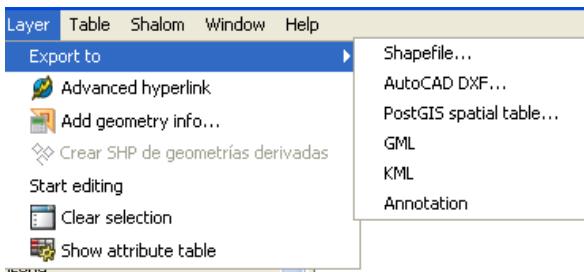


Figure 12.18 GvSig basic export options

#### **12.5.4 Connecting to other spatial databases**

Although we did not try it, GvSig supports MySQL and Oracle Spatial as well as ArcSDE.

##### **CONNECT TO MYSQL**

To connect to a MySQL db, you do it the same as you would the PostGIS connection, pick the JDBC driver from the drop down.

##### **CONNECT TO ORACLE SPATIAL**

The drivers needed for Oracle Spatial are not packaged with GvSig, and to see Oracle Spatial appear in the Add Layer JDBC and Export To options, you need to download the Oracle jdbc drivers from Oracle site.

[http://www.oracle.com/technology/software/tech/java/sqlj\\_jdbc/htdocs/jdbc\\_10201.html](http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/htdocs/jdbc_10201.html)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

and then copy them to the bin/gvSIG/extensions/com.iver.cit.gvsig/lib folder of your install. This is all explained in the manual.

#### **12.5.4 Summary**

We found GvSig a bit tricky to get started with, mostly because its conventions were a little different from the other tools we were used to. Once started, the rest was fairly intuitive. We only brushed the surface of what GvSig has to offer. In addition to its extensive support for various formats, web services, spatial databases, it also has many geometric processing options. These we didn't touch on since they are fairly trivial to do in PostGIS, but not so trivial with shape files. In addition to that it has nice printing options, themeing, editing and measuring capabilities out of the box. We found its export features much easier to use than that of QGIS and the fact it makes AutoCAD export a simple click away should be inviting to AutoCAD users or spatial analysts who work with AutoCAD users.

### **12.6 Summary**

In this chapter we quickly covered the most common free and open source desktop tools used to view and edit PostGIS data and provided a feature matrix that compared them based on functionality, installation and ease of use. While these tools have PostGIS capabilities, they support that functionality in varying degrees and can also be used to view other kinds of data including other spatial databases. We only touched the iceberg of what each of these tools provide.

All these tools have end-user features you can use straight out of the box, as well as developer features that allow one to enhance the functionality via scripting or building of Plug-ins. Sadly we did not have time to go deeply into what these tools offer. We encourage you to explore these tools further to see what gems they have in store. Each of these tools have fairly extensive user manuals.

We hope at the very least, that you were able to get a sense of what each of these tools offer, what audience of user they try to target, and which ones will serve your needs best.

# 13

## *PostGIS WKT Raster*

This chapter covers

- Difference between raster and vector data
- Using PostGIS WKT Raster
- Future of PostGIS WKT Raster

Up to this point, we have focused our attention on spatial vector data. This is primarily because most of the functionality built into PostGIS is for storing and analyzing vector data. However there is another kind of spatial data support that people commonly use perhaps even more than vector data. This is raster data.

Much of vector data we use and get is a refined form of data generated from the raw material we call "raster data" via various expensive and involved third party processing options. As such there is probably a larger wealth of free raster data in the world than vector data and data you can vectorize in any way you like to match your specific area and needs if only you had easy enough to use inexpensive tools to do so.

Although much of vector data is generated from a raster form, it would be unfair to characterize raster data as just the raw material that makes mass production of vector data possible. Raster data is used in various other ways. Each pixel of a raster can have one or more different values. Geometry vector data on the other hand, is continuous with no distinct difference in value from one part of the geometry to the other aside from what you can encode in z and m coordinates. When you view or print vector data you are really viewing a rasterized form of the geometries suitable for the dpi resolution of your viewing device. So in short, raster data in many cases is both the beginning and the end of all kinds of spatial analysis.

In this chapter, we will first look at raster data as a form of information expression and how it differs from vector data. We will look at how the new raster companion data type makes analyzing raster data in a PostGIS enabled database possible and easier than you can

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

do outside of the database. We will also look at the various kinds of cross-querying you can accomplish by having both vector and raster support in the same database. It is this cross-querying that makes having vector and raster coexist in the same database so compelling. We will conclude by summarizing the various enhancements planned and being advanced in PostGIS raster support.

## **13.1 What is PostGIS WKT Raster**

PostGIS WKT Raster is a subproject of the PostGIS project started by Pierre Racine and others. It was created to support spatial analysis and processing of raster data as well as making processes that involve both formats possible to do with one tool. PostGIS WKT Raster introduces a new PostgreSQL data type called "raster" that stores raster data in a binary format in PostgreSQL similar to how the PostGIS geometry and geography types store vector data.

PostGIS WKT Raster introduces analysis and processing functions to work with raster data and to also allow raster data to comingle with vector data. The main reason for this focus is that a lot of operations you normally do with vector data you would want to do with raster data. Some processes involve both types of data as input or one type of data as input and the other type as output. Currently, PostGIS WKT Raster is installed as a separate PostgreSQL module that you install in a PostGIS enabled database. In later releases of PostGIS, the plan is to merge this in to the core PostGIS project and distribute as a single PostGIS module. You can experiment with PostGIS WKT Raster now if you are using PostGIS 1.3.5 or above.

Before we go into specifics about what you can do with rasters using PostGIS WKT Raster, we'll cover some basic concepts about raster and its common use cases.

### **13.1.1 What is raster data and how is it different from vector data**

Raster data is data that is expressed as a rectangular grid of pixels (sometimes called cells). When this data is pinned down to a particular region of the world, we refer to this data as georeferenced raster data and we can georeference raster data much the way we georeference vector data.

#### **GEOREFERENCE RASTER DATA**

Georeferenced raster data is data where the rows and columns usually the upper left corner is pinned down to a specific geographic location and expressed in some spatial reference system. The pixels are generally modeled as equal sized and the pixel width and height represent x meters/ft/degrees or y meters/ft/degrees of geographic space in some spatial reference system.

Pixels have things called bands, which are also sometimes referred to as channels. When we look at a picture such as a jpeg, png or tiff, it is generally composed of 1 - 4 bands

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

expressed as the Red Green Blue Alpha (RGBA) channels we see on the screen. When you look at the picture in a microscope, you will realize, it stores a lot of information. It stores a matrix with 1 or more numeric values per cell where each cell has exactly the same number of values.

Vector data is continuous data represented by x,y, and sometimes m and z coordinates and a linear equation that defines the continuity of the pattern. You can break vector data into infinitesimally small segments that still satisfy the equation, but raster data on the other hand, can only be broken down to the pixel (cell level).

Visually speaking, this means you can blow up a vector to any resolution and it will still maintain its original smooth form, whereas a raster you will begin to see the individual pixels that make up the image. The smaller you make your pixels, the more storage space you need to hold your raster data, but the more crisp your image will be and the more sampling options you have to make lighter weight rasters (bigger pixels) or less accurate vector geometries. Below is a figure showing a portion of a 1960s USGS Massachusetts raster topographical map we downloaded from <ftp://data.massgis.state.ma.us/pub/images/usgs/> and the same portion of the map after select pixel value ranges have been vectorized using ST\_DumpAsPolygons and further smoothed with PostGIS geometry simplify and buffering operations. When zoomed in, the original raster begins to show its pixel formation whereas vectorized will continue to show straight lines.

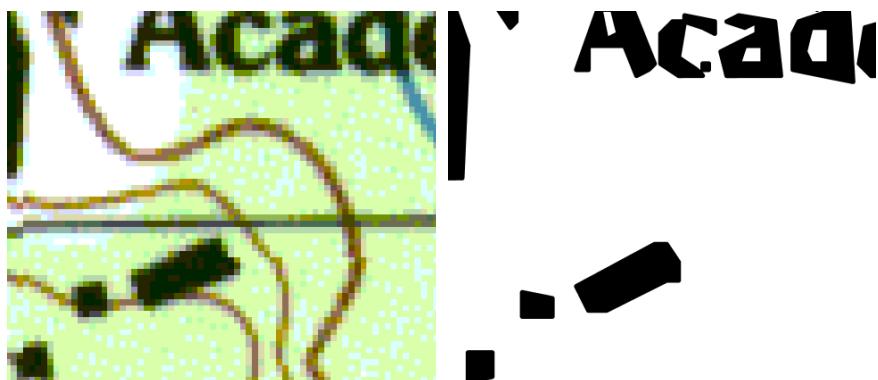


Figure 13.1 Example of a raster topo map and select pixel ranges vectorized. In raster you can make out the pixels while in the vector you can not

Although rasters we generally look at are limited to 1 to 4 bands, with each pixel band storing whole integers, they need not be. The raster universe is much bigger than that. You can have rasters with many bands and those store floating point numbers or larger integers in each pixel band. The PostGIS raster data type supports all these various pixel band types and can even store rasters with different kinds of pixel bands in the same raster. These

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

kinds of rasters do not always have a direct viewable format. Each band can be encoding a different kind of information about that specific region of space such as observations from various instruments that are synchronized to analyze the same section of grass at each moment in time. This kind of capture is often referred to as remote sensing.

### **Remote sensing**

Remote sensing is a field of work that involves acquisition of information about objects and phenomenon using recorded or real-time sensing devices. Much of the automated geographic collection of information is done using remote sensing technologies. Kinds of devices include Light Detection and Ranging (LIDAR), Radio detection and Ranging (RADAR), space probes, ultrasound, Magnetic Resonance Imaging (MRI), Positron emission Tomography (PET), and many others. From these raw outputs we get more refined rasters such as those for aerial imagery or even vector data such as building footprints.

These raster band pixel values can be further reclassified via various algebraic calculations that map pixel band values from the one set of bands to another or converge pixel ranges based on values or proximity to other values. From these processes we get viewable rasters, easier to analyse rasters, or crisper vector geometries. When you hear the word raster or picture, we encourage you not to just admire the pretty image before you, but also think about the matrices behind that image and the power you can wield with matrix like algebra.

What makes expression of data in raster format particularly alluring is that the human brain is designed to analyze images very quickly; more so than other forms of information. Our minds can scan millions of pixels on a screen and grasp the relationships between various factors simply by the shades of the colors used or the proximity of one shade of color to another. We can do all of this in a split second. We use this for facial recognition and character recognition among other things, and with an engineer or doctor's attuned sense even surmise subtle changes in color of thermal camera or other imagery to diagnose problem regions, root causes of explosions, and cancer. Our minds are automatically reducing the information overload of rasters into vector pattern simplifications. This is also what makes optical illusions possible; when our minds can come up with 2 vastly different pattern formations based on the same set of pixel data.

Raster data in some ways is more versatile than vector data because it makes fewer assumptions about patterns. Vector already assumes a specific pattern formation. You have many options for arriving at patterns that you can't with vector data. For example you can decide that information in one band is significant for a particular purpose only if its other bands have some specific range of values or if this pixel has neighbors with values within a particular range. You can then vectorize based on these assumptions. Vector data doesn't have these other levels of analysis. What vector does provide are simpler faster ways of thinking about things. You can more easily take areas and measurements with vector data.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

A pixel in raster data can be modeled as a rectangle with a band1value, ... bandnvalue that you can then relate to vector data via the x and y start coordinate of the pixel. Each rectangle in a raster has a width and height and in most cases, the pixels in a raster are equally sized, though they need not be. Keep in mind that this is simply a model of a pixel, a pixel is really a measurement of a section of space and as such you can just as easily say that is a triangle or a star shape, but the math to deal with that would be harder than a rectangle.

### **REGULAR VS. IRREGULARLY BLOCKED RASTER**

When each pixel of a raster is equal sized and each row of the raster has the same number of pixels and each column of the raster has the same number of pixels, we refer to that as being regularly blocked. If any of these is not the case, then the raster may not form a rectangle and is considered irregularly blocked.

PostGIS raster type supports both kinds of rasters, but can't currently export out irregularly blocked rasters.

#### **13.1.2 Why analyze raster data**

Why would you want to analyze raster data? Much of the machine generated data in the world comes in a raster format. This is increasing as tools such as LIDAR imagery, thermal and infra red camera imagery, electron microscopes, etc. become cheaper and easier to use. Even the old map you scanned from an 18<sup>th</sup> century drawing or Nautical Navigation charts you download from NOAA are raster. By analyzing such drawings, you can convert it to a smaller, crisper and easier to manipulate vector form. Much of the data we consider as vector data are extruded from raster data whether that be paper survey maps scanned in, or other kinds of imagery that form the imprint of an object. In fact lots of GIS businesses make much of their money converting raw raster data into other forms.

Vector data is generally smaller than raster data for the same region because it is the result of line fitting various observation points of data (the raster). Raster analysis is done quite frequently in the real world to analyze land use, soil, bacteria and plant growth, wind, digital elevations and terrains using digital elevation and terrain model format (DEM/DTM). Much of which is best expressed in its raster form. In addition, raster data such as aerials is used to overlay on top of maps and giving higher and lower resolutions as you zoom in and out by changing the sampling of the pixels. There are also many non-GIS uses for raster analysis, many of which have not been fully explored. Any analysis where matrices are useful or machine generated data is expressed in a pixel/cell format, is suitable for analysis and processing into other forms.

In the rest of this chapter, we will learn how to load raster data, and how to do some common exercises such as reading bands from raster data, polygonizing raster data, and getting various attributes about raster data. We'll then summarize with what you can expect to do with rasters in the future.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

Keep in mind that at this point PostGIS WKT Raster is a moving target accelerating in motion. By the time you read this, it will do much more than what we have summarized here.

To find out more about PostGIS WKT Raster, please refer to <http://trac.osgeo.org/postgis/wiki/WKTRaster>.

## 13.2 Getting started with PostGIS WKT Raster

In order to use PostGIS WKT Raster, you need the following items

1. A working PostGIS database preferably 1.4 or above though 1.3.5+ should work.
2. The rtpostgis.so or rtpostgis.dll compiled against your version of Postgres and a version of PostGIS. If you are on Linux, you can download the source from <http://www.postgis.org/download/>. If you are on Windows, there are fairly recent precompiled binaries for 8.3 and 8.4 at that should work fine for PostGIS 1.4-1.5 <http://www.postgis.org/download/windows/experimental.php>  
You can expect in PostGIS 2.x for WKT Raster to be part of the core postgis.dll/.so file and not require separate installation.
3. As of this writing, WKT Raster relies on the gdal library -- <http://www.gdal.org/> for its more advanced processing features such as ST\_DumpAsPolygons and ST\_Polygon. If you are compiling yourself, you will need the source for gdal 1.6+ and compiled and reference it in your configure.
4. Copy the .so or .dll file into your PostgreSQL ..lib folder. This step is usually done by the install process.
5. Run the rtpostgis.sql in your PostGIS enabled database.

Now that we have covered how to install raster support, we'll go on to how you can load raster data into your PostgreSQL database and various storage considerations you need to make.

## 13.3 Storing and Loading raster data

Before you can start working with raster data, you need a mechanism to either import raster data into your database or reference it from outside of your database.

In the sections that follow, we'll go over the ways you can store your raster data and considerations you need to make about how you store the data.

### 13.3.1 Options for storage

PostGIS WKT Raster supports in db storage and out of db storage.

#### IN DB STORAGE

If you choose to store your rasters in the database, you get the following benefits:

##### Pros

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

- Your raster data gets backed up with your db
- More tested than out of db
- Transactional editing goodness of the db
- Faster reading of data, aggregation, and vectorization
- Pretty much any format of data supported by gdal you can load in.

Storing rasters in the database is not without its issues:

#### **Cons**

- Rasters are big and generally bigger than vectors that cover the same space. This will make your database backups longer
- You can't easily share the rasters with tools designed to read them from their flat file storage.

### **READING POSTGIS WKT RASTER WITH RENDERERS**

The planned ST\_AsJPEG, ST\_AsTiff functions have not been created yet, but should exist shortly will make this process easier. The popular GDAL library used by Mapserver, QGIS and other GIS open source and closed source tools also has a driver for PostGIS WKT Raster . Since many open source as well as commercial GIS tools build on top of GDAL, this means you should see support for PostGIS WKT Raster in third party tools soon if its not there already.

#### **OUT OF DB STORAGE**

In out of database storage mode, the rasters are stored outside the database and the database stores the path to each tile and the geometric extents and other meta data about the tile. The pros and cons are as follows.

#### **Pros**

- You can share the rasters with other applications that need them and don't know how to read raster from the database
- Backup is simpler since you just backup the files and restore them

#### **Cons**

- Not very well tested
- Don't get transactional benefits of db
- The rasters can get deleted or moved apart from the db and make the db records useless
- General path annoyance. Need to make sure the postgres server process can access the files and path setting is in a form that is relative to server.
- Analysis of rasters such as forming polygons, reading pixel values, etc. will be slower

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

than in db.

### **13.3.2 Using loader to load data**

Currently, PostGIS WKT Raster relies on Python, GDAL bindings for python and numpy for loading raster data. In order to use the packaged loader called gdal2wktraster.py, you need Python 2.5 installed with Python bindings GDAL 1.6+ and numpy.

#### **gdal2wktraster.py current and future**

In later versions of WKT Raster, will include an executable called raster2pgsql with few dependencies that will mirror much of the functionality of shp2pgsql except for loading simple raster data such as tiff files similar to how shp2pgsql loads ESRI shape file vector data. The GDAL PostGIS WKT Raster will also be enhanced to import data to PostgreSQL. Currently GDAL PostGIS WKT Raster can only read PostGIS raster data type and export to other raster formats.

gdal2wktraster.py can load in any raster format supported by GDAL as well as coverages of tiles. This covers quite a range of formats such as TIFF, Jpeg, DEMS, PNGs, Gifs, Arc GIS Ascii grids to name a few. Some are not compiled in by default and may depend on your particular install. Refer to [http://www.gdal.org/formats\\_list.html](http://www.gdal.org/formats_list.html) for details.

Instructions on how to get started with that are here  
<http://trac.osgeo.org/gdal/wiki/GdalOgrInPython>

If you are on windows and don't want to mess with compiling your own, you can use the binaries from [http://pypi.python.org/pypi/GDAL/1.6.1\\_The\\_ReadMe.txt](http://pypi.python.org/pypi/GDAL/1.6.1_The_ReadMe.txt) file packaged with the wkt raster windows experimental builds covers configuring your Python environment to support raster loading.

The loader supports the ability to:

- Load single rasters
- Chunk a single raster into various raster records
- Import raster coverages (several rasters all at once into same raster table)
- Just reference rasters and load in the meta data and path info for that.

In the examples that follow we will demonstrate some of these.

To get more help about what is supported by gdal2wktraster use:

```
python gdal2wktraster.py --help
```

#### **LOAD OF SMALL RASTER**

For this first example we'll load in our celebrity PostGIS elephant. We'll call our PostGIS elephant "Pele" (short for PostGIS elephant) and assume for our exercises that Pele is a girl.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

Pele will guide us thru various raster exercises. Although Pele lives in her own undefined coordinate system unencumbered by the earthly weight of the world, she is a useful specimen to start with. She is small but big enough to be interesting, cute, and easy to manipulate. She also doesn't mind being fattened, stretched, cloned, and moved to a point.

The following snippets of code generate the pele.sql file and installs the sql file. We can do this in one step too by using the | command we demonstrated with shp2pgsql.

```
python gdal2wktraster.py -r pele.png -t ch13.pele -o pele.sql  
psql -h localhost -U someuser -d postgis_in_action -f pele.sql
```

Both of these steps you run from command line.

#### **LOADING LARGER AND GEOREFERENCED RASTERS**

Many rasters are big in the mega byte or gigabyte range. The raster type uses gist indexes similar to the geometry and geography data types we have seen before. The bounding box of that is the bounding box of the raster in each record. Since index seeks are much faster than the final non-box intersects checks, you generally want to split raster files into smaller rasters and store one raster fragment per record. Smaller rasters are also easier to manipulate if you only need to work on one small part at a time.

In the last example, we brought Pele in into a table called ch13.pele, and also as a single record. Ideally you probably want your chunks to be no more than 200x200 pixels if you expect to do a lot of processing and analysis with them. Pele was a little bigger, but not too bad.

In terms of choosing your tile sizes, you don't want to make them too small such that you are always doing a lot of record seeks and pulls to get the records you need to work with. You also don't want them too big such that most of the raster information you would intersect with your geometry data is not needed and is just dead weight to your processing functions.

For this next example, we are going to load in an ArcView image file (BIL) representing elevations of the Hawaiian Island of Kauai. This particular elevation model we grabbed from <http://duff.geology.washington.edu/data/raster/tenmeter/hawaii/index.html>

The BIL we are using is measured in meters using UTM zone 4, NAD 83. A query of our spatial\_ref\_sys table:

```
SELECT srid, proj4text  
FROM spatial_ref_sys  
WHERE proj4text ILIKE '%utm%' AND proj4text ILIKE '%zone=4 %'  
AND proj4text ILIKE '%datum=NAD83%'
```

informs us the SRID of this raster is 26904.

With this information in hand, we will load Kauai BIL into our database.

If you want to see what this file looks like, you can view it in QGIS using the Add Raster Layer option. Below is a snapshot of the image. The first is the original file using the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

pseudocolor coloring option in QGIS. The second is the same file with the envelopes of the PostGIS raster rows overlaid on top to show how the rasters in the database are stored.

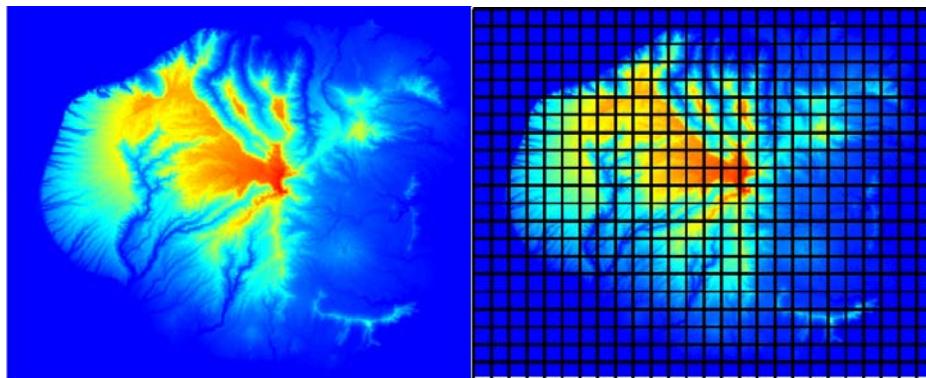


Figure 13.2 The Kauai BIL elevation model file in pseudocolor as single file and after chunking

Since this file is a good size 14mb, we are going to break it into pieces of 200x200 pixels. as part of the loading process and use the -I option to have the gist index created after load and -M to force a vacuum of the table. Here is our command to generate and load the sql file

```
python gdal2wktraster.py -r kauai.bil -t ch13.kauai \
-s 26904 -k 200x200 -I -M -o kauai.sql
psql -h localhost -U someuser -d postgis_in_action -f kauai.sql
```

Now to get a quick summary of what we have loaded, we run this query:

#### **Listing 13.1 Getting general summary info about rasters in a kauai table**

```
SELECT count(*) As num_rasters, ST_Height(rast) As height,
       ST_Width(rast) As width, ST_SRID(rast) As srid,
       ST_NumBands(rast) As num_bands,
       ST_BandPixelType(rast,1) As btype
  FROM ch13.kauai
 GROUP BY ST_Height(rast),
           ST_Width(rast), ST_SRID(rast),
           ST_NumBands(rast),
           ST_BandPixelType(rast,1);
```

In this small snippet of code, we demonstrate the various common meta data you would want to know about each raster tile in your table. Most of these are specific to raster data except for ST\_SRID which applies to both vector and raster data.

The code gives us an output of:

num_rasters	height	width	srid	num_bands	btype
546	200	200	26904	1	16BUI

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

We will go over some of the above functions and various other useful functions in the coming sections. We will demonstrate simple examples using Pele since she is compartmentalized in just one record. We will also demonstrate how we can bring Pele to Kauai.

## **13.4 Raster Maintenance tables and functions**

Many of the raster maintenance functions are similar to the functions you find for geometry support. If you are already familiar with geometry maintenance functions, learning the raster maintenance functions will be trivial.

Similar to the geometry type, the raster type has a parallel registration table that catalogs raster tables that are registered. This table is called **raster\_columns**. The functions for adding constrained raster columns are also similar in naming to the geometry add/drop column functions.

### **13.4.1 raster\_columns meta data table**

When you import rasters with the gdal2wktraster utility, it also registers the raster table in the raster\_columns meta table using the AddRasterColumn function. We can query this table much like any other

```
SELECT r_table_name, r_column, srid,
       pixel_types,
       nodata_values,
       pixelsize_x, pixelsize_y
  FROM raster_columns
 WHERE r_table_schema = 'ch13';
```

The raster\_columns table is similar to geometry\_columns and geography\_columns, except it has a lot more fields. We are just showing a subset here. These are similar to the geometry\_columns f... of the same name except they are prefixed with r\_ instead of f\_. The pixel\_types is most similar to geometry\_type except since each band has its own type, it is an array representing the type of each band. nodata\_values is also an array with one element for each band. In some cases nodata\_values is null, meaning the raster set does not have a single value representing no data. In these cases the ST\_NoDataBandValue will return 0.

The above example returns a table listing basic information about the rasters. This is not the only information available in the raster\_columns table.

**Table 13.1 Query of raster\_columns**

r_table_name	r_column	srid	pixel_types	nodata_values	pixelsize_x	pixelsize_y
pele	rast	-1	{8BUI,8BUI,8BUI,8BUI}		1	1
kauai	rast	26904	{16BUI}		10	-10

As we can see from the raster\_columns query table, Pele is composed of 4 bands that store pixels that are 8 bit unsigned integers. Since we didn't denote an srid, pele's SRID is unknown. Kauai on the other hand is composed of one pixel band with values that are 16 bit unsigned integers representing elevation levels on the island. Its srid is 26904 as we specified, and gdal2wktraster read from the file that each pixel size is a 10x10 meters per pixel. Since our units are in meters, each pixel represents a 10x10 square meter of coordinate space. Also note that kauai has a -10 for the height of the pixel which we shall explain shortly.

Note that although this data talks about a whole table, you can define a table with a mixed bag of rasters just as we defined tables with a mixed bag of geometries. Each raster object has its own meta data associated with it. By default gdal2wktraster calls the raster column rast and the id column rid, though you are free to name them whatever you want. The downside of using a mixed bag of rasters is that the GDAL driver we shall discuss later, does not know how to export such a monster to other raster formats.

### **13.4.2 AddRasterColumn function**

The function used to create a uniform raster column and register it in raster\_columns table is called AddRasterColumn. Similar to its AddGeometryColumn sibling, it also adds constraints to the created raster column to ensure only rasters of the specified srid can be inserted in that column. It currently does not constrain blocksize and other properties designated when registering the column. It also will not go back and correct the raster\_columns table when changes are made.

Although you can use AddRasterColumn to add a raster column to a table, you can also use the standard CREATE TABLE construct if you don't care about constraining the spatial reference system. Here is an example that adds a secondary raster column to the pele table.

```
ALTER TABLE ch13.pele ADD COLUMN rast_in_kauai raster;
```

The gdal\_translate tool can only handle regularly blocked and tables with single raster column, so to accommodate this restriction for later, we will create a new table to store Pele for future export use.

```
CREATE TABLE
ch13.pele_in_kauai(rid serial primary key,
twin varchar(30));
SELECT AddRasterColumn('ch13', 'pele_in_kauai', 'rast', 26904,
'{8BUI,8BUI,8BUI,8BUI}', false, true, '{255,255,255,255}', 10,-10,299,439, null);
```

The AddRasterColumn function returns a message:

```
ch13.pele_in_kauai.rast
srid:26904 pixel_types:{8BUI,8BUI,8BUI,8BUI}
out_db:false
regular_blocking:true
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```

nodata_values:'{255,255,255,255}'
pixelsize_x:'10'
pixelsize_y:'-10' blocksize_x:'299' blocksize_y:'439' extent:NULL

```

We'll use this column later to hold georeferenced versions of Pele in Kauai.

### **13.4.3 Other management functions**

In addition to the AddRasterColumn function, there is a DropRasterColumn and DropRasterTable that are parallels to the DropGeometryColumn and DropGeometryTable functions we have seen before.

## **13.5 Simple exercises with raster data**

Once we have Pele and Kauai transported into our database, we can access some meta data information about them.

### **13.5.1 Common accessors**

To figure out the number of rows and columns in Pele's extent, we use the ST\_Width and ST\_Height raster functions. These functions return the width and height of the image in units of pixels and the spatial reference system the coordinates of the raster are georeferenced to.

```

SELECT ST_Height(rast) As nrows, ST_Width(rast) As ncols,
       ST_NumBands(rast) As numbands, ST_SRID(rast)
FROM ch13.pele;

```

From the above we learn that Pele's image is 439 pixels tall, by 299 pixels wide, and she has 4 channels of information in her image. Since we didn't specify a spatial reference system for her, her spatial reference system came in as -1 (unknown).

#### **ST\_VALUE AND ST\_BANDNODATAVALUE**

WKT Raster doesn't have extensive polygonizing functions yet, but with the very useful ST\_Value function, we can pull lots of polygons out of Pele by selectively picking cells in her body using pixel ranges and/or a geometric range in the raster. By looking at Pele's image, we can tell the upper quadrant is white space and that white space is most likely what we should consider No data band pixel value. So to confirm the pixel value of that white space - we run this query:

```

SELECT ST_Value(rast,1,1,1) As b1val,
       ST_Value(rast,2,1,1) As b2val,
       ST_Value(rast,3,1,1) As b3val,
       ST_Value(rast,4,1,1) As b4val,
       ST_BandNoDataValue(rast,1) As blnodata
FROM ch13.pele;

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

The ST\_Value function is a function that for a given band and row/col in a raster, will return the value of that band pixel. This query tells us that the no data band value in all bands of Pele should be 255, but at least for the first band, it is currently set to 0. Note its not really 0 either, but the ST\_BandNoDataValue function currently returns 0 if no BandNoDataValue is exists for a raster.

### **ST\_SetBandNoDataValue**

The ST\_SetBandNoDataValue is a complement to the ST\_BandNoDataValue that allows you to redefine the pixel value in each band that represents no data.

The no data value is used by some functions to determine pixels to skip over during analysis.

To convert our no data value from 0 to 255, we use the ST\_SetBandNoDataValue function to reset this. Note that this setting like many others can be set before you load the data using gdal2wktraster.py script.

