PostGIS IN ACTION



Regina O. Obe
Leo S. Hsu

Foreword by Paul Ramsey

MANNING

PostGIS in Action

PostGIS in Action

REGINA O. OBE
LEO S. HSU

MANNING
Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
180 Broad Street
Suite 1323
Stamford, CT 06901
Email: orders@manning.com

©2011 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without elemental chlorine.



Manning Publications Co.
180 Broad Street
Suite 1323
Stamford, CT 06901

Development editor: Sebastian Sterling
Copyeditor: Linda Recktenwald
Typesetter: Marija Tudor
Cover designer: Marija Tudor

ISBN: 9781935182269
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 17 16 15 14 13 12 11

brief contents

PART 1 LEARNING POSTGIS 1

1 ■ What is a spatial database? 3
2 ■ Geometry types 33
3 ■ Organizing spatial data 53
4 ■ Geometry functions 80
5 ■ Relationships between geometries 117
6 ■ Spatial reference system considerations 153
7 ■ Working with real data 173

PART 2 PUTTING POSTGIS TO WORK 201

8 ■ Techniques to solve spatial problems 203
9 ■ Performance tuning 241

PART 3 USING POSTGIS WITH OTHER TOOLS 277

10 ■ Enhancing SQL with add-ons 279
11 ■ Using PostGIS in web applications 312
12 ■ Using PostGIS in a desktop environment 345
13 ■ PostGIS raster 371

contents

foreword xv
preface xvii
acknowledgments xix
about this book xxi
about the cover illustration xxviii

PART 1 LEARNING POSTGIS 1

1 ***What is a spatial database? 3***

1.1 Thinking spatially 3

Introducing the geometry data type 6

1.2 Modeling 7

Imagine the possibilities 8

1.3 Introducing PostgreSQL and PostGIS 9

PostgreSQL strengths 9 ▪ PostGIS, adding GIS to PostgreSQL 13 ▪ Alternatives to PostgreSQL and PostGIS 14 ▪ What works with PostGIS 15

1.4 Getting started with PostGIS 16

Verifying version of PostGIS and PostgreSQL 17 ▪ Creating geometries with PostGIS 17

1.5 Working with real data 20

Loading comma-separated data 21 ▪ Spatializing flat file data 22 ▪ Loading data from spatial data sources 25

1.6 Using spatial queries to analyze data 28

Proximity queries 29 ▪ Viewing spatial data with OpenJUMP 30

1.7 Summary 31

2 *Geometry types 33*

2.1 Geometry columns in PostGIS 34

The geometry_columns table 34 ▪ Interacting with the geometry_columns table 37

2.2 A panoply of geometries 38

What's a geometry? 38 ▪ Points 39 ▪ Linestrings 40 Polygons 41 ▪ Collection geometries 43 ▪ Curved geometries 47 ▪ 3D geometries 51

2.3 Summary 52

3 *Organizing spatial data 53*

3.1 Spatial storage approaches 53

Heterogeneous geometry columns 54 ▪ Homogeneous geometry columns 56 ▪ Table inheritance 57

3.2 Modeling a real city 60

Modeling using a heterogeneous geometry column 61 Modeling using homogeneous geometry columns 64 Modeling using inheritance 66

3.3 Using rules and triggers 69

Rules versus triggers 69 ▪ Using rules 71 Using triggers 73

3.4 Summary 78

4 *Geometry functions 80*

4.1 Constructors 81

Creating geometries from well-known text and well-known binary representations 81 ▪ Autocasting in PostgreSQL/PostGIS 83

4.2 Outputs 84

Well-known text and well-known binary 85 ▪ Keyhole Markup Language 85 ▪ Geography Markup Language 86

Geometry JavaScript Object Notation 86 ▪ Scalable Vector Graphics 86 ▪ Geohash 87 ▪ Examples of output functions 87

4.3 Accessor functions: getters and setters 88

Getting and setting spatial reference system 88 ▪ Transform to a different spatial reference 89 ▪ Geometry type 90 Coordinate and geometry dimensions 91 ▪ Geometry validity 92 ▪ Number of points that define a geometry 93

4.4 Measurement functions 94

Planar measures for geometry types 95 ▪ Geodetic measurement for geometry types 96 ▪ Measurement with geography type 98

4.5 Decomposition 99

Boxes and envelopes 99 ▪ Coordinates 101 Boundaries 102 ▪ Point marker for a geometry: centroid, point on surface, and nth point 103 ▪ Breaking down multi and collection geometries 105

4.6 Composition 108

Making points 108 ▪ Making polygons 110 ▪ Promoting single to multi geometries 112

4.7 Simplification 112

Coordinate rounding using ST_SnapToGrid 113 Simplifying geometries 114

4.8 Summary 115

5 *Relationships between geometries 117*

5.1 Introducing spatial relationship functions 118

5.2 Intersections 119

Segmenting linestrings with polygons 120 ▪ Clipping polygons with polygons 121

5.3 Specific intersection relationships 123

Interior, exterior, and boundary of a geometry 123 ▪ Contains and Within 125 ▪ Covers and CoveredBy 127 ContainsProperly 128 ▪ Overlapping geometries 129 Touching geometries 129 ▪ Crossing geometries 130 Disjoint geometries 131

5.4 The remainder: ST_Difference and ST_SymDifference 131

5.5 Nearest neighbor 134

Intersects with tolerance 135 ▪ Finding N closest objects 135 Using SQL Window functions to number results 137

5.6 Bounding box and geometry comparators 139

The bounding box 139 ▪ Bounding box and geometry operators 140

5.7 The many faces of equality 141

Spatial equality 142 ▪ Geometric equality 142 ▪ Bounding box equality 144

5.8 Underpinnings of relationship functions 147

The intersection matrix 147 ▪ Equality and the intersection matrix 148 ▪ Using the intersection matrix with ST_Relate 149

5.9 Summary 152

6 *Spatial reference system considerations 153*

6.1 Spatial reference system: What is it? 154

The geoid 154 ▪ Ellipsoids 156 ▪ Datum 158 Coordinate reference system 158 ▪ Projection 158 Different kinds of projections 159

6.2 Selecting a spatial reference system to store data 162

Pros and cons of using EPSG:4326 162 ▪ Geography data type for EPSG:4326 163 ▪ Mapping just for presentation 164 ▪ Covering the globe when distance is a concern 166

6.3 Determining the spatial reference system of source data 168

Guessing at a spatial reference system 169 ▪ When the spatial reference system is missing 172

6.4 Summary 172

7 *Working with real data 173*

7.1 Tools for importing/exporting data 174

PostgreSQL built-in tools 174 ▪ PostGIS packaged tools 174 OGR2OGR: all-purpose vector data loader 175 ▪ Quantum GIS Shapefile to PostGIS Import Tool 177 ▪ osm2pgsql: OpenStreetMap to PostGIS loader 179

7.2 Loading data 179

Getting and extracting compressed files 180 ▪ Using PostGIS and PostgreSQL tools to load data 182 ▪ Loading data with OGR2OGR 187 ▪ Importing OpenStreetMap data with osm2pgsql 193

7.3 Exporting data from PostGIS 195

Using pgsql2shp to dispense PostGIS data 196 ▪ Using OGR2OGR to dispense PostGIS data 197

7.4 Summary 199

PART 2 PUTTING POSTGIS TO WORK. 201

8 *Techniques to solve spatial problems 203*

8.1 Proximity analysis 204

Check for intersections and measuring distances 204 Convert to different units of measurement 207 ▪ Measure large distances 209 ▪ Choose spatial reference systems when measuring area 212

8.2 Data tagging 215

Techniques for generating dummy data 215 ▪ Tag data to a specific region 216 ▪ Snapping points to closest linestring 217 ▪ Geocoding an address to a point on a street 219

8.3 Slicing and splicing linestrings 221

Create linestrings from points 221 ▪ Break linestrings into smaller segments 223

8.4 Slicing and splicing polygons 227

Create a single multipolygon from many multipolygon records 227 ▪ Tessellate areas 228 ▪ Create equal-area slices 231

8.5 Translating, scaling, and rotating geometries 235

Move a geometry along X, Y, Z 236 ▪ Increase and decrease size of geometry 238 ▪ Rotate a geometry 239

8.6 Summary 240

9 *Performance tuning 241*

9.1 The query planner 242

Planner statistics 243

9.2 Using explain to diagnose problems 245

Text explain versus pgAdmin III graphical explain 246 The plan without an index 247

9.3 Indexes and keys 250

The plan with a spatial index scan 250 ▪ Options for defining indexes 253

9.4 Common SQL patterns and how they affect performance 257

SELECT subselects 258 ▪ FROM subselects and basic common table expressions 263 ▪ Window functions and self-joins 264

9.5 System and function settings 265

Key system variables that affect plan strategy 266
Function-specific settings 268

9.6 Optimizing geometries 269

Fixing invalid geometries 269 ▪ Reducing number of vertices with simplification 269 ▪ Removing holes 272
Clustering 273

9.7 Summary 275

PART 3 USING POSTGIS WITH OTHER TOOLS..... 277

10 Enhancing SQL with add-ons 279

10.1 Georeferencing with the TIGER geocoder 280

Installing the TIGER geocoder 281 ▪ Loading TIGER data 281 ▪ Geocoding and address normalization 283
Summary 286

10.2 Solving network routing problems with pgRouting 286

Installation 286 ▪ Shortest route 286 ▪ Traveling salesperson problem 288 ▪ Summary 289

10.3 Extending PostgreSQL power with PLs 290

Basic installation of PLs 290 ▪ What can you do with a non-native PL 290

10.4 Graphing and accessing spatial analysis libraries with PL/R 292

Getting started with PL/R 292 ▪ Saving datasets and plotting 293 ▪ Using R packages in PL/R 296 ▪ Quick primer on rgdal 298 ▪ Getting PostGIS geometries into R spatial objects 301 ▪ Outputting plots as binaries 304

10.5 PL/Python 304

Installing PL/Python 304 ▪ Our first PL/Python function 306 ▪ Using Python packages 306 ▪ Geocoding with PL/Python 309

10.6 Summary 311

11 *Using PostGIS in web applications 312*

11.1 GIS and the web 313

Limitations of conventional web technologies 313 ▪ Mapping servers 314 ▪ Mapping clients 317 ▪ Proprietary services 318