```
UPDATE ch13.pele
  SET rast =
    ST_SetBandNoDataValue(ST_SetBandNoDataValue(
      ST_SetBandNoDataValue(
        ST_SetBandNoDataValue(
          rast,1, 255)
        ,2,255),
      3,255),4,255) ;
```

### **ST\_Envelope AND SPATIAL INDEX**

Before we continue, we overlay the geometric envelope of Pele with her original self, to make sure all looks good. To get the geometric envelope of Pele we do this:

```
SELECT ST_Envelope(rast) FROM ch13.pele;
```

Now we overlay in OpenJump our envelope with our original png image to make sure they line up.

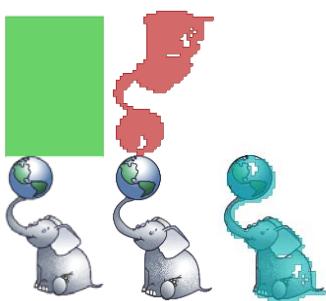


Figure 13.2 Pele with her envelope, her 10 sampling polygon self, and herself flipped back correctly.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

It appears something went wrong when we pulled in Pele. She's not lining up with her envelope. We have also fast forwarded a bit and shown a polygonized form of Pele we will generate with our upcoming exercises. To produce the polygon version, we are sampling every 10 pixels in x and y direction. From the polygonized form, you can see that she came in as an upside down mirror image of her former self.

In the last shot, we correct Pele by setting her Y pixel size to a -1 which we will demonstrate shortly.

#### **CREATING A SPATIAL INDEX ON RASTER DATA**

There is currently no gist operator defined for raster, so to use a gist index we use a functional one. If you included the -I option in gdal2wktraster load, then this is automatically done for you. If you forgot or need to load additional data later and left it out, you can create one after the fact with a command such as this.

```
CREATE INDEX idx_gist_ch13_pele_rast
    ON ch13.pele USING gist (ST_Envelope(rast));
```

#### **ST\_CONVEXHULL**

The ST\_ConvexHull returns more or less the same answer for regularly blocked and non-skewed rasters. The only difference is that the envelope returns a slightly larger box with the coordinates rounded.

ST\_ConvexHull does not exclude no data band values from the mix. It becomes interesting if you rotate your raster or your raster is not an even wxh filled pixels. In all our cases we have regularly blocked (where all blocks are filled in across and down).

In the next example, we will rotate Pele and overlay her new envelope and convex hull.

```
SELECT ST_Envelope(rast_rot) As envelope,
       ST_ConvexHull(rast_rot) As convexhull
    FROM (SELECT ST_SetSkew(rast,0.5) As rast_skew
          FROM ch13.pele) As foo;
```

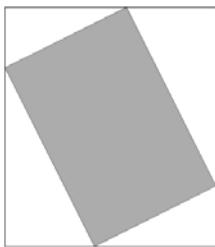


Figure 13.3 The extent of a rotated Pele overlaid with the convex hull. The shaded in gray is the convex hull of the new rotated image.

Now that we have demonstrated some simple things you can do with raster data, we will move on to more exciting things; combining the power of raster processing with vector processing.

### **13.5.2 Georeferencing functions**

Raster data usually has an origin that starts at the upper left. Spatial coordinates on the other hand usually has an origin that starts at the lower left. Raster data that goes in the same direction as the spatial coordinate system will have a positive X pixel size and a positive Y pixel size. Since common files such as pngs, gifs and tiffs are usually have an upper left origin, the Y pixel size is negative to denote it is in the opposite direction of the spatial coordinate system.

#### **WHY DID PELE COME IN THE WRONG WAY?**

Pele came in upside down because the default YPixelSize for non-georeferenced images used to default to 1 in gdal2wktraster if not explicitly passed in. This happens if your jpeg/png/tif is missing a world file. A world file is a meta data file that denotes various attributes such as the coordinate reference system, direction of pixels and pixel size in coordinate units. For some kinds of raster data this meta data is embedded directly in the file rather than as a separate text file. If no information is provided gdal2wktraster guesses at the x and y pixel sizes. This has since been changed so that gdal2wktraser will default to -1 for y pixel size to support the standard convention.

In this section, we'll explore the various functions used to set the orientation and sizing of pixels relative to spatial coordinates. These are often referred to as georeferencing functions.

Below is a table of georeferencing edit functions currently supported by WKT Raster. These are all described in the WKT Raster Editor Function section of reference.

**Table 13.1 Georeferencing raster edit functions**

<b>Function</b>	<b>Purpose</b>
ST_SetGeoReference	Sets the basic 6 georeferencing numbers in 1 statement. This includes PixelSize (x and y), skew (x y), and upper left corner x and y coords
ST_SetSRID	Sets the spatial coordinate system the raster uses. The UpperLeftCorner coordinates and pixel size should be expressed in coordinates and units of this system
ST_SetUpperLeftX	Sets the upper left corner of the raster to spatial ref coordinates
ST_SetUpperLeftY	Sets the upper left y position of the raster to geometric coordinates

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

ST_SetPixelSize	Sets the pixel width and height (x,y) size and direction in units of the coordinate system.
-----------------	---

#### **PIXELSIZEX AND PIXELSIZEY FAMILY OF FUNCTIONS**

The PixelSizeX and PixelSizeY represent the ratio of pixels to a spatial coordinate system. The PixelSizeX is most always positive since raster coordinates and spatial coordinates go in the same direction on the X axis. The PixelSizeY is most often negative since raster pixel origin often go in the opposite direction spatial coordinate system. By setting the Y pixel size of Pele to -1 we are saying that she is vertically oriented in the opposite direction of what we consider our spatial coordinate system.

In this next example we correct the mistake we had made of bringing her in upside down.

First to see how we screwed up we run an informational query:

```
SELECT ST_PixelSizeX(rast) As pixx, ST_PixelSizeY(rast) As pixy
FROM ch13.pele;
```

The query we just ran informed us that the x and y pixel sizes are both 1 meaning our raster coordinate direction is assumed to be the same as our spatial coordinate system and each x width of pixel and y width of pixel is assumed to represent one unit of x and y of our spatial coordinate system.

Next we correct our mistake by switching to state our raster y direction goes in opposite direction of our spatial coordinate system.

```
UPDATE ch13.pele SET rast = ST_SetPixelSize(rast, 1,-1);
```

Next we vectorize our raster using a function we created that we shall describe shortly.

```
SELECT rutility.upgis_rastertopolygon(rast, ARRAY[1,2,3,4], 10)
FROM ch13.pele;
```

The 10 at the end we use for our sampling ratio - which means for every 10 pixels in x and every 10 pixels in y, check only 1 (so only one pixel for every 100) 10x10 (100) will be considered and will be treated as they represent the whole population of 100.

We can do other interesting things with these settings to make Pele taller or fatter than she is in real life.

Below is the same exercise repeated but creating a version of our Pele that is Y pixel size to -1.5 and our X pixel size to 2. Which means the width of a pixel represents 2 units in our spatial coordinate system and a height of a pixel represent 1.5 units in our spatial coordinate system but oriented in the opposite direction.

```
SELECT rutility.upgis_rastertopolygon(
ST_SetPixelSize(rast,2,-1.5)
), ARRAY[1,2,3,4], 10
FROM ch13.pele;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>



Figure 13.4 A fatter and taller shadow of Pele overlaid with her original self.

#### USING ST\_SETGEOREFERENCE AND ST\_SETSRID TO SET SPATIAL COORDINATES

Pele lives in her own coordinate system detached from everything else. For this next example we are going to create clones of her that are transported to Kauai. These new Peles will have a pixel size that are 10 meters by 10 meters per pixel in UTM zone 4, NAD 83 spatial reference system (SRID: 26904). Ideally we want to place her clones in spots that seem bright and happy to each of them. Since we don't know, we'll just place them all together and let them find their way.

Then we use the `ST_SetGeoreference` and `ST_SetSRID` functions using these new coordinates and set that as the Upper left corner and also use `generate_series` to create 3 copies of pele separated by their body widths.

#### **Listing 13.2 Creating 3 clones of Pele in Kauai**

```
-- 1
INSERT INTO ch13.pele_in_kauai(twin, rast)
SELECT 'pele' || i,
       ST_SetSRID(
           ST_SetGeoReference(rast,
           '10 0 0 -10 '
           || 443149.653768 + i * ST_Width(rast) * 10
           || ' ' || 2440440.99542, 'GDAL'),
           26904)
FROM ch13.pele
CROSS JOIN generate_series(1,3) As i;
-- 2
UPDATE raster_columns
SET extent = (SELECT
                  ST_Union(ST_ConvexHull(rast))
                  FROM ch13.pele_in_Kauai)
WHERE
    raster_columns.r_table_name = 'pele_in_kauai'
    and r_table_schema = 'ch13';
-- 3
CREATE INDEX idx_gist_ch13_pele_in_kauai_rast
    ON ch13.pele_in_kauai USING gist (ST_Envelope(rast));
vacuum analyze ch13.pele_in_kauai;
-- 1 create 3 peles across
-- 2 correct raster_columns
-- 3 index and analyze for better performance
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

We (1) insert 3 copies of Pele using PostgreSQL built in generate\_series and using the raster ST\_SetGeoReference to reposition the coordinates in Kauai. (2) We then update the raster\_columns table to correct the extent of the table. This is currently necessary for gdal\_translate to be able to correctly export.

To see where Pele's clones are relative to the Kauai rasters we overlay the envelopes against the envelope of the Kauai rasters:

```
SELECT twin, ST_Envelope(rast)
  FROM ch13.pele_in_kauai;
```

With that of our Kauai table:

```
SELECT ST_Envelope(rast)
  FROM ch13.kauai;
```

The diagram below shows the relative positioning of the clones to the Kauai raster tiles.

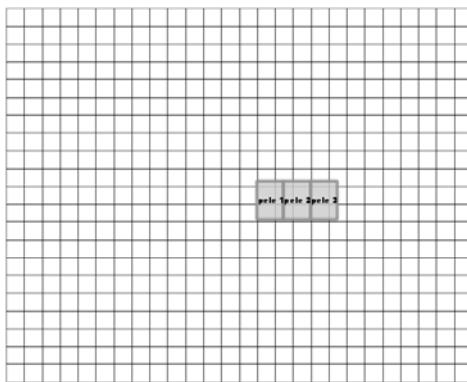


Figure 13.5 The clone envelopes of Pele overlaid on the envelopes of the kauai raster tiles

In the next section we'll demonstrate some common processes you can perform on rasters as well as how you can relate rasters with vector data.

## 13.6 Combining Raster processing with vector processing

In this section we'll demonstrate various ways you can combine the vector functions we've discussed in previous chapters with the raster functions currently supported in WKT Raster.

### 13.6.1 Polygonizing rasters

One of the useful things you can do by combining raster functions with vector functions, is to polygonize rasters. This is for example useful if you have paper drawings of maps that you need to convert to vectors for doing various measurements and other massagings. We'll start off by showing you the function we created to polygonize Pele in the prior examples.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

This version of raster to polygon will integrate all the cells in Pele's body, but skipping every n cells across and down and using the sampled cell to represent the skipped cells. This n we call our sampling factor. It allows us to do a faster polygonization at the expense of accuracy. We consider all the cells that have values not equal to the nodatavalue for that band as part of Pele's body.

### **WKT RASTER ROADMAP**

Eventually you will be able to convert geometries to JPG and other kinds of rasters as well using the WKT Raster functions. Also on the road map list are functions for doing intersections between rasters and vectors as well as map algebra. As such many of these functions we are demonstrating will be unnecessary in later use cases.

All the functions that follow will be installed in new schema we created called rutility. We run the function as follows:

```
SELECT rutility.upgis_rastertopolygon(rast, ARRAY[1,2,3,4], 10)
  FROM ch13.pele;
```

What this will do is return a multipolygon that unions all the pixels in Pele's body that have at least one band in passed in bands that are not equal to the BandNoDataValue. What does this function look like:

#### **Listing 13.3 upgis\_rastertopolygon converting rasters to polygons**

```
--1
CREATE OR REPLACE FUNCTION rutility.upgis_rastertopolygon(param_rast
raster, param_bands integer[], param_sampling integer)
RETURNS geometry AS
$$
DECLARE
--2
var_rows integer := ST_Height(param_rast);
var_cols integer := ST_Width(param_rast);
var_pixsizex float := ST_PixelSizeX(param_rast);
var_pixsizey float := ST_PixelSizeY(param_rast);
var_pixpoly geometry :=
ST_MakeEnvelope(ST_UpperLeftX(param_rast),ST_UpperLeftY(param_rast),
                 ST_UpperLeftX(param_rast) + var_pixsizex*param_sampling,
                 ST_UpperLeftY(param_rast) + var_pixsizey*param_sampling,
                 ST_SRID(param_rast));
var_result geometry;
var_nodatavalue float[] := ARRAY(SELECT
                                     ST_BandNoDataValue(param_rast, param_bands[i])
                                     FROM generate_series(1, array_upper(param_bands,1)) As i
                                );
BEGIN
--3
  SELECT ST_Union(ST_Translate(var_pixpoly, x*var_pixsizex,
                                y*var_pixsizey))
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```

        INTO var_result
        FROM generate_series(1,var_cols,param_sampling) As x
            CROSS JOIN
            generate_series(1,var_rows,param_sampling) As y
        WHERE
        EXISTS(SELECT 1
            FROM
            generate_series(1, array_upper(param_bands,1)) As i
            WHERE ST_Value(param_rast, param_bands[i],x,y) <>
var_nodatavalue[i]);
        RETURN var_result;
END
$$
LANGUAGE 'plpgsql' IMMUTABLE;
-- 1 array list of bands passed in
-- 2 extrude common values we will need
-- 3 array of nodata values for selected bands
-- 4 integrate and return in var_result

```

(1) This particular version of our function takes in several arguments. First we pass in the raster object. One argument of importance is the list of bands we wish to consider the values for. (2) We store meta data values we will be reusing in easily accessible variables. PostgreSQL has built in support for arrays which comes in handy here. It allows us to store all the nodata values for the select bands in an array we call var\_nodatavalue. We are also using the function ST\_MakeEnvelope introduced in PostGIS 1.5 to represent a pixel rectangle we will use to integrate. The ST\_PixSizeX, ST\_PixSizeY and ST\_UpperLeft/ST\_UpperRight, and ST\_SRID are important for converting pixels to geometric coordinates. (3) Finally we union all pixel rectangles (using 1 pixel to represent value of every var\_sample) and only collect and union those in that for at least one selected band is not equal to the no data value for that band.

In the prior examples, we treated Pele as a one dimensional girl by collecting all her bands and coloring in the rectangle if any of those bands in that rectangle had data. She is however more complicated than that. She can respond on 4 different channels. Below are a couple of uses that generate different polygons by looking at each channel separately. We are also reducing our sampling to get crisper looking polygons. Below is an example that just considers the first band and samples one pixel for every 4x4 (16) pixels.

```

SELECT ST_AsBinary(rutility.upgis_rastertopolygon(
rast
, ARRAY[1], 4))
FROM ch13.pele;

```



Figure 13.6 Pele's first band considering all values that are not equal to the NoDataValue.

If you look at each band of Pele and aggregate all the pixels that have data, they are all the same except for the 4<sup>th</sup> band which seems to have no data. They all look as shown above. Generally when you do raster analysis, you want to select regions that contain a single value or a range of pixel values and that cover a specific region of space. Doing ranges will allow us to summarize in vector form regions of space which for example have high amounts of toxic gas or chemical in the soil or air. All this you can do by introducing range values and intersecting with regular geometries. You can think of the geometry you intersect with as the rule that defines what raster cells to consider in your final output.

### **ST\_Polygon and ST\_DumpAsPolygons alternative**

ST\_Polygon is a function that returns the polygon/multipolygon formed by unioning all the pixel values that are not the NoDataValue. It does what we have just described but faster if you don't want to selectively sample pixels.

ST\_DumpAsPolygons returns a set of single polygon, pixvalue pairs for a given raster band and relies on the GDAL library. In many cases this function will be easier and faster to use than going down to the level of the pixel using ST\_Value. The trade off is that it doesn't do sampling as we demonstrated so if you just need a rough answer, may be slower than the function we demonstrated, but you will get more accurate answers and faster if you need a more accurate answer. Here is a simple query that will polygonize select pixel ranges and give the average pixel value for that range.

```
SELECT ST_Union(geom),
       SUM(val*ST_Area(geom)) / SUM(ST_Area(geom)) As avgval
  FROM (SELECT (ST_DumpAsPolygons(rast)).*
    FROM ch13.pele
   ) As foo
 WHERE val BETWEEN 0 and 200;
```

In the next section we'll delve a little more into combining raster functions with PostGIS vector and PostgreSQL functions.

### **13.6.2 Getting a pixel value at a geometric point**

Currently the ST\_Value function takes a raster, a band number, a column, and a row and returns the pixel band value for that column and row. Often times you will want to pass in a point geometry and return the band value corresponding to the cell that the point falls in.

#### **MISSING FUNCTION: VALUE AT GEOMETRIC LOCATION**

Please note that the function we are about to demonstrate will soon be implemented in PostGIS WKT Raster. This function as a result will become obsolete, but we hope the concepts in the function will still be useful for understanding how to build your own custom functions.

#### **Listing 13.4 Get the approximate pixel band value at a geometric point**

```
CREATE OR REPLACE FUNCTION rutility.upgis_ptval(param_rast raster,
param_bnum integer, param_pt geometry)
RETURNS float AS $$

DECLARE
    var_result float = 0; var_rid integer;
    var_cols integer; var_rows integer; var_c integer; var_r integer;
    var_upx float; var_upy float; var_sizex float; var_sizey float;
BEGIN
    -- 1 does point intersect chosen tile
    SELECT ST_Width(param_rast), ST_Height(param_rast),
    ST_UpperLeftX(param_rast), ST_UpperLeftY(param_rast),
    ST_PixelSizeX(param_rast), ST_PixelSizeY(param_rast)
        INTO var_cols, var_rows, var_upx, var_upy, var_sizex, var_sizey
    WHERE ST_Intersects(ST_Envelope(param_rast), param_pt)
    LIMIT 1;
    -- 2 if no return null
    IF var_cols IS NULL THEN -- doesn't intersect
        var_result := NULL;
    ELSE
        -- 3 find row and column
        SELECT 1 + floor( ( (ST_X(param_pt) - var_upx) / (var_sizex*var_cols)
    ) *var_cols )::integer, 1 + floor( ( (ST_Y(param_pt) -
    var_upy)/(var_sizey*var_rows) ) *var_rows )::integer
        INTO var_c, var_r;
        -- 4 edge case
        IF var_c = 0 THEN var_c := 1; END IF;
        IF var_r = 0 THEN var_r := 1; END IF;
        IF var_c = var_cols + 1 THEN var_c := var_cols; END IF;
        IF var_r = var_rows + 1 THEN var_r := var_rows; END IF;
        -- 5 get cell value
        SELECT ST_Value(param_rast,param_bnum, var_c , var_r )
            INTO var_result;
        END IF;
        RETURN var_result;
    END $$
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```

LANGUAGE 'plpgsql' STABLE;
-- 1 does point intersect chosen tile
-- 2 if no return null
-- 3 find row and column
-- 4 edge case
-- 5 get cell value

```

In (1) we first determine if the point intersects our raster tile, if it doesn't (2) our work is done and no need to inspect further. If it does (3) When then figure out the percentage along the raster tile it is and multiply by height and width of the tile to get the row and column position. (4) Because of floating point aberrations we may fall outside of the tile in which case we return the upper or lower cell value. (5) Finally we get the pixel value from the raster tile.

To use this function we can then do a query as follows:

```

SELECT p.twin, rutility.upgis_ptval(k.rast,1,p.geom)
FROM ch13.kauai As k INNER JOIN
(SELECT twin, ST_Centroid(ST_Envelope(rast))
 As geom FROM ch13.pele_in_kauai) As p
ON ST_Intersects(ST_Envelope(k.rast), p.geom)
ORDER BY p.twin;

```

Which will give us an answer of: pele 1: 529, pele 2: 2, pele 3: 1299

### **13.6.3 Adding additional attributes to raster records**

One of the nice features about having raster data in the database, is that you can attach on various additional attributes to each record. You may want to attach on where you got the data or even what rows can be safely ignored for most processing purposes.

In the Kauai data file we imported, after cutting up, a good chunk of the tiles are just water around Kauai and not terribly useful to us. We may want to keep these to maintain even blocking, but we probably don't care to use them for vector processing or other kinds of analysis. One easy way to keep them and yet flag them as not useful or not useful for certain kind of operations is to add another attribute. For this example we'll add another column we will call `is.filler`.

```
ALTER TABLE ch13.kauai ADD COLUMN is.filler boolean;
```

Now we are going to mark the tiles as filler or not. The first thing we can tell by looking at the tiles overlaid with the raster is that tiles that have all pixel values of 0 are junk. We can in fact consider more than one value of pixels as junk so we'll consider pixel values 0-2 as junk. So we do this first update which samples 1 pixel for every 20x20 pixels in a raster tile to find one with a pixel value greater than 2 and those ones we know have information. This updates 400 tiles.

```

UPDATE ch13.kauai
SET is.filler = false
WHERE EXISTS

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```
(SELECT 1
  FROM generate_series(1, ST_Width(rast),20) As X
  CROSS JOIN generate_series(1, ST_Height(rast),20) As y
  WHERE ST_Value(rast,1,x,y) > 2 );
```

Then we mark the rest as filler.

```
UPDATE ch13.kauai SET is_filler = true
where is_filler is null;
```

Our filled in section represents the tiles with information.

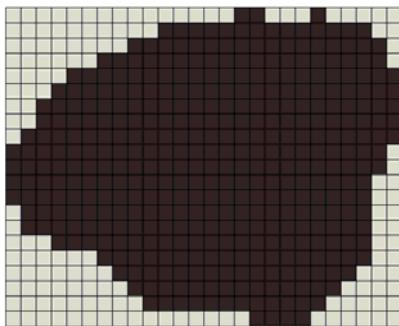


Figure 13.7 Dark Tiles represent tiles that have pixels with information

Now that we have demonstrated some simple exercises using raster, we will demonstrate how to use raster in real world modeling.

#### **13.6.4 Using raster with vector to solve real world problems**

For this next section, we are going to demonstrate how to use PostGIS geometry/geography spatial functions in conjunction with raster functions for modeling. These next set of exercises require PostGIS 1.5+ (since we are using geography type) and PostgreSQL 8.4+ since we are using some CTEs. We will also be reusing our utility.upgis\_ptval function.

Imagine that we are a railroad company that is planning the best trails to lay down our rail road tracks for a given start and end city destination. For our model, we clipped a very zoomed out National Elevation dataset (NED) relief from a PDF we generated using the USGS National Map Seamless Server <http://seamless.usgs.gov/>. You can read the details of how we massaged it so we could use it for our very rough modeling in the code ReadMe.txt.

Our NED relief looks as shown and it is in WGS 84 long lat (SRID: 4326). We have the option of keeping it in this projection or reprojecting it using GDAL Warp. For this exercise, we will keep it in its native projection and use the geography data type (introduced in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

PostGIS 1.5) to do our measurements. Although it has 4 bands, most have more or less the same value. For our example, we are just going to analyse the first band.

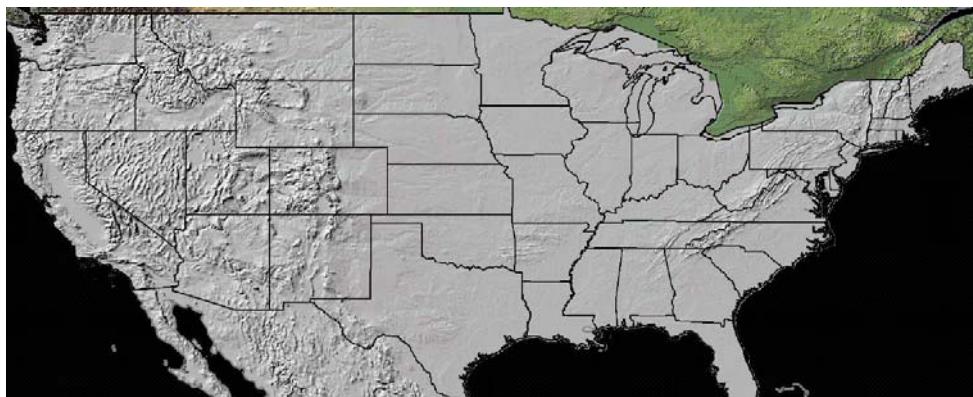


Figure 13.8 The shaded NED relief we will use to determine terrain elevations for our trip simulation

We will use this raster data to determine rough elevations along our simulated paths at each step of the way. Keep in mind that although we are using this for determining slopes (changes in elevation), you can use it for any kind of observations you store in pixel bands or even multiple factors such as wind grassiness if you want to avoid dessert etc.

The basic idea is that you try to minimize distance and also minimize the sloping. We will penalize more for going up (increasing elevation) and less for going down. We will also penalize for the longer the distance. The weights we use for these are a bit fuzzy and even our map is not the greatest of maps to use for this kind of analysis. If you have sturdier trains that can withstand greater slopes you can penalize less for slope change than we are doing.

We start off by setting up a table to hold our travel constants. Each record will define a start and end point location and various other knobs we want to tweak in our simulation.

```
CREATE TABLE ch13.travels(travel_id integer PRIMARY KEY,
    title text, pt_start geography(POINT(4326)),
    pt_end geography(POINT,4326),
    path geography(LINESTRING,4326),
    max_dev_m integer, move_inc integer, max_duration interval);
```

- **travel\_id** - Just a random number we will use to denote the simulation.
- **pt\_start, pt\_end** - Our starting and ending points. Note we are using geography which uses a spheroidal model instead of geometry. This will allow us to extend to cover the whole world if we want.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

- **path** - This is a place holder for our path. It starts out NULL and our simulation will fill in the optimal path once it has gotten to the target location or it runs out of time.
- **move\_inc** - This is the number of meters we want to move for each simulation step. We are keeping this constant for simplicity.
- **max\_dev\_m** - Max number of meters we will allow our path to deviate from the straight path
- **max\_duration** - This is the maximum amount of time we allow our simulation to run. If our simulation still hasn't achieved an optimal path in the allotted time, it will store how far it has gotten in the path field.

#### **PIXEL VALUE SAMPLING**

We first create a function to do elevation samplings that will take a target geography point and will return the pixel value at this location. For the sake of argument, we will assume this represents the elevation at that location.

#### **Listing 13.5 Getting pixel value at a travel point**

```
CREATE OR REPLACE FUNCTION ch13.travel_pixval(param_pos
geography(POINT,4326))
RETURNS float AS
$$
DECLARE
    var_result float = 0;
BEGIN
    SELECT rutility.upgis_ptval(k.rast, 1, geometry(param_pos)) As val
    INTO var_result
    FROM ch13.usdem AS k
    WHERE ST_Intersects(geometry(ST_Envelope(k.rast)), param_pos)
    LIMIT 1;
    RETURN var_result;
END
$$
LANGUAGE 'plpgsql' STABLE;
```

In this example we simply take the pixel value for band 1 at first raster tile that intersects our point.

#### **SIMULATIONS**

For this next part we are going to create a function that tries various paths along the way and at each step of the way picks the best next step and adds it to its accumulating path.

Our objectives

- Never cross our current traveled path to prevent from running around in circles
- Always move closer to our desired end spot and optimize for shortest distance
- minimize on bumpiness (differential in slope between where we are and our next move)

To achieve our stated objectives, we have this function which does the simulation.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

### Listing 13.6 Travel Planner function to minimize on slope and distance

```

CREATE OR REPLACE FUNCTION ch13.travel_planner(param_travel_id integer)
RETURNS text AS
$$
DECLARE
    var_starttime timestamp with time zone := clock_timestamp();
    var_result text := ' Start ' || var_starttime;
    :
BEGIN
    SELECT * INTO var_tv FROM ch13.travels WHERE travel_id =
param_travel_id;
    IF var_tv.path IS NOT NULL THEN
        var_curpos := geography(ST_EndPoint(geometry(var_tv.path)));
    ELSE
        var_curpos := var_tv.pt_start;
    END IF;
    var_curpv := ch13.travel_pixval(var_curpos);
    -- 1
    var_corridor := ST_Buffer(geography(ST_MakeLine(geometry(var_tv.pt_start),
geometry(var_tv.pt_end))), var_tv.max_dev_m);
    -- 2
    WHILE (var_starttime + var_tv.max_duration) > clock_timestamp() AND
ST_Distance(var_curpos,var_tv.pt_end) > 1000    LOOP
    -- 3
        var_circle := geometry(ST_Buffer(var_curpos, var_tv.move_inc)) ;
        WITH pos_runs AS
            (SELECT geography(foo.geom) As pot_loc FROM
ST_DumpPoints(var_circle) As foo WHERE foo.path[2] < ST_NPoints(var_circle)
AND ST_Intersects(foo.geom, var_corridor) AND (var_tv.path IS NULL OR NOT
ST_Crosses(ST_MakeLine(geometry(var_curpos),foo.geom),
geometry(var_tv.path))));
        pos_runs_pd AS
            (SELECT pot_loc, ch13.travel_pixval(pot_loc) As pixval,
ST_Distance(pot_loc, var_tv.pt_end) As dist_end
            FROM pos_runs
            WHERE ST_Distance(pot_loc, var_tv.pt_end) < ST_Distance(var_curpos,
var_tv.pt_end) )
        -- 4
        SELECT pot_loc
        INTO var_tarpos
        FROM pos_runs_pd
    -- 5
        ORDER BY CASE WHEN pixval > var_curpv THEN (pixval - var_curpv)*100
ELSE (var_curpv - pixval)*20 END + dist_end*0.1 LIMIT 1;

        IF var_tv.path IS NULL THEN
            var_tv.path := geography(ST_MakeLine(geometry(var_curpos),
geometry(var_tarpos)));
        ELSEIF var_tarpos IS NOT NULL THEN
            var_tv.path := geography(ST_AddPoint(geometry(var_tv.path),
geometry(var_tarpos)));
        ELSE
            EXIT;
        END IF;
    END WHILE;
END;
$$

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```

    var_cupos := var_tarpos; var_loops := var_loops + 1;
    var_curpv := ch13.travel_pixval(var_cupos);
END LOOP;
--6
UPDATE ch13.travels SET path = var_tv.path WHERE ch13.travels.travel_id
= var_tv.travel_id;
var_result := var_result || ' To End: ' || clock_timestamp() || ' in
loops: ' || var_loops || ' Duration %' || age(clock_timestamp())
, var_starttime) ;
RETURN var_result;
END
$$
LANGUAGE 'plpgsql' VOLATILE;
-- 1 keep path within buffer of straight
-- 2 loop til 10 m of destination or time out
-- 3 test points around a circle
-- 4 find best new point
-- 5 penalize more for going up than going down
-- 6 update path with accumulated

```

The travel planner (1) creates a corridor around the shortest path between our start and end which it uses to ensure we don't deviate too much (2) we keep adding points to our path until we reach our destination or we run out of time (3) To determine next points to test, we approximate a circle around our current point (of radius the distance we want to travel from current) and then we test those points that make up the boundary (note this is 8\*4 points for each loop since we are using default 8 points to approximate a quarter circle). Of these 32 points, we only keep those that would result in an additional line path that does not cross our current path and that is still within our corridor. (4) From the points that are left, we pick the point that based on our weighting has the lowest slope relative to where we are now and shortest distance to our destination (5) penalizing more for going up than for going down. (6) We end by updating the travel record with our full path.

### **CASTING BETWEEN GEOGRAPHY AND GEOMETRY**

We do a lot of back and forth casting between geography and geometry because most functions are not yet supported for geography, but those functions are fairly safe to use for geodetic data so you could create wrappers for them if you use them a lot and save some code. In later versions of PostGIS, this ugly back and forth casting will probably not be necessary.

To test our function, we insert these two routes into our table:

```

INSERT INTO ch13.travels(travel_id, title,
    pt_start, pt_end, max_dev_m, move_inc, max_duration)
VALUES(2, 'SF CA to Reno NV',
ST_GeogFromText('POINT(-122.6837.75)'), ,
ST_GeogFromText('POINT(-119.78 39.5)'), 50000, 1000,
interval '10 minutes');

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

We run with `SELECT ch13.travel_planner(2);`  
 Which outputs: Start 2010-05-03 00:52:01.218-04 To End: 2010-05-03 00:52:26.515-04  
 in loops: 419 Duration 00:00:25.297

The path generated is shown in the next figure.

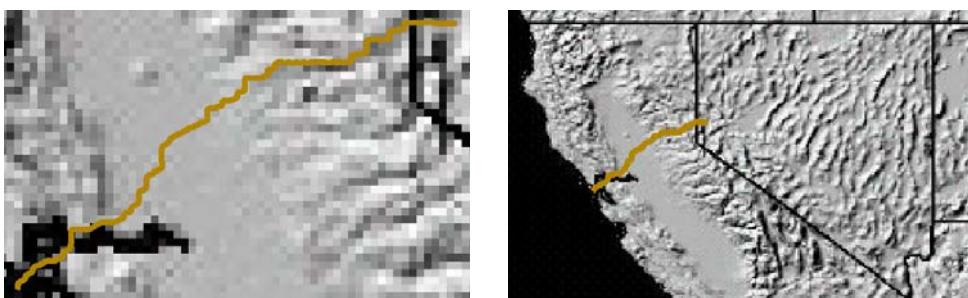


Figure 13.9 Demonstrates how the pixel formation affects the path generated by travel planner

If you use PostGIS to store your raster data, you may want to export subsections of it or show it on a map. In the next sections, we'll demonstrate how to export PostGIS raster data into various formats as well as show it on a map.

### 13.7 Exporting Raster data into other raster formats

The GDAL library version 1.7.1+ has a driver called "PostGIS WKT Raster". You can enable this during compile by compiling with

`--with-pg=path/to/pg_config`

If you are on Windows, you can get already pre-built binaries. The FW Tools package we discussed in Loading Real data chapter includes this driver from version 2.4.6 on.

There is also a nightly build of gdal that has the newer trunk version (currently 1.8) with all the improvements being added to the driver maintained by Tamas Szekeres <http://vbkto.dyndns.org/sdk/>. These ones are built nightly and contain the latest and greatest of gdal, mapserver, and python bindings for windows so are probably the best to use to get the newest changes for PostGIS WKT Raster.

To verify you have the driver compiled in your version of the binaries do:

`gdalinfo --formats`

The driver only supports regularly blocked PostGIS rasters though that will be enhanced with time.

To get basic information about our kauai raster type we do a:

```
gdalinfo "PG:host=localhost port=5432 dbname='postgis_in_action'  
user='postgres' password='whatever' schema='ch13' table=kauai"
```

Which gives us output as shown below:

Driver: WKTRaster/PostGIS WKT Raster driver

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

```

Files: none associated
Size is 5174, 4169
Coordinate System is:
PROJCS[ "NAD83 / UTM zone 4N",
    GEOGCS[ "NAD83",
        DATUM[ "North_American_Datum_1983",
            SPHEROID[ "GRS 1980", 6378137, 298.257222101,
:
:
Origin = (418205.0000000000000000, 2459785.0000000000000000)
Pixel Size = (10.00000000000000, -10.00000000000000)
Image Structure Metadata:
    INTERLEAVE=BAND
Corner Coordinates:
Upper Left  ( 418205.000, 2459785.000) (159d47'37.54"W, 22d14'29.81"N)
Lower Left   ( 418205.000, 2418095.000) (159d47'29.98"W, 21d51'54.00"N)
Upper Right  ( 469945.000, 2459785.000) (159d17'30.02"W, 22d14'35.84"N)
Lower Right  ( 469945.000, 2418095.000) (159d17'27.24"W, 21d51'59.92"N)
Center       ( 444075.000, 2438940.000) (159d32'31.20"W, 22d 3'15.59"N)
Band 1 Block=200x200 Type=UInt16, ColorInterp=Undefined
    NoData Value=0

```

In order to do output, you would use either gdal\_translate or gdalwarp. Gdal\_translate is a command line tool packaged with gdal that will convert from one raster format to another and will also output to various resolutions. Gdal\_warp is a tool that can also convert from one raster format to another but also does a spatial coordinate transformation. We'll demonstrate these in this section.

### **13.7.1 Gdal\_translate basics to convert to other formats**

To export our data into some other raster format, we use gdal\_translate:

#### **Listing 13.7 Exporting Raster data from PostGIS raster type**

```

--1
gdal_translate -of PNG -outsize 10% 10% PG:"host=localhost
dbname='postgis_in_action' user='postgres' password='whatever'
schema='ch13' table=kauai" kauai_small.png

--2
gdal_translate -of GTiff PG:"host='localhost' port='5432'
dbname='postgis_in_action' user='postgres' password='whatever'
schema='ch13' table='kauai' where='rid BETWEEN 1 and 200'" subset.tif

--3
gdal_translate -of GTiff PG:"host='localhost' port='5432'
dbname='postgis_in_action' user='postgres' password='whatever'
schema='ch13' table='kauai'
where='ST_Intersects(rast,(SELECT ST_Union(ST_Envelope(p.rast)) As pgeom
FROM ch13.pele_in_kauai))'" pelespots.tif

```

**--1 export at 10% of original size**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

--2 export select rows  
--3 export parts with Pele clones

In (1) we export all of kauai table into a single file but shrunk to 10% $\times$ 10% of original size and export as a PNG. (2) We only export raster rows 1-200 into a single GeoTiff image. (3) We do intersects with out pele\_in\_kauai to get only those tiles that contain Pele clones.

The GDAL PostGIS WKT Raster driver makes it possible to export PostGIS raster data out to many of the formats that GDAL supports. As a side benefit of this, it also makes it possible for tools that build on GDAL to view this data without need for export. In the next section, we'll demonstrate how to view raster data using MapServer. What makes this interesting is that no change to MapServer was needed to accomplish this. Since MapServer is built with GDAL/OGR, it natively reads any data supported by the compiled in GDAL/OGR driver. In the next section, we'll demonstrate how to define a MapServer PostGIS WKT Raster layer.

### **13.7.2 Gdalwarp to transform from one spatial ref to another**

If we had our data in a space preserving spatial reference system such as US National Atlas Equal Area meters, we would have lost some precision, but saved ourself some time casting back and forth from geography to geometry.

You can use the GDAL tool kit to transform from one spatial reference system to another using the gdalwarp binary. For raster, this terminology is often referred to as Warping instead of transformation, but it means more or less the same thing.

As of this time the PostGIS WKT Raster only supports reading, not writing and no functions as of yet for raster type to transform the spatial reference system. As a result you can't as of this time directly transform PostGIS raster without going thru an intermediary step.

#### **REPROJECTING DATA BEFORE LOAD**

If we did this initially, we would have warped our US.tif raster before loading. Below is a snippet of code to warp our original US.tif to US national atlas equal area meters. Gdalwarp is capable of taking either EPSG codes or a proj4text transformation expression. To get the proj4text, we looked up the proj4text field for SRID = 2163 in our spatial\_ref\_sys table.

```
gdalwarp -s_srs "EPSG:4326" -t_srs "+proj=laea +lat_0=45 +lon_0=-100 +x_0=0 +y_0=0 +a=6370997 +b=6370997 +units=m +no_defs" US.tif US_laea.tif
```

Which transforms our original US map to:

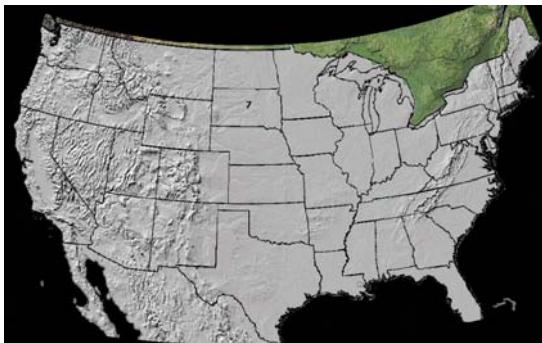


Figure 13.10 Our original WGS 84 map after being warped to NA LAEA (SRID:2163)

#### **EXPORTING RASTER DATA IN DIFFERENT PROJECTION**

Although you can't use gdalwarp to reproject PostGIS raster data, you can use it to export the data in a different projection. For this next example, we are going to export a subset of our data and transform it in one command. For this example you need GDAL 1.8+ compiled with PostGIS WKT Raster support. The GDAL 1.7.1+ will run, but is a bit buggy and cuts off a good chunk of the image.

#### **EPSG VS PROJ4**

The gdalwarp -t\_srs is the same as what we had in previous example except we are using the EPSG code instead of the proj4text code. Gdalwarp will accept both forms, but the EPSG version requires that your gdal\_data path environment variable is set and you have the epsg to proj4 mappings available for it to lookup the proj4text. As such writing out the proj4 is more verbose but more likely to work and guaranteed to match with what you have in spatial\_ref\_sys.

```
gdalwarp -s_srs "EPSG:4326"
-t_srs "EPSG:2163"
PG:"host='localhost'      port='5432'      dbname='postgis_in_action'
user='postgres'    password='whatever'    schema='ch13'    table='usdem'
where='ST_Intersects(ST_Envelope(rast),ST_MakeEnvelope(-115.60,32.54,
-112.96, 26.03,4326))' " usdem_sub.tif
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

While it is great to be able to analyze raster data programmatically, it is also nice to be able to view your data to spot check it. Right now there are few tools that can view PostGIS raster data directly. MapServer is one tool that can currently. We will go over how to use Mapserver to do that in the next section.

## 13.8 Viewing raster data with MapServer

If you are using GDAL 1.7+ with the PostGIS WKT Raster driver enabled, you should be able to view PostGIS raster layers via Mapserver. Below is an example layer.

### **Listing 13.8 Example Mapserver PostGIS WKT Raster Layer**

```
LAYER
  NAME kauai
  TYPE raster
  STATUS ON
  DATA "PG:host=localhost port=5432 dbname='postgis_in_action'
user='postgres' password='whatever' schema='ch13' table='kauai'