11.2 Using MapServer 319

Installing MapServer 319 ▪ Creating WMS and WFS services 320 ▪ Calling a mapping service using a reverse proxy 322

11.3 Using GeoServer 324

Installing GeoServer 324 ▪ Setting up PostGIS workspaces 325 ▪ Accessing PostGIS Layers via GeoServer WMS/WFS 326

11.4 Basics of OpenLayers and GeoExt 327

Using OpenLayers 328 ▪ Enhancing OpenLayers with GeoExt 333

11.5 Displaying data with server-side web scripting 337

Using PostGIS output functions with PHP 337 ▪ Displaying data in Google Earth 340 ▪ Loading custom layers with GeoExt 341 ▪ Proximity queries with PostGIS geography 342

11.6 Summary 343

12 *Using PostGIS in a desktop environment 345*

12.1 At a glance 346

Capsule review 346 ▪ Spatial database support 347 Format support 349 ▪ Web services supported 350

12.2 OpenJUMP Workbench 351

Feature summary 352 ▪ Register data source 353 Rendering PostGIS geometry data 355 ▪ Exporting data 357 Summary 357

12.3 Quantum GIS 357

Feature summary 357 ▪ Adding a PostGIS connection 359 Viewing and filtering PostGIS data 360 ▪ Connecting with other spatial databases 361 ▪ Loading other vector and raster layers 361 ▪ Exporting data 362 ▪ Summary 362

12.4 uDig 362

Feature summary 363 ▪ Connecting to PostGIS and other spatial databases 364 ▪ Viewing and filtering PostGIS data 365 ▪ Exporting data 365 ▪ Summary 366

12.5 gvSIG 366

Feature summary 366 ▪ Adding a PostGIS layer to a view 368 ▪ Exporting data 369 ▪ Connecting to other spatial databases 370

12.6 Summary 370

13 *PostGIS raster 371*

13.1 What is PostGIS raster? 372

What is raster data and how is it different from vector data? 373 ▪ Why analyze raster data? 376 ▪ Getting started with raster support in PostGIS 376

13.2 Storing and loading raster data 377

Options for storage 377 ▪ Using a loader to load data 378

13.3 Raster maintenance tables and functions 383

raster_columns metadata table 384 ▪ AddRasterColumn function 385 ▪ Other management functions 385

13.4 Commonly used functions 385

Common accessors 385 ▪ Georeferencing functions 389

13.5 Combining raster processing with vector processing 392

Pixel value getters and setters 392 ▪ Intersects and Intersections 392 ▪ Adding bands 395 ▪ Adding additional attributes to raster records 397

13.6 Exporting raster data into other raster formats 398

Gdal_translate basics to convert to other formats 399
Using gdalwarp to transform from one spatial ref to another 400

13.7 Viewing raster data with MapServer 401

13.8 The future of PostGIS raster support 402

Input/output functionality 402 ▪ Open source viewing tools 403 ▪ Database raster functions 403

13.9 Summary 404

appendix A Additional resources 405
appendix B Installing, compiling, and upgrading 419
appendix C SQL primer 430
appendix D PostgreSQL features 451
index 483

foreword

As children, we were 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 smartphones 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 as 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 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 geographic 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 has 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. 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 1,000 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, you can 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's truly one of a kind.

PAUL RAMSEY
CHAIR, POSTGIS
PROJECT STEERING COMMITTEE

preface

PostGIS (pronounced *post-jis*) is a spatial database extender for the PostgreSQL open source relational database management system. It's 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, what 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'll find in these other OGC-compliant databases as well as numerous additional ones that are unique to PostGIS.

Readers coming from other ANSI/ISO–compliant spatial databases or other relational databases such as those we've mentioned, will feel right at home with PostgreSQL/PostGIS. PostgreSQL is the most ANSI/ISO SQL–compliant database management system around, and it supports most of the ANSI-SQL92/2003 standards and some of the 2006/2008 standards. In a similar vein, PostGIS supports many of the industry-standard OGC/ISO SQL/MM 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—to serve 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've come across and various strategies for solving them with PostGIS.

Above and beyond our primary mission, we hope to lay the foundation for thinking spatially. We hope that readers 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.

acknowledgments

We thank each other for making this book possible. If only one of us was writing this book, it would have been either a random stream of consciousness or an obsessively organized masterpiece that would never have been finished in our lifetime.

We thank our technical reviewer, Dr. Jan Hartmann, from the University of Amsterdam Department of Geography, who went above and beyond the call of duty in reviewing all chapters of our book, testing the code, and providing invaluable constructive criticism. We'd also like to thank Paul Ramsey for contributing the foreword and our illustrators Gary Battiston and Alejandro Gomez.

We thank everyone at Manning Publications. In particular, we acknowledge Marjan Bace and Karen Tegtmeyer for reviewing our proposal, organizing reviewer feedback, and giving us the opportunity to be published authors; our development editor, Sebastian Stirling, who endured many revisions of our chapters; and our production team of Linda Recktenwald, Mary Piergies, Barbara Mirecki, and others for keeping us focused during the production process.

Our exposure to PostGIS would not be possible without the City of Boston Department of Neighborhood Development (DND), particularly the MIS and Policy Development and Research divisions where Regina first got exposed to GIS and PostGIS. A special thanks to fellow members of the PostGIS development team and Steering Committee: Kevin Neufeld, Mark Cave-Ayland, Paul Ramsey, Sandro Santilli, Nicklas Avén, Olivier Courtin, Mark Leslie, Mateusz Loskot, Pierre Racine, Jorge Arévalo, and others; each ensures that every new release of PostGIS has great features and that bug reports get immediate attention. We also thank the PostGIS community of newsgroup

subscribers who answer questions as best and as quickly as they can, PostGIS bloggers, and package maintainers; each in their own way gives newcomers to PostGIS a warm and fuzzy feeling.

Finally, we thank our early access readers and reviewers who flagged errors and ambiguities in our text and code before publication, in particular, Brent Wood, Stephen Woodbridge, Dylan Beaudette, Rick Wagner, Sandro Santilli, Kevin Neufeld, James Fee, Paul Ramsey, Bruce Rindahl, Amos Bannister, Paolo Corti, Richard Greenwood, Bill Dollins, Pierre Racine, Mark Leslie, Mark Cave-Ayland, Andy Saurin, Dane Springmeyer, Katie Filbert, and Jeff Addison.

about this book

This book isn't 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 won't tell you how to combine all these functions into a recipe to solve your problems. That is the purpose of our book. Although it doesn't cover all functions available in PostGIS, this book does cover the more commonly used or interesting ones and gives you the skills you need to combine them to solve classic and more esoteric but interesting problems in spatial analysis and modeling.

While 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 focuses on two-dimensional non-curved Cartesian vector geometries. Although it is primarily about writing spatial queries against 2D vector geometries, we provide introductions to the following ancillary topics:

- Creating 3D vector geometries
- Creating curved geometries
- Creating and querying the geodetic geography data type
- Working with raster data using the companion raster data type (integrated in PostGIS 2.0)

While the main purpose of this book is the use of PostGIS, we'd 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 the following:

- An extensive appendix that covers PostgreSQL in great detail from setup, to backup, to 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?

This book provides an introduction to PostGIS and it assumes a basic comfort level with programming and working with data. The types of people we've found most attracted to PostGIS and best suited for reading this book are listed here.

GIS PRACTITIONERS AND PROGRAMMERS

You know everything about data, 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're adept at generating data sources in ESRI shapefiles, using 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're stuck. Being able to draw all the Wal-Marts in the United States on a map is one thing, but being able to answer the question of how many Wal-Marts 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?

- It provides the ability to easily intermingle spatial data with other corporate data such as financial information, observation data, and marketing information. Yes, you can do these with ESRI shapefiles, 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 speed 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 user data, whether that user is drawing a geometry on the screen and inputting related information or clicking a point on the map, there's so much infrastructure built around databases that the task is much easier if you're using one. Take, for example, rolling your own web application whether in .NET, PHP, Perl, Python, Java, or some other language. 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 functions, and other processing functions that PostGIS provides, along with the geometry-

generation and -manipulation functions that platforms like OpenLayers, MapServer, 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 a text attribute or a geometry.
- It offers triggers that can allow generation of other things like related geometries in other tables when certain database events happen.
- PostgreSQL has a multi-version concurrency control (MVCC) transactional system to ensure that when 100 users are reading or updating your data at the same time, your system doesn't come screeching to a halt.
- It provides the 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're married to your preferred GIS desktop tools, don't worry. Choosing a spatial DBMS such as PostGIS doesn't mean you need to abandon your tools of choice. Manifold, Cadcorp, MapInfo 10+, AutoCAD, and various commonly used desktop tools have built-in support for PostGIS. ArcGIS does as well via the SDE offering or via Obtuse Software's zigGIS plug-in. Safe FME, a popular extract, transform, load (ETL) 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're forced to limit your thinking in terms of coordinates, location names, or other geographical attributes that can be reduced to numbers and letters. This works fine for point data, but you're at a complete loss once areas and regions come into play. You may be able to find all the people 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'd be 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, documents, 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.

SCIENTISTS, RESEARCHERS, EDUCATORS, AND ENGINEERS

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. Although we don't consider ourselves the same as them, 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 they aren't trained in database management, relational database use, or GIS. If you're 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?

- It gives you the ability to integrate with statistical packages such as R, and you can 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 in a plain Python environment.
- While many think of PostGIS as a tool for geographic information systems, and that's implied by the name, we see it as a tool for spatial analysis. The distinction is that while geography focuses on the earth and the reference systems that bind the earth, spatial analysis focuses on space and the use of space. That space and coordinate reference system may be specific to an anthill, or to a map of a nuclear plant whose location is yet to be defined, or it may be used as a visualization tool to model the inherently non-visual, such as in process modeling. So while you may think of your particular area of interest as not 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're dealing with data of a spatial nature or using space as a visualization tool, PostGIS provides more functions to extend that analysis.
- Much of this data can be easily collected by machines (GPS, alarm systems, remote sensing devices) and directly piped to the database via automated feeds or standard import formats.
- Portions of data are easily distributed. A relational database is ideal for creating what we call "data dispensers," which allow 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.

These profiles are the basic groups of spatial database users, but they're not the only ones.

If you've 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 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 appendixes.