  PROJECTION
    "init=epsg:26904"
  END
  PROCESSING "NODATA=0"
  PROCESSING "SCALE=-100.5,100.5"
  PROCESSING "SCALE_BUCKETS=201"
  METADATA
    ows_title "Kauai Elevations"
    gml_include_items "all"
    ows_featureid "rid"
    "wms_srs"      "EPSG:4269 EPSG:4326 EPSG:26904 EPSG:3785 EPSG:900913"
    "wfs_version"  "1.0.0"
    "wfs_srs"      "EPSG:900913"
  END
  CLASS
    NAME "red"
    EXPRESSION ([pixel] < 10)
    COLOR 255 0 0
  END
  CLASS
    NAME "green"
    EXPRESSION ([pixel] >= 10 AND [pixel] < 15000 )
    COLOR 0 255 0
  END
  CLASS
    NAME "blue"
    EXPRESSION ([pixel] >= 15000 )
    COLOR 0 255 0
  END
END
```

### **Mapserver display limitations**

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=XYZ>

The Mapserver implementation relies on the PostGIS WKT Raster GDAL driver. Since the driver can't currently export irregularly blocked rasters, they will either show up as black squares on the map or not at all. This is an issue in the GDAL PostGIS WKT Raster that is currently being improved on. You can expect to have this ability in the very near future.

Although MapServer is the only tool as of this time we have tested that can directly read PostGIS raster data type, we expect other tools, particularly ones that are built with GDAL to follow shortly. One such tool we expect to see support soon is QuantumGIS. QuantumGIS already sports an Oracle GeoRaster plugin which piggy back's on using the GDAL driver for rendering Oracle GeoRaster. We expect a similar plugin can be created fairly easily for PostGIS WKT Raster since the GDAL connection conventions for Oracle GeoRaster and PostGIS WKT Raster are fairly similar.

## **13.9 The future of PostGIS Raster support**

Although the PostGIS WKT Raster project is still a young project, we hope we have presented enough here so you can get a feel for what it can do for you now. We also hope you are as impressed as we are at the amount of functionality and speed it has already achieved.

In the examples above we demonstrated building additional helper functions with the functionality already present in PostGIS with raster support and geometry/geography functions. In the future, many of these tricks will be unnecessary.

The full set of milestones is itemized here

<http://trac.osgeo.org/postgis/wiki/WKTRaster/PlanningAndFunding#OtherImportantMilestones>

As of this writing, the current 0.1.6 series of milestones is being finished off and about 70-80% there.

### **13.9.1 What is left in milestone 0.1.6**

#### **ST\_SetValue**

The ST\_SetValue and other related functions that will allow you to edit raster data directly in the database are currently missing.

#### **INTERSECTS AND INTERSECTIONS**

In future versions of the raster functions, you can expect to see functions such as ST\_Intersection, ST\_SelectByValue, etc. that will allow for both returning geometries as well as intersections of rasters. With this new animal you can create cookie cut rasters by clipping existing rasters with geometries. You can also create intersections that return polygons created by raster band ranges and intersection with other polygons and rasters, with much less effort and better performance than we demonstrated here.

**RASTER OUTPUT FORMATS**

ST\_Band, ST\_AsJPEG, ST\_AsTiff, ST\_AsPNG are output functions in the works that will return binary data (bytea). The ST\_Band function will return a 1 band raster by selecting a specific band.

The ST\_AsJPEG,Tiff,PNG will return raw images of those types. These will be useful for displaying raster portions without any need for additional tools. These functions will also allow you to rasterize geometries since they will support both geometry and raster types. This means you can conceivably create a mapping server directly in PostGIS without need for anything else.

**GDAL DRIVER**

Currently the PostGIS WKT Raster GDAL driver can only export to other raster formats. You need to use gdal2wktRaster to import. This driver is being continually enhanced and will be changed to include ability to import raster data into PostGIS without need for Python.

It also has a couple of rough spots that need to be ironed out when dealing with exporting irregularly blocked rasters.

**TOOLKITS SUPPORTING POSTGIS WKT RASTER**

Many API tools build on top of GDAL. Given this fact, these API tools are already capable of leveraging PostGIS raster data if their GDAL library is compiled with PostGIS WKT Raster support. Tools in this family are PyGDAL for interfacing with Python, Rgdal for interfacing with R which we covered earlier, SharpMap.Net for interfacing with the .Net Framework. Although we have not tested these except for PyGDAL, we expect this to be just a compile away against the latest GDAL 1.8+ source.

**13.9.2 Milestone 0.2.4****VIEWING TOOLS**

Currently MapServer is the only tool tested that can render PostGIS WKT Raster. Most of the other tools that leverage GDAL such as QGIS you should see support in the menu options for viewing PostGIS WKT Raster, similar to what you can do with Oracle GeoRaster.

**AGGREGATE FUNCTIONS**

In milestone 0.2.4 you'll begin to see aggregate functions that work natively with rasters. Aggregates on the list are ST\_Union and ST\_Accum. Right now these functions do work with rasters, because of the silent casting from raster to polygon that is currently present but treat each raster as a square geometry rather than a raster.

**RASTER2PGSQL**

This is a companion to shp2pgsql. It is a lighter weight importer that will be distributed with PostGIS. It will only handle JPEGs and TIFFs but will not have any heavy dependencies.

### 13.9.3 Later Milestones

Of most interest in the later milestones are functions that allow for more sophisticated analysis and also allow you to do much of the functionality on rasters directly that you can do on geometries.

#### GEOMETRY LIKE OFFERINGS

ST\_Area, ST\_Centroid, ST\_Count, ST\_Transform

#### RASTER SPECIFIC OFFERINGS

- ST\_Resample -- will allow you to recompute pixel size and origin based on 'NEAREST NEIGHBOR', 'LINEAR', and 'BICUBIC' algorithms
- ST\_SelectByValue(raster|geometry, 'expression') - allow for selecting pixels based on band value or intersection with a geometry.
- ST\_MapAlgebra(raster|geometry, [raster|geometry,...], 'mathematical expression', 'raster' |'geometry') -> raster/geometries - allow to do mathematical calculations using map algebra expressions.

[http://www.quantdec.com/SYSEN597/GTKAV/section9/map\\_algebra.htm](http://www.quantdec.com/SYSEN597/GTKAV/section9/map_algebra.htm)

## 13.10 Summary

In this chapter we demonstrated how to use rasters in PostGIS with the PostGIS WKT Raster functions and new raster data type. These examples built on top of what we have already learned to do with PostGIS vector operations. We demonstrated the fluidity with which geometry and raster functions can interoperate and how the already present geometry functions enhance the power of using raster functions.

The future of PostGIS is bright and exciting. There will be integration of raster as demonstrated here. Also planned for the PostGIS 2.0 series is more support for curved geometries, 3d surfaces, and export options for Polyhedral surfaces. All these new developments will help extend the reach of PostGIS from its humble GIS beginnings to a multimedia tool bonanza well suited for virtual modeling, simulations and other kinds of physical science and engineering analysis.

# Appendix A:

## *Additional Resources*

This Appendix covers

- PostGIS focused Tutorials, Sites and Blogs
- Open Source Tools and Offerings
- Sources of Free Data
- Information on spatial reference systems
- Commercial offerings that support PostGIS

This appendix includes links to resources useful for PostGIS users of all walks of life. Some of these links we may have already covered in various chapters but they're also listed here so you can have all of these resources in one place.

### **A.1 PostGIS focused tutorials and sites**

This section lists listings of good tutorials as well as sites focused on PostGIS content.

#### **A.1.1 Getting started tutorials**

This section lists a compendium of tutorials accumulated over the years. We try to list the most up to date ones first.

- FOSS4G 2009 Sydney Australia -- Introduction to PostGIS by Mark Leslie complete with sample data and instructions with viewing in uDig.  
<http://revenant.ca/www/postgis/workshop/>

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- BostonGIS - Part 1: Getting Started With PostGIS: An almost Idiot's Guide -

This is mostly geared to the windows user and covers how to install PostGIS, load data and quick-use examples. It is still useful to Mac and Linux users since it covers a few basics about loading and using which are pretty much the same across all Oses.

[http://www.bostongis.com/PrinterFriendly.aspx?content\\_name=postgis\\_tut01](http://www.bostongis.com/PrinterFriendly.aspx?content_name=postgis_tut01)

- OpenGeo's Open Source stack tutorial -- includes setting up PostGIS, Mapserver, Geoserver, QuantumGIS and creating an application. Also include link to download the stack and tutorial data <http://workshops.opengeo.org/stack-intro/> and PostGIS specific tutorial <http://workshops.opengeo.org/postgis-spatialdbtips/>
- Foss4g2007 – Paul Ramsey - Introduction to PostGIS

These are powerpoint and data documents for the Intro Workshop given by Paul in FOSS4G 2007 conference. This is complete with using UMN Mapserver and Canadian data examples

<http://www.foss4g2007.org/workshops/W-04/>

- Paolo Corti - Installing PostGIS on Ubuntu

Paolo goes thru not only how to install PostGIS and PostgreSQL on Ubuntu, but also how to install QGIS, uDig and gvSig. He also covers a little about creating databases, users, and roles in PostgreSQL, as well as a quickie tour of QGIS, uDig, gvSig. Its worth a read even if you are not using Ubuntu.

<http://www.paolocorti.net/public/wordpress/index.php/2008/01/30/installing-postgis-on-ubuntu>

- Webb Sprague's PostgreSQL 2007 talk on PostGIS complete with slides, audio and code <http://www.postgresqlconference.org/2007/talks/>
- Lincoln Ritter - [Installing PostgreSQL, PostGIS and more on OS-X Leopard](http://www.lincolnritter.com/blog/2007/12/04/installing-postgresql-postgis-and-more-on-os-x-leopard/)  
<http://www.lincolnritter.com/blog/2007/12/04/installing-postgresql-postgis-and-more-on-os-x-leopard/>

### **A.1.2 Important GIS sites**

- OSGEO - OSGEO is the foundation that spearheads and cradles many open source GIS projects. <http://www.osgeo.org/>
- Open Geospatial consortium (OGC) - is the body that defines standards for interoperability between GIS products that are both open source and commercial. They define data portability, web service standards, and spatial sql standards. <http://www.opengeospatial.org/>
- PostGIS main site - <http://www.postgis.org/> (<http://postgis.refractions.net> )
- PostGIS User Wiki and Bug Tracker - <http://trac.osgeo.org/postgis>

© Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- PostgreSQL main site - <http://www.postgresql.org>
- Free GIS - <http://www.freegis.org> this is a site put together by folks at Intevation GmbH (<http://intevation.net/>) and is a directory listing of free and open source gis software, data, documents and projects.
- Spatial Reference Org - <http://spatialreference.org> - this is an invaluable site for looking up spatial reference systems. It is so important we are listing it twice.

### **A.1.3 Noteworthy PostGIS blogs and sites**

The blogs and sites in this list are blogs that are high in PostGIS material and helpful tips and tricks or are just too good on their overall breath of GIS not to be mentioned.

- PostGIS in Action book site <http://www.postgis.us> - This is a site we have set up where you can download data and code discussed in this book. We will also be posting our presentations, chapter summaries, other chapter related information , links and demos.
- Paul Ramsey - One of the original co-developers of PostGIS and many think of as the face of PostGIS. He does a fair amount of blogging about what's going on in PostGIS land. More specifically, he focuses on open source GIS and how it fits into the overall GIS ecosystem. He is also a member of the PostGIS steering committee and the 2008 recipient of the Sol Katz Award for Geospatial Free and Open Source leadership. He is also a contribute to Mapserver and founder of uDig desktop kit <http://blog.cleverelephant.ca>
- Martin Davis aka (Dr. JTS) - Martin is the lead architect behind Java Topology Suite (JTS) which GEOS (Geometry Open Source) is a C++ port of. A lot of the great geometry manipulation algorithms you will find in PostGIS and other commercial and open source packages that rely on GEOS and JTS, are due a large part to his efforts. He blogs about some of the algorithms behind these processes as well as general GIS and sometimes just interesting random technology topics. <http://lin-ear-thinking.blogspot.com/>
- Simon Greener – Spatial DbAdvisor  
  
If you want to see how its done in other spatial databases such as Oracle and SQL Server and other tools such as Manifold as well as some extra tips for getting the most out of PostGIS, then Simon's your man.  
  
His blog is full of freely available functions he has written to explore spatial sql in all its beautiful forms. He has functions for PostGIS, Oracle Locator/Spatial, SQL Server 2008 Spatial, and Manifold <http://www.spatialdbadvisor.com>
- California Soil Resource Lab - Dylan Beaudette - this site largely authored by Dylan is full of PostGIS/R/GRASS/GDAL tutorials. It's a must read for anyone doing analytical work with these tools. <http://casoilresource.lawr.ucdavis.edu/drupal/blog/2>
- Bill Dollins – GeoBabble

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Bill's blog is one of the best in terms of its breadth of examples of interoperability between commercial and open source GIS. He blogs about SQL Server 2008 spatial, PostGIS, SpatiaLite and integration of other open source GIS with other commercial GIS primarily ESRI ArcGIS. He is also one of the developers of ZigGIS (a Plug-in for ArcGIS Desktop 9.1 and above for editing and displaying PostGIS data) along with Abe Gillespie and Paolo Corti.

<http://geobabble.wordpress.com/>

- Thinking in GIS - Paolo Corti

I don't think there is any web gis particularly of an open source nature that Paolo hasn't tried and often times blogged about with helpful tutorials. His site contains everything from ArcGIS, using PostGIS in ArcGIS, and open source topics such as using GeoDjango, KML Overlays, UMN Mapserver, Ruby on Rails, TileCache and OpenLayers, and even NoSQL databases. Also lots of useful tutorials on installing these packages and getting up and running on Ubuntu

<http://www.paolocorti.net/>

- Postgres OnLine Journal

Check out our more or less monthly journal (available as individual articles on line, full month in html, and in full month in PDF format). It covers, general PostgreSQL tips and tricks such as doing automated backups, Integrated Tsearch Full-Text engine as well as PostgreSQL integration with other tools such as MS Access, Open Office, Web Programming (PHP, ASP.NET, Flex), and comparisons of different databases and administration tools.

<http://www.postgresonline.com/>

- Boston GIS - this is our other satellite site focused on OpenGIS standards and open source GIS. We try to pack a lot of PostGIS tutorials in here, but also provide tips, tricks, and tutorials on other open source GIS and OpenGIS concepts. You'll find not only PostGIS here, but tutorials on SpatiaLite, SQL Server 2008, SharpMap.NET, PL/R and various cheat sheets we've developed over the years. The theme of the site centers around using Boston data to demonstrate spatial concepts.  
<http://www.bostongis.com>

- Nicklas Avén - Nicklas is a new core PostGIS developer. He made major contributions to the distance functions in PostGIS 1.5. He improved the efficiency of the existing on large geometries and also introduced some new ones such as ST\_MaxDistance, ST\_ClosestPoint, ST\_LongestLine, ST\_ShortestLine and various others. In his blog, he chronicles some of his thought processes in adding to PostGIS code base.  
<http://blog.jordogskog.no/>

- Mateusz Loskot - Mateusz is a core GEOS developer and WKT Raster developer. He blogs a lot about various Geo processing kits, packaging and also about PostGIS and WKT Raster. <http://mateusz.loskot.net/>

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- Sandro Santilli aka (strk) - Sandro is a long time PostGIS, WKT Raster and GEOS developer. He is responsible for integrating much of the GEOS functionality you find in PostGIS. His blog content is both technical - PostGIS, OpenStreetMap as well as familial things like bats. <http://strk.keybit.net/blog/>
- James Fee - no GIS site list would be complete without James, the king of GIS blogging. James is all over the map from commercial GIS to open source GIS to GIS data services and combining all. He still manages to throw in a bit of PostGIS as well. He's not afraid to give a candid view of what he thinks is hot and what is not. He is probably the most read GIS blogger around. He is also the maintainer for Planet Geospatial. <http://www.spatiallyadjusted.com/>
- Planet OSGEO - <http://planet.osgeo.org/> This is a blog aggregator of OSGEO community bloggers. Many key OSGEO movers and shakers can be found in this list. Lots of Project community blogs like Quantum GIS and OpenLayers team. Good for staying abreast of OSGEO projects.
- Planet Geospatial - <http://www.planetgs.com/> This is an aggregator of the more popular and up and coming GIS focused blogs and news sites.

#### A.1.4 Noteworthy R, PL/R sites and Newsgroups

These are sites rich in R and PL/R content

- PL/R official site - this is where you can download the source and binaries for PL/R PostgreSQL language handler. You can also subscribe to the PL/R mailing list from here. <http://www.joeconway.com/plr/>
- PL/R Wiki - <http://www.joeconway.com/web/guest/plr> as of this writing, this is a work in progress that already contains useful install manuals and snippets of PL/R code. The main PL/R page will eventually be merged in here.
- R - <http://www.r-project.org/> This is where you can download the R software package and basic tutorials on R.
- California Soil Resource Lab - <http://casoilresource.lawr.ucdavis.edu/drupal/blog> these pages are chuck full of GIS related R scripts as well as PL/PgSQL scripts. Keep in mind that although many of the R scripts are raw R, they can be easily flipped into PL/R scripts with minor changes.
- Quick-R - <http://www.statmethods.net/> Lots of snippet R recipes you can cut and paste from. This is authored by ,Robert Kabacoff, the author of the Manning book "R in Action"
- Boston GIS Getting started with PL/R.

These are three quick tutorials we have written on PL/R -

[http://www.bostongis.com/PrinterFriendly.aspx?content\\_name=postgresql\\_plr\\_tut01](http://www.bostongis.com/PrinterFriendly.aspx?content_name=postgresql_plr_tut01)

[http://www.bostongis.com/PrinterFriendly.aspx?content\\_name=postgresql\\_plr\\_tut02](http://www.bostongis.com/PrinterFriendly.aspx?content_name=postgresql_plr_tut02)

[http://www.bostongis.com/PrinterFriendly.aspx?content\\_name=postgresql\\_plr\\_tut03](http://www.bostongis.com/PrinterFriendly.aspx?content_name=postgresql_plr_tut03)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- Connecting R with PostGIS demonstrates using output of PostGIS queries in R. This particular example doesn't use PL/R but instead uses R directly. - [http://wiki.intamap.org/index.php/PostGIS#Connecting\\_PostGIS\\_with\\_R](http://wiki.intamap.org/index.php/PostGIS#Connecting_PostGIS_with_R)
- R-Sig-Geo - this is a newsgroup focused on using R in geoinformatics and geographical mapping. Its a good group to join if you want to learn tricks of the trade and what R packages are out there for doing geospatial analysis with R.  
<https://stat.ethz.ch/mailman/listinfo/r-sig-geo>
- Spatial Packages for R - this is descriptive listing of R packages commonly used in spatial analysis in R. <http://cran.r-project.org/web/views/Spatial.html>
- Applied Spatial Data Analysis with R - <http://www.asdar-book.org/> This is the book site for the 2008 Springer publication Applied Spatial Data Analysis with R. The book is written by Roger S. Bivand (manager of R-Sig-Geo newsgroup) and others. The book site contains downloadable code and data sets from the book. It demonstrates various exercises using the R geospatial packages.
- A Practical Guide to Geostatistical Mapping - <http://spatial-analyst.net/book/order> This is a 2009 publication by Tomislav Hengl. It comes as free e-book or \$13 hard print course workbook. It has many examples of using R geostatistical packages as well as using GRASS and is licensed under Creative Commons Attribution Noncommercial-No Derivative Works 3.0.

### **A.1.5 PgRouting Installation and Examples**

These are sites specific to PgRouting where you can download source, binaries, or learn how to compile and use it.

- PgRouting official site - Here you will find links to download the source and various binaries available for different OS. <http://pgrouting.postgis.org>
- PgRouting on Ubuntu Netbook Remix 9.10 - Details how to compile and install PgRouting on Ubuntu - <http://www.mkgeomatics.com/wordpress/?p=312>
- Foss4G 2009 Tokyo PgRouting Workshop - Workshop tutorial slides on getting up and running with PgRouting - [http://www.osgeo.jp/wordpress/wp-content/uploads/2009/11/workshop\\_manual.pdf](http://www.osgeo.jp/wordpress/wp-content/uploads/2009/11/workshop_manual.pdf) by Daniel Kastl
- <http://pgrouting.postgis.org/wiki/WorkshopFOSS4G2008> complete documents and sample data for workshop which details using PgRouting with OpenLayers and MapFish

### **A.1.6 PL/Python Installation and Examples**

- Our Postgres OnLine Journal - We have a cheatsheet and collection of intro articles on PLPython detailing installing, samples and basic flow. <http://www.postgresonline.com/journal/index.php?archives/106-PL-Python.html>
- Official Docs for PostgreSQL 8.4 on PL/Python. Similar docs exist for 8.2 and 8.3

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

<http://www.postgresql.org/docs/8.4/interactive/plpython.html>

- GDAL - Geographic Data abstraction library has both a c and Python interface. It is probably the most common Python library used by GIS open source python crowd. <http://www.gdal.org>
- Enabling GDAL in Python and GDAL Python helper packages - <http://pypi.python.org/pypi/GDAL/>. As of this writing, there are no precompiled windows binaries for 1.7 and above. If you want precompiled, use the 1.6 <http://pypi.python.org/pypi/GDAL/1.6.1> which is available for Python 2.5/2.6. This will allow you to access the ogr2ogr and gdal objects in PL/Python similar to what we demonstrated in PL/R.
- NumPy - this is an open source numerical processing library for python that is similar in functionality to things like MatLab. It is used for dealing with complex matrices. It is a common favorite among scientific professionals. It also has some useful GIS bindings and is commonly combined with GDAL. <http://numpy.scipy.org/>
- GNUPlot.py -- <http://gnuplot-py.sourceforge.net/> this is a python library for interfacing with GNUPlot that allows generating attractive graphical plots in Python.
- Windows 32-bit and 64-bit nightly build binaries by Tamas Szekeres contain the latest python PyGDAL. <http://vbkto.dyndns.org/sdk/>

### A.1.7 WKT Raster related information

- WKT Raster information page - This page will give you links to other WKT Raster resources, provide you with status of the project, where you can download source or binaries, and link to the road map <http://trac.osgeo.org/postgis/wiki/WKTRaster>
- The gdal 1.6+ libraries have a WKT Raster driver that will allow you to export data out of PostGIS raster format. This page will also give you more extensive details of using the gdal2wktraster.py script we described. [http://trac.osgeo.org/gdal/wiki/frmts\\_wtkraster.html](http://trac.osgeo.org/gdal/wiki/frmts_wtkraster.html)
- GDAL raster formats list - This page gives listing of raster formats GDAL supports you can load in any of the formats supported by your version of GDAL into the PostGIS raster data type using gdal2wktraster.py. [http://www.gdal.org/formats\\_list.html](http://www.gdal.org/formats_list.html)
- RasterLite - This is an raster extender for SQLite similar to how SpatialLite is a vector extender for SQLite. Its focus is more for storing raster in database for rendering rather than WKT Raster for analysis. <http://www.gaia-gis.it/spatialite/rasterlite-man.pdf> . GDAL 1.7+ has drivers that support it.
- Pre-compiled binaries of GDAL and Python needed for WKT Raster loading. <http://trac.osgeo.org/gdal/wiki/DownloadingGdalBinaries> . For windows the aforementioned Tamas Szekeres binaries are the latest and greatest and built nightly.

## A.2 Open source tools and offerings

This section includes listings of prepackaged open source tools that include PostGIS as a core component of their mix. They include one-click installers, fully contained application stacks, and single download virtual machines.

### A.2.1 Installers and self-contained suites that include/work with PostGIS

- For Mac Users – there are binaries for Mac OSX graciously supplied by King Chaos. It includes PostgreSQL, PostGIS, and some other useful open source gis toolkits  
<http://www.kyngchaos.com/wiki/software:postgres>
- PostgreSQL Yum Repository  
For Red Hat Linux (Enterprise and Fedora) and CentOS, there is the recently released PostgreSQL Yum repository that has packages for PostgreSQL, PostGIS and several other PostgreSQL accessories and as of this writing, planned offerings for OpenSUSE.  
<http://yum.pgsqrlpms.org>
- OpenGeo Stack - <http://opengeo.org/community/suite/download/> this stack contains Geoserver, GeoExt, GeoEditor, GeoWebCache, PostGIS, PostGIS GUI shapefile loader and optional extensions for ArcSDE and Oracle spatial. It has one-click installers available for Windows, Mac OSX, and soon Linux. The GeoEditor is a web-based GIS editor that allows you to edit PostGIS data via the web interface. It comes in both Enterprise and Community Edition. The main differences being the Enterprise includes support, training and Service Level Assurances (SLA), and hand-holding help with upgrades for those new to GIS or that need more predictable professional support. The stack also comes prepackaged with sample data to get you started. The Community Edition is free open source and binary download for the more experienced user, student user, or consultant looking for an easy to configure stack for a client and to extend with their own web product. Compare can be found  
<http://opengeo.org/products/suite/compare/>
- Portable GIS  
If you are a windows user, check out Portable GIS managed by Jo Cook. It is really cool and comes packaged with PostgreSQL, PostGIS, MySQL, Quantum, GRASS, FWTools, MapServer, GeoServer, FeatureServer, OpenLayers all of which can be run from a thumb drive. <http://www.archaeogeek.com/blog/portable-gis/>
- GIS VM  
<http://www.gisvm.com/>  
If you want a fully contained GIS Virtual Machine that you can play with a VMPlayer such as VMWare's freely available VMWare VM Player, that contains best of the breed Open Source GIS tools - check out - GIS VM. This is an Ubuntu VM that comes in 3 flavors – GIS VM Basic English, GIS VM Geostatistics English, and Portuguese version.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

The basic International English version comes packaged with

PostgreSQL/PostGIS, GeoServer, Mapserver, FWTools,

QGIS/Grass, gvSIG, uDig, OpenJump, Kosmo

The GeoStatistical Version contains all the above plus PL/R and R Statistical Environment (similar to SAS and S-Plus), SAGA and MySQL 5.

- DebianGIS - provides binary packages for MapServer, PostgreSQL/PostGIS, GDAL, QGIS and GEOS for Debian Linux. <http://wiki.debian.org/DebianGis>

- EnterpriseDb One-Click PostgreSQL/PostGIS installer

If you are on a Windows system or a desktop Linux or Mac OSX, the easiest way to get started is to use the respective One-Click installers provided by EnterpriseDb. These we cover in the installation guide appendix

<http://www.enterprisedb.com/products/pgdownload.do>

### **A.2.3 Free open source desktop GIS**

The following desktop tools in this list have integration features with PostGIS to allow viewing and editing PostGIS data.

- OpenJump -- <http://www.openjump.org/> - this is one of our favorite desktop GIS tools and what we use to render many of the ad hoc spatial queries you see in this book. It is a java-based GIS desktop toolkit based on a plugin architecture. Many user contributed plugins. Runs on Linux/Windows/MacOSX
- QuantumGIS (QGIS) - <http://www.qgis.org/> Perhaps the most popular of the free open source desktop tools. It also has a plugin architecture. QGIS is written in C++ but offers a rich Python scripting environment and various GRASS integration options. It also includes drivers for connecting to PostGIS data as well as various other GIS data sources. GNU GPL
- GvSIG - <http://www.gvsig.gva.es/> java based desktop platform with lots of integration features to ArcIMS and other ESRI services.
- uDig - <http://udig.refractions.net> an Eclipse based Java desktop application and SDK. Lots of integration features with OGC compliant webservices and more advanced cartography.
- OSGeo4W Installer - <http://trac.osgeo.org/osgeo4w/> this is an on-line installer for MS windows that packages QuantumGIS, GDAL/OGR, Python bindings for mapserver, Gdal and apache webserver with web apps and sample data in a single install that allows you to pick and choose what you want. If you want to use the osgeo/gdal package under Python 2.5+ and don't want to compile yourself, this is currently the easiest package to use.
- Geographic Resources Analysis Support System (GRASS) - <http://grass.osgeo.org/> GRASS is probably the oldest and one of the most advanced free and open source

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

tools for analyzing vector, raster and other GIS data. It is designed more for the advanced GIS analyst rather than a new GIS or pure spatial database user. Although it is not set up specifically for PostGIS, there are many avenues of integration such as the PostGRASS driver, JGRASS and QGIS GRASS integration tools.

#### **A.2.4 Extract Transform Load (ETL)**

- GDAL/OGR - <http://gdal.org/ogr2ogr.html> - this is the most popular of all purpose open source free ETL tools. It is licensed under the MIT license which is similar to BSD. Binaries can be downloaded from <http://trac.osgeo.org/gdal/wiki/DownloadingGdalBinaries>
- shp2pgsql, pgsql2shp, shp2pgsql-gui -- packaged with PostGIS for dumping and loading data from ESRI shapefile format.
- OSM2PGSQL - this is a command line tool specifically designed for converting OpenStreetMap XML (OSM) format to PostgreSQL/PostGIS. There are binaries for most OS including Windows and Mac OSX that can be downloaded from <http://wiki.openstreetmap.org/wiki/Osm2pgsql>
- Spatial Data Integrator - <http://www.spatialdataintegrator.com/> is an GPL v2 licensed open source ETL tool with geospatial capabilities spearheaded by CamptoCamp and Talend. It is based on Talend Open Studio, Talend's generic ETL solution, and extends it with geospatial components. Some Talend Geospatial use case examples can be found at  
<http://www.talendforge.org/wiki/doku.php?id=sdi:examples>  
<http://www.talendforge.org/wiki/doku.php?id=sdi:geocomponentslist>
- As of this writing current formats supported are ESRI shapefile, MapInfo, WKT, WFS, GPX, and OSM (OpenStreetMap XML format), and PostGIS.
- Geokettle - Is LGPL released Open Source ETL loader based on Pentaho Kettle ETL. It currently has built in support for PostGIS, Oracle Spatial, MySQL, ESRI shapefiles.  
<http://sourceforge.net/projects/geokettle/>

#### **A.3 Proprietary Tools that support PostGIS**

- CadCorp SIS <http://www.cadcorp.com/> - a suite of products that includes desktop GIS and web mapping OGC compliant WMS, WFS, great raster and CAD support
- Safe FME (ETL) <http://www.safe.com/> the most recognized name in the industry for spatial ETL and automating spatial ETL workflows.
- ESRI ArcGIS 9.3 - requires ArcSDE license to work with PostGIS

© Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

<http://webhelp.esri.com/arcgisdesktop/9.3/index.cfm?TopicName=The%20PostGIS%20geometry%20type>

- ZigGIS for ArcGIS - a plugin for ArcGIS useful if you just need to do desktop work and don't want to shell out money for an ArcSDE license. The code is technically commercial open source. <http://pub.obtusesoft.com/>
- Manifold - <http://www.manifold.net> Pretty nice all in one package. Also has some neat SQL functions for dealing with Raster. Has its own dialect of Spatial SQL very similar in style to Microsoft Access Jet (e.g. supports cross tabs using Transform PIVOT, TOP etc. and lots of spatial functions). Works with all the popular spatial databases without additional cost - PostGIS, Oracle, DB2, SQL Server 2008
- Pitney Bowes MapInfo 10 - A favorite among data analysts and casual GIS users because of its ease with which you can link to datasources, import data, and run basic SQL queries  
<http://www.pbinsight.com/products/location-intelligence/applications/mapping-analytical/mapinfo-professional/>
- MapDotNet - <http://www.mapdotnet.com/Pages3.0/default.aspx> -  
web mapping toolkit for ASP.NET similar in style to UMN Mapserver and in fact mapfile format follows similar scheme. Includes wizards to build map etc.

## A.4 Places to get “free” vector data

Below are some useful places to find data. In chapter 6 we grab data from some of these places to demonstrate how to load up on spatial data.

### A.4.1 All geographic regions

- OpenStreetMap  
<http://www.openstreetmap.org> is a community driven spatial database and map repository that has contributions from people all over the world. You can think of it as a free and open source google map that has both web services and data you can download. It has got base map information you can access via tile services as well as other crowd-sourced information such as biking trails and other GPS traces and way points in GPX format. You can use it as an overlay directly with your maps using something like OpenLayers. In addition to that you can load some of this data right into your PostGIS enabled PostgreSQL database using the osm2pgsql command line tool <http://wiki.openstreetmap.org/wiki/Osm2pgsql> .
- Natural Earth Data - public domain map datasets that contain both raster and vector data. Most data can be used in any manner for private or commercial consumption to build upon. Data currently offered are world administrative boundaries, city and town points with population, and various natural land, water geometries.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

<http://www.naturalearthdata.com/>

- Center for disease Control Administrative boundary files

The United States center for disease controls maintains boundary files for all the continents and countries in ESRI Shapefile format. These are circa 2000.

<http://www.cdc.gov/epiinfo/shape.htm> and are all in WGS 84 long lat spatial reference system (SRID = 4326).

Some ones of notable interest

Country boundaries this contains the multipolygon boundaries for each country as well as information such as name of country, currency, population, and iso-code. We will be using this layer in some of our future exercises - **Cntry00.zip**

By State – this is a more granular than country boundaries and breaks the various countries into states and provinces. This list the administrative name, country, population, type of boundary. **Admin00.zip**

- <http://www.gadm.org/> (some of these files used to be located at <http://biogeo.berkeley.edu> and that now redirects here)

Global Administration Areas is a fairly new site that tries to maintain a fairly up to date version of administration boundaries at various resolutions. Data is licensed for free use for non-commercial and educational purposes. Data is stored in ESRI Shapefile, ESRI geodatabase, and Google KMZ and some in RData format for R statistical package. You can download files by country <http://www.gadm.org/country> or download the whole set <http://www.gadm.org/world> .

- Geocommons

Geocommons is a directory of both free and non-free GIS datasources in ESRI Shape, KML, and GeoRSS. <http://finder.geocommons.com/>

It contains spatial data containing statistical information on a wide variety of topics from health, demographics, and boundaries. Many of these are user contributed.

- Infochimps <http://infochimps.org>

This is a search engine specifically for finding datasets. Many of the data sets are of a geospatial nature.

#### A.4.2 North America

- U.S. census data and Tiger Street Data

By far the most popular, complete and free source of data for United States is the U.S. census Topologically Integrated Geographic Encoding and Referencing system (Tiger) data. The most recent distribution of this data is the 2008 version which was released

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

on December 8, 2008. This can be downloaded from  
<http://www.census.gov/geo/www/tiger/>

The latest release of Tiger data are distributed in ESRI Shapefile and Dbase DBF format so can be loaded easily with the shp2pgsql tools provided with PostGIS.

Versions prior to 2007 are released in a Tiger proprietary format which can be loaded with OGR2OGR.

The Tiger data set includes the following items in EPSG:4269 (US NAD 83 long lat)

- National Layers – US State boundaries
- Census block groups, blocks, and tracts and are broken out by state-county – a census tract is the smallest demarcation for population. The US census, tries to maintain the same population across all census tracts or tracts of a particular type. A lot of statistics such as employment, disease etc. are calculated against census blocks and tracts. Census block polygons are recalculated every 10 years or so and populations in them may change drastically.
- 2002 5-Digit and 3-Digit Zip Code tabulation areas – these are polygons that approximate zip code area routes, for all US as single files. Keep in mind that US Postal Zone Improvement Plan (Zip) routes are really street segments, so the ZCTA is a simplification of these into polygons by aggregating census blocks that intersect these street segments.
- Other state county files: Streets, Roads, Water lines, Water polygons, address ranges, points of interest, voting districts (aka Ward and Precincts), congressional and senatorial districts
- New England Cities and towns
- Data.gov - <http://www.data.gov/> this is a new site launched by the Obama administration as part of an open data government initiative. It contains both geographic data in ESRI Shapefile and KML formats as well as various statistical data in csv tabular format. All of these are fairly easy to import and analyze in PostgreSQL. You'll find all sorts of interesting data such as spending, toxic waste zones, air emissions. We encourage you to explore it.
- US National Atlas

The US National Atlas offers numerous geographic layers for United States such as railroads, airports, political boundaries. Most are in ESRI shape file format.

<http://nationalatlas.gov/atlasftp.html>

- US National Weather

<http://www.nws.noaa.gov/geodata/>

The National Weather service has a catalog listing of ESRI shapefile boundaries, and weather related data mostly of US, but some covering the globe.

- Nature Serve

If you are into the study of animals and ecological affects, Nature Serve has an extensive assortment of ecological data for the US and Canada and some other regions of North America. All in ESRI Shapefile format.

<http://www.natureserve.org/getData/animalData.jsp>

These are mostly point and polygon regions where these species are naturally found.

- GeoBase.ca

GeoBase is a portal that provides free spatial data for Canada.

<http://www.geobase.ca>

It provides fairly up to date data and for following types of features:

- Administrative boundaries
- Digital elevation data
- Hydrology
- Satellite Imagery
- Roads for each province

- Canada – Statistics Canada – National Statistical Agency

Statistics Canada has data for free as well as for cost download including Canadian boundary and Road Network files from (2005 – 2007).

<http://www.statcan.gc.ca/mgeo/boundary-limite-eng.htm>

- GeoGratis - Natural Resources Canada

<http://geografatis.cgdi.gc.ca/>

Maintains mostly boundary files for Canada as well as raster data in ESRI Shapefile , Raster Tiffs, and tabular data. Data is free for download for both commercial and non-commercial. Probably one for most general use is the framework data files

<http://geografatis.cgdi.gc.ca/geografatis/en/download/framework.html>

#### **A.4.3 Other Countries and Continents**

- UK Ordnance Survey

Ordnance Survey recently launched their open data site that offers both free with very unrestrictive licenses as well as for purchase data. You can find both vector and raster data here.

<http://www.ordnancesurvey.co.uk/oswebsite/opendata/>

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

#### A.4.4 Regional

These are some locations that cover a small region such as a state or city but have a lot of spatial data for that region.

- Mass GIS

We couldn't talk about data without talking about our favorite state and the state we live in. Mass GIS has an extensive inventory of both vector data and high resolution aerial imagery for most of Massachusetts. All free for download. All the vector data comes in ESRI format for easy loading into PostGIS and other spatial databases. In addition to downloading, they have various web services you can use to consume their spatial data if you just need to use in your mapping applications. We demonstrate this in chapter 11.

- Raw Data <http://www.mass.gov/mgis/laylist.htm>
- Web Mapping Services

<http://lyceum.massgis.state.ma.us/wiki/doku.php?id=history:home>

- DataSF.org

If you live in San Francisco, California, or just looking for data to play with, you'll want to check out DataSF.org. It is part of a pilot project called CivicDb which hopes to provide a reference implementation for other government agencies. Details here

<http://it.toolbox.com/blogs/database-soup/datasforg-is-now-up-33563?rss=1>

As DataSF, you can find lots of spatial vector and aerial data for San Francisco. Data such as streets, shortelines, bridges, zipcodes, city projects such as renewal and revitalization, city parcels, and zones.

<http://www.datasf.org>

#### A.4.4 Sample data for training

- North Carolina has provided a free data set for training purposes that contains both vector and raster data. This can be downloaded from  
<http://www.grassbook.org/ncrexternal/index.html>
- This is from our PGCon 2009 presentation and more of an exercise on how NOT to build a town. The data is made up and free to use and improve on.  
[http://www.bostongis.com/PrinterFriendly.aspx?content\\_name=pgcon2009\\_postgis\\_spatial](http://www.bostongis.com/PrinterFriendly.aspx?content_name=pgcon2009_postgis_spatial)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

## A.5 Spatial Reference Systems Resources

In this section we list some resources on spatial reference systems that we have found useful.

- <http://www.enchantedlearning.com/geography/glossary/projections.shtml> - gives a good primer on map projections.
- <http://spatialreference.org>

This is an invaluable site for looking up spatial reference systems and adding them to PostGIS. Especially when you have a somewhat obscure one. This site contains both EPSG defined (many of the US state plane feet which do not come packaged with the default PostGIS spatial\_ref\_sys table) and also many user contributed ones from around the globe. The nice thing about this site is it will provide you an insert statement for PostGIS. You can submit an SRS Text of an obscure projection and it will calculate the PostGIS/Proj4text equivalent. You can also search for already user submitted ones.

- <http://www.sharpgis.net/post/2007/05/Spatial-references2c-coordinate-systems2c-projections2c-datums2c-ellipsoids-e28093-confusing.aspx> - summary by Morten Nielsen
- [http://geology.isu.edu/geostac/Field\\_Exercise/topomaps/map\\_proj.htm](http://geology.isu.edu/geostac/Field_Exercise/topomaps/map_proj.htm) - gives a good description of conical/cylindrical and (oblique, equatorial, transverse)

<http://en.wikipedia.org/wiki/Ellipsoid> - gory details of the mathematical definition of an ellipsoid

- [http://en.wikipedia.org/wiki/Figure\\_of\\_the\\_Earth](http://en.wikipedia.org/wiki/Figure_of_the_Earth) - Figure of earth and various ellipsoids used over the years
- <http://msdn.microsoft.com/en-us/library/cc749633.aspx> - SQL Server 2008 documentation explaining spatial coordinate systems by Isaac Kunen and difference between flat and round earth models. It is a surprisingly good description complete with pictures.
- Proj4 Wiki -- Proj4 is the projection library used by PostGIS. This website includes documentation on how to use the raw API and will be of value to those wanting to create their own custom spatial reference systems. <http://trac.osgeo.org/proj/wiki>
- This article is a quick primer on common spatial reference systems and the proj4 equivalents. Again this is of use for those who want to look at by example how to define a custom spatial reference system with proj4 syntax.  
[http://www.remotesensing.org/geotiff/proj\\_list/](http://www.remotesensing.org/geotiff/proj_list/)

# **Appendix B:**

## *Installing, Compiling and Upgrading*

This Appendix covers

- Installing PostgreSQL and PostGIS
- Spatial enabling a new PostgreSQL database
- Spatially enabling an old PostgreSQL database
- Upgrading an existing install

There are several ways to install PostgreSQL/PostGIS. When we were starting out, the only way was to compile these things yourself. Today life has become much simpler and the general user need not know the joys and frustrations of compiling your own source code. Compiling from source is still an adventurous journey and builds character, but most take the easy, sedated road.

### ***B.1 Installing PostgreSQL and PostGIS***

In order to use PostGIS, it goes without saying, that you need a functioning PostgreSQL server. The below install options we will describe offer the basic PostgreSQL install package as well as the additional PostGIS package.

#### ***B.1.1 Desktop Linux, Windows, MacOSX Using One-click Installers***

If you are on a Windows system or a desktop Linux or MacOSX, the easiest way to get started is to use the respective One-Click installers provided by EnterpriseDb.

<http://www.enterprisedb.com/products/pgdownload.do>

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

EnterpriseDb one-click installers will work for any desktop Linux system (32 bit and 64 bit), Windows system (2000, XP, 2003,2008), and MacOSX. The installer comes with the following prepackaged goods:

- PostgreSQL Server
- PgAdmin III (GUI Database administration tool)
- Application Stack Builder – which allows you to install other PostgreSQL add-ons such as PostGIS, JDBC, ODBC, and application development environments such as Apache and Ruby on Rails, PostgreSQL Tuning Wizard (to help you quickly configure the PostgreSQL memory and other settings based on your desired profile), MySQL migration Wizard (requires java be installed)

If you don't want to use the built-in Stack builder because you need to install on a system not connected to the internet, then you can download the PostgreSQL binaries directly from the PostgreSQL website

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

and the pre-compiled stackbuilder PostGIS pieces (windows pre-compiled, and only source for linux) from

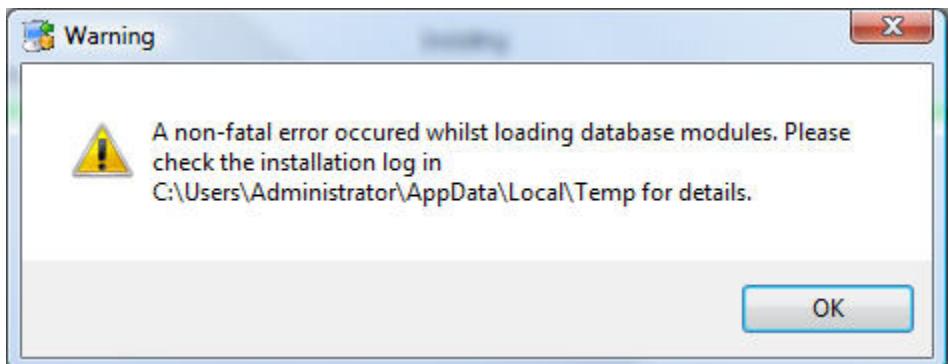
- <http://pgfoundry.org/projects/stackbuilder/>

For those trying to install using the Linux one-click installer, make sure to make the .bin file executable by running chmod 777 on the .bin file.

For further help on getting started, check out the Additional Resources appendix.

#### **WINDOWS VISTA GOTCHAS**

When trying to install on Windows Vista – you may get an error something of the form



© Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

**Figure B.1 Common Error on Windows Vista**

And then it proceeds to uninstall PostgreSQL. This is because of the security measures added to Vista. If you are installing on Windows Vista, follow the instructions outlined in the PostGIS Wiki

<http://trac.osgeo.org/postgis/wiki/UsersWikiWinVista>

**TURN OFF USER ACCOUNT CONTROL (UAC)**

It may be sufficient to just turn off User Account Control located in Control Panel --> User Accounts. In most cases the newer PostgreSQL installers seem capable of creating a postgres account on their own.

**B.1.2 Installing on Server Linux (Red Hat EL, CentOS) Using YUM**

The PostgreSQL Yum repository has the latest and greatest for PostgreSQL and beta versions of PostgreSQL if you are running a variant of Red Hat or CentOS. In addition to the core PostgreSQL the Yum repository has additional packages for PostGIS and other add-ons.

<http://yum.pgsqrlpms.org/>

The Yum repository is most suitable for command-line Server installs, but can also be used for desktop installs via the Yum installer.

Details on how to install can be found at the following links we have written  
Postgres OnLine Journal - Almost Idiot's Guide to PostgreSQL YUM -  
<http://www.postgresonline.com/journal/index.php?archives/45-An-Almost-Idiots-Guide-to-PostgreSQL-YUM.html>

For other topics we have written on YUM check out:

<http://www.postgresonline.com/journal/index.php?categories/53-yum>

OpenSUSE and SUSE are not available yet thru this repository, but these are a planned addition in the near future. For OpenSUSE you can use the EnterpriseDb one-click installers.

**B.1.3 Mac OSX specific Installers**

If you are a Mac User, you might want to check out Kyng Chaos.

<http://www.kyngchaos.com/software/postgres>

Kyng Chaos has packages for the latest stable releases of PostGIS/PostgreSQL and PgRouting for Tiger and Leopard as well as a number of other interesting GIS open source packages. Kyng Chaos generally stays up to date with the latest and greatest PostGIS offerings and has them available as soon as the source is officially released.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

## B.1.4 Other Available Binaries and Distros

Most of the other distros such as Ubuntu and Debian have PostgreSQL and PostGIS available via the distro package manager. For Ubuntu you would use apt-get to get these. These don't always have the latest version binaries, though things are improving these days so they are more up to date. As of this writing, we would suggest using the already stated PostgreSQL binaries if you can since those are specifically maintained by PostgreSQL high-end users so are most always up to date.

## B.1.5 Compiling and Installing from PostGIS source

If you want the most bleeding edge version of PostGIS, compiling yourself is still the way to go, and this for the most part, is covered in the PostGIS manual and Wiki and is kept fairly up to date, so we won't waste space covering it again.

- The official PostGIS manual in Chapter 2 covers standard compilation on Linux systems.

<http://www.postgis.org/documentation/manual-svn/ch02.html#firsttimeinstall>

While you can compile your own PostgreSQL, if you are on Linux you don't need to even if you want to compile PostGIS yourself. However you will need the PostgreSQL development headers (usually packaged in something called postgresql-devel in addition to installing postgresql,postgresql-server) which you can install using your Linux packager – e.g. YUM, YatZ, apt-get or whatever your distro uses for software install. The YUM, YatZ, apt-gets of the world are similar in concept to Windows Update in Windows and provide already precompiled binaries of PostgreSQL and PostGIS for your particular distro from well-defined repositories.

- There is a whole section on the PostGIS Wiki contributed by users that details how to compile PostGIS on various Oses.

<http://trac.osgeo.org/postgis/wiki/UsersWikiMain>

- For Windows – as of this writing – because headers are not available for Windows since Windows PostgreSQL versions are compiled with Visual C++, you must compile your own PostgreSQL under MingW if you want to compile PostGIS. The PostGIS User Wiki covers this:

<http://trac.osgeo.org/postgis/wiki/UsersWikiWinCompile>

If you don't want to suffer the joys and pains of compiling code and don't mind waiting for a package maintainer to compile and prepare a package for general consumption, then stick with the compiled versions.

## B.2 Creating a PostGIS database

The windows one click installer already creates a template postgis database for you, but many of the binary packages will not create one.

A template database is a database you can use as a model for new databases. If you create one specifically for PostGIS work (and perhaps add to it other stuff you commonly use – e.g. if you are a python programmer you may want to enable plpython in your template), you can use this template as a quick way for creating PostGIS enabled databases. For those coming from SQL Server backgrounds, the idea of a template database is similar in concept to SQL Server's model database, except that PostgreSQL allows you to create multiple template databases you can use for various use cases. PostgreSQL comes prepackaged with 2 template databases – template0 and template1. template0 is the super plain vanilla with the absolute minimal you need to have a functioning PostgreSQL database. template1 is the more commonly used one that has common function libs already preinstalled.

Of course before you can even create a spatial database or any database for that matter you need to be able to log into your PostgreSQL server via PgAdmin III or psql. If you have problems doing that – then please refer to appendix D.

### B.2.1 Creating template\_postgis under PostGIS 1.3.x

Below is a simple script to create a template\_postgis database for PostGIS 1.3.x installs using psql. Please note the paths may be different from install to install so you may need to change that.

For RedHat EL Linux installs, the path locations are the ones listed below. For Windows it's usually C:/Program Files/PostgreSQL/8.something/share or C:/Program Files (x86)/PostgreSQL/8.something/share (for 64-bit windows)

To launch psql – do the following from the command line on the PostgreSQL server:

```
psql -d postgres -U postgres
```

If psql is not accessible without the full path, you may need to include the full path to your postgresql/bin folder.

Once in psql do the following:

#### **Listing B.1 Creating a template\_postgis for 1.3**

```
CREATE DATABASE template_postgis WITH TEMPLATE = template1 ENCODING =
'UTF8';
\c template_postgis;
CREATE LANGUAGE plpgsql;
\i /usr/share/pgsql/contrib/lwpostgis.sql;
\i /usr/share/pgsql/contrib/spatial_ref_sys.sql;
\i /usr/share/pgsql/contrib/postgis_comments.sql;
UPDATE pg_database SET datistemplate = TRUE WHERE datname =
'template_postgis';
(1)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
GRANT ALL ON geometry_columns TO PUBLIC;
GRANT ALL ON spatial_ref_sys TO PUBLIC;
(2)
\q
```

The above code listing you use in PSQL commandline or PgAdmin III to create a new template postgis 1.3 database and (1) give permissions to the geometry\_columns and spatial\_ref\_sys so that when a database is created with it, new geometry columns can be registered by the user and new spatial\_ref\_sys records can be added as needed. (2) The \q is a psql only command not available in PgAdmin, that exits out of the psql commandline.

### B.2.2 Creating template\_postgis under PostGIS 1.4, 1.5+

Below is a simple script to create a template\_postgis database for a PostGIS 1.4+ installs using psql. In the 1.4 and above versions, the paths and names of files changed a little.

For RedHat EL Linux installs, the path locations are the ones listed below. For Windows its usually C:/Program Files/PostgreSQL/8.something/share/contrib/postgis-1.4 (postgis-1.5) or C:/Program Files (x86)/PostgreSQL/8.something/share/contrib/postgis-1.5 (for 64-bit windows)

To Launch psql – do the following from the command line on the PostgreSQL server:

```
psql -d postgres -U postgres
```

If psql is not accessible without the full path, you may need to include the full path to your postgresql/bin folder.

Once in psql do the following:

#### **Listing B.2 Creating template\_postgis for 1.5**

```
CREATE DATABASE template_postgis WITH TEMPLATE = template1 ENCODING =
'UTF8';
\c template_postgis;
CREATE LANGUAGE plpgsql; --this may not be needed if running 8.4
\i /usr/share/pgsql/contrib/postgis.sql;
\i /usr/share/pgsql/contrib/spatial_ref_sys.sql;
\i /usr/share/pgsql/contrib/postgis_comments.sql;
UPDATE pg_database SET datistemplate = TRUE WHERE datname =
'template_postgis';
GRANT ALL ON geometry_columns TO PUBLIC;
GRANT ALL ON spatial_ref_sys TO PUBLIC;
\q
```

The above code listing you use in PSQL commandline or PgAdmin III to create a new template postgis 1.4 database and give permissions to the geometry\_columns and spatial\_ref\_sys so that when a database is created with it, new geometry columns can be registered by the user and new spatial\_ref\_sys records can be added as needed. The \q is a psql only command not available in PgAdmin, that exits out of the psql command line.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### B.2.3 Creating a new spatially enabled database

If you have a template\_postgis database created, you can now use this to create future PostGIS enabled databases. You can create a new spatially enabled database with PgAdmin III or PSQL command line/shell tool.

#### USING PGADMIN III

If you are using PgAdminIII – right click on the Database tree icon and – create a new database and choose the template\_postgis database as your template as the following picture shows.

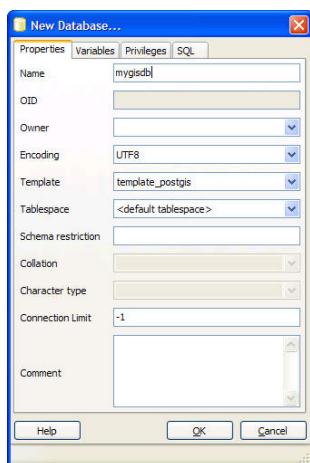


Figure B.2 New database dialog window in PgAdmin III with template\_postgis database selected

#### USING PSQL OR SHELL VIEW

If you don't have PgAdmin III handy or prefer to live in the shell, then you have two options which you can access usually by straight command in Linux or in windows launching the Start->Programs->PostgreSQL->SQL Shell

Connect via psql to any database and run:

```
CREATE DATABASE mygisdb WITH TEMPLATE = template_postgis;
\q
```

Or use the createdb command which is in the PostgreSQL/..bin folder or in Linux /usr/bin createdb

## B.2.4 Spatially Enabling an Existing PostgreSQL database

To enable a PostgreSQL database with no postgis install.

- Install PostGIS binaries – which you can get from One-Click Installer – Stack Builder (should be on your PostgreSQL menu), or available via Yum or your distro, or which you compile and install following directions on PostGIS wiki.
- Then run the same scripts we defined for creating a template\_postgis
- To verify you install run the following query after connecting to the newly created database:

```
SELECT postgis_full_version();
```

From either PSQL or PgAdmin III

## B.3 Upgrading an Existing Install

Before you upgrade your PostGIS install, you should first verify which version of PostGIS you are running.

```
SELECT postgis_full_version();
```

After verifying your current install, compile or upgrade the PostGIS binaries as described earlier and then follow the below steps to upgrade your database.

### PostGIS 1.3, PostGIS 1.4, PostGIS 1.5 can coexist

As of PostGIS 1.4, it is possible to have both PostGIS 1.3 and PostGIS 1.4 and PostGIS 1.5 installed on the same PostgreSQL server, but different PostgreSQL databases, so would be useful for testing out functionality. If you go with having both versions, you will want to name your templates something like template\_postgis13 and template\_postgis14, template\_postgis15.

They all have to be sharing the same GEOS and Proj libraries however so keep that in mind.

### B.3.1 Upgrading database from 1.3.x to 1.3.x+

If you are running a PostGIS version of 1.3 or above and are upgrading to a point release say 1.3.3 to 1.3.6 you can run the /path/to/pgsql/share/contrib/postgis/lwpostgis\_upgrade.sql to upgrade your database.

### **B.3.2 Upgrading database from 1.3.x to 1.4.x or 1.3.x to 1.5.x**

If you are running a PostGIS version of 1.3 or above, you can use the soft upgrade script located in the postgis (version folder)

`postgis_upgrade_13_to_14.sql`

for 1.5 it would be `postgis_upgrade_13_15.sql` and so on.

Then once you are done --- to verify you are running 1.4, do a

`SELECT postgis_full_version();`

### **B.3.3 Hard Upgrades**

Hard Upgrades are required when upgrading from a PostGIS major to major (e.g. 0.9 to 1.3). While they are not required from semi-major to semi major -- if your database is small enough you should probably do a hard-upgrade. There are degrees of this.

All Hard Upgrades require a backup of your database (dump), followed by a restore. The manual covers a somewhat cleaner way of doing a hard upgrade which is a bit more time consuming than what we will demonstrate and also requires that you have Perl installed which may not be an option on a Windows Server.

Official Hard Upgrade instructions are

1.3: <http://www.postgis.org/documentation/manual-1.3/ch02.html#upgrading>

1.4: [http://www.postgis.org/documentation/manual-svn/ch02.html#hard\\_upgrade](http://www.postgis.org/documentation/manual-svn/ch02.html#hard_upgrade)

#### **HARD UPGRADE FROM 0.X, 1.X TO 1.3.X OR 1.4 OR 1.5**

If you are running an older version between 1.1 and 1.2.2 (or lower) or are going from upgrading to a major release (e.g. 1.2 to 1.3) you should do a hard upgrade even though running the upgrade script will lead you to believe that you can do a soft-upgrade. The reason being that certain things such as changes to CASTS and types can not be accomplished with a soft-upgrade. Also you can't drop things in use without destroying the dependents. When disk storage changes happen, you also really need to reload the data for the old format to be stored in the new.

For versions after 1.1. you can alternatively use our cutting corners way. It will leave some junk from prior versions.

You'll want to dump your database with (replace localhost with your server's name if you are not local to it). All this step you do from the command line or if you are local to the server and are using a server with PgAdmin III installed, you can use the right-click backup (compressed) feature of PgAdmin III as shown in accompanying diagram.

```
/path/to/pgsql/bin/pg_dump -i -h localhost -p 5432 -U youruser -F c -b -v -f "/path/to/backup/yourdbhere.backup" yourdbhere
```

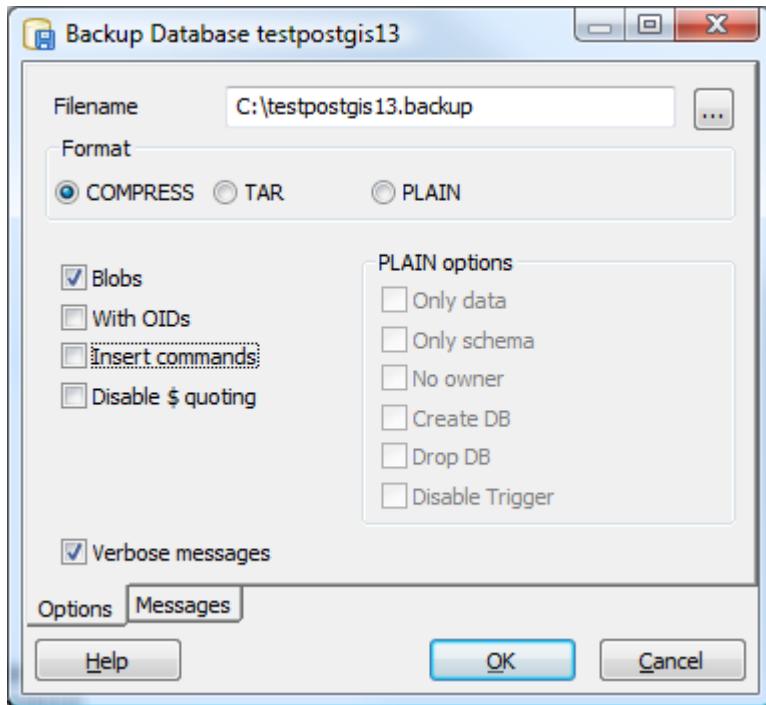


Figure B.3 If you are using PgAdmin III for backup – should look like above.

Launch psql (or you can use PgAdmin III):

```
/path/to/pgsql/bin/psql -i -h localhost -p 5432 -U postgres
```

Then perform the following from the psql prompt

Install new PostGIS and create a new template\_postgis based on new PostGIS install. If you are on the same server as your old, then drop the old database (but make note of the database encoding since you want to create your new database with the same encoding).

Note the below example uses UTF8, you should replace with whatever your old database was encoded with.

### **Listing B.3 Hard Upgrade**

```
(1) DROP DATABASE yourdbhere;
(2) CREATE DATABASE yourdbhere WITH ENCODING='UTF8' TEMPLATE=template_postgis;
(3)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
\q
(1) Drop your old database
(2) Create a fresh new version based on template_postgis
(3) Quit out of psql
```

or again you can use PgAdmin III

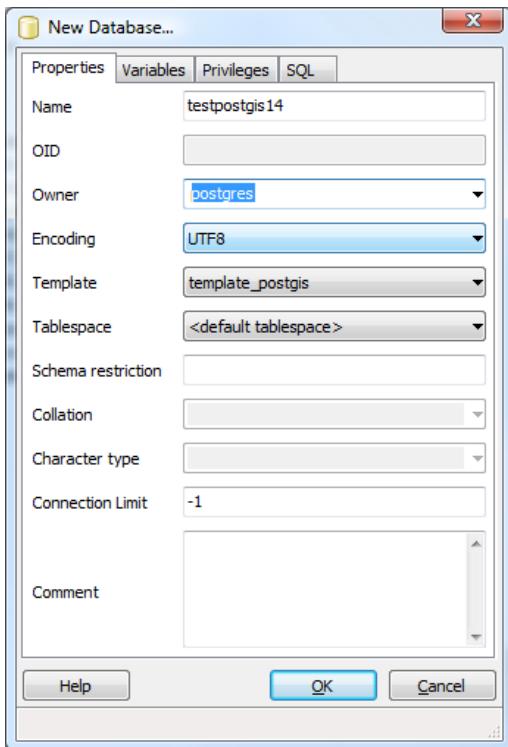


Figure B.4 Creating a new database with PgAdminIII using template\_postgis

Once you have a fresh PostGIS database you can now restore your old data on top of it. The pg\_restore will not put in the old functions (it will skip over functions and aggregates already present) so the new functions you had installed will take precedence. As for data, it will not recreate tables already present, but will insert data into those tables if the structure is the same.

#### **OLD SPATIAL REFERENCE RECORDS NOT PRESERVED**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=565>

Since spatial\_ref\_sys has a primary key and data already, it will fail trying to add records – so any custom records you added you will need to read. Alternatively you can delete the spatial\_ref\_sys table before restore so your old spatial\_ref\_sys is restored, but then you lose the corrections made in the newer spatial\_ref\_sys

You can restore using the command line pg\_restore.exe packaged with PostgreSQL

```
/path/to/bin/pg_restore.exe --host=localhost --port=5432 --username=postgres --dbname=yourdbgoeshere --verbose "/path/to/yourbackupfile"
```

or PgAdmin III if network bandwidth between the PostgreSQL server and your workstation are good (or your db is small).

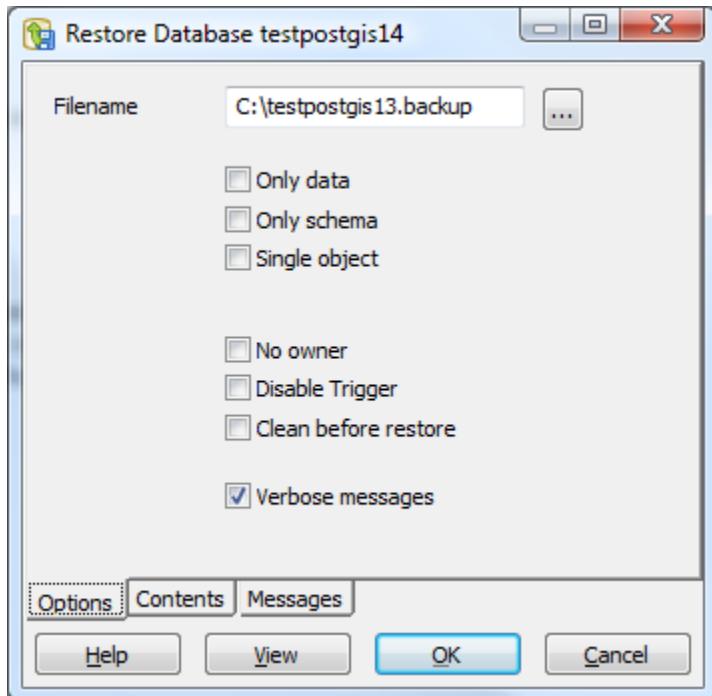


Figure B.5 Doing restore with PgAdmin III

# Appendix C:

## *SQL Primer*

This Appendix covers

- Information Schema
- Basic SQL functions available in most relational databases
- JOINS (INNER, OUTER, LEFT, RIGHT, FULL, INTERSECT, EXCEPT)
- Using Aggregates
- Using subselects
- Window functions and Window aggregates

PostgreSQL has pretty much all the ANSI SQL:1992, 1999 standard logic supported as well as much of the SQL:2003, SQL:2006 and some of the SQL:2008 constructs. In this section we shall cover these as well as some PostgreSQL specific SQL language extensions. Since we are keeping this section fairly focused on standard functionality, the content in this chapter is applicable to other standards compliant relational databases.

### **C.1 Information Schema**

The information schema is a catalog introduced in SQL:1992 and that has been enhanced in each subsequent version of the spec. While it is a standard, sadly most of the commercial and open source databases do not support it. The following common ones we know that do are PostgreSQL (7.3+), MySQL 5+ (not sure about 4) , and Microsoft SQL Server 2000+.

The most useful views in this schema are the “tables”, “columns”, and “views” and provide a catalog of all the tables, columns, and views you will find in your database.

To get a list of all non-system tables in PostgreSQL, you can run this query and will work equally well in MySQL (except for MySQL schema means database and there is only one information\_schema shared across all MySQL databases on a MySQL cluster). MS SQL Server behaves more like PostgreSQL in that each information\_schema is unique to each database

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

except that in SQL Server, the system views and tables are not queryable from the information\_schema whereas they are in PostgreSQL. The tables view in PostgreSQL will only list tables that you have access to:

### **Listing C.1 List names of all non-system tables**

```
SELECT table_schema, table_name, table_type
FROM information_schema.tables
WHERE table_schema NOT IN('pg_catalog', 'information_schema')
ORDER BY table_schema, table_name;
```

The columns view will give you a listing of all the columns in a particular table or set of tables. In the below example we are listing all the geometry columns found in our hello schema that houses our hello town created in Chapters 1 and 2.

### **Listing C.2 List all columns in hello schema**

```
SELECT c.table_name, c.column_name, c.data_type, c.udt_name AS uname,
c.ordinal_position AS ord_pos,
c.character_maximum_length AS cmaxl ,
c.column_default AS cdefault
FROM information_schema.columns AS c
WHERE table_schema = 'hello'
ORDER BY c.table_name, c.column_name;
```

The results of this query look something like:

Table C.1 Results of query in Listing C.2

table_name	column_name	data_type	u_name	ord_pos	cmax	cdefault
coastline	coastline_id	integer	int4	1		nextval('hello....')
coastline	coastline_name	character varying	varchar	2	150	
coastline	line_geom	USER-DEFINED	geometry	3		
:						

One important way that PostgreSQL is different from databases such as SQL Server and MySQL server that support the information schema is that it has an additional field called "udt\_name" that denotes the PostgreSQL specific data type. Since PostGIS geometry is an add-on module and not part of PostgreSQL, you will see the standard ANSI data\_type listed as USER-DEFINED and the udt\_name storing the fact that it is a geometry.

There are numerous other fields that this view provides so we encourage you to explore it. We have listed below what we consider the most useful fields.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- **table\_name and column\_name** – which should be obvious
- **data\_type** – the ANSI standard data type name for this column
- **udt\_name** – the PostgreSQL specific name – except for user defined types, you can use the data\_type or the udt\_name when creating these fields except in the case of series. Recall we created coastline\_id as a serial data type and PostgreSQL behind the scenes created an integer column, a sequence object and set the default of this new column to the next value of the sequence object.  
`nextval('hello.coastline_coastline_id_seq')::regclass`
- **ordinal\_position** – this is the order that the column appears in the table
- **character\_maximum\_length** - if this is a kind of character field, then this tells you the maximum number of characters allowed for this field.
- **column\_default** – the default value assigned to new records. This can be a constant or the result of a function.

The tables view lists both tables and views (virtual tables). The views view, will give you the name of the view and the view\_definition for each view defined in your database that you have access to. The view\_definition gives you the SQL that defines the view and is very useful for scripting the definitions. In PostgreSQL, you can see how the information\_schema views are defined, though you may not be able to in other databases such as SQL Server since the information\_schema is excluded from this system view.

#### **Listing C.3 View definitions that define information\_schema views**

```
SELECT table_schema, table_name, view_definition,  
is_updatable, is_insertable_into  
FROM information_schema.views  
WHERE table_schema = 'information_schema';
```

In the above examples, we have demonstrated the common meta tables you would find in the ANSI information\_schema. We have also demonstrated the most fundamental of SQL statements. In the next section, we'll tear apart the anatomy of an SQL statement and describe what each part means.

## **C.2 Querying data with Structured Query Language (SQL)**

The cornerstone of any relational database is the declarative language called Structured Query Language (SQL). Although each relational database has slightly different syntax, the fundamentals are pretty much the same across all relational DBMS.

One of the most common things done with SQL is to query relational data. SQL of this nature is often referred to as Data Manipulation Language (DML) and consists of clauses specifically designed for this. The other side of DML is updating data with SQL which we shall cover in the next section.

### **C.2.1 SELECT, FROM, WHERE, and ORDER BY clauses**

For accessing data, you use a SELECT statement usually accompanied with a FROM and a WHERE clause. The SELECT part of the statement restricts the columns to return, the FROM clause determines where the data comes from and the WHERE restricts the number of records to return.

When returning constants or simple calculations that come from nowhere, the FROM is not needed in PostgreSQL, SQL Server or MySQL where as in databases such as Oracle and IBM, you need to do FROM dual or sys.dual or some other dummy table.

#### **BASIC SELECT**

A basic select looks something like the below

```
SELECT gid, item_name, the_geom
FROM feature_items
WHERE item_name LIKE 'Queens%';
```

Keep in mind that PostgreSQL is by default case sensitive, and if you want to do a non-case sensitive search, you would do the below or use the non-portable ILIKE PostgreSQL predicate:

```
SELECT gid, item_name, the_geom
FROM feature_items
WHERE upper(item_name) LIKE 'QUEENS%';
```

You can't guarantee the order that results are returned with the above queries, and often you care about order. The SQL ORDER BY clause satisfies this need for order.

Below is an example that lists all items starting with Lion and orders them by item\_name.

```
SELECT DISTINCT item_name
FROM feature_items
WHERE upper(item_name) LIKE 'LION%'
ORDER BY upper(item_name);
```

For pre PostgreSQL 8.4, you will want to upper case your ORDER BY field for sure but PostgreSQL 8.4 provides a new per database collation feature that makes this not so necessary depending on the collation order you designate for your database.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

## **SELECT \* is not your friend**

Within a SELECT statement you can use the term \* which means select all the fields in the FROM tables. There is also the variant sometable.\* if you only want to select all fields from one table and not all fields from the other tables in your FROM. We highly recommend you stay away from this with production code. This is useful for seeing all the columns of the table when you don't have the table structure in front of you, but can be a real performance drain especially with tables that hold geometries. The reason for that is that if you have a table with a column that is unconstrained by size such as a large text field or geometry field, you'll be pulling all that data across the wire and pulling from disk even when you don't care about the contents of that field.

## **INDEXES**

The WHERE clause often relies on an index to improve row selection. If you have a large number of distinct groupings, it is useful to put an index on that field. For few distinct groupings of records by a column, the index is more harmful than good since it will be ignored by the planner because it is faster to do a table scan and incurs update penalty when updating data. For example putting indexes on things like boolean fields is pointless. They will almost always be ignored.

## **ALIASING**

In the examples on information\_schema, we demonstrated the concept of aliasing. Aliasing is giving a table or a column a different name in your query than how it is defined in the database. It is indispensable when doing what are called SELF JOINS (where you join the same table twice) and need to distinguish between the two or where the two tables you have may have field names in common. The other use is just to make your code easier to read and also reducing typing by shortening longish table names and field names.

Aliasing is done with a statement AS. For table aliases, AS is optional for most ANSI-SQL standard databases including PostgreSQL. For column aliases, AS is optional for most ANSI SQL databases and PostgreSQL 8.4+, but required for PostgreSQL 8.3 and below.

## **Why put AS when you don't need to**

Although AS is an optional clause, we like to always put it in for clarity. To demonstrate which is more understandable?

```
SELECT b.somefield a FROM sometable b;
```

or

```
SELECT b.somefield AS a FROM sometable AS b;
```

## **USING SUBSELECTS**

The SQL language has built into it, support for what are called subselects. Much of the expressiveness and complexity of SQL is keeping subselects straight and knowing when and ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

when not to use them. For PostgreSQL except noted below, any valid SELECT ... can be used as a subselect, and when used in a FROM clause, must be aliased. For some databases such as SQL Server, there are some minor limitations – for example SQL Server doesn't allow an ORDER BY in a subselect without a TOP clause.

A subselect statement is a full SELECT ... FROM .... statement that appears within another SQL statement. It can appear in the following locations of an overall SQL statement:

- a UNION, INTERSECT, EXCEPT as we shall learn about shortly
- in the FROM clause - where you would normally put a table name and behaves like a virtual table. This needs to have an alias name to define how it will be called in other parts of the query and can't reference other FROM table fields as part of its definition. Some databases allow you to do this under certain conditions such as SQL Server's CROSS APPLY.
- In the definition of a calculated column - When used in this context, the subselect can only return one column and one row. This pretty much applies to all databases. PostgreSQL has a somewhat unique feature because of the way it implements rows. A row is a data type and as such can be used as the data type of a column. This allows you to get away with returning a multi-column row as a column expression. Since this is not a feature you will commonly find in other databases and is of limited use, we won't cover it in this chapter.
- In the WHERE part of another SQL query in clauses such as IN, NOT IN, and EXISTS
- In a WITH clause – this is loosely defined as a subquery but is not strictly thought of that way. Note that the WITH clause is only available in PostgreSQL 8.4+. You will also find it in Oracle, SQL Server 2005+, IBM DB2, and Firebird. You will not find it in MySQL.

## WHAT IS A CORRELATED SUBQUERY

A correlated subquery is a subquery that uses fields from the outer query (next level above the subquery) to define the subquery. Correlated subqueries are often used in column expressions and WHERE clauses. They are generally slower than non-correlated subqueries since they have to be calculated for each unique combination of fields and have a dependency on the outer query.

Below are some examples of subselects in action. Do not worry if you don't completely understand the following examples since some require understanding topics we will cover shortly.

### **Listing C.4 Subselect used in a table alias**

```
SELECT s.state, r.cnt_residents, c.land_area
FROM states As s LEFT JOIN
  (SELECT state, COUNT(res_id) As cnt_residents
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

        FROM residents
        GROUP BY state) As r ON s.state = r.state
LEFT JOIN (SELECT state, SUM(ST_Area(the_geom)) As land_area
           FROM counties
           GROUP BY state) As c
          ON s.state = c.state;

```

The above statement uses a subselect to define the derived table we define as r. This is the common use case. We will demonstrate the same using the PostgreSQL 8.4 WITH clause. The WITH clause, sometimes referred to as a Common Table Expressions (CTE), is an advanced ANSI-SQL feature that you will find in SQL Server, IBM DB2, Oracle, and Firebird to name some.

#### **Listing C.5 Same statement written using the WITH clause**

```

WITH
    r AS (
        SELECT state, COUNT(res_id) As cnt_residents
              FROM residents
              GROUP BY state),
    c AS (
        SELECT state, SUM(ST_Area(the_geom)) As land_area
              FROM counties
              GROUP BY state)
SELECT s.state, r.cnt_residents, c.land_area
  FROM states As s LEFT JOIN
       r ON s.state = r.state
  LEFT JOIN c
     ON s.state = c.state;

```

In the next example, we demonstrate how to write the same question using a correlated subquery.

#### **Listing C.6 Same statement written using a correlated subquery**

```

SELECT s.state,
       (SELECT COUNT(res_id)
          FROM residents
         WHERE residents.state = s.state) As cnt_residents
 , (SELECT SUM(ST_Area(the_geom))
        FROM counties
       WHERE counties.state = s.state) AS land_area
  FROM states As s ;

```

Although one can use any of the above to express the same desired result, the strategies used by the planner are very different and depending on what you are doing, one can be much faster than another. For large number of row returns, the correlated subquery approach should be avoided, but in certain cases to prevent duplication of count etc, it is sometimes necessary to use a correlated subquery.

## C.2.2 Joins

PostgreSQL supports all the standard joins and sets defined in the ANSI SQL Standards.

A JOIN is a clause that relates two tables usually by a primary and a foreign key, although the join condition can be arbitrary and in a spatial database you will find that the join is often a proximity one rather than one defined by keys. The clauses LEFT JOIN, INNER JOIN, CROSS JOIN, RIGHT JOIN, FULL JOIN, and NATURAL JOIN exist in the ANSI SQL specs. PostgreSQL supports all of these. SQL Server does too as well. MySQL lacks FULL JOIN support. Oracle does support these too, but also has its own proprietary syntax as well of in WHERE \*= etc that is non-standard and still often used today by longtime Oracle users.

### LEFT JOIN

The LEFT JOIN returns all records from the first table (M) and only records in the second table (N) that match records in table (M). The max number of records returned by a LEFT JOIN is  $(m \times n)$  where m is the number of rows in M and n is the number of rows in the N). The number of columns is the number of columns selected from M + the number of columns selected from N.

Generally speaking if your M table has a primary key which is the joining field, you can expect the minimum number of rows returned to be m and the maximum to be  $m + m \times n - n$ .

NULL placeholders are put in on N table's columns where there is no match in the M table.

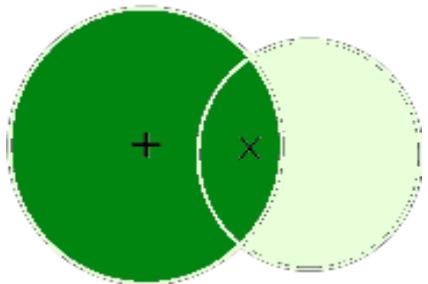


Figure C.1 Diagram of a Left JOIN. The darkened region represents the portion of records returned by a LEFT JOIN. x stands for multiplication and + is additive. The first circle is M and the second circle is N.

Below are a couple of examples of a LEFT JOIN:

```
SELECT c.city_name, a.airport_code,a.airport_name, a.runlength
FROM city As c
LEFT JOIN airports As a ON a.city_code = c.city_code;
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

The above query will list both cities that have airports and cities that don't have airports based on the city\_code which we assume here to be the city primary key and a foreign key in the airports table. If the LEFT JOIN was changed to an INNER JOIN, only cities with airports would be listed. For cities that have no airports, it puts in a null placeholder for the airport fields.

One trick commonly used with LEFT JOINs is to return only unmatched rows by taking advantage of the fact that a LEFT JOIN will return NULL place holders where there is no match. When using this, make sure the field you are joining with is guaranteed to be filled in when there are matches, otherwise you will get spurious results. So for example a good candidate would be the primary key of a table. Below is an example of such a trick.

```
SELECT c.city_name
  FROM city As c
    LEFT JOIN airports As c.city_code
 WHERE a.airport_code IS NULL;
```

In the above example code we are returning all cities with no matching airports. We are making the assumption here that the airport\_code is never NULL in the airports table. If it were ever NULL, this would not work.

#### **INNER JOIN**

The INNER JOIN only returns records which are in both M and N tables. The maximum number of records you can expect from an inner join is  $(m \times n)$ . Generally speaking if your M table has a primary key which is the joining field, you can expect the maximum number of rows to be  $n$ . A classic example is customers joined with orders. If a customer has only 5 orders, the number of rows you will get back with that customer id and name is 5.