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. By the end of this part, you should have a solid foundation in the various geometry types, a basic understanding of spatial reference systems and database storage options, and, most important, the ability to load and query spatial data in a PostGIS-enabled PostgreSQL database.

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

In chapter 2 we go through all the geometry types that PostGIS has to offer, most of which are standard across most high-end spatial databases. You'll learn how to create these on the fly using well-known text (WKT) representations. You'll 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 won't find in most other relational databases. In this chapter we explore using table inheritance, examine heterogeneous/homogeneous geometry columns, and take a brief look at the hstore key-value data type. We'll also demonstrate how to compartmentalize business logic in the database using PostgreSQL rules and triggers.

Chapter 4 discusses 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 one or more geometries as input.

Chapter 6 is a basic primer on the very important topic of spatial reference systems. It discusses how to determine which reference system your data is in and how to select suitable reference systems 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 shapefiles, MapInfo, KML, and 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'll learn about common mistakes people make when writing queries and how to avoid them. You'll also 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 encompasses the tools most commonly used with PostGIS for building applications.

Chapter 10 covers add-ons you can use with PostGIS directly in spatial queries. It demonstrates the TIGER geocoder and 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 and MapServer, provides the fundamentals of the WMS/WFS OGC web services, and discusses using the 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'll learn the pros and cons of each and you'll find quick primers on installing and working with each. Covered are OpenJUMP, Quantum GIS, uDig, and gvSIG.

Chapter 13 is an introduction to the PostGIS raster data type. Raster support isn't packaged in with PostGIS 1.5 or below but is packaged with PostGIS 2.0. This chapter will teach you how to load raster data using GDAL, do intersections with geometries, polygonize rasters, and do basic analysis with raster pixels.

APPENDIXES

There are four appendixes.

Appendix A provides additional resources for getting help on PostGIS and the ancillary tools discussed in the book.

Appendix B shows how to get up and running with PostgreSQL and PostGIS.

Appendix C is an SQL primer that explains the concepts of JOIN, UNION, INTERSECT, and EXCEPT. It discusses the fundamentals of rolling up data with aggregate functions and aggregate constructs as well as the more advanced topic of using Window functions and frames.

Appendix D covers features of 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 callouts 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 ❶ that are referenced later in the text.

Code downloads

The examples and data for all chapters of this book can be downloaded via http://www.postgis.us. On the book site you'll also find chapter code downloads, data downloads, and descriptions of each chapter with related links for each chapter. Each chapter page listing has a link where you can download the full data and code for that chapter.

The code can also be downloaded from the publisher's website at http://www.manning.com/PostGISinAction.

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're 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's being learned. People understand and remember new things, which is to say they master them, only after actively exploring them. Humans learn in action. An essential part of an *In Action* book is that it's example driven. It encourages the reader to try things out, to play with new code, and to explore new ideas.

There's 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.

about the cover illustration

The figure on the cover of *PostGIS in Action* is captioned "A woman from Ubli, Croatia." The illustration is taken from a reproduction of an album of Croatian traditional costumes from the mid-nineteenth century by Nikola Arsenovic, published by the Ethnographic Museum in Split, Croatia, in 2003. The illustrations were obtained from a helpful librarian at the Ethnographic Museum in Split, itself situated in the Roman core of the medieval center of the town: the ruins of Emperor Diocletian's retirement palace from around AD 304. The book includes finely colored illustrations of figures from different regions of Croatia, accompanied by descriptions of the costumes and of everyday life.

Ubli is the main ferry port on the island of Lastovo, located in an archipelago of islets in the Adriatic Sea off the coast of Croatia. The main characteristic of an Ubli woman's costume is the rich and colorful embroidery. Over a white linen dress that is trimmed with red bands, women typically wear a long blue vest decorated with red woolen roses as well as an embroidered apron. Colorful woolen socks and a little red hat decorated on the edges complete the costume. Live flowers are often added to the back of the hat.

Dress codes and lifestyles have changed over the last 200 years, and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone of different hamlets or towns separated by only a few miles. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

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'll learn how to perform actions with single lines of SQL code that you thought were possible only with a desktop GIS system. By using spatial SQL, much of the heavy lifting that would require many manual steps in desktop GIS tools can be scripted and automated.

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

Chapter 1 covers the fundamentals of spatial databases and what you can do with a spatially enabled database that you can't do with a standard relational database. It concludes with a fast-paced example of loading fast food restaurant longitude/latitude data and converting it 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'll learn how to create these using well-known text (WKT) representations and the concepts of validity and simplicity.

Chapter 3 covers different approaches for storing data in PostgreSQL/PostGIS. You'll learn how to store multiple kinds of geometries in a single geometry column as well as how to control 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 it to enhance flexibility and for partitioning data. We'll end with a real-world example using data from Paris, France.

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 and functions for processing that can simplify and morph geometries. You'll also learn the fundamental accessor and measurement functions.

Chapter 5 covers relationships functions. These are the most common 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 Dimensionally Extended 9 Intersection Model (DE9IM) that most spatial relationship functions are based on.

Chapter 6 is an introduction to spatial reference systems, and we'll 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, and shp2pgsql-gui, as well as the 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 you learned about in part 1 as well as optimizing for performance.

In part 3 we'll conclude the book with an overview of various common open source tools used to enhance the power of PostGIS and PostgreSQL.

1 What is a spatial database?

This chapter covers

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

In this chapter we'll first introduce you to spatial databases. You'll learn what they are and how they allow you to model space and perform proximity queries that are impossible with a plain relational database system. We'll then focus on PostGIS, a spatial database extender for the PostgreSQL database management system. We'll dive in with quick examples of loading spatial data and performing proximity analysis with PostGIS.

1.1 Thinking spatially

Popular mapping sites such as Google Maps, Virtual Earth, MapQuest, and Yahoo have empowered people in many walks of life to answer the question "Where is something?" by showing it on a gorgeously detailed, interactive map. No longer are we restricted to textual descriptions of "where" like "Turn right at the supermarket

and it's the third house on right." Nor are we faced with the perennial problem of not being able to figure out where we currently are on a paper map.

Going beyond getting directions, organizations large and small have discovered that mapping can be a great resource for analyzing patterns in data. By plotting the addresses of pizza lovers, a national pizza chain can visibly see where to locate the next grand opening. Political organizations planning on grassroots campaigns can easily see on a map where the undecided or unregistered voters are located and concentrate their route walks accordingly. While these mapping sites have given unprecedented power to interactive mapping, using them still requires that users gather point data and place it 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 showing pizza lovers by means of pushpins, but suppose we further differentiate pizza lovers by income level. If the chain has more of a gourmet offering, it would 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, as shown in figure 1.1. Not only does the planner need to look at the concentration of pushpins, but 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.

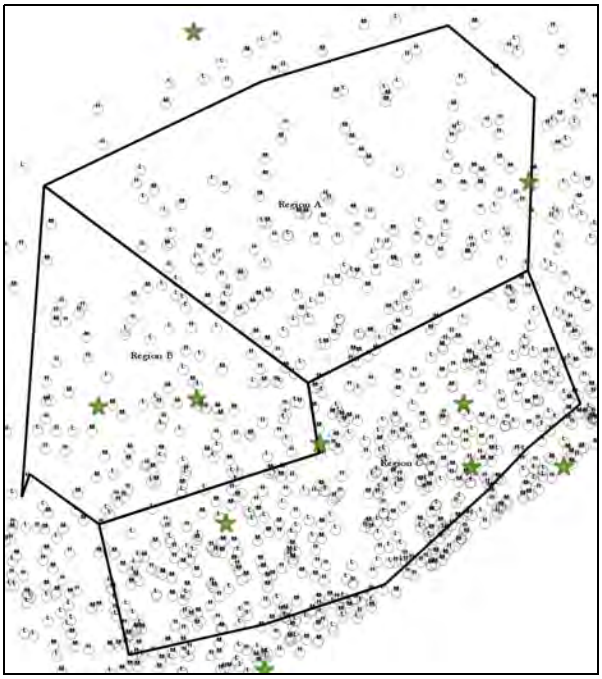


Figure 1.1 Pushpin madness!

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). A spatial database is often used as just a storage container for spatial data, but it can do much more than that. Although a spatial database need not be relational in nature, most of the well-known ones are.

A spatial database gives you both a storage tool and an analysis tool. Presenting data visually isn't a spatial database's only goal. The pizza shop planner can store an infinite number of attributes of the pizza-loving household, including income level, number of children in the household, pizza-ordering history, and even religious preferences and cultural upbringing (as they relate to topping choices on a pizza). More important, the analysis need not be limited to the number of variables that can be juggled in the brain. The planner can ask questions like "Give me a list of neighborhoods ranked by the number of high-income pizza lovers with more than two children." Furthermore, she can add unrelated data such as location of existing pizzerias or even the health-consciousness level of various neighborhoods. Her 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 shows what the results of such a spatial query might look like.

Table 1.1 Result of a spatial query

Region	#Households	#Restaurants	#Avg Travel
Region A	194	1	17.1 km

Suppose you aren't 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're 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 you have the latitude and longitude of all the employees' addresses; you 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 question posed is "Give me the number of houses within two miles of the coastline requiring evacuation in the event of a hurricane" or "How many households would be affected by the noise of a newly proposed runway?" Without spatial support, these questions would require collecting or deriving additional values for each data point. For the coastline question, you'd need to determine the distance from the beach house by house. This could involve algorithms to find

the shortest distance to fixed intervals along the coastline or using a series of SQL to order all the houses by proximity to the beach and then making a cut. With spatial support all you need to do is to reformulate the question slightly as "Find all houses within a two-mile radius of the coastline." A spatially enabled database can intrinsically work with data types like coastlines (modeled as linestrings), buffer zones (modeled as polygons), and houses (modeled as points).

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

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

1.1.1 Introducing the geometry data type

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

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

You need not limit yourself 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 doghouse in the back yard could be represented by a point. See how interesting this could be? Just by reducing the landscape to two-dimensional points, linestrings, and polygons, as shown in figure 1.2, you have the tools to model everything.

Don't 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'll never perfectly mimic the real thing. Don't be overly concerned with geometries that you feel should be included but are missing. Two good examples are football stadiums and 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 aren't defined (at least for now), but they could be approximated closely enough with polygons.