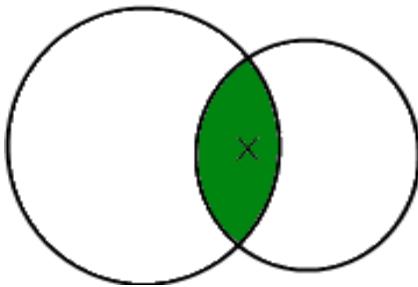


Figure C.2 Diagram of an INNER JOIN. The darkened region represents the portion of records returned by the INNER JOIN. x denotes its multiplicative. The first circle is M and the second circle is N.

Below is an example of an INNER JOIN.

```
SELECT c.city_name, a.airport_code, a.airport_name, a.runlength
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
FROM city AS c
    INNER JOIN airports a ON a.city_code = c.city_code;
```

In the above example we only list cities that have airports and only the airports in them. If we had a spatial database, we can do a JOIN using a spatial function such as ST\_Intersects or ST\_Dwithin if we want to find airports in proximity to a city or in a city region.

#### **RIGHT JOIN**

The RIGHT JOIN returns all records in the N table and only records in the M table that match records in the N. In practice, RIGHT JOIN is rarely used because a RIGHT can always be replaced with a LEFT and most people find reading join clauses from LEFT to RIGHT easier to comprehend. Its behavior is a mirror image of the LEFT JOIN flipping the table order in the clause.

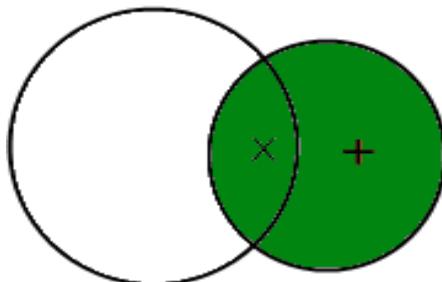


Figure C.3 Diagram of a RIGHT JOIN. The darkened region represents the portion of records returned by a RIGHT JOIN. x stands for multiplication and + is additive. The first circle is M and the second circle is N.

#### **FULL JOIN**

The FULL JOIN returns all records in M and N and puts in NULLs as placeholders in fields where no matching data. There is a lot of debate about the usefulness of this. In general it is rarely used, and some people are of the opinion that it should never be used because it can always be simulated with a UNION [ALL]. Although we rarely use it, in some cases, we find it clearer to use than a UNION [ALL].

The number of columns returned by a FULL JOIN is the same as for a LEFT, RIGHT,INNER join and the number of rows minimum number of rows returned is  $\max(m,n)$  and the maximum is  $(\max(m,n) + mxn - \min(m,n))$ .

#### **Full Joins on Spatial Relationships – forget about it**

While in theory it is possible to do a FULL JOIN using something like the ST\_DWithin or ST\_Intersects spatial function in the FULL JOIN clause, in practice, this is not currently supported even as of PostgreSQL 8.4, PostGIS 1.4.

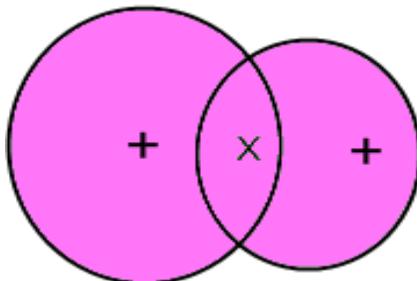


Figure C.4 Diagram of a FULL JOIN. The darkened region represents the portion of records returned by a FULL JOIN. x stands for multiplication and + is additive. The first circle is M and the second circle is N.

#### CROSS JOIN

The CROSS JOIN is the cross multiplication of every record in the M table with every record in the N table. The result of a cross join without a WHERE clause is  $m \times n$  rows. It is sometimes referred to as a cartesian product.

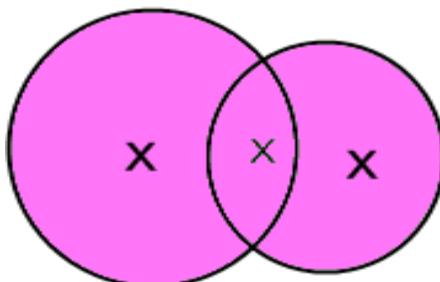


Figure C.5 Diagram of a CROSS JOIN. The darkened region represents the portion of records returned by the CROSS JOIN. x stands for multiplication. The first circle is M and the second circle is N.

Below is an example of a good use for a CROSS JOIN. The below calculates the total price of a product including state tax for each state.

#### **Listing C.7 Calculate the price of every product in every state based on state tax**

```
SELECT p.product_name, s.state, p.base_price * (1 + s.tax) AS total_price
FROM products AS p
CROSS JOIN state AS s;
```

-- 1 Can also be written as

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
SELECT p.product_name, s.state, s.tax) As total_price  
FROM products AS p, state AS s
```

Note that an INNER JOIN can be written with CROSS JOIN or (.) syntax and WHERE part, but we prefer the more explicit INNER JOIN because it is less prone to mistakes. When doing an INNER JOIN with cross join syntax, you put the join fields in the WHERE clause. Primary keys and foreign keys are often put in the INNER JOIN ON clause, but in practice you can put any joining field in there. There is no fast rule about it. The distinction becomes important when doing LEFT JOINS as we saw with the LEFT JOIN orphan trick.

### NATURAL JOIN

A NATURAL JOIN is like an INNER JOIN without an ON clause and supported by many ANSI compliant databases. It auto-magically joins same named fields between tables thus no need for an ON clause.

#### Just say no to the NATURAL JOIN

We highly suggest you stay away from using this. It is a lazy and dangerous way of doing joins that will come to bite you when you add new fields with same names that are totally unrelated. We feel so strongly about not using this, that we will not even demonstrate its use. So when you see it in use instead of thinking COOL, just say NO.

### CHAINING JOINS

The other thing with joins is that you can chain them almost ad-infinitum. You can also combine multiple join types, but when joining different types, either make sure to have all your INNER JOINS first before the LEFTS or put parentheses around them to control order.

#### **Listing C.8 Example of join chaining**

```
SELECT c.last_name, c.first_name, r.rental_id, p.amount, p.payment_date  
FROM customer As C  
    INNER JOIN rental As r ON C.customer_id = r.customer_id  
    LEFT JOIN payment As p  
        ON (p.customer_id = r.customer_id AND p.rental_id = r.customer_id);
```

The above example is from the popular PostgreSQL pagila database. In the above example we find all the customers who have had rentals and list the rental fields as well (note inner join kicks out all customers who haven't made rentals). We then pull the payments they have made for each rental and have nulls if no payment was made but still list the rentals.

### C.2.3 Sets

A set is sort of like a JOIN and often lumped in with joins. What distinguishes a set class of predicates from a JOIN is that it chains together SQL statements that can normally stand by themselves to return a single dataset. The set class defines the kind of chaining behavior. Keep in mind when we talk about sets here, we are not talking about the SET clause you will find in UPDATE statements.

SQL Clauses we consider sets are UNION [ALL], INTERSECT, and EXCEPT. PostgreSQL supports all three of these though many databases only support the UNION [ALL].

One other distinguishing thing about sets is that the number of columns in each SELECT has to be the same and the data types in each column should be the same too or auto cast to the same data type in a non-ambiguous way.

#### Spatial Parallels

One thing that confuses new spatial database users is the parallels between the two terminologies. In general SQL lingua franca you have UNION, INTERSECT, EXCEPT which talks about table rows and when you add space to the mix, you have parallel terminology for geometries (ST\_Union (which is like a UNION), ST\_Collect (which is like a UNION ALL), ST\_Intersection which is like INTERSECT, and ST\_Difference which is like EXCEPT) which serve the same purpose for geometries. So when a new user asks – I want to union my data? It has a lot of possible meanings in spatial SQL.

#### UNION AND UNION ALL

The most common type of set is the UNION and UNION ALL sets. Most relational databases have at least one of these and most have both. A UNION takes 2 SELECT statements and returns a DISTINCT set of these. Which means no two records will be exactly the same. A UNION ALL on the other hand always returns  $n + m$  rows. Where  $n$  is the number of rows in table N and  $m$  is the number of rows in table M.

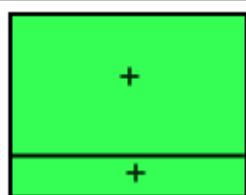


Figure C.6 UNION ALL

A union can have multiple chains each separated by a UNION ALL or UNION. The ORDER BY can only appear once and must be at the end of the chain. The ORDER BY is often denoted by numbers where the number denotes the column number to order by.

A union is generally used to put together results from different tables. Below is an example that will list all water features and land features > 500 units in area and all architecture monuments greater than 1000 dollars. Order results by item name.

### **Listing C.9 Combining water and land features**

```
SELECT water_name As label_name, the_geom,  
       ST_Area(the_geom) As feat_area  
  FROM water_features  
 WHERE ST_Area(the_geom) > 10000  
UNION ALL  
SELECT feat_name As label_name, the_geom,  
       ST_Area(the_geom) As feat_area  
  FROM land_features  
 WHERE ST_Area(feat_geometry) > 500  
UNION ALL  
SELECT arch_name As label_name, the_geom,  
       ST_Area(the_geom) As feat_area  
  
  FROM architecture  
 WHERE price > 1000  
 ORDER BY 1,3;
```

The above example will pull data from 3 tables (water\_features, land\_features, and architecture) and return a single data set ordered by the name of the feature and then the area of the feature.

### **UNION IS OFTEN ACCIDENTALLY USED**

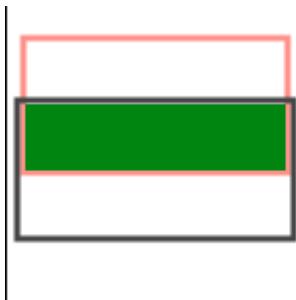
The plain UNION statement is often accidentally used because it is the default option when ALL is not specified. As stated it does an implicit DISTINCT on the data set which makes it slower than a UNION ALL. It also has another side effect of losing geometry records that have the same bounding boxes. This we covered in Chapter 4. In short, be careful. In general you want to use a UNION ALL except when deduping data.

### **INTERSECT**

Intersect is a clause for joining multiple queries together similar to union all. It is defined in the ANSI-SQL standard, but not all databases support it (e.g MySQL does not support it, SQL Server 2000 does not (though SQL Server 2005 and above do).

Intersect returns only the set of records that are common between the two result sets. How it is different from INNER JOIN is that it is not multiplicative and both queries must have the same number of columns. In the diagram below, the green represents what is returned

by an SQL Intersect. Keep in mind later we will look at a spatial intersection which involves intersection of geometries rather than intersection of row spaces.



**Figure C.7 Intersect.** The darkened region is the intersection of two data sets returned by a intersect clause.

INTERSECT is something that is rarely used. There are a couple of reasons for that.

- Many relational databases do not support it
  - It tends to be slower than doing the same trick with an inner join. In PostgreSQL 8.4, the speed of Intersects supposedly has been vastly improved, though in prior versions it was not that great.
  - In some cases, it looks convoluted when you are talking about the same table.
- In some cases it does make your code clearer such as when you have two disparate tables and is useful when chaining more than 2 queries.

#### **Listing C.10 Intersect example**

```
--1
SELECT feature_id, label_name, the_geom
  FROM water_features
 WHERE ST_Area(the_geom) > 500
INTERSECT
SELECT feature_id, label_name, the_geom
  FROM protected_areas
 WHERE induction_year > 2000;
--2
SELECT wf.feature_id, wf.label_name, wf.the_geom
FROM water_features As wf
INNER JOIN
protected_areas As pa ON wf.feature_id = pa.feature_id
WHERE ST_Area(wf.the_geom) > 500
AND pa.induction_year > 2000;
```

(1) The query lists all water features greater than 500 sq units that are also protected areas inducted after year 2000.

Note if the feature id field is not unique, the INNER JOIN runs the chance of multiplying records. To overcome that, you may do a subselect as shown in (2).

The next example demonstrates chaining intersect clauses.

### C.11 Chaining INTERSECT clauses

```
SELECT r
    FROM generate_series(1,3) AS r
INTERSECT
SELECT n
    FROM generate_series(3,8) AS n
INTERSECT
SELECT s
    FROM generate_series(2,3) AS s
```

Keep in mind that you can mix and match with UNION ad EXCEPT as well. The order of precedence is from top query down unless you have subselects parenthetical expressions.

#### EXCEPT

An EXCEPT chains queries together such that the final result only contains records in A that are not in B. The number of columns and type of columns in each chained query must be the same similar to UNION and INTERSECT. The green in the below diagram represents the result of the final query.



Figure C.8 Except

EXCEPT is rarely used, but does come in handy when chaining multiple clauses.

### Listing C.12 Chaining Except and Intersect

```
SELECT r
    FROM generate_series(1,3) AS r
EXCEPT
SELECT n
    FROM generate_series(3,8) AS n
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
INTERSECT
SELECT s
  FROM generate_series(2,3) AS s;
```

### C.2.4 Using SQL Aggregates

Aggregate functions are those that take a group of records and roll them up into one record. In PostgreSQL the standard SUM, MAX, MIN, AVG, COUNT and various statistical aggregates exist out of the box. PostGIS adds approximately 9 to the list and of these, ST\_Collect, ST\_Union, ST\_Extent are the most commonly used. We'll revisit them in the Spatial Aggregates section and provide various use cases for them. In this section we'll just focus on using Aggregates. How you use aggregates is pretty much the same regardless of if they are spatial or not.

When using aggregates in SQL, there are generally the following parts:

- **SELECT and FROM** – This is where you select the fields and where you are pulling data from. You also include the aggregated functions in the select field list.
- **SOMEAGRREGATE(DISTINCT somefield)** – on rare occasions, you will use the DISTINCT clause within an aggregate function to denote only using a distinct set of values to aggregate. This is commonly done with COUNT aggregate to count a unique name only once.

NOTE: with geometries, what is DISTINCTED is the bounding box so is not truly distinct if you have different geometries with the same bounding box.

- **WHERE** – Non aggregate filter – this gets applied before the HAVING
- **HAVING** – similar to WHERE, except used when applying filtering on the already aggregated data
- **GROUP BY** – all fields in the SELECT that are non-aggregate function calls must appear here.

#### FAST FACTS ABOUT AGGREGATE FUNCTIONS

- For most if not all aggregate functions, NULLS are simply ignored. This is important to know because it allows you to do things such as COUNT(the\_geom) as num\_has\_geoms, COUNT(neighborhood) as num\_has\_neighborhoods in the same SELECT statement.
- If you want to count all records, use a field that is never null to count, e.g. COUNT(gid) or a constant such as COUNT(1). You can also use COUNT(\*). Prior to PostgreSQL 8.1, the COUNT(\*) function was really slow, so long time PostgreSQL users tend to avoid that syntax out of habit.
- When grouping by geometries – which is very rare, it is the bounding box of the geometry that is actually grouped on (although the first geometry with that bounding box is used for output), so be very careful, and avoid grouping by geometry if possible unless you have another field in the group by that is distinct for each geometry like

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

the primary key of the table the geometry is coming from.

Here is an example that mixes aggregate SQL functions and Spatial Aggregates together

### **Listing C.13 Combining standard SQL and Spatial Aggregates**

```
-- 1
SELECT n.nei_name,
       SUM(ST_Length(road.the_geom)) AS total_road_length,
       ST_Extent(road.the_geom) AS total_extent,
       COUNT(DISTINCT road.road_name) AS count_of_roads
  FROM neighborhoods AS n
 INNER JOIN roads ON
    ST_Intersects(neighborhoods.the_geom, roads.the_geom)
 WHERE n.city = 'Boston'
   GROUP BY n.nei_name
   HAVING ST_Area(ST_Extent(road.the_geom)) > 1000;
```

- (1) The query above for each neighborhood, specifies the total length of road, and the extent of road way. It also includes count of unique road names and only count neighborhoods where the total area of the extent covered is greater than 1000 square units.

### **C.2.5 Window Functions and Window Aggregates**

PostgreSQL 8.4 introduced the ANSI standard Window functions and aggregates. These allow you to do useful things such as sequentially number results by some sort of ranking, do running subtotals based on a subset of the full set using the concept of a “window frame” and for PostGIS 1.4+ do running geometry Unions and MakeLines which is perhaps a solution in search of a problem, but nevertheless intriguing.

A window frame defines a subset of data within a subquery using the term PARTITION BY and then within that window, you can define orderings and sum results within the window to achieve rolling totals and counts. Microsoft SQL Server, Oracle, and IBM also support this feature with Oracle’s feature set being the strongest and SQL Server’s being weaker than IBM or PostgreSQL. Check out our brief summary comparing these databases to get a sense of the differences.

<http://www.postgresonline.com/journal/index.php?/archives/122-Window-Functions-Comparison-Between-PostgreSQL-8.4,-SQL-Server-2008,-Oracle,-IBM-DB2.html>

PostgreSQL has an additional feature of allowing one to name window frames and reusing those frames by name.

Below is an example of using the ROW\_NUMBER() window function to number streets sequentially that are within 1 kilometer of a police station by their proximity to the police station.

### **Listing C.14 Find roads within 1 km from each police station and number sequentially**

```
SELECT
  -- (1)
  ROW_NUMBER() OVER (
  -- (2)
  PARTITION BY loc.pid
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
-- (3)
        ORDER BY ST_Distance(r.the_geom, loc.the_geom)
                  , r.road_name) As row_num,
    loc.pid, r.road_name, ST_Distance(r.the_geom, loc.the_geom)/1000 As
    dist_km
        FROM land As loc
LEFT JOIN
        road As r
ON ST_DWithin(r.the_geom, loc.the_geom, 1000)
WHERE loc.land_type = 'police station'
ORDER BY pid, row_num;
```

In the above listing we are using (1) the window function called ROW\_NUMBER() to number the results. The (2) partition by clause forces numbering to restart for each unique parcel id (identified by pid) which uniquely identifies a police station. The (3) order by defines the ordering. In this case we are incrementing based on proximity to police station. If two streets happen to be at the same proximity then one will be arbitrarily be n and the other n+1. Our order by includes road\_name as a tie breaker.

In the table below – we show a subset of our resulting table just for two police stations.

<b>row_num</b>	<b>  pid</b>	<b>  road_name</b>	<b>  dist_km</b>
1	000010131	Main Rd	0.228687666823197
2	000010131	Curvy St	0.336867955509993
3	000010131	Elephantine Rd	0.959190964077745
1	000040128	Elephantine Rd	0.587036350160092
2	000040128	Main Rd	0.771250583026646

**Table C.2 Results of window query in Listing C.14**

In the next section we'll learn about another key component of SQL. SQL is good for querying data, but also useful for updating and adding data as well.

## C.3 Update, Insert, and Delete

The other feature of DML is the ability to update, delete and insert data. An update, delete, and insert can combine the aforementioned predicates we learned for selecting data to do cross updates between tables or to formulate a virtual table (subquery) to insert into a physical table. In the exercises that follow, we will demonstrate simple constructs as well as ones that are more complex.

### C.3.1 Updates

#### SIMPLE UPDATE

A simple update will update data to a static value based on a where condition. Below is a simple example of this.

```
UPDATE things
SET status = 'active'
WHERE last_update_date > (CURRENT_TIMESTAMP - '30 day'::interval);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### UPDATE FROM OTHER TABLES

A simple update is one of the more common update statements used. In certain cases however, you will need to read data from a separate table based on some sort of related criteria. In this case you will need to utilize joins within your update statement. Below is a simple example that updates the region code of a point data set if the point falls in the region.

```
UPDATE things
    SET region_code = r.region_code
    FROM regions As r
    WHERE ST_Intersects(things.the_geom, r.the_geom);
```

### UPDATE WITH SUB SELECTS

A subselect as we learned earlier is like a virtual table. It can be used in update statements similar to the way you use regular tables. In a regular update statement even involving ones with table joins, you can not update a table value to the aggregation of another table field. A way to get around this limitation of SQL is to use a subselect. Below is such an example that tallies the number of objects in a region.

#### **Listing C.15 Update region table with total number of objects in each region**

```
UPDATE regions
    SET total_objects = ts.cnt
    FROM (SELECT t.region_code, COUNT(t.gid) As cnt
        FROM things AS t
        GROUP BY t.region_code) As ts
    WHERE regions.region_code = ts.region_code;
```

If you are updating all rows in a table, it is often more efficient to build the table from scratch and use an insert statement rather than an update statement. The reason for this is that an UPDATE is really an INSERT and a DELETE. Because of the MVCC nature of PostgreSQL, PostgreSQL will remove the old row and replace with the new row in the active heap. In the next section we'll learn how to perform inserts.

### C.3.2 Inserts

Just like the UPDATE statement, we can have simple inserts that insert constants as well as more complex ones that read from other tables or aggregate data. We'll demonstrate some of these constructs.

#### SIMPLE INSERT

The simple insert just inserts constants and it comes in 3 basic forms.

There is the single value constructor approach that has been in existence in PostgreSQL since the 6.0 days and is pretty well supported across all relational databases.

#### **Listing C.16 Simple value insert – insert a single point**

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
INSERT INTO points_of_interest(fe_name, the_geom)
VALUES ('Highland Golf Club',
        ST_SetSRID(ST_Point(-70.063656, 42.037715), 4269));
```

The next popular is the multi-row value constructor syntax introduced in SQL-92 which we demonstrated a lot in this book. This syntax was introduced in PostgreSQL 8.2, IBM DB2, has been supported for a long time in MySQL (I think since 3+), and was introduced in SQL Server 2008. As of this writing Oracle has yet to support this useful construct. The multi row constructor is useful for adding more than a single row or as a faster way of creating a derived table with just constants. Below is such an example excerpted from earlier chapters. The multi row insert is similar to the single. It starts with the word VALUES and then each row is enclosed in (...) and separated with ,.

#### **Listing C.17 Multi value row insert – 2 insert facilities**

```
INSERT INTO hello.poi(poi_name, poi_geom)
VALUES ('Park',
        ST_GeomFromText('POLYGON ((86980 67760,
        43975 71292, 43420 56700, 91400 35280, 91680 72460, 89460
        75500, 86980 67760))' ),
        ('Zoo',
        ST_GeomFromText('POLYGON ((41715 67525, 61393 64101, 91505 49252, 91400
        35280, 41715 67525))' ));
```

The last kind of simple insert is one that uses the SELECT clause. In the simplest example it does not have a FROM. Some people prefer this syntax because it allows you to alias what the value is right next to the constant. It is also a necessary syntax for the more complex kind of inserts we will demonstrate in the next section. Note: this syntax is supported by PostgreSQL (all versions), MySQL, and SQL Server. To use in something like Oracle or IBM DB2 you need to include a FROM clause like FROM dual or sys.dual.

#### **Listing C.18 Simple value insert using SELECT instead of VALUES**

```
INSERT INTO points_of_interest(fe_name, the_geom)
SELECT 'Highland Golf Club' AS fe_name,
       ST_SetSRID(ST_Point(-70.063656, 42.037715), 4269) AS the_geom;

INSERT INTO hello.poi(poi_name, poi_geom)
SELECT 'Park' AS poi_name,
       ST_GeomFromText('POLYGON ((86980 67760,
        43975 71292, 43420 56700, 91400 35280, 91680 72460, 89460
        75500, 86980 67760))' AS poi_geom
UNION ALL
SELECT 'Zoo' AS poi_name,
       ST_GeomFromText('POLYGON ((41715 67525, 61393 64101, 91505 49252, 91400
        35280, 41715 67525))' AS poi_geom;
```

The above is the standard way of inserting multiple values into a table. This was the only way to do a multi row in pre PostgreSQL 8.2. This is also the only way to do it in SQL Server 2005 and below.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

## ADVANCED INSERT

The advanced insert is not that advanced. You use this syntax to copy data from one table or query to another table. In the simplest case, you are copying a filtered set of data from another table. It uses the SELECT syntax usually with a FROM and sometimes accompanying joins.

### **Listing C.19 Insert a subset of rows from one table to another**

```
INSERT INTO polygons_of_interest(fe_name, the_geom, interest_type)
SELECT pid, the_geom, 'less than 300 sqft' As interest_type
FROM parcels WHERE ST_Area(the_geom) < 300;
```

A slightly more advanced insert is one that joins several tables together. In this scenario the SELECT FROM is just a standard SQL SELECT statement with joins or one that consists of subselects. Below is a somewhat complex case. Given a table of polygon chain link edges, constructs polygons and stuffs it into a new table of polygons.

### **Listing C.20 Construct Polygons from line work and insert into polygon table**

```
INSERT INTO polygons(polyid, the_geom)
SELECT polyid, ST_Multi(final.the_geom) As the_geom
FROM (SELECT pc.polyid, ST_BuildArea(ST_Collect(pc.the_geom)) As the_geom
      FROM
      (SELECT p.right_poly as polyid, lw.the_geom
        FROM polychain p INNER JOIN linework lw ON
              lw.tlid = p.tlid
        WHERE (p.right_poly <> p.left_poly OR p.left_poly IS NULL)
      UNION ALL
      SELECT p.left_poly as polyid, lw.the_geom
        FROM polychain p INNER JOIN linework lw ON
              lw.tlid = p.tlid
        WHERE (p.right_poly <> p.left_poly OR p.right_poly IS NULL)
    ) As pc
  GROUP BY polyid) As final;
```

## SELECT INTO AND CREATE TABLE AS

Another form of the insert statement is what we commonly refer to as a Bulk Insert. In this kind of insert, not only are you inserting data, but you are also creating the table to hold the data as well in a single statement. PostgreSQL supports two basic forms of this:

- One is the standard SELECT ... INTO which a lot of relational databases support. We prefer this since because its more cross platform (will work on SQL Server as well as MySQL for example).
- The other is a CREATE TABLE AS ..SELECT, which is not as well supported by other relational databases.

In both cases any valid SELECT statement or WITH statement can be used. Below are

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

examples of the same statement written using SELECT INTO and CREATE TABLE As

### **Listing C.21 Example SELECT INTO and CREATE TABLE**

```
-- 1 SELECT INTO
SELECT t.region_code, COUNT(t.gid) AS cnt
    INTO thingy_summary
    FROM things AS t
GROUP BY t.region_code;

-- 2 CREATE TABLE AS
CREATE TABLE thingy_summary AS
    SELECT t.region_code, COUNT(t.gid) AS cnt
        FROM things AS t
    GROUP BY t.region_code;
```

(1) is the standard more cross database platform way of creating a table and inserting the data in one go. (2) is more of a PostgreSQL specific way that is a bit clearer in style, but not as cross platform. If you need your code to support multiple vendor databases, you are better off with (1).

### **C.3.3 Deletes**

Deletes are the most limiting as far as joins go. When doing a delete you can not join with any data so to define a subset of data to be deleted based on other information, you generally need to use an [NOT] EXISTS or [NOT] IN clause.

#### **SIMPLE DELETE**

A simple delete has no sub selects but usually has a WHERE clause. ALL the data in a table is deleted and logged if you are missing a WHERE clause. Below is an example of a standard DELETE.

```
DELETE FROM streets WHERE fe_name LIKE 'Mass%';
```

#### **TRUNCATE TABLE**

In cases where you want to delete all the data in a table you can use the much faster TRUNCATE TABLE statement. The TRUNCATE TABLE is considerably faster since it does much less logging, but it can only be used in tables that are not involved in a foreign key relationship. Below is an example of it at work.

```
TRUNCATE TABLE streets;
```

#### **ADVANCED DELETE**

An advanced delete involves sub selects in the WHERE clause. These are useful for cases where you need to delete all data in your current table that is in the table you are adding from or you need to delete duplicate records. Below is an example to delete duplicate records.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

**Listing C.22 Deleting duplicate records**

```
DELETE
  FROM      sometable
 WHERE     someuniquekey NOT IN
 (SELECT      MAX(dup.someuniquekey)
   FROM      sometable As dup
  GROUP BY    dup.dupcolumn1, dup.dupcolumn2, dup.dupcolumn3);
```

Now that we have covered the basics of SQL in PostgreSQL, that concludes our SQL Primer. In the next appendix, we'll cover PostgreSQL unique features such as its powerful language and stored function functionality, its extensive array support and how security and backup are managed.

# Appendix D:

## PostgreSQL Features

This Appendix covers

- Useful PostgreSQL resources
- Connecting to PostgreSQL Server
- Controlling access to data
- Backup and Restore
- Data Structures and PostgreSQL Objects
- Writing PostgreSQL Stored functions

In this appendix, we cover features and behavior that are fairly unique to PostgreSQL that make it a little different from working with other relational databases.

### **D.1 Useful PostgreSQL Resources**

#### **D.1.1 General**

- PostgreSQL Wiki -- [http://wiki.postgresql.org/wiki/Main\\_Page](http://wiki.postgresql.org/wiki/Main_Page)  
User contributed articles about various PostgreSQL topics from Administration, Performance Tuning and Writing queries to using PostgreSQL in various application and programming environments. Check out the Code Snippets repository for lots of useful PostgreSQL functions you can copy and paste in your database  
<http://wiki.postgresql.org/wiki/Category:Snippets>
- Planet PostgreSQL - <http://www.planetpostgresql.org/> blog roll of PostgreSQL specific blogs. Learn from hard-core long time PostgreSQL users how to get the most out of

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

PostgreSQL. Also learn what's new and hot in the PostgreSQL front.

- Our blog / Journal -- <http://www.postgresonline.com> - We try to cater to new PostgreSQL users, programmers and databases users coming from other database systems such as MySQL, SQL Server or Oracle.
- PostgreSQL Main site: <http://www.postgresql.org> - you can download the source from here as well as getting flash news. You can also download the manual in PDF form or leaf through the HTML version on-line. The manual is huge and consists of 5 volumes. As of this writing you can also purchase printed versions of each volume. The latest versions of these are offered by linlibrary.com -- <http://www.linlibrary.com/postgresql/index.html> or purchasable through Amazon.com <http://www.amazon.com/exec/obidos/search-handle-url/index=books&field-author=The%20PostgreSQL%20Global%20Development%20Group>
- PostgreSQL Yum repository -- If you are a centos, fedora, or redhat Enterprise Linux user, this is a painless way of installing PostgreSQL service and keeping it up to date. They provide yum updates for even the latest beta versions of PostgreSQL -- [http://yum.pgsqrlrms.org/reporpms/repoview/letter\\_p.group.html](http://yum.pgsqrlrms.org/reporpms/repoview/letter_p.group.html)

### **D.1.2 Performance**

- Explaining Explain -- [http://wiki.postgresql.org/images/4/45/Explaining\\_EXPLAIN.pdf](http://wiki.postgresql.org/images/4/45/Explaining_EXPLAIN.pdf) covers planner basics up through PostgreSQL 8.3 and some of PostgreSQL 8.4

### **D.1.3 PostgreSQL specific tools**

- Packaged with PostgreSQL psql, pg\_dump, pg\_dump\_all, pg\_restore. Command line use for querying PostgreSQL, backing up and restoring. You can get from your Linux distro, Mac OSX distro, EnterpriseDb one-click installers or download source from PostgreSQL core site and compile yourself.
- PgAdmin III -- comes packaged with PostgreSQL, but binaries and source can be downloaded separately if you need to install on a workstation not running PostgreSQL server service. <http://www.pgadmin.org/> These are also available via common OS distros.
- PhpPgAdmin -- php web based database administration tool for PostgreSQL patterned after phpMyAdmin. <http://phppgadmin.sourceforge.net/>

## **D.2 Connecting to PostgreSQL Server**

Before you can even create a spatial database or any database for that matter you need to be able to log into your PostgreSQL server via PgAdmin III or psql. In this section, we'll cover the basics, most common problems and how to work around them.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### D.2.1 Core configuration files

If you are just starting out and have just installed your PostgreSQL server, you will want to pay attention to the following key files which are all located in the data cluster of your install.

#### LOCATION OF DATA CLUSTER

- For windows users -- this is the data directory you are prompted for during install, which if you don't change it is located in C:\Program Files\PostgreSQL\8.4\data (for 32-bit systems) and for 64-bit systems is located in C:\Program Files (x86)\PostgreSQL\8.4\data.
- For other users, most likely you had to do an initDb and the path you gave to the -D is the location of the data cluster.

#### POSTGRESQL.CONF

The postgresql.conf is the most important file. It contains all the memory configurations, defaults etc and also contains the listening addresses and ports. If you are running multiple versions of PostgreSQL or just multiple instances, you need to have them listening on different ports or different addresses otherwise the first one to start will prevent others from starting. However you can have multiple instances sharing the same binaries as long as they have a different data cluster. In practice that is rarely done.

The two settings of most importance for getting started are the listen\_addresses and port

If you want to be able to allow your server to be accessed by remote computers without need for ssh tunneling then set it to:

```
listen_addresses = '*'                      # what IP address(es) to listen on;
                                              # comma-separated list of addresses;
                                              # defaults to 'localhost', '*' = all
                                              # (change requires restart)
```

The port setting is defaulted to 5432, but if you want to run multiple PostgreSQL services, you will need to have each one set differently. For example we run PostgreSQL 8.2,8.3,8.4 on our servers for testing. We have our port for 8.4 set to something like:

```
port = 5434
```

Some of the other important settings in the postgresql.conf file we discuss in Chapter 9 on Performance Tuning.

#### PG\_HBA.CONF

The pg\_hba.conf file controls what users on what ip ranges can connect to the PostgreSQL service/daemon as well as what authentication scheme is used for them.

### D.2.2 Launching PSQL

If you are using a windowless server such as a Linux server with no windowing installed (standard for most production web/app servers), you will need to use PSQL at least once on the server to get everything rolling.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

From the server do the following:

```
psql -h localhost -U postgres
```

to verify you can connect. If you can't refer to the Connection difficulties section below.

### **D.2.3 Launching PgAdminIII**

For new users, we highly recommend the PgAdmin III gui. If you installed with one of the one-click desktop installers, PgAdmin III is usually included. On windows you can find it under

```
Start->Programs->PostgreSQL 8.4->PgAdmin III
```

For Linux distros it varies.

You can also install PgAdmin on a regular desktop pc that doesn't have PostgreSQL server installed. Simply download one of the available binaries from:

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

For versions of PgAdminIII 1.10 and above, you can also launch psql for that specific database right from PgAdminIII.

This is useful to take advantage of psql specialty features like dumping out to file or importing from files. To access psql from

PgAdmin III.

1. Select the database
2. Under the Plugins icon -- psql. If your psql is launching the wrong version, you may want to change it in the plugins.ini file in the PgAdminIII install folder.

### **D.2.4 Connection difficulties**

If you are connecting to the server from a separate desktop pc, then you want the server to be listening on an IP address or all addresses it has and to allow remote connections. After you make changes to configuration files to allow these things, then you must restart the PostgreSQL service. On windows you just go into Services Manager and restart.

On Linux, if you have it installed as a service which it usually is if you installed from YUM or one-click installer , you can usually do the below from a shell prompt.

```
service postgresql restart
```

#### **CONNECTION REFUSED**

If you get an error in PgAdminIII something of the form –

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

could not connect to server: Connection refused (0x0000274D/10061)  
 Is the server running on host "blah blah blah" and accepting TCP/IP  
 connections on port 5432?

Then most likely one or more of the following situations is at fault

- Your PostgreSQL Server is not started
- Your PostgreSQL Server service is only listening on localhost or non-accessible IP
- Your PostgreSQL Server is not listening on the port you think its listening on
- Your firewall is getting in the way. Generally for firewall issues you can usually set up an SSH tunnel which we describe a little here: <http://www.postgresonline.com/journal/index.php?archives/38-PuTTY-for-SSH-Tunneling-to-PostgreSQL-Server.html>. Some third party PostgreSQL interfaces also have SSH tunneling built in. PgAdmin III does not.

#### **NO ENTRY IN PG\_HBA.CONF**

If you get a no entry error, then it means your pg\_hba.conf is not configured right to allow remote connections or you have an error in your file. We generally set our pg\_hba.conf to something like the below which allows all local connections to be trusted – meaning you do not need to provide a password to connect just a valid postgresql username if you are connecting locally. If you are very security conscious -- you could leave this line out and just require md5 or some other more secure security scheme. To make connection a bit easier, you can set up a .pgpass file we will discuss shortly.

#### **Listing D.1 Example pg\_hba.conf – trust all local connections**

```
# TYPE   DATABASE   USER           CIDR-ADDRESS      METHOD
# IPv4 local connections:
host    all        all            127.0.0.1/32    trust
# IPv6 local connections:
#host   all        all            ::1/128          md5
host   all        all            0.0.0.0/0       md5
```

Keep in mind that the order of these statements is important because PostgreSQL will check each one in order and apply the first matching rule that meets the profile of the person trying to connect. This means that if you accidentally put the 0.0.0.0/0 md5 rule above all the others, then you will have to provide a password even when connecting locally.

#### **D.2.5 Enabling advanced administration for PgAdmin III**

Once you have connected for the first time, you will be able to administer the postgresql.conf, pg\_hba.conf and also check server status and other goodies remotely from

another computer using PgAdmin III, if you run the adminpack.sql which is located in the contrib folder of your PostgreSQL install.

Do the following from the command line on the PostgreSQL server. The below example uses the default windows install path for PostgreSQL 8.4, if you are on linux or running a lower version of PostgreSQL the install paths of things will be different and you may not even need to include psql path since its usually in the default bin folder on Linux.

```
"C:/Program Files/PostgreSQL/8.4/bin/psql" -h localhost -U postgres -d postgres -p 5434  
--file="C:/Program Files/PostgreSQL/8.4/share/contrib/adminpack.sql"
```

If you have your PostgreSQL installed on a 64-bit version of Windows, it gets installed by default in the Program Files (x86) since PostgreSQL as of this writing does not yet run in 64-bit mode on Windows. It does however run fine in 64-bit mode on Linux and does use 64-bit memory in Windows since it delegates memory management to the server.

```
"C:/Program Files (x86)/PostgreSQL/8.4/bin/psql" -h localhost -U postgres -d postgres -p  
5432 --file="C:/Program Files (x86)/PostgreSQL/8.4/share/contrib/adminpack.sql"
```

From then on, to access the administrative postgresql.conf, pg\_hba.conf from within PgAdmin from any desktop.

1. Register the server in PgAdmin III
2. Go under menu Tools->Server Configuration (you should see both config files there).

## **D.3 Controlling Access to data**

There are two parts of access control in PostgreSQL. There is control of who can login and how they can login which we saw a glimpse of earlier. Once a person is logged in, there is control of what kind of data can be accessed, created, deleted and edited.

### **D.3.1 Connection Rules**

The connection rules are controlled by three files:

- **postgresql.conf** – controls what ports and ips the server listens on and whether Secure Socket Layer (SSL) connection is required. For LDAP like connectivity, it also holds such information as the Kerberos settings to connect to a Kerberos authenticating server.
- **pg\_hba.conf** - controls whether people can connect based on their IP range and what kind of authentication is supported for each.

Common ones are

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

- o **md5** – using md5 encryption – what most people use particularly for web apps.
- o **trust** – ignores password. Never use this except in a tightly defined local network or on local pc that has good firewall protection from ip spoofers or that only listens on local port.
- o **ident** – trust a user based on their local identity determined by the OS. Again generally only used for local authentication.
- o **reject** - this is the kind of authentication that doesn't allow you to authenticate. You use this if you want to ban certain IP ranges or everyone that is not on your network but that would otherwise be allowed by a broader ip range rule. In such cases, you would put this rule above the broader rule so that it is resolved first.

Less common ones – but particularly useful if you are in an enterprise network using LDAP or Active Directory (there are even MORE such as pam and some others we've never heard of)

- o **krb4/5** – Kerberos connection – deprecated – don't use.
- o **sspi** – only supported on windows and requires the PostgreSQL server to be running on Windows. It is designed for connecting via windows authentication or NT authentication. Sits on top of Kerberos.
- o **gss** – Industry defined protocol similar to sspi but not requiring a Windows server – sits on top of Kerberos.
- o **ldap** – authentication via a ldap directory service such as Active Directory or Novell directory service. The user must exist in the PostgreSQL server, but the password verification uses ldap.
- **pg\_ident.conf** – allows you to map an authenticated user to a database defined user/login. For example you might want root to login as postgres.
- **pgpass.conf , .pgpass** - This is a local configuration file that stores the usernames, server, and passwords that you connect to. If you are using PgAdmin, then pgAdmin creates this for you automatically and you can export the contents of it using the File -> Open pgpass.conf (for windows users this generally exists in %APPDATA%\postgresql\pgpass.conf ) and on Linux/Unix systems, the file is called .pgpass and should be put in ~/.pgpass . This file will be used by psql, pg\_dump, pg\_restore to automatically log you into a PostgreSQL server without prompting for the password. It is useful to have if you don't have trust enabled and you need to schedule backup jobs etc. Keep in mind that the file must exist under the account doing the work so if you are doing backup under a service account such as postgres, or Administrator, then you need to copy the file into the respective home directories of these accounts.

### **D.3.2 Users and Groups (Roles)**

The PostgreSQL security model from PostgreSQL 8.1+ is composed of Roles and Roles sit on the server level - NOT the database level. In prior versions of PostgreSQL we had groups and users instead of roles. Roles can inherit from each other, can have login rights, and can contain other member roles. A user is a role with login rights. Unlike most databases, PostgreSQL does not make a distinction between a user and a group. You can easily morph a user into a group simply by adding members to the roles. Roles are all there is.

In a relational database system, you create users (roles) and grant rights using a kind of SQL called, Data Control Language (DCL). DCL varies most significantly from database product to database product because of the idiosyncrasies of how each database manages security. There do exist ANSI SQL standards dictating the syntax, but this is much less followed than those for Data Manipulation Language (DML) and Data Definition Language (DDL). PostgreSQL does try to follow the standard as much as possible, but also deviates much like most relational databases. In this section, we'll go over PostgreSQL security concepts, and also demonstrate PostgreSQL's specific dialect of DCL. In terms of roles -- Oracle is probably closest in syntax to PostgreSQL of the databases most people will be familiar with.

Below is a listing of general user database concepts and their equivalent in PostgreSQL

**Table D.1 PostgreSQL role concepts and parallels to other databases**

<b>General Concept</b>	<b>PostgreSQL equivalent</b>
User (login)	Role with login rights and generally contains no member roles
Group	A role with member roles (usually no login rights)
Database User	User with grant rights to a database object.
Sys Dba	A role that has SUPERUSER rights
public	The built-in role that all authenticated users belong to. SQL Server has such a role too and it coincidentally is also called public.

There also exist ANSI standard information\_schema tables for interrogating roles, privileges and so forth, but each database system we have worked with arbitrarily implement the ones they feel like in this regard, to the point of relying on any of the role/privilege based tables in information\_schema not terribly portable between databases.

### **D.3.3 Rights management**

PostgreSQL roles can in addition to containing other roles, can be contained by many other roles. In other words, each role/user/group can have many parents or belong to many

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

groups. Roles in PostgreSQL do not necessarily inherit rights from their parent roles, which is a bit of cause of confusion for many people. We'll go over this shortly.

#### CORE SERVER RIGHTS

There are a couple of core rights a role can have which are granted at the server level. These rights are not inheritable so if you add a user to a group role with these rights, then that user will not by default be able to do these things even if you mark them as inheriting from their roles. To relinquish these simply prefix with a NO such as NOSUPERUSER, NOINHERIT ..

- SUPERUSER – has super powers (to relinquish super user rights use NOSUPERUSER)
- INHERIT – when marked as inherit, this means the role inherits the rights of its parent roles. In later versions of PostgreSQL this is the default behavior.
- CREATEDB – this gives the role ability to create a database
- CREATEROLE – this gives a role the ability to create other roles that are not super user roles and that it is not a member of. Only a super user can create other super users
- LOGIN – gives the role rights to login. Generally speaking people create group roles by not giving the group role rights to login, though in theory you can have a group role that has rights to login. In practice having group roles that can login is confusing and PgAdmin in fact prevents you from doing this via the GUI interface.

#### THE POWER WITHOUT THE POWER USING SET ROLE

People often make the mistaken assumption that a member of a role with super powers always has super powers. This is never the case as mentioned above because SUPERUSER rights and the like are never inheritable. It is also useful to prevent yourself from shooting yourself in the foot or to appease a boss. You can add yourself or the boss to a super user role, but not cause damage casually; How? When the boss asks -- "I need power to do everything," as bosses often demand -- you can nod and say -- "Yes I have added you to a group that has power to do everything," and blissfully walk away. Let us demonstrate this super user without super user powers with this simple exercise.

#### **Listing D.2 Creating super users and groups**

```
CREATE ROLE office_of_president SUPERUSER;

CREATE ROLE regina INHERIT LOGIN PASSWORD 'queen';
GRANT office_of_president TO regina;

CREATE ROLE leo LOGIN PASSWORD 'lion king' SUPERUSER;
```

We have a simple script above that creates a group called "office\_of\_president", and two users "leo" and "regina". Leo has super user rights and Regina is a member of a group that

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

has super user rights. Leo is always omnipotent. Regina is only omnipotent when she summons her powers of omnipotence. Let us demonstrate with these scenarios:

Leo logs in and creates a database called kingdom by running this command:  
CREATE DATABASE kingdom;

He is successful.

Regina logs in and tries to create a database called fortress  
CREATE DATABASE fortress;

She gets a message:  
ERROR: permission denied to create database

She is frustrated. She is a member of the mighty role of "office\_of\_president" and she is marked as inheriting rights. She must be able to create a database, but how? First recall that SUPERUSER rights are never inheritable, but they can be summoned. Regina summons her powers of office\_of\_president and then creates the database:

```
SET ROLE office_of_president;
CREATE DATABASE fortress;
```

She succeeds.

Being dissatisfied with the state of affairs -- she summons her powers to put things into order.

```
SET ROLE office_of_president;
ALTER ROLE leo NOSUPERUSER;
ALTER ROLE regina SUPERUSER;
```

And now Leo is powerless and she is always omnipotent without need to summon super powers.

#### **TO INHERIT OR NOT TO INHERIT**

One thing that makes PostgreSQL stand out from other databases is this idea of INHERIT and NOINHERIT as well as mentioned above that some rights are never inheritable. You can define a user that belongs to many groups, but does not inherit the permissions of those groups. This little idiosyncracy dumb-founds people because people often accidentally mark their login roles as not inheriting rights from their parent roles and scratch their heads when the user complains that they can't do anything.

Why you ask would anyone ever create a user that doesn't inherit rights of its membership groups?

One reason is for testing. Lets imagine you create a user, that is a member of every single group role under the sun, but that user (login role) does not inherit rights from any

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

role it is a member of. What can this user do? It can for a specific session, promote itself to have rights of any role it is a member of much like our queen promoted herself to the rights of her powerful group. This is somewhat useful for testing different membership rights or only giving a user certain rights within an application. It also as we demonstrated earlier -- prevents you from doing superuser damage without trying to deliberately do super user damage.

#### **SESSION AUTHORIZATION**

Session authorization is similar to set role but the distinction is that in SET ROLE you are summoning your powers as a member of a role, but with SET SESSION AUTHORIZATION you are becoming that role. Basically you are impersonating another user. Only a superuser can impersonate another user, but any member of a role can do a SET role to the roles they are a member of. Impersonation is useful when you are creating a bunch of objects that you want owned by a specific user, but you don't want to have to change owner to that person for each creation or you want to run commands but limit yourself only to the rights that user has just to verify what a user can do.

#### **GRANTING RIGHTS TO OBJECTS**

As with other databases, PostgreSQL allows you to grant rights to specific objects in a database. The database owner, owner of an object can grant rights to others and in addition to granting rights to objects you can give others the right to grant rights to objects using the WITH GRANT OPTION. GRANT as we saw in the previous examples is also used to add a user to a group role. If a user is GRANTED rights to a role WITH ADMIN OPTION, then they can add or remove users to that role.

#### **POSTGRESQL 8.4 COLUMN LEVEL PERMISSIONS**

PostgreSQL 8.4 introduced column level permissions, which allows for granting read/write/update permissions to individual columns of a table. Largely in part to the work of Stephen Frost who is also a long time contributor of the PostGIS project (tiger geocoder).

In the following exercise, we'll list the common GRANT usages:

#### **Listing D.3 Common Grant options**

-- 1  
GRANT ALL PRIVILEGES ON DATABASE postgis\_in\_action to leo WITH GRANT OPTION;

-- 2  
GRANT ALL PRIVILEGES ON SCHEMA world to leo;

-- 3  
GRANT SELECT, INSERT ON geometry\_columns TO leo;

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

**--4**

```
GRANT SELECT ON spatial_ref_sys TO public;
GRANT UPDATE(proj4text,srtext) ON spatial_ref_sys TO leo;
```

(1) We are granting all rights to Leo for our database and also allowing him to give grant rights to others he chooses, but this just means Leo can connect to the database and create new schemas in the database. It doesn't give him rights to view existing data of existing tables for example or create objects in schemas he doesn't have rights for. (2) We give leo all rights to schema "world". This allows him not to view or edit existing data, but allows him to create new objects in world. (3) We give leo the rights to view and add data in geometry\_columns, but not update or delete. (4) We give everyone rights to view data in spatial\_ref\_sys and leo rights to update the proj4text and srtext columns in spatial\_ref\_sys (only works for PostgreSQL 8.4+).

As you can see in the above, the process of granting rights in PostgreSQL is a bit annoying. Annoying in the sense that often times you want to grant rights to a whole database or schema, and there is no one liner in PostgreSQL for doing such a thing. To get around this annoyance, one can do one of the following:

- Use SQL to script desired rights as we describe in this article <http://www.postgresonline.com/journal/index.php?archives/30-DML-to-generate-DDL-and-DCL--Making-structural-and-Permission-changes-to-multiple-tables.html>
- Use PgAdmin III -- Grant wizard which allows you to select a list of objects and grant rights to specified roles. To use the Grant Wizard
  - Select a schema
  - Select Tools -> Grant Wizard
  - Select Objects you want , privileges and roles

### **PostgreSQL 9.0 enhancement to GRANT and REVOKE**

In PostgreSQL 9.0, there will be an extension to the GRANT and REVOKE feature that will allow one to GRANT ALL to ALL tables/function etc. in a schema or across the database. This is described in <http://www.depesz.com/index.php/2009/11/07/waiting-for-8-5-grant-all/>

### **REVOKING RIGHTS**

You can revoke rights just as easily as you can Grant rights. You revoke rights with the REVOKE command. In this section we'll demonstrate common REVOKE statements.

The revoke command is used to revoke any kind of permission that is granted with the GRANT command. We'll demonstrate our favorite revoke command, and that is revoking connection rights from public group. As we mentioned, the public group is the group that ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

everyone belongs to and that in general most databases allow connect access to when created. What this means is that any authenticated user can connect to the database and browse the structure of the tables etc. This is not always desirable. To prevent this, you can initiate the following command:

```
REVOKE CONNECT ON DATABASE postgis_in_action FROM public;
```

Now that we have covered the basics of security, we will cover something perhaps even more important, and that is backup and restore.

## D.4 Backup and Restore

PostgreSQL has one of the richest backup and restore tools of perhaps any open source database and rivals and often surpasses those offered by the commercial relational database systems. Backup is accomplished with

- **pg\_dump** – which can do custom compressed backups, sql backups as well as selective backup of schemas and other objects all in a single command line. SQL backups are restored with psql and compressed and tar backups are restored with pg\_restore.
- **pg\_dumpall** – does only sql backups and system server configuration backups such as backups of users and tablespaces and the like. It can also do a whole backup of the server – all databases included. The backup is a regular sql backup, so does not allow selective restore that you can do with pg\_dump. Backups done with pg\_dumpall are restored with psql.

For restoring data, PostgreSQL comes packaged with psql and pg\_restore

- **pg\_restore** – used for restoring compressed and tar backups created with pg\_dump. pg\_restore will allow you to restore select objects and also to generate a list of objects backed up in a backup. You can then edit this list to fine grain what you would like to restore from the backup.
- **psql** – used for restoring or running an sql file such as those generated by pg\_dumpall or pg\_dump when sql mode is chosen or PostGIS shp2pgsql shapefile import tool.

## IMPROVEMENTS TO PG\_RESTORE IN 8.4

In PostgreSQL 8.4, pg\_restore was enhanced to include a jobs= option. This option is particularly useful for large backups and defines the number of parallel threads used to do a restore. If you backup a database with PostgreSQL 8.4+ pg\_dump, then you can specify a jobs=2 or more. This does a parallel restore. Depending on your disk io and cpu, setting this can half or even greater reduce the time of a restore. For example a restore of a PostGIS 800 gig byte database would take about 12 hours in prior versions and in 8.4 would be reduced to 6 hours or less.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

### D.4.1 Backup

The pg\_dump command line tool that comes packaged with PostgreSQL, is our preferred for doing database backups. The main reason being is that it creates a nice compressed backup and allows for selective restore of objects. Most of the time when you need to restore things, its because a user accidentally destroyed data and in those cases, you don't want to have to restore the whole database. In this section we'll go through some common pg\_dump and pg\_dumpall statements used for backing up data.

#### **Listing D.4 Common Backup statements**

```
-- 1
pg_dump -i -h localhost -p 5432 -U someuser -F c -b -v -f
"/pgbak/somedb.backup" somedb

-- 2
pg_dump -i -h someserver -p 5432 -U someuser -E latin1 -F c -b -v -f
"/pgbak/somedb.backup" somedb

-- 3
pg_dump -i -h someserver -p 5432 -U postgres -F p -o -v -n pgagent -f
"C:/pgagent.sql" postgres
-- 4
pg_dumpall -i -h someserver -p 5432 -U someuser -c -o -f
"/pgbak/all dbs.sql"

-- 5
pg_dumpall -h localhost -p 5432 -U postgres --globals-only >
/pgbak/globals.sql

-- 6
pg_dump -h localhost -p 5432 -U postgres -F c -b -v -f
"/pgbak/work_poi.backup" -t "work.poi" somedb
```

- (1) Dump database in compressed include blobs show progress and show verbose progress (-v).
- (2) Dump database in latin1 encoding - useful if you want to restore a database but want to use a different encoding to store the data in the new database.
- (3) Backup pgagent schema or any schema of postgres db in plain text copy format, maintain oids.
- (4) Dump all databases - note pg\_dumpall can only output to plain text.
- (5) Backup users/roles and tablespaces.
- (6) Backup a single table in compressed format

### D.4.2 Restore

In order to restore a backup of PostgreSQL, you use pg\_restore to restore compressed and tar backups and you use psql to restore sql backups. If you have a compressed or tar

backup, you can use pg\_restore to restore select portions of a backup file if you desire. In this section, we'll demonstrate some common examples.

#### **Listing D.5 Common restore statements**

```
-- 1
psql -h localhost -p 5432 -U postgres -c "CREATE DATABASE somedb"
pg_restore -h localhost -p 5432 -U postgres --dbname=somedb --jobs=2
/pgbak/somedb.backup