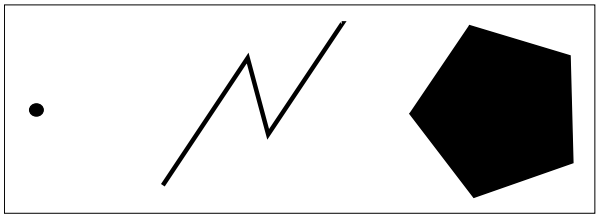


Figure 1.2 The basic geometries: a point, a linestring, and a polygon

1.2 Modeling

We've introduced our building blocks of points, linestrings, and polygons. We've given an example of how to model real-world geographies using these basic geometries and have invited you to come up with your own. Modeling isn't the only step in problem solving. Let's go back to the salt flats of Utah. We know the interstate traverses the salt flats 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 driver may instruct her navigator to measure the distance from the mile marker where the interstate entered the salt flats to the mile marker where the interstate exits the salt flats, as shown in figure 1.3. Unbeknownst to the motorist, she has just formulated a spatial query.

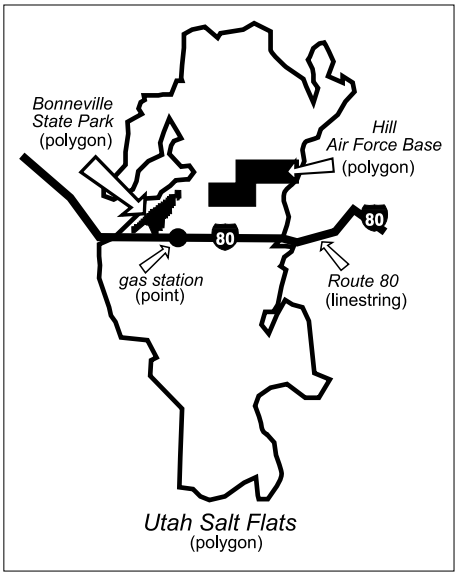


Figure 1.3 The Utah salt flats—we can model them with linestrings, points, and polygons.

With our points, linestrings, and polygons in hand, let's dissect this simple act of measurement and then see if we can ask it in a way that a spatially enabled database could easily answer. To start the measurement, the navigator looks for the point on the map where the interstate first hits the salt flats. This point is the intersection of the linestring with the polygon. The navigator then proceeds to find the place where the interstate leaves the salt flats and comes up with another point of intersection. Given that the highway is completely straight during its crossing of the salt flats, the navigator can use a ruler to measure from the first intersection point to the second intersection point. Let's 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 find the intersection of the two geometries. A line intersected with a polygonal area yields the linestring contained within the polygon. We finish by taking the length of the linestring.

One linestring through a polygon seems more like an exercise in geometry than a database query, but it's 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 search for two tables: one with the polygons for all the states and one with all interstate highways in the United States, represented as linestrings. Next we extract the polygon that represents Utah from the states table 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 of that query would be those portions of all highways within the state of Utah. Finally, we aggregate those portions using the SQL LENGTH and SUM functions, and there's our answer. With the Utah example we demonstrated

that geometric data types and functions can leverage the querying power of a relational database and can easily lead to solutions for problems that at first sight looked insurmountable.

Don't worry if you aren't 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, and UPDATE statements will get you through 85% of the spatial queries that you'll need to write.

1.2.1 Imagine the possibilities

We've demonstrated the power of spatial queries without 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 a body of functions to the standard SQL language 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 answer 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 previously we talked about points, linestrings, and polygons in two dimensions. These are the fundamental building blocks. In a spatially enabled database such as PostGIS/PostgreSQL, more complex objects exist such as MULTIPOLYGONS, MULTIPOINTS, MULTILINESTRINGS, GEOMETRYCOLLECTIONS, and curved geometries. In addition to the 2D world we've described, you can also have these 2D objects sitting in 3D space. This is often referred to as 2.5D and is the first step to real 3D modeling.

In the PostGIS 2 series of releases we'll start seeing true 3D support in the form of polyhedral surfaces and triangulated irregular network (TIN) as well as relationship functions to work with 2.5D and 3D. Also introduced in PostGIS 2.0 is another kind of data called raster data. 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. 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 them up, creating derivatives, and overlaying them on a map. We'll cover these more complex objects including the currently available PostGIS raster functionality in the later chapters of this book.

1.3 Introducing PostgreSQL and PostGIS

In the rest of this chapter, we'll talk more about PostgreSQL and PostGIS. PostGIS is a free and open source (FOSS) library that spatially enables the free and open source PostgreSQL object-relational database management system (ORDBMS).

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, and text and that allows the user to define new custom data types, new functions, and operators that manipulate these new custom types.

The major reason PostgreSQL was chosen as the platform on which to build PostGIS was the ease of extensibility it provided for building new types and operators and for controlling the index operators.

1.3.1 PostgreSQL strengths

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

A brief history of PostgreSQL

If you were to look at the family tree of PostgreSQL, it would look something like this:

```
System-R (1973)
  Ingres (1974)
      Postgres (1988)
        Illustra (1993)
          Informix (1997)
            IBM Informix (2001)
        Postgres95 (1995)
          PostgreSQL (1997)
```

PostgreSQL is a cousin of the databases Sybase and Microsoft SQL Server because the people who 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 Ingres and 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.

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

POSTGRESQL UNIQUE FEATURES

PostgreSQL has many features that are rarely found in other databases. Some of its features don't exist in other databases at all.

- *Various languages to choose from for writing database functions that can return simple scalar values as well as data sets and for building aggregate functions*—No open source or commercial database to our knowledge can compete with PostgreSQL in this regard. Commonly used ones are built-in SQL, PL/PGSQL, and C. In addition to the three built-in languages, PL/Perl, PL/Python, PL/TCL, PL/SH, PL/R, and PL/Java are also often used. These require additional environment installs such as Perl, Python, TCL, Java, and R in order to take advantage of them. IBM DB2 and Microsoft SQL Server come close with allowing .NET functions, but this isn't quite as elegant as being able to write the code right in the database. Oracle supports only PL/SQL and Java. In addition, the PostgreSQL PL platform is the most extensible of any database platform, making it easy register new language handlers. Watch for PL/Parrot, a procedural language handler for the Parrot system that allows for combining multiple dialects of languages in one procedural language.
- *Support for arrays*—PostgreSQL, Oracle, and IBM DB2 are fairly unique among databases in that arrays are first-class citizens. In PostgreSQL, you can define any table column as comprising an array of strings, numbers, dates, geometries, or even your own data type creations. This comes in handy for matrix-like analysis or aggregation. In addition, you can convert any single-column row list to an array, which is particularly useful when manipulating geometries.
- *Table inheritance*—PostgreSQL has a feature called table inheritance, which is something like object multi-inheritance. Table inheritance allows you to treat a whole set of tables as a single table as well as define nested inheritance hierarchies. It's often used for table-partitioning strategies. We'll demonstrate the power 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 isn't commonly exploited and 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:

  How to create multi-column aggregates: http://www.postgresonline.com/journal/archives/105-How-to-create-multi-column-aggregates.html

  Making SVG plots with PLPython and multi-column aggregates:

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

BASIC ENTERPRISE FEATURES

In addition to the unique features native to PostgreSQL, PostgreSQL also sports basic enterprise features that make managing mission-critical information easier.

- *Exceptional ANSI-SQL compliancy, even when compared with 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's good for optimizing intricate joins and aggregations without the need for hints*—The speed is comparable to enterprise-class DBMS for even the hairiest of SQL statements.
- *Ability to define new data types fairly easily in both C and the built-in languages.*
- *Relational views with the 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 and Microsoft SQL Server 2005+ use. It also has features such as transaction save points.
- *Thousands of built-in functions and contributed functions*—These are for anything from string manipulation, regular expressions, and regression analysis to analysis of astronomical data.
- *Similarity to PL/SQL*—Those coming from an Oracle background 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, which other databases lack and which is much easier to write for simple set returning or calculation functions. Unlike other language functions, an SQL function isn't a black box to the PostgreSQL planner. The major benefit of this is that it can be incorporated in the plan strategy. The logic is frequently in-lined in the query, similar to a macro in C. This makes it often more efficient than a PL/PgSQL or other language function while still hiding the complexity from the person utilizing the function.
- *Ability to run on pretty much any OS you can think of.*
- *Ability to define column-level permissions, introduced in PostgreSQL 8.4.*
- *Ability to write variadic functions, also introduced in PostgreSQL 8.4*—This allows you to write a single function that has a default argument if it's not passed in. So `getMyElephant('blue'), getMyElephant()` would use the same function, but `getMyElephant` would use the default color defined in the function. PostgreSQL 9.0 extends the ways you can call functions by allowing named argument call notation similar to what you'll find in languages like VB and Python: `getMyElephant(color := 'blue')`.

ADVANCED ENTERPRISE FEATURES

In addition to the basic enterprise features PostgreSQL sports, it offers advanced enterprise features you'll rarely find in other open source databases. Some of these features are also more advanced than the equivalent you'll find in commercial database offerings.

- *Ability to easily write your own aggregate function in most any supported language including SQL*—This feature is particularly useful for something like spatial analysis.

The simplicity and ease of writing aggregates will come as a shock for those who have come from other databases that allow this, but it require immense amounts of code to do it.

- Windowing functionality introduced in *PostgreSQL 8.4*—Many of the high-end commercial databases such as IBM DB2 and Oracle have had this functionality, and Microsoft SQL Server introduced it in its SQL Server 2005 offering. This is useful for OLAP and data warehouse applications and even more important for nearest-neighbor searches, as we'll demonstrate. In PostgreSQL 9.0 this feature was enhanced to include numbered ROW RANGES, thus coming closer to the capabilities of Oracle's windowing functionality and far surpassing SQL Server 2008 R2's windowing functionality.
- *Recursive common table expressions for writing recursive queries (useful for navigating trees) and common table expressions functionality, introduced in PostgreSQL 8.4*—These are found in the popular high-end commercial databases. We'll 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 it's almost exactly what you'd write in Microsoft SQL Server and IBM DB2. Oracle has had its own variant for doing hierarchical queries almost since its inception called CONNECT BY that doesn't follow the standard. Oracle introduced in its 11GR2 offering ANSI-compliant recursive common table expressions that follow the same CTE syntax as PostgreSQL, SQL Server, and IBM DB2.
- *Database restore supports parallel restore of tables, introduced in PostgreSQL 8.4*—This makes the database restore four times faster than it was in 8.3 and below, which is important for huge databases. In addition, the compressed backup storage format of PostgreSQL has always supported selective restore of objects.
- *Column-level triggers and conditional triggers as well as simplified security management, introduced in PostgreSQL 9.0.*
- *Anonymous functions in any PL language via the new DO command, introduced in PostgreSQL 9.0*—This allows for running Python, Perl, TCL, and PL/PgSQL code without defining a function.
- *Built-in warm standby and streaming replication introduced in PostgreSQL 9.0.*
- *64-bit support on Windows introduced in PostgreSQL 9.0.*
- *In-place upgrade introduced in PostgreSQL 8.4 and enhanced in later versions.*

MORE FEATURES IN POSTGRESQL 9.1

PostgreSQL 9.1 introduces many more sought-after features, both for enterprise and for ease of use.

- *Support for ANSI SQL–compliant triggers on views.*
- *Enhancements to CTEs to support UPDATE/INSERT/DELETE.*

- *Functional dependency on foreign keys, which will simplify GROUP BY clauses by not requiring you to group by additional fields in a table if the primary key is already in the GROUP BY.*
- *CREATE TABLE IF NOT EXISTS similar to what MySQL has long had.*

1.3.2 PostGIS, adding GIS to PostgreSQL

PostGIS is a project spearheaded 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 complementary features that PostgreSQL and other PostgreSQL-related projects provide, then you have one jam-packed powerhouse at your disposal that's well suited for hard-core 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 Inc.'s Java Topology Suite (JTS), historically incubated by Refractions Research and now a project of Open Source Geospatial Foundation (OSGeo). The foundation of PostGIS, the PostgreSQL object-relational database management system (ORDBMS), which provides transactional support, gist index support used to index spatial objects, and a query planner out of the box for PostGIS, is perhaps the most important of all. It's 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. This includes both open source and commercial tools that can work with it and numerous toolkits 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're 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, Quantum GIS, deegree, UMN MapServer, GeoServer, MapDotNet,

(continued)

and the like—then everyone can use the tools they feel most comfortable with or fit their work process and/or can afford and share information with one another. 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 whose salaries and tools are paid for with tax dollars, students who have a lot of will and the intelligence 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-name 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 aren't simply learning how to use a set of products; you're 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 how it works 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 won't find even in the commercial offerings. In general it has more output formats than the commercial offerings and its speed is on par with and sometimes better than the commercial ones for common spatial needs.

1.3.3 Alternatives to PostgreSQL and PostGIS

Admittedly, PostGIS/PostgreSQL isn't 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 its included Locator, priced add-on Oracle Spatial, and before those its spatial data option (SDO)), IBM DB2 and IBM Informix (with their add-on priced options Spatial DataBlade and Geodetic DataBlade). Recently SQL Server 2008 provided spatial functionality with its built-in Geometry and Geodetic Geography types and companion spatial functions. Ingres, the older cousin of PostgreSQL, is also enhancing its spatial support, using some of the same plumbing that underlies PostGIS. The new kid on the block is SpatiaLite, which is an add-on to the open source SQLite portable database. SpatiaLite is especially interesting because it's well positioned to be used as a low-end companion to PostGIS and other high-end spatially enabled databases. It can be used to create master/slave applications to provide basic lightweight spatial support for clients such as portable devices. It also has a companion, RasterLite, that's for the most part focused on raster data storage and display and so makes a great companion to the raster analysis that's being developed in

PostGIS. SpatiaLite and RasterLite also use many of the core libraries that PostGIS uses: GEOS, PROJ, and GDAL. This fact makes it an even more fitting companion to PostGIS, because many of the conventions are the same and much of the ecosystem around PostGIS also supports or is starting to support SpatiaLite/RasterLite. What SpatiaLite lacks is a strong enterprise database behind it that allows for writing advanced functions and spatial aggregate functions. That's why some spatial queries possible in PostGIS are harder to write or not even possible in SpatiaLite. SpatiaLite's single file and embedded engine make it less threatening and easier to deploy for users new to databases or GIS.

MySQL has had spatial support for quite some time, but its spatial development is fairly stagnant and even weaker than that of SpatiaLite. Now that it's a property of Oracle, it's questionable how interested Oracle will be in beefing up MySQL spatial support, because this competes directly with its Oracle Locator/Spatial offering. MySQL 4 and 5 have built-in spatial functionality, with one major drawback: Their geometric functions have until now worked only 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 its spatial database engine (SDE) with its ArcGIS product. The SDE engine is integrated into the ArcGIS line of products and was often used to spatially enable external databases such as Microsoft SQL Server 2005 and below that 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, because in order to use the ST_Geometry type and functions that ArcGIS provides, you need to go through the middleware layer of ArcGIS Server.

1.3.4 What works with PostGIS

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

- *Cadcorp SIS*—This vendor is partially funding the raster support in PostGIS and is a favorite among modelers for both desktop and web-based apps. Cadcorp supports more than 160 formats, including direct support for all other high-end spatial database offerings.
- *Safe FME*—It contributes both monetary and developer support for GEOS and makes extract transform load (ETL) tools for GIS data, which makes moving GIS data transport to different formats and databases a simple drag, drop, and schedule exercise. It's the favorite for high-end ETL transactions.
- *Manifold*—It released support for PostGIS in its version 8.0 and above product, and it's 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 the ESRI ArcGIS desktop that works with 9.2 and above and allows you to access PostGIS data without an ArcSDE license. It doesn't work with ArcGIS Server as of this writing.
- *ArcGIS*—In ArcGIS 9.3, ESRI introduced support for PostGIS. Although this requires an ArcSDE Server license for PostGIS and works only with PostGIS 1.4 and below (as of ArcGIS 10), it may not be suitable for people on a limited budget or who want to use more recent versions of PostGIS or PostgreSQL. ArcGIS is best known for its cartography. See http://resources.arcgis.com/content/arcsde/10.0/postgresql-system-requirements.
- *Pitney Bowes MapInfo 10*—Pitney Bowes introduced support for PostGIS in its 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's a favorite of lightweight GIS users and database analysts because of its rich query options and easy data import menus.

Its commercial vendor support is now just as strong as what you'll find available for Oracle, SQL Server, or IBM DB2. PostGIS has garnered more support in the free open source GIS arena than any other spatial database, far exceeding the spatial offerings of MySQL. There are too many PostGIS open source tools to list. We'll cover the more common offerings in our desktop and web tools chapters. As you can see, PostGIS has an already strong and growing commercial support belt as well.

1.4 Getting started with PostGIS

In this section we'll show some simple examples for creating geometries with PostGIS; in later sections of this chapter we'll cover loading and querying spatial data. Before going further, you'll need to have a working copy of PostGIS 1.3 or higher and PostgreSQL 8.2 or higher, as well as ancillary tools such as pgAdmin III to compose and execute your queries. Information about acquiring and installing these can be found in appendix B. As always, if you're starting completely from scratch, we recommend that you install the latest versions.

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. Precompiled binaries are available for most operating systems. pgAdmin III 1.9 and above support a plug-ins architecture, which allows you to call shp2pgsql-gui and any other executables you like from within pgAdmin. The psql command-line client is packaged with it as a plug-in. All this is configurable by editing the plugins.ini file.

1.4.1 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 that we describe in chapter 12. For most of these exercises we use OpenJUMP for visualization. For detailed installation guides on PostgreSQL, PostGIS, and pgAdmin III, please refer to appendix B. The following examples require PostGIS 1.3 or above and PostgreSQL 8.2 or above. In later sections, we'll 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 shown here:

```
POSTGIS="1.5.2" 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're running:

```
SELECT version();
```

If all is well, you should see the PostgreSQL version and operating system:

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

1.4.2 Creating geometries with PostGIS

We'll 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 didn't specify a spatial reference system for our simple point. The default coordinate system type used is the basic Cartesian grid you learned in childhood. In most real-world applications, you'll 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's used to define geometry points. This is a bit of a simplistic definition but will do for now. You can be assured that if two geometries are in the same spatial reference system, they can be overlaid without distortion. PostGIS packages about 3,000 of these, and these are all denoted by numbers (currently just European Petroleum Survey Group (EPSG) standard codes that are common in the industry) and can be looked up in the included spatial reference table. They define how geographic data is represented on a flat map and what units of measurement (degrees, feet, meters) the coordinate system uses. Spatial reference systems are in general good for only a specific region of the globe.

With this in mind, let's create yet another point that has real geographical relevance:

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

Here we added a function to indicate that our point is using a spatial reference system known as WGS 84 Lon Lat. This refers to the longitude and latitude that most people are familiar with. Incidentally, this 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 all distances to be measured from this point. You can read more about it at http://en.wikipedia.org/wiki/Zero_Milestone.

The function ST_GeomFromText offers a more generic method for creating various geometry types from a textual representation. This function is slower and less accurate than the ST_Point function, but it's intuitive and applies to all geometric types. Here's how you'd create a point with ST_GeomFromText:

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

This approach can be used to create any geometry with what is called well-known text (WKT) representation of a geometry.

Well-known text representation of geometries

Well-known text representation is an OGC standard for representing geometries as text. ST_AsText and ST_GeomFromText are OGC functions found commonly in many OGC-compliant spatial databases that convert back and forth between a database's native binary format to a textual display format.

You should have noticed that the result of this geometry constructor would look like this:

```
0101000020E6100000FD2E6CCD564253C0A93121E692724340
```

Most spatial databases store geometries in some a binary format that's 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 previous PostGIS binary format using this function:

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

You'll 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 doesn't include the spatial reference identifier.

Lon lat vs. lat lon

You'll note that the points are stored in longitude, latitude and not latitude, longitude. This storage is 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's move on to more complex examples of creating linestrings and polygons. PostGIS is primarily used to store and query geographic data, but we can also use PostGIS to represent any data that can be drawn using a Cartesian coordinate system. We'll start by creating a linestring without specifying a spatial reference system. We'll use the ST_GeomFromText function without the spatial reference parameter to create the geometries shown in figure 1.4.

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

This creates a check mark–like linestring through the origin. As you may be able to observe, 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 in the figure is closed. To draw a polygon, we specify the linestring forming its boundary. Here's a triangle:

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

Because 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;
```

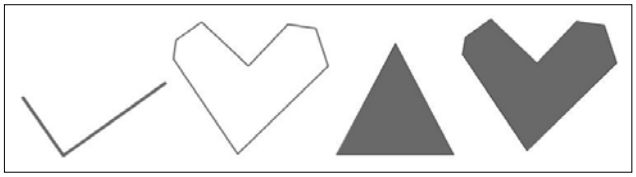


Figure 1.4 Linestrings and polygons created in the following code snippets

The first interesting thing about the WKT representation of a polygon is that it has one additional set of parentheses that a linestring doesn't have. Why is this? A polygon can have holes, and each hole needs its own set of parentheses. Our triangle and our heart happen to not have any holes, so the extra parentheses appear redundant. When we explore polygons in detail in the next chapter, we'll show you how to create donut-like polygons.