-- 2
pg_restore --schema=us --dbname=somedb -U postgres /pgbak/somedb.backup

-- 3
pg_restore --list /pgbak/somedb.backup --file=/pgbak/somedb_list.txt

-- 4
psql -h localhost -p 5432 -U postgres -d postgres -f /pgbak/globals.sql

-- 5
pg_restore -h localhost -p 5432 -U postgres /pgbak/somedb.backup -t
"work.poi"
```

(1) Create a new database and restore backup file to this new database using 2 threads for restore. (remember jobs only works for PostgreSQL 8.4+) (2) Restore only a specific schema in this case us schema. (3) Generate a table of contents for a backup file and store in file somedb\_list.txt. (4) Restore user accounts and custom table spaces. You can specify any sql file here such as the one we created that backups up all databases. (5) Restore a single table from backup in this case we are restoring the table poi in the schema work.

In the next section we'll provide some tips for automating the backup process.

#### **D.4.3 Setting up automated jobs for backup**

There are two common ways for automating backups for PostgreSQL and it varies slightly depending on your OS

- Use an OS specific scheduling agent such as cronjob in Unix/Linux or Windows Scheduler in Windows
- Use PgAgent which is a scheduling agent for PostgreSQL that is a free download and is manageable from PgAdmin III.

We prefer the PgAgent way because its cross platform, allows us to manage like we manage and view other parts of PostgreSQL (via the PgAdminIII tool), and is also designed for easily running sql jobs. On the downside, it is sometimes more finicky to set up. We describe the details of setting this up and also how to define a backup script for windows and linux.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

<http://www.postgresonline.com/journal/index.php?/archives/19-Setting-up-PgAgent-and-Doing-Scheduled-Backups.html>

Note that since the backup scripts we describe just rely on shell commands of the respective Unix/Linux/Windows environment, you can also run them with your scheduling agent of choice.

## D.5 Data structures and objects

PostgreSQL like many sophisticated relational databases, has a rich collection of different kinds of objects to accomplish different tasks. In addition to database objects, it has built-in data types, many of which you will find in other relational database, as well as some data types that are fairly unique to it. If that were not enough, PostgreSQL allows you to extend the system to define new data types to suit your needs. The PostGIS family of data types is an extension of the core set. In this section we'll go over all this.

### D.5.1 PostgreSQL objects

When we talk of the term "objects", we are not talking about data types but rather a class of objects of which a data type is one class. Data types are used to define columns in a table, but objects are things that are part of the core makeup of PostgreSQL. They are tables, views, schemas etc. Some of these we have already been exposed to. In the table below we give a brief synopsis of the core PostgreSQL objects and their function.

- **Server service/Daemon** - PostgreSQL itself that houses everything.

**Tablespaces** - Physical locations of data that map to a named location in the server. Objects can be moved to different locations on disk if you run out of disk space on one by moving them to a different tablespace. Again these you can define the default for by setting the GUC variables default\_tablespace, temp\_tablespace and can even be set at the user level so that you can control disk space used by groups of users or use fast non-redundant disks for temporary tables and so forth. It is also fast and trivial to move even a single table to a different tablespace using PgAdmin or with the SQL command ALTER TABLE sometable SET TABLESPACE newtablespace

which we describe in

<http://www.postgresonline.com/journal/index.php?/archives/123-Managing-disk-space-using-table-spaces.html>

- **Database** - both a physical and a logical entity. A database has a root folder in the file system and database data in any tablespace is always stored in a folder <tablespace path>/<databaseoid>/objectoid
- **Schemas** - Logical location of tables, views, functions -- no relation to physical location, but SQL statements reference the logical name and you can control the default schemas at the server, database, or user level in versions of PostgreSQL 8.2+. In 8.3+ you can also control default at the function level via the search\_path

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

configuration. This allows you to maintain a logical separation without having to schema qualify commonly used schemas. Think of schemas as a database within the database. The first schema in the search\_path is the one where new objects are created by a user. If you have two objects with the same name in different schemas, and you reference them without qualifying the schema, then the first one in the search path is what is chosen.

- **Roles** - Users and groups -- sit at the database level and are granted rights to objects in a database.
- **Rules** - Rules rewrite SELECT, INSERT, UPDATE statements. They are unique to PostgreSQL and serve a similar purpose as triggers. In some cases such as the way PostgreSQL implements views, rules are the only option.
- **Views** - Views are virtual tables. They are a window into the real data and allow you to see summaries or a subset of data by selecting from an abstracted virtual table. A view generally only consists of an instead of SELECT Rule. An updateable view will also have Rules on the UPDATE, INSERT, DELETE action of a view.
- **Triggers** - triggers are actions that are triggered based on a change in data. They are often used to update additional data. Common examples in PostGIS are if you have an application that updates a long and lat field, you may have a trigger to update the geometry field when these values change. You may have another trigger to store a line in a separate table when points are added to one table.
- **Data Types** - Data types are the micro storage structure of data and their definition can be composed of other data types. Table columns are composed of a specific collection of things with the same data type. The rows of a table itself is implemented in PostgreSQL as a composite data type. You will notice in the types section of PostgreSQL, that every PostgreSQL table has a corresponding data type with the same name as the table.
- **Casts** - Casts are the objects that allow you to implicitly or explicitly convert from one data type to another. PostgreSQL is fairly unique among relational database in that it allows you to define casting behavior for your custom created data types. If an implicit cast is in place (a cast with no qualification), then when data of a specific data type is fed to a function expecting a different datatype, the data will be automatically cast for you if there is one unambiguous type it has an autocast for. Care should be taken when doing this. Because of the overloading features of PostgreSQL, it is possible to have two functions with the same name but take different data types and if an object that has an auto cast for both is used without an explicit casting, you get an "ambiguous" error. You do an explicit CAST by using the ANSI SQL compliant CAST(mybox As geometry) or the PostgreSQL non-ANSI SQL specific short-hand of mybox::geometry.
- **Operators** - these are things like =,> , < and again PostgreSQL allows you do define custom operator behavior for your custom types. There are some operators that have

special meaning such as = > < that are used by internal SQL querying to define how ORDER BY , GROUP BY DISTINCT and so forth are achieved. These are useful to override if you are building custom data types and want them to behave a certain way.

- **Functions** - These are functions you can use within an SQL statement. PostgreSQL comes with a lot of built-in ones, contrib ones such as the soundex we saw earlier and those provided by PostGIS. You can also build your own. These can return simple data types, sets, or arrays. There are 3 core classes of functions -- regular functions, aggregate functions, and trigger functions. We'll touch on each of these with examples in this appendix.
- **Sequences** - If you have worked with Oracle, then a sequence object will be very familiar to you. It is basically a counter that can be incremented and used to get the next id for a column. MySQL folks will recognize this as AUTO\_INCREMENT, except in PostgreSQL, a sequence object need not be tied to a single table. You can use it for multiple tables and increment it separately from a table. SQL Server people will recognize this as an IDENTITY field -- again unlike SQL Server which has the incrementation tied to a table, PostgreSQL does not, though the incrementing is just as transparent in most cases as it is in SQL Server and MySQL since the next increment is set as the default value for the column. If you want a sequence to be tied to a specific table in PostgreSQL, then you would create the column as serial or serial8.

### D.5.2 Built-in data types

PostgreSQL comes packaged with a lot of built-in data types. Some of these are pretty standard across all relational databases. These are:

- **int4, int8** - which go by more familiar names such as int, integer, bigint.
- **float, double precision** - another class of number types that don't necessarily exist in other databases but are common.
- **serial, serial8** - you can use this in the CREATE TABLE statement, but it is not a true type. It's a short hand way of saying give me an integer with a sequence object that increments it. It is still an integer. So the parallel in MySQL would be marking the column as an AUTO\_INCREMENT or in SQL Server setting the identity property to Yes.
- **numeric** -- has a scale and precision. This is named the same in other relational databases as well. Also often referred to as decimal in other databases.
- **varchar, text** - character varying. PostgreSQL unlike most other relational databases does not put a limit on the maximum length of a varchar or text. varchar and text behave much the same except text has no specified max limit and varchar may or many not have a specified max limit. Some other databases also decide on storage handling based on the specified size of a field -- so for example in SQL Server if text is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

noted, then pointer to text field is stored and data is stored elsewhere not with the table. PostgreSQL does not care about this and bases storage considerations on the size of data actually stored in the field and relegating to toast tables if a field goes beyond allotted storage size. As such many PostgreSQL people will argue there is no penalty of using text over varchar with a limit. If you care about interoperability we argue there is a big penalty. For exporting purposes such as tab-delimited and so forth -- its important to have a limit on size of field and if you export to another system often - you want your size limits to mirror the other side. For users who use auto generated screens with screen painters, such apps interrogate the system tables to determine the width to make a certain field on the screen, if everything is text, then you end up with big textboxes everywhere. If you use an ODBC driver, ODBC e.g. in MS Access treats a varchar very differently from a text. It will allow you to sort by varchar but not text.

- **char** - padded characters. If you say it's a char(8) then the field will always be of length 8. This exists in most other relational databases too. This is more a presentation feature than storage consideration in PostgreSQL.
- **date** - This is a date without time. MySQL has this, Oracle has this, SQL Server 2008+ has this (prior versions of SQL Server you could not have a date without time).
- **timestamp, timestamp with timezone** - Again very similar in other relational databases though may be called datetime or other name. SQL Server introduced timezone in SQL Server 2008, so prior to that you had no timezone information stored.
- **arrays** - arrays are not quite so common in other relational databases except perhaps for Oracle and IBM DB2. Arrays in PostgreSQL are typed. For example date[] would be an array of dates. Any custom type you build you can define a table column as an array of that type and use in functions as well. Arrays play an important role in building aggregate functions, since many of the tricks for building aggregate functions involve wrapping data in an array to be processed by a terminal function. Some quick ways of building arrays are ARRAY(SELECT somefield FROM sometable WHERE something\_is\_true), ARRAY[1,2,3,4] or in PostgreSQL 8.4+ the **array\_agg** ANSI SQL compliant aggregate function that will create an array for each row in a (GROUP BY ...). Note that IBM DB2 also has an array\_agg function as defined by the ANSI SQL 2003 specs.
- **row** - this is more of an abstract data type similar to array. A typed row is a row in a table or a specific type. You can cast compatible rows to compatible types as we shall demonstrate shortly.

### **D.5.3 Anatomy of a database function**

Relational database stored functions and procedures are useful for compartmentalizing reusable nuggets of functionality and embedding them in SQL statements. Unlike most ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

relational databases, PostgreSQL even as of PostgreSQL 8.4 does not make a distinction between a stored procedure and stored function. In other databases, stored procedures are things that can update data and generally return a cursor or nothing for their output. In PostgreSQL, there only exist functions and functions may return nothing (void) or something and can update data as well as return something at the same time.

PostgreSQL allows you to write stored functions in various languages. Its language offering is probably richer than any relational database system you will find both commercial and open source. Common favorites are sql, plpgsql, plperl, and for GIS users plpython and plR are additional favorites. There are more esoteric ones that are designed more for a specific domain such as pl/sh (which allows you to write stored functions that run bash/shell commands), pl/proxy (designed by Skype Corporation and freely provided that designed to replicate commands between PostgreSQL servers).

The only languages pre-installed in all PostgreSQL databases is sql and c. C is not really a procedural one but allows you to wrap a C function in a C library in a stored function wrapper so that it can be used in an SQL statement. Most PostGIS functions are C functions. PLPgSQL is not always installed, but is always packaged with PostgreSQL and is required to run PostGIS.

A PostgreSQL database function has a couple of core parts regardless of what language the function is written in.

- The function argument declaration
- The RETURNS declaration which dictates the return of the function, for functions that don't return anything -- its void
- The body -- which is the meat of the function. From PostgreSQL 8.1+ the general convention is to use what is referred to as \$ quoting syntax to encapsulate the body. Dollar quoting has form \$somename\$. Often times people leave out the somename so it reduces down to \$\$ body goes here \$\$ . This works for all languages. Prior versions required quoting with ' which required a lot of escaping of ' if you had that in the function. \$\$ quoting is a much more readable and painless way of writing functions.
- The language which is always LANGUAGE 'somelanguage'
- For PostgreSQL 8.3+ the ROWS expected and COST as a function of cpu cycles.
- The cachability of it -- designated as IMMUTABLE, STABLE, VOLATILE which allows PostgreSQL to know under what conditions the results can be cached. IMMUTABLE meaning with same inputs you can expect always the same output. STABLE within the same query you can expect the same inputs to result in same outputs, and VOLATILE means never cache because it either updates data or the results vary even given the same function inputs.
- The security context. If not specified, the function is assumed to be run using the security rights of the user. If you denote a function as SECURITY DEFINER, then that

means the function is allowed to do anything that the owner of the function can do. This allows you for example to create logic that can be executed by a non-superuser, that has logic that requires super user rights such as reading files from the file system.

### **POSTGRESQL 9.0 DO COMMAND**

In PostgreSQL 9.0+, the DO command was introduced. This allows you to write one-off anonymous functions that just contain a body and no name and can be run straight from command line without building a function for it.

#### **D.5.4 Defining custom data types**

Defining custom data types is fairly trivial in PostgreSQL. As mentioned earlier, when you create a new table, you are creating a new data type as well. Below is a simple example of a data type we will call vertex that contains an x and y attribute and that we then create instances of and then pull out just one attribute of.

#### **Listing D.6 Create a simple type and use it**

```
-- 1
CREATE TYPE vertex AS
(x double precision,
 y double precision);

-- 2
SELECT CAST(ROW(x,y*0.02) As vertex) As myvert
FROM generate_series(1,10) As x
CROSS JOIN generate_series(10,20,2) As y;

-- 3
SELECT ((myvert)).y
FROM (
SELECT CAST(ROW(x,y*0.02) As vertex) As myvert
FROM generate_series(1,10) As x
CROSS JOIN generate_series(10,20,2) As y
) As foo;
```

In (1) we define a new type called vertex that has an x and a y attribute. In (2) we create a query that returns two columns and then cast that to a vertex by first packaging each as an anonymous row. In (3) We pull out the y attribute of our fictitious table. This is similar to what we do often with ST\_Dump. You will recognize we often do a (ST\_Dump(the\_geom)).geom to grab just the geom attribute or a (ST\_Dump(the\_geom)).\* to explode all the attributes into separate columns.

#### **D.5.5 Creating tables and views**

Creating tables and views is done just like in any other relational database. Below are some simple examples that create a table and view in the assets schema.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

**Listing D.7 Creating a table and a view**

```
-- 1
CREATE TABLE assets.poi(poi_gid serial PRIMARY KEY,
                        the_geog geography(POINT,4326),
                        poi_name varchar(100),
                        is_active boolean DEFAULT true NOT NULL);

-- 2
CREATE VIEW assets.vwpoi_active AS
    SELECT poi_gid, the_geog, poi_name, is_active
        FROM assets.poi
       WHERE is_active = true;

-- 3
DROP TABLE assets.poi CASCADE;
```

In (1) We create a table with a geography field (required PostGIS 1.5+) that is of type POINT and WGS 84 long lat, it also has an auto increment primary key call poi\_gid and an active flag that is defaulted to true for new entries. In (2) we create a view against this new table that will only list active records. In (3) We drop the table and include the CASCADE command which will drop all dependent objects such as the view we created in (2). When using CASCADE proceed with caution since you could be dropping a lot of things. Without the CASCADE we would be informed that assets.vwpoi\_active depends on assets.poi and there can not be dropped. We would then have to drop the view first and then the table.

Now that we have covered the basic features of PostgreSQL, we shall get into more specifics about creating functions and rules.

## D.6 Writing functions in SQL

PostgreSQL is probably the only relational database system that allows you to write stored procedures in pure SQL. This is very different from what the PL/SQL supported by IBM DBII and MySQL in that PostgreSQL sql function language has no support for procedural control structures. What other databases call PL/SQL is closer in family to PostgreSQL's PLPgSQL.

It would seem on first glance that not allowing procedural control in a stored function language would be an undesirable thing, but the main benefit of such a language is that it can be treated like any other SQL statement and in many cases very useful pieces of reusable code can be compartmentalized in such a simple structure.

### D.6.1 When to use SQL functions

The most important attribute about SQL functions that makes them stand out from functions written in other procedural languages is that they are often inlined in the overall query. What does this mean? It means the query planner can see inside an SQL function and embeds its definition in the query and treats it like a macro similar to the way C macros are stuffed where they are used. This means that if your function uses an indexable expression, then the planner can use an index and if your sql function contains a sub expression already

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

in a query, then the planner can collapse the expression. A common example is the `&&` operator which is built into many PostGIS functions. If you use two functions that have this in the statement, the planner will see something of the form `&& AND &&` and collapse the two into a single `&&`.

So general rule of thumb as to when to use an SQL function:

- Uses constructs that could benefit from an index
- Logic is fairly simple and short
- In a rule -- you can only write rules with SQL. Rules aren't really functions per se, but they serve similar purpose.

There is one situation where you absolutely can't use SQL to write a function even if you wanted to and that is for a trigger function. This may change in later versions of PostgreSQL, but as of PostgreSQL 8.4, you can not write triggers with SQL language.

### **D.6.2 Creating an SQL function**

An SQL function like all other functions, contains an argument list, a return argument type and a function body. Unlike other languages, SQL functions can't have variables and they can at most have only one SQL statement.

#### **SQL AND VARIABLES**

While it is true you can't declare variables in an SQL function, for PostgreSQL 8.4+, you can significantly compensate for this by using CTEs to define sub work steps as we have demonstrated through out this book.

This makes them fairly limited but easy to fold into a larger SQL statement. The other disadvantage of them is that you can't use the argument inputs by their names, you have to reference them by \$1, \$2 ... In other PL languages such as PL/PgSQL, PL/Perl, PL/Python and PL/R you can reference by position or name. This restriction will most likely change in PostgreSQL 8.5.

Below is a very trivial function that returns a square of numbers starting with the first and ending with the last.

#### **Listing D.8 Example SQL function returns square and use**

```
-- 1
CREATE OR REPLACE FUNCTION fnsquare(param_start integer, param_end integer)
RETURNS SETOF integer
AS
$$
SELECT CAST(POWER(i,2) As integer)
FROM generate_series($1,$2) As i;
$$
language 'sql'
IMMUTABLE;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```
-- 2
SELECT i, fnsquare(i,i + 3) As squared_range
  FROM generate_series(1,3) As i;

-- 3
SELECT *
  FROM fnsquare(1,10) As foo;
```

In (1) we define our function that takes a range returns the square of each number in the range. (2) We use our function in the SELECT part of a query. This is only legal with SET returning functions in PostgreSQL prior to 8.4 if it is written in SQL or C. For PostgreSQL 8.4+, you can use this with PLPgSQL and other functions as well. (3) Standard way for calling set returning functions.

### D.6.3 Creating rules

Rules are objects that are bound to tables or views. They are often used in place of triggers and for views, you can only use rules -- not triggers. Rules don't actually perform any action, but help in rewriting SQL statements to do something in addition or instead of what the SQL statement would normally do.

The classic use of RULES is in defining views. When you create a view in PostgreSQL using standard ANSI syntax of the form:

```
CREATE OR REPLACE VIEW assets.vwpoi_active AS
  SELECT poi.poi_gid, poi.the_geog, poi.poi_name, poi.is_active
    FROM poi
   WHERE poi.is_active = true;
```

PostgreSQL is behind the scenes changing it to something that has a SELECT rule in it. If you were ever nosy enough to inspect your view -- you would see this curious thing attached to it.

```
CREATE OR REPLACE RULE "_RETURN" AS
  ON SELECT TO vwpoi_active
    DO INSTEAD
      SELECT poi.poi_gid, poi.the_geog, poi.poi_name, poi.is_active
        FROM poi
       WHERE poi.is_active = true;
```

So in short - the concept of a VIEW in PostgreSQL is really a packaging of a set of rules that has at least one DO INSTEAD SELECT rule, and if updateable accompanying DO INSTEAD UPDATE,INSERT, OR DELETE rules. Whenever someone calls for the virtual table vwmyview the SELECT rule will rewrite the SQL statement to use (SELECT a.gid, a.the\_geom FROM mytable As a) ... instead of the virtual table they were calling for.

How do we make a view updateable, we create an UPDATE rule that rewrites the update to update the raw tables.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

**Listing D.9 Making a view updateable**

```
-- 1 update rule
CREATE RULE updvwpoi_active AS
    ON UPDATE TO assets.vwpoi_active
        DO INSTEAD (
            UPDATE poi
                SET poi_name = NEW.poi_name ,
                    poi_gid = NEW.poi_gid,
                    the_geog = NEW.the_geog,
                    is_active = NEW.is_active,
                    poi_gid = NEW.poi_gid
                WHERE poi.poi_gid = OLD.poi_gid;
        );

-- 2 insert rule
CREATE RULE insvwpoi_active AS
    ON INSERT TO assets.vwpoi_active
        DO INSTEAD (
            INSERT INTO poi(poi_name, the_geog)
                VALUES(NEW.poi_name, NEW.the_geog)
        );
-- 3 delete rule

CREATE RULE delvwpoi_active AS
    ON DELETE TO assets.vwpoi_active
        DO INSTEAD (
            DELETE FROM poi WHERE poi.poi_gid = OLD.poi_gid
        );
```

In a rule or trigger there exist two records called NEW and OLD. NEW exists when there is an insert or update to an object. OLD exists when there is an UPDATE or DELETE to an object. In the above examples we make our view updateable by pushing updates to the base tables the view is based on.

**D.6.4 Creating aggregate functions**

Aggregate functions are functions you can use just like MAX, MIN and AVG. PostgreSQL allows you to create your own custom aggregate functions and even with a language as simple as SQL. This is simply one of the coolest features of PostgreSQL. Part of the power of doing this is because of the malleability of PostgreSQL array model. We have a couple of examples of creating aggregate functions in PostgreSQL using plain SQL language.

To demonstrate the ease with which you can create an aggregate function in PostgreSQL, here is an example that simulates (but we think better), MS Access "first" and "last" aggregate functions excerpted from one of our articles entitled "Who is on first and who is on last" <http://www.postgresonline.com/journal/index.php?archives/68-More-Aggregate-Fun-Whos-on-First-and-Whos-on-Last.html>

**Listing D.10 Creating First and Last Aggregate functions**

```
-- 1
CREATE OR REPLACE FUNCTION first_element_state(anyarray, anyelement)
RETURNS anyarray AS
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

$$
    SELECT CASE WHEN array_upper($1,1) IS NULL THEN array_append($1,$2)
ELSE $1 END;
$$
LANGUAGE 'sql' IMMUTABLE;

-- 2
CREATE OR REPLACE FUNCTION first_element(anyarray)
RETURNS anyelement AS
$$
    SELECT ($1)[1] ;
$$
LANGUAGE 'sql' IMMUTABLE;

-- 3
CREATE OR REPLACE FUNCTION last_element(anyelement, anyelement)
RETURNS anyelement AS
$$
    SELECT $2;
$$
LANGUAGE 'sql' IMMUTABLE;

-- 4
CREATE AGGREGATE first(anyelement) (
    SFUNC=first_element_state,
    STYPE=anyarray,
    FINALFUNC=first_element
)
;

-- 5
CREATE AGGREGATE last(anyelement) (
    SFUNC=last_element,
    STYPE=anyelement
);

```

As we can see in the above code, an aggregate function is composed of at the minimum a state function (SFUNC) (1,3) and a State Type (SType). The FINALFUNC (2) is sometimes present and needed if the result of each subsequent state is not enough or the datatype of the final is different from the data type of the state. In 4 and 5 we define our first and last aggregate functions with these elements and now we take it for a test drive.

#### List D.11 Putting our first and last to work

```

SELECT max(age) As oldest_age, min(age) As youngest_age, count(*) As
numinfamily, family,
    first(name) As firstperson, last(name) as lastperson
FROM (SELECT 2 As age, 'jimmy' As name, 'jones' As family
      UNION ALL SELECT 50 As age, 'c' As name , 'jones' As family
      UNION ALL SELECT 3 As age, 'aby' As name, 'jones' As family
      UNION ALL SELECT 35 As age, 'Bartholemu' As name, 'Smith' As family
      ) As foo
GROUP BY family;

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

We put our functions to use with a simple stand alone query. This example and creation of aggregates works in most versions of PostgreSQL even back to 8.1.

## D.7 Writing functions in PL/PgSQL

The PostgreSQL PL/PgSQL procedural language is probably closest in form to Oracle's PL/SQL. It, like Oracle PL/SQL and the other relational database procedural languages, is a language that allows you to declare variables, other control flow such as FOR and WHILE loops, cursors, RAISE errors etc. and also write SQL. Unlike the pure SQL language, it is not transparent to the planner, and is treated like a black box. Inputs go in and outputs come out. It like the SQL language and other PL languages allows you to dictate attributes such as volatility, cost and security so that the planner can decide if a choice of order is allowed how costly the function is to evaluate relative to other functions and what kind of rights are allowed within the function.

### D.7.1 When to use PL/PgSQL functions

PL/PgSQL is desirable for writing functions where there is no benefit to using an outer index such as in cases where the values that go into the function are already filtered by a where condition and where very fine grained step by step control is needed

So general rule of thumb as to when to use an PL/PgSQL function:

- No construct that could benefit from an outer index check.
- Logic is complex needing several breaks, storage of variables, ability to raise errors.
- In a trigger. You can't use SQL in a trigger. Though you can use other languages such as PL/Python or PL/Perl for writing triggers, PL/PgSQL tends to be more stable and also has more integration with PostgreSQL so generally a better language for writing triggers unless you need to leverage specific functionality only offered via a Python or Perl library or construct.

As mentioned earlier, you can't write rules with PL/PgSQL and in earlier versions of PostgreSQL (pre 8.4), you can not use a set returning PL/PgSQL function in the SELECT clause of a statement where as you can with an SQL function.

### D.7.2 Creating a PL/PgSQL function

Below is a simple PL/PgSQL function. This is the `utmzone` function we've used often in the book and a good example of when to use a PLPGSQL function. Lets study its parts.

#### **Listing D.12 utmzone**

```
-- 1
CREATE OR REPLACE FUNCTION utmzone(geometry)
RETURNS integer AS
$$
-- 2
DECLARE
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

geomgeog geometry;
zone int;
pref int;
-- 3
BEGIN
    geomgeog:= ST_Transform($1,4326);

    IF (ST_Y(geomgeog))>0 THEN
        pref:=32600;
    ELSE
        pref:=32700;
    END IF;

    zone:=floor((ST_X(geomgeog)+180)/6)+1;

    RETURN zone+pref;
END;
$$ LANGUAGE 'plpgsql' IMMUTABLE
COST 100;

```

(1) The first part of a plpgsql function like any function is the envelop which defines the parameters that go into the function and the return type. (2) Then there is the DECLARE which is part of the body and is where we declare the variables we will use through the rest of the function. SQL functions do not have this though other languages may but specify it differently. (3) Then comes the meat of the function which is encapsulated between a BEGIN AND END and generally ends with a RETURN that returns the output.

We haven't gone through any of the control flow logic but the BEGIN and END for more complex things often has FOR loops and return when returning a set may have RETURN NEXT to return each element in the set. For 8.3+ there is also RETURN QUERY which allowed returning results of precompiled SQL and 8.4 introduced RETURN QUERY EXECUTE which allowed returning results of dynamic SQL. Example of the 8.4 construct is demonstrated in Pavel Stehule's blog -- <http://okbob.blogspot.com/2008/06/execute-using-feature-in-postgresql-84.html>

### D.7.3 Creating triggers

Triggers like rules have an available record called NEW or OLD or both and also have a variable called "TG\_OP" which holds the kind of operation that triggered the trigger. There are other TG\_ variables provided. If you are reusing the same trigger across multiple tables TG\_TABLE\_NAME AND TG\_TABLE\_SCHEMA are useful ones as well. The NEW and OLD objects have the same column structure as the table the trigger is being applied to. Trigger functions can be shared across tables. Triggers in PostgreSQL 8.4 and below can not be written using SQL, they must be written in plpgsql or some other language. Not all languages support triggers, but PL/Python, PL/Perl, and PL/R to name a few do. How the NEW and OLD data is referenced varies from language to language. Below are listings of kinds of triggers and what data is available to each. A trigger is either a ROW level or

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

statement level trigger and is triggered on the UPDATE/INSERT/DELETE event or combination of those events.

- Statement trigger -- gets run for each kind of SQL statement on a table. No data is available to it, so the best you can do is log that a statement has been run and what kind of statement. It is not that often used.
- INSERT row-level trigger -- gets run on insert of data and once for each row. The NEW object is available to it and contains the new data. An INSERT trigger can be marked as BEFORE INSERT or AFTER INSERT. In a BEFORE INSERT you can change the values in the NEW object and these will get propagated to the actual insert. In an AFTER INSERT -- PostgreSQL lets you still set values in the NEW, but this data gets thrown away and doesn't get propagated to actually affecting the insert. This is a common mistake people make that they try to set values of NEW.. in the AFTER INSERT trigger.
- UPDATE row-level trigger -- One can think of an update as a delete followed by an insert. Therefore the UPDATE trigger has both an OLD and NEW variable available to it. The OLD contains data that is deleted or to be deleted and the NEW has data to be added or is added. Again an UPDATE trigger can be marked as BEFORE or AFTER and for BEFORE, changes to the NEW record will get propagated to the table and AFTER your NEW changes go into a black hole when the trigger is completed.
- DELETE -- just the OLD object is available.

Below is an example trigger that will update a geography column whenever a longitude and latitude are updated or a new record is changed and will also log changes to a log table. Keep in mind you can do other useful things such as geocode records on the update of address information. In PostgreSQL, triggers are a kind of function and the function is separate from the actual trigger that the trigger function is bound to. The benefit of this approach is that a trigger function can be shared across many tables that have same named fields. The downside is you can't just write the trigger function as part of the table definition as you can in some other databases.

#### **Listing D.13 Trigger function applied to geography table in plpgsql**

```
-- 1
CREATE TABLE poi(gid serial PRIMARY KEY,
                 the_geog geography(POINT,4326),
                 poi_name varchar(100),
                 longitude float, latitude float);

CREATE TABLE poi_log(logid SERIAL PRIMARY KEY,
                    logdt timestamp with time zone DEFAULT CURRENT_TIMESTAMP,
                    logtype varchar(20), geogtable varchar(100), geog_gid integer,
                    old_geog geography, new_geog geography);
```

-- 2

```
CREATE OR REPLACE FUNCTION trig_set_thegeog_pt() RETURNS trigger AS
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

```

$$
DECLARE
    changed boolean := false;
    oldgeog geography := NULL;
BEGIN
-- 3
    IF tg_op = 'INSERT' AND NEW.longitude IS NOT NULL AND NEW.latitude IS
NOT NULL THEN
        changed = true;
-- 4
    ELSIF COALESCE(NEW.longitude, -1000) != COALESCE(old.longitude, -1000)
        OR COALESCE(NEW.latitude, -1000) != COALESCE(old.latitude, -1000) THEN
        changed = true;
    END IF;

    IF changed THEN
        IF NEW.longitude IS NOT NULL AND NEW.latitude IS NOT NULL THEN
            NEW.the_geog := ST_GeographyFromText('SRID=4326;POINT(' || |
NEW.longitude || ' ' || NEW.latitude || ')');
        ELSE
            NEW.the_geog = NULL;
        END IF;
-- 5
    INSERT INTO poi_log(logtype, geogtable, geog_gid, old_geog, new_geog)
        VALUES(TG_OP, TG_TABLE_NAME, NEW.gid, oldgeog, NEW.the_geog);
    END IF;
    RETURN NEW;
END;
$$
LANGUAGE 'plpgsql' VOLATILE;

-- 6
CREATE TRIGGER step01_trigupdpt
    BEFORE INSERT OR UPDATE
    ON poi
    FOR EACH ROW
    EXECUTE PROCEDURE trig_set_thegeog_pt();

-- 7
INSERT INTO poi(poi_name, longitude, latitude)
    VALUES('My back yard', -72.1234, 41.3456);

SELECT gid, ST_AsText(the_geog) As wktgeog
FROM poi;

UPDATE poi SET longitude = -72.555 WHERE gid = 1;

SELECT gid, ST_AsText(the_geog) As wktgeog
FROM poi;
SELECT * FROM poi_log;

```

- (1) We create a test table that we will later apply our trigger to. (2) We create a trigger function and the first part declares a state variable we initialize to false because we don't want to make any unnecessary updates. (3) checks to see what kind of event caused the  
 ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

trigger to fire, if its an insert we know we need to update if the longitude and latitude values are not NULL (4) if its an update, then we need to either update the geography column or wipe out the contents. We use the COALESCE here to set NULLS to a random -1000 so that we are never comparing NULLS. NULLS are tricky things to compare since when compared even to itself returns false so to shorten our code, we write the COALESCE hack. (5) We then log the change to our log table. In general logging should be done as an AFTER TRIGGER event so that the logging sees the final record data. In this case since we have only one trigger its simpler to just combine into one. (6) We bind our trigger function to the INSERT and UPDATE events of the table. Note that a table can have multiple triggers and they run in alphabetical order based on the triggering event. Sometimes especially if you have complex triggers shared by many tables, its advantageous to categorize your triggers by functionality rather than trying to write a big body of logic. In those cases you just have to keep track of the order in which the triggers are fired by naming them accordingly say step01... step02... etc. Each subsequent BEFORE trigger will see the change of the previous if the previous makes sure to return NEW, otherwise trigger execution stops. (7) Then we test our trigger to make sure its working. Our select should show something like POINT(-72.555 41.3456) and we should see two records in our log table which includes the name of the table from the TG\_TABLE\_NAME variable.

The above example just scratches the surface of what you can do with triggers in PostgreSQL in general and plpgsql in particular. Triggers are also capable of triggering other triggers so that you can have recursive triggers. Recursive triggers are particularly useful for maintaining the positioning of a parent object so that when you move a parent object, its child component parts move accordingly. Hopefully we have demonstrated enough here that you can envision the potential it holds.

## D.8 Performance

In chapter 9 we talked a bit about performance. In this section, we'll cover some of the loose ends we left out of that chapter.

### D.8.1 Index

Just like in other databases, PostgreSQL uses indexes to improve performance. There are various flavors of indexes you can choose from as well as various additional options you can specify for an index that you may or may not find in other relational databases. The key ones are

- Partial Indexes -- these are indexes with a where clause where the WHERE constrains the data that is actually indexed.
- Functional -- you can choose to index a function calculation such as UPPER, LOWER, SOUNDEX, ST\_Transform etc. where the arguments to the function can be any of the columns in a row. You can't go across rows, but your function can take

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

multiple columns. The function must also be marked immutable, which means given a set of inputs, the function is guaranteed to return the same output.

- Kinds of indexes -- btree, gist, gin, hash. The most common is the btree index. The kind of index controls how the index leaves are structured.

### **BTREE INDEX GOTCHAS**

A btree index is the most commonly used of all index types in PostgreSQL. Under certain conditions where you would expect it to be used, it is not. These conditions are probably very unintuitive to people coming from databases that are not necessarily case sensitive.

- Trying to do a functional compare e.g. UPPER/LOWER. Some people think that when you do a query `upper(item_name) = 'DETROIT'` then that condition should be able to use a btree index of the form

```
CREATE INDEX idx_item_name ON items USING btree(item_name);
```

Well it can't because your compare `upper(item_name)` does not match what is indexed. You need to change the index to be:

```
CREATE INDEX idx_item_name ON items USING btree(upper(item_name));
```

- The varchar\_pattern\_ops gotcha. If you want to do something like this:

```
WHERE upper(item_name) LIKT 'D%'
```

Then you can't use an unqualified btree. You need one with varchar\_pattern\_ops. Prior to PostgreSQL 8.4 a varchar\_pattern\_ops will not be able to service an equality like above. So to handle both, you need two indexes in prior versions. Read Tom's note in this thread of our article to get the gory details.  
<http://www.postgresonline.com/journal/index.php?archives/78-Why-is-my-index-not-being-used.html#c503>

### **FUNCTIONAL INDEX GOTCHAS**

Functional indexes are indexes that are against a function rather than raw data. Common functional indexes are things like:

```
CREATE INDEX idx_item_name ON items USING btree(upper(item_name));
```

The main gotcha with functional indexes is that you can only index functions marked as IMMUTABLE. This means the function outputs do not change given the same arguments. The main reason for that is that once an index is calculated, the index value is only changed if the input fields to the functions change.

You can of course lie about a function being immutable by marking it as immutable to get around this restriction and Postgres even as of 8.4, will not try to validate whether it demands on dynamic things such as tables. We demonstrated that with doing an index on ST\_Transform. If you do such a thing, you need to be careful to ensure that at least in most cases, such things are for your use cases, immutable.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=565>

The other gotcha with functional indexes is that if you redefine a function to do something other than what it was doing before, that changes the output value, PostgreSQL will not go back and reindex the affected tables. As such - if you change a function used in an index and you know that change will affect output, you need to go back and reindex those affected tables.

Its fairly easy to determine which tables are affected, particularly in PgAdmin using the dependents tab of a function.

### **D.9 Summary**

In this appendix, we have given you a small taste of the uniqueness and versatility of PostgreSQL that makes it stand apart from some other databases you may be familiar with. We have also demonstrated its also not so nice and cumbersome features such as the RIGHTS management. This is by no means the extent of what is offered by PostgreSQL and we encourage you to reference its very detailed extensive volumes of manuals available when you need to learn more about a specific feature.