Now that we've covered the basics of geometry creation, we'll cover the more common case of loading preexisting spatial data from other sources and querying the loaded data.

1.5 Working with real data

In this section, we'll cover how to load data from the two most common formats: delimited ASCII data and ESRI shapefile data. For delimited data, we'll demonstrate how to convert those to spatial points, using the lessons you learned in the previous section. For shapefile data, we'll load a data file of road linestrings, using the ESRI shapefile loader packaged with PostGIS.

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, like longitude/latitude coordinates, into spatial data types such as geometry or geography. For geometry data, we often need to transform lon lat values to a planar-based spatial reference system that's suitable for planar measurement.

PostGIS geography vs. geometry measurement

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

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

If you're 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's going on. These processes will be repeated in later chapters and become clearer when you see them again.

1.5.1 Loading comma-separated data

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

Before we load in data, we create the schema and table to house our data. In listing 1.1 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'll be loading.

Listing 1.1 Set up fastfoods and franchises lookup

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

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');

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

❶ Create logical container
❷ Create lookup table
❸ Add franchise types
❹ Table holds locations

❶ First we create a schema to hold our data. A schema is a container you'll find in most high-end databases. It logically segments objects (tables, views, functions, and so on) for easier management. Refer to appendix D for more details. ❷ Next we create a lookup table to map franchise codes to meaningful names ❸. We then add all the franchises we'll be dealing with ❹. Finally, we create a table to hold the data we'll be loading. We define only columns that are in our source dataset. We'll 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 have 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 function, 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 the PostgreSQL-packaged psql command-line tool. It requires that the computer you're running psql on have access to the file path and also that your OS account have access to read the file. The path is relative to the client computer. We offer a more thorough description of these distinctions 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 method, we'd do the following. Note that when we use the SQL COPY command, the /data/ path references the path on the PostgreSQL server:

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

Using psql, the command-line tool, we'd do the following, and the path would reference the folder on our local computer from which we started psql:

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

After we've finished 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 the psql command line 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 option.

1.5.2 Spatializing flat file data

Until now, our lon lat values have been plain database numbers. We need to convert these numbers into either geometry or a geography data type in order to take advantage of PostGIS's spatial functionality.

If you're using PostGIS 1.4 or lower, you can choose only the geometry data type. With 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 values. This means that you can cover the whole world with geography data and never have to learn anything about 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. Also, fewer functions are available for it than for geometry. The speed of functions such as ST_Intersects and other relationship functions for geography is currently lower than for the geometry data type, though this will change in time. On the other hand, it's fairly easy to convert between these two spatial types.

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. The similarity extends beyond nomenclature to include functionality as well. The SQL Server 2008 geometry type is an OGC type similar to the PostGIS OGC geometry type. Just like the PostGIS geometry type, the SQL Server 2008 geometry type treats all data as planar data. PostGIS and SQL Server geography data types aren't OGC types, though they try to follow many of the same function and naming conventions as OGC geometry functions. OGC is mainly concerned with planar geometries and operations. 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're 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, whereas PostGIS has robust support for this for its geometry datatype. As for geographies, SQL Server 2008 supports many lon lat–based spatial reference systems. PostGIS geography currently supports only EPSG:4326, also referred to as WGS 84 Lon Lat. WGS 84 Lon Lat is the most common spatial reference system for longitude latitude data.

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

USING THE GEOMETRY DATA TYPE

If we were to use the geometry data type, we'd do the following.

Listing 1.2 Using the geometry data type to store data

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

CREATE INDEX idx_fastfoods_geom
  ON ch01.fastfoods USING gist(geom);
```

❷ Use equal area planar

❶ Add column

❸ General maintenance

❶ We first add a geometry column to the fastfoods table to house our future spatial points. ❷ Because geometry is a planar projection, we want our points to be in a projected coordinate system that covers our area of interest. We've picked EPSG:2163, which is an equal area projection covering the continental United States. The measurements of its coordinate system are in meters and are a little less accurate when modeling the earth as a sphere. ❷ The PostGIS ST_Transform takes our data from lon lat degrees and projects it to North American Equal Area. The main drawback of this particular spatial reference system is that because it covers a fairly large area, measurements will less accurate than if we had used geography, which by default assumes a spheroid model of the earth. Speed will be faster and we'll be free to use

the vast array of geometry functions. We then ❸ do some general housecleaning by creating a spatial index on our new column.

USING THE GEOGRAPHY DATA TYPE

If we were to use the geography data type, we'd do the following with our data.

Listing 1.3 Using the geography data type to store data

```
ALTER TABLE ch01.fastfoods ADD COLUMN geog geography(POINT,4326);
UPDATE ch01.fastfoods
        SET geog =
                ST_GeogFromText('SRID=4326;POINT(' || lon
          || ' ' || lat || ')');
CREATE INDEX idx_fastfoods_geog
➥ ON ch01.fastfoods USING gist(geog);
```

❶ Add column

❷ Use geodetic

❸ General maintenance

In this example we explicitly use SRID=4326 in our construction. We do this for clarity and also in case PostGIS in the future supports more spatial reference systems for geography. In practice, you can leave it out and PostGIS will assume 4326.

The steps we follow for creating geography columns are similar to what we did for geometry, but there are some differences. ❶ 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 constrained geography columns without the need for an AddGe.. function. ❷ Instead of using ST_GeomFromText, we use the parallel ST_GeogFromText, but the WKT representation is more or less the same. Because geography measures along a spheroid rather than a plane, we don't need to transform to get meaningful measurements. ❸ As we did with geometry, we create our spatial index.

It's good to follow up any bulk load process with a vacuuming:

```
vacuum analyze ch01.fastfoods;
```

The vacuum-analyzing step is a process that gets rid of dead rows and updates the planner statistics. It's good practice to do this after bulk uploads. If not done, the vacuum daemon process, which is enabled by default, will eventually do it for you.

ADDING CONSTRAINTS

Our fast-foods data has no primary key index. Unfortunately, nothing in the data file lends itself to a good natural primary key. For our later analysis, we'll need to uniquely identify restaurants so that we don't double-count them. Also certain mapping applications and viewers, such as QGIS and MapServer, have issues with tables without primary keys. They also complain if that primary or unique key isn't an integer. So we'll create an autonumber primary key on our fastfoods table.

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

Although not necessary for this particular data set because it won't be updated, we'll create a foreign key relationship between our fastfoods franchise column and our

lookup table. This helps prevent people from introducing franchises we don't know about in our restaurants table. Adding CASCADE UPDATE DELETE rules when we add foreign key relationships will allow us to change the franchise codes for our franchises if we want and have those update the fastfoods table automatically. By restricting deletes, we can prevent people from inadvertently removing franchises with extant records in the fastfoods table. One added benefit of foreign keys is that relational designers such as what you'll find in OpenOffice Base and other ERD tools will automatically draw lines between the two tables to visually alert you to the relationship.

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

We then 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 the pgAdmin GUI. There isn't any need to memorize how to do this in SQL, although it's standard Data Definition Language (DDL) code across many relational databases.

1.5.3 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, a GUI tool called shp2pgsql-gui makes loading this kind of data simple and user friendly. It can also be used as a plug-in to pgAdmin III. Details on how to install and get started with it are provided in appendix B. It's packaged with the Windows Stack Builder installs as well as the OpenGeo Suite 1.9+ GIS stack installs.

When shp2pgsql-gui is used as a plug-in in pgAdmin III, it reads your database credentials directly from pgAdmin III for the selected database. Even if you're using a version of PostGIS below 1.5, 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 without PostGIS.

For our next example we'll use shp2pgsql-gui. to demonstrate loading the road network data we downloaded from http://www.nationalatlas.gov/atlasftp.html#roadtrl. For this example we'll be going through pgAdmin and accessing from the Plugins menu, as shown in figure 1.5.

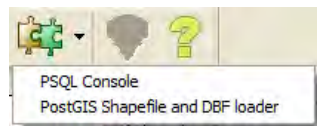


Figure 1.5 The Plugins menu of pgAdmin III shows the PostGIS Shapefile and DBF loader.

The Plugins option 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 dialog box shown in figure 1.6 comes up with the credentials of the selected database already filled in.

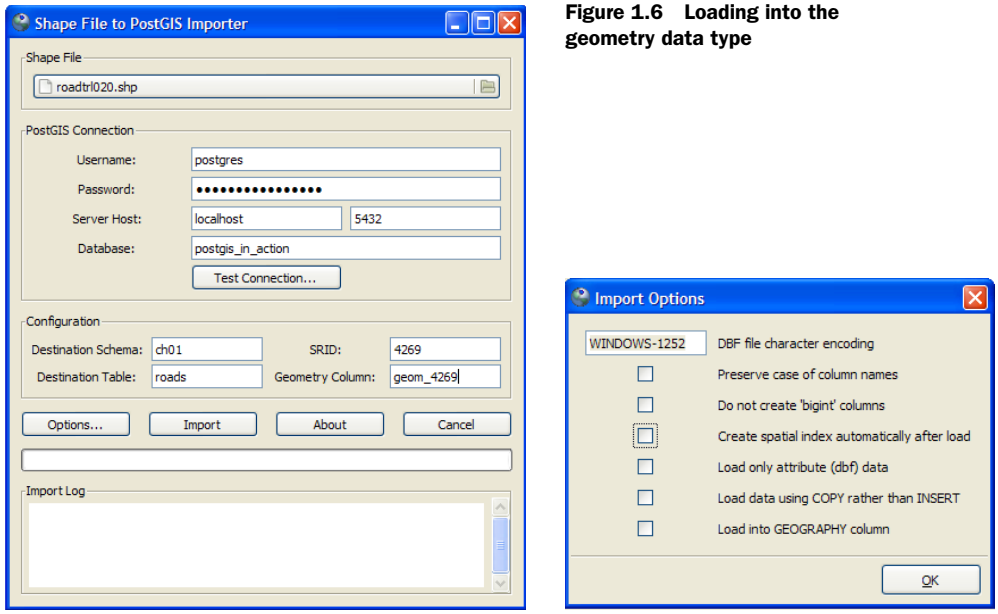


Figure 1.6 Loading into the geometry data type

If you don't have shp2pgsql-gui available, you can load the data using the command-line loader shp2pgsql. Shp2pgsql is included with all PostGIS packages for all versions of PostGIS. The equivalent command using shp2pgsql is

```
shp2pgsql -s 4269 -g geom_4269 /data/roadtrl020.shp ch01.roads |
➥ psql -h localhost -U postgres -p 5432 -d postgis_in_action
```

LOADING DATA INTO THE GEOMETRY DATA TYPE

In order to load into the geometry data type, you must browse to the file. Make sure to indicate that you'll be loading into the ch01 schema and into a new table called roads. Shp2pgsql-gui will create the table for you. Note that we also changed the default geometry column name to geom_4269. Click the Options button and uncheck Create Spatial Index Automatically After Load, and make sure Load Into GEOGRAPHY Column is unchecked. Normally you'd want a spatial index created when loading data, but in this case we don't because geom_4269 is a temporary column that we'll be dropping once we've finished with it. Suffice for now that the road data is in a lon lat spatial reference system called North American Lon Lat Datum 1983 (4269), which is similar to WGS 84 Lon Lat (4326). They're so similar that in most cases you can treat them the same. When you've finished setting the options, click the Import button.

Neither WGS 84 Lon Lat nor NAD 83 Lon 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 "rectangularization" is often referred to as a Plate Carrée projection. The other reason won't be using lon lat in geometry is that the measurements would return degrees. Unless you're a mariner,

degrees of longitude and latitude probably don't mentally translate well into distance measures. That being said, in listing 1.4 we'll transform this column into the same spatial reference system we're using for our fast-foods data so that it will be useful for doing spatial comparisons between them and being able overlay them both on the same map without distortion.

Listing 1.4 Using the geometry data type to store roads data

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

CREATE INDEX idx_roads_geom ON ch01.roads USING gist(geom);
```

1 Add column
2 Use equal area planar meters
3 Drop old column
4 General maintenance

In order to load the roads data into the geometry data type format, we use shp2pgsql-gui to load it into its native spatial reference system. Neither the shp2pgsql command line nor the GUI has transformation abilities. To make the input suitable for measurement, we transform it after we've loaded it in the database. To do so, ❶ we add a new column that holds a Cartesian spatial reference that covers our area of interest and is suited for doing measurements ❷. We update this new column by transforming our imported geometry data to the new spatial reference system ❸. After that, we no longer need the original column, so we drop it. ❹ Then we do our usual creating an index.

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

```
vacuum analyze ch01.roads;
```

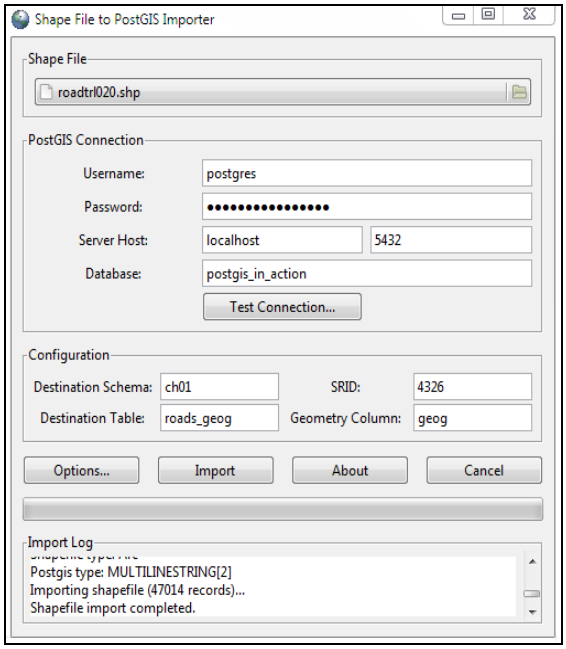
In the next example, we'll repeat the same exercise, but we'll load into the geography data type.

LOADING SPATIAL DATA INTO THE GEOGRAPHY DATA TYPE

For this particular data set, loading into the geography data type is much simpler than loading into geometry. The reason is that our data is already in lon lat units, and although NAD 83 Lon Lat isn't the same as WGS 84 Lon Lat, we can pretend it is because for most areas the differences are extremely small. Remember that not all lon lat data can be treated as similar. For instance, NAD 27 Lon Lat (4267) is sufficiently different that you would have to bring your in data as geometry, transform it to 4326, and cast it to geography—something along the lines of `geography(ST_Transform (geom_4267,4326))`.

We use the loader, as shown in figure 1.7, to ease the import.

As you can see in the figure, our settings are more or less the same as those we chose for geometry, except that we chose to create the index, load into geography, and load into a different table, roads_geog. We also changed the name of the column



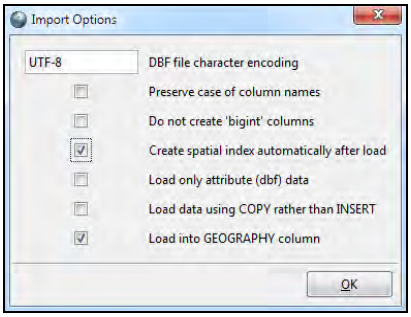


Figure 1.7 Loading data into the geography data type. We pretend our data is 4326 instead of 4269 because they're similar. We go ahead and index and check the load into geography.

so we know it holds geography data type data instead of geometry. The equivalent command using shp2pgsql is

```
shp2pgsql -G -g geog /data/roadtrl020.shp ch01.roads_geog |
➥ psql -h localhost -U postgres -p 5432 -d postgis_in_action
```

There's nothing else we need to do after this except to vacuum analyze ch01.roads_geog.

1.6 Using spatial queries to analyze data

Now that we have spatial data loaded in our database, we're able to spatially analyze this data with standard statistical SQL queries, yielding just raw numbers and text, as well as with queries returning geometric objects that can be rendered on a map. We'll start off by doing statistical queries to count restaurants by their proximity to roads and to determine which roads have the largest number of restaurants.

Before we got to this point, we transformed all the disparate data sets into a common planar spatial reference system, in this case North America Equal Area Meter (2163). Because we chose a meter-based reference, all our units are in meters. When we ask questions using miles, keep in mind that the conversion factor of 1609 meters equal one mile.

1.6.1 Proximity queries

One of the most common uses of PostGIS and other spatial databases is to compute the proximity of objects to each other. For the next couple of examples, we'll demonstrate how to combine the ST_DWithin function of PostGIS with standard SQL operations like COUNT, JOINS, and GROUP BY.

HOW MANY FAST-FOOD RESTAURANTS BY CHAIN ARE WITHIN ONE 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're using the geometry type in the following listing, the query itself would be written exactly the same for a geography column because an ST_DWithin function is available for both geometry and geography types.

Listing 1.5 List franchise name, count of restaurants on a principal highway

```
SELECT ft.franchise_name,
        COUNT(DISTINCT ff.ff_id) As tot
FROM ch01.fastfoods As ff
        INNER JOIN ch01.lu_franchises As ft
        ON ff.franchise = ft.franchise_code
         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

In this example we 1 use an ANSI-SQL COUNT(DISTINCT) construct to ensure we count a fast-food restaurant only once even if it's 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 pick up only restaurants within one 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 HALF-MILE RADIUS?

In this next example we'll demonstrate a slight twist to the earlier example and determine which highway has the highest number of restaurants along a half-mile boundary.

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

```
SELECT r.name,
  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*0.5)
WHERE r.feature LIKE 'Principal Highway%'
GROUP BY r.name
ORDER BY tot DESC LIMIT 1;
```

1 Distinct count

2 Within a half mile

3 One with greatest total

1 Because we're joining with a roads table with multiple records, it's possible for a restaurant to be within a half mile of more than one record, so we use the SQL DISTINCT clause to prevent double-counting. 2 We use the PostGIS ST_DWithin to consider only those points within a half mile of a road with a feature type of Principal Highway. 3 We care only about the road with the largest number of restaurants, which we can get by ordering by total count per road and picking the one with largest count (DESC).

This gives us an answer of U.S. Route 1 with a total of 414 restaurants. Anyone who has driven any segment of this old highway can attest to the abundance of roadside food shops.

1.6.2 Viewing spatial data with OpenJUMP

Although PostGIS is good for doing quick spatial analysis that's next to impossible to do by inspecting a map, it can also be used as a visualization 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 the geometry (always 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 U.S. Route 1 that runs through 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';
```

This gives us an answer of 3.

To spot check our results, we used OpenJUMP to overlay the portion of U.S. Route 1 that's in Maryland, the Hardee's restaurants within 10 miles of it, and the 10-mile

corridor. In order to display geometries with the default 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 U.S. 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 that we don't use INNER JOIN here because it would result in duplicates where a Hardee's restaurant is within 10 miles of more than one U.S. route record in Maryland.

```
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)
        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';
```

The results are shown in figure 1.8.

As you can see, not only is PostgreSQL/PostGIS a good analytical tool for doing simple number stats on spatial data, but it can also 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.



Figure 1.8 U.S. Route 1 in Maryland, with three Hardee's restaurants in the 10-mile buffer, and the 10-mile buffer around the route

1.7 Summary

In this chapter, we've given you a small taste of a spatially enabled database and how it fits in a relational database system. We sowed the budding idea of how to model real-world objects in space with spatial constructs.

We championed PostgreSQL and its spatial companion PostGIS. We demonstrated how PostgreSQL and PostGIS can be used together to analyze spatial patterns in data. We hope we've convinced you that

the PostgreSQL/PostGIS combination is one of the best choices (if not the best) for spatial analysis.

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

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

Before we can continue our journey, we must first analyze the different geometry types that PostGIS offers us and show you how to create these and when it's appropriate to do so. We'll explore geometries in greater detail in chapter 2.

2 Geometry types

This chapter covers

- PostGIS geometry_columns metatable
- 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 which basic geometries it supports. This chapter continues by explaining how PostGIS manages geometry data stored in the database. You'll learn about those tables in all PostGIS-enabled databases that provide an inventory of the geometry table columns and of the available spatial reference systems. We then show you the definition and characteristics of points, linestrings, and polygons and how to work with them in a PostGIS-enabled database. After covering these single geometries, we move on to 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), some are specific to linestrings such as the linear referencing functions, some ignore the third and fourth coordinates, and some reject curved geometries or don't work well with geometrycollections. For all intents and purposes, you can treat geometry on a par with other PostgreSQL data types such as dates, numbers, and text.

Native PostgreSQL geometry data types versus PostGIS geometry data type

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 don't follow the OpenGIS Consortium standards, nor do they support spatial coordinate systems. These types are divided into individual types called point, polygon, lseg, box, circle, and path. The built-in PostgreSQL box type and the PostGIS box2d support type have similar names but are different, incompatible data types.

2.1.1 The geometry_columns table

PostGIS uses a table named geometry_columns to store metadata 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. All other OGC-compliant spatial databases have a table with a similar or the same name, because this table is defined in the OGC Simple Features for SQL (SF SQL) specs.

Geometry columns versus layers

Geometry columns in a spatial table are often referred to as layers or feature classes when displayed in mapping applications.

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, and 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 four dimensions. The fourth 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 treat the fourth dimension as an extra attribute of a point in the geometry rather than as another spatial dimension. The fourth dimension can be a temporal dimension, but you could use it as an index for anything. We'll talk more about this M coordinate when we get to points.

What's a dimension?

In spatial speak, there are two kinds of dimensions.

The coordinate dimension 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 they have, always have a geometric dimension of 0. A linestring and multilinestring are one dimensional, and a polygon and multipolygon are two dimensional. You'll notice that we have no three-dimensional geometry types. Such types would be volumetric surfaces such as boxes and spheres and amorphous objects you'd find in real 3D space. These aren't supported in PostGIS 1.* versions, but in PostGIS 2+, these will be supported in new types called polyhedral surfaces and triangulated irregular network (TIN).

SRID

SRID stands for spatial reference identifier and is an integer that relates back to the primary key of the metatable spatial_ref_sys. PostGIS uses this table to catalog all the spatial reference systems available to the database. The spatial_ref_sys metatable contains the name of the spatial reference system, the parameters needed to reproject to another system, and by which authority the system has been defined.

SRID versus SRS ID

In common GIS lingo, there's another term with similar meaning called SRS ID (spatial reference system identifier), which is usually represented as the authority name plus the unique identifier used by the authority for the spatial reference system.

For example, the common WGS 84 lon lat has an SRS ID of EPSG:4326, where EPSG stands for European Petroleum Survey Group (www.epsg.org). 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 isn't the case with all spatial databases, and the same spatial reference system can go under multiple identifiers.

Keep in mind that using a different SRID doesn't 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're trying to measure the distance between two points that 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 return the distance calculated using the Pythagorean theorem, not the great circle distance that you might expect. Even if we were to use spherical coordinates like latitudes and longitudes to map our features, the underlying calculations would treat these as planar. So to correctly calculate 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 introduced in PostGIS 1.5 to store lon lat data. The PostGIS geography data type has similar functions for inserting data and uses the same text representations, except that all coordinates are input and expressed in WGS 84 lon lat, measurements are always in units of meters/square meters, there are fewer functions, and it uses a view called geography_columns that's always in sync with the tables. We'll briefly cover this type in later chapters of this book.

The default SRID in pre-2.0 versions of PostGIS is -1 to represent the unknown SRID. Should you use the unknown SRID? The answer is no if you're 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're using PostGIS for non-geographical purposes, such as modeling a localized architecture plan or demonstrating analytic geometry principles, it's perfectly fine keeping your spatial reference as unknown.

Unknown SRID in OGC

In OGC, the unknown spatial reference system is 0 instead of -1. Future versions of PostGIS after 1.5 will use the more standard 0 to represent the unknown spatial reference system. For most functions that requires SRID (minus ST_Transform), you can leave out the SRID if you wish the spatial reference system to be treated as unknown.

You should also know that even though the spatial_ref_sys table has close to 4,000 entries, you'll encounter plenty of instances where you have to add 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's 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 by 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 won't be revalidated. You'll end up with metadata that's out of sync with the actual data. For this reason, we recommend that you not edit the geometry_columns table directly.

2.1.2 Interacting with the geometry_columns table

To avoid editing the records directly in the metadata tables, PostGIS offers five functions, which, 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.
- *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 as long as PostGIS itself. It doesn't destroy any information already in the geometry_columns table but only adds new valid entries and gives you some stats as to the number it adds. It doesn't inspect views or tables that lack PostGIS constraints, so these kinds of tables and views can't be added with probe_geometry_columns. Nor does it add missing constraints to tables.
- *Populate_Geometry_Columns* (introduced in PostGIS 1.4)—A bit more sophisticated than Probe_Geometry_Columns, it can be used to populate the geometry_columns metadata 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. Therefore, 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, only Populate_Geometry_Columns can be used to register views in the PostGIS geometry_columns table; however, you're still free to register these and other tables manually by directly inserting into geometry_columns.

In all of our examples thus far, we used the AddGeometryColumn function to handle the creation of new geometry columns. Although you don't need to use these functions for creating and maintaining the geometry columns, for pre-2.0 versions of PostGIS we highly recommend doing so because the functions will automatically register and maintain entries in the geometry_columns table. To demonstrate the advantage of using the maintenance functions, we'll add a new geometry column of points

to a table without using the AddGeometryColumn function. We'd need to go through the following steps to achieve the same result:

1. Use 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 permit only SRID of -1.
5. Add an entry to the geometry_columns metatable noting that the new points column is a coordinate two-dimensional point layer with an unknown spatial reference system.

As you can see, using maintenance functions wherever possible greatly simplifies matters and reduces the likelihood of you forgetting a step. Now that you have a general idea of the supporting structures and maintenance functions PostGIS offers for geometries, we'll explore what kinds of geometries PostGIS offers and how to create and add them to your 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'll explore the geometric data types in PostGIS in detail. We'll concentrate on the defining attributes of geometries, addressing questions such as "What makes a linestring a linestring?"

2.2.1 What's 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 allows you to apply the knowledge you learn in working with one spatial database to another so long as they're 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 isn't precise enough. You have to go further and must make distinctions between floating point and fixed numbers. With fixed numbers, you can specify more attributes, like the number of places after the decimal point or whether a column will be signed or unsigned. Like numeric data types, a geometry data type is

akin to a base data type from which you can derive more specific data types. At the root of geometry data types, you find points, linestrings, polygons, and curves, each with its own defining attributes.

Let's delve into some concrete examples. We start by creating a table to store all the geometries to be shown:

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

For now, our table has nothing more than an autonumbered column (also known as a serial column in PostgreSQL parlance) and a column to store the name of the geometries. For the sake of brevity, we're placing our new table in the default public schema. Without prefixing the table name, PostgreSQL follows the schema search path. Unless you've changed the search path order, the default search always starts with $user followed by 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'll find it easier when it comes time to upgrade PostGIS and for performing selective backups and restores.

2.2.2 Points

All PostGIS geometries are based on the Cartesian coordinate system. A point in 2D coordinate space is specified by its X and Y coordinates. In 3D space, a point has X, Y, and Z coordinates; in 2DM space, a point has X, Y, and M coordinates and 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 coordinates. (In OGC nomenclature, this is often represented as a distinct data type called Point MZ, but PostGIS prior to the 2.0 series represents this as data type POINT with four coordinates.)

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 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 variants of such geometries: 2DM and 3DM. 2DM has X ,Y, M and 3DM has X, Y, Z, M. You can use the measure coordinate to store additional information associated with the spatial coordinates. Scientific data often uses the extra variable to hold a measurement taken at the point, hence the term *measure*.

The benefit of using M to store additional information directly becomes clear as soon as 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 add an additional table that divides the linestring into points for the sake of storing the measure data.

Several functions in PostGIS (many of which are defined by the OGC SFS standard) deal specifically with M coordinate data. We'll explore these in a later chapter.

The following call to the AddGeometryColumn function creates a new geometry point column, after which we add two pizza parlors and your 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)
VALUES ('Pizza 2',ST_GeomFromText('POINT(1 -1)'));
```

The code in listing 2.1 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 shown in figure 2.1.

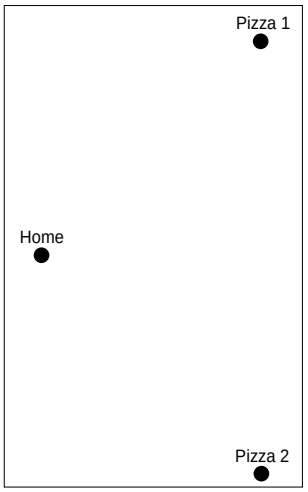


Figure 2.1 Three points created using the code in listing 2.1

2.2.3 Linestrings

Linestrings are defined by at least two distinct points. Like points, there are four 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's use the following listing 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 in listing 2.2 adds a linestring starting at the origin, going to (1,1) and terminating at (1,-1). This is an example of an open linestring where the starting and end points aren't the same. The second INSERT statement adds 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 can see in figure 2.2, are the basis for constructing polygons.

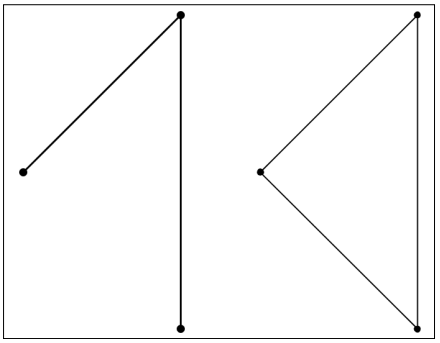


Figure 2.2 Open and closed linestrings created using the code in listing 2.2. The points that make up the lines are shown as well.

The concept of simple and non-simple geometries comes into play when describing linestrings. A simple linestring can't have self-intersections except at the starting and end points. A linestring that crosses itself isn't simple. PostGIS provides a function, ST_IsSimple, that tests to see if a geometry is simple. The following query will return false:

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

The output of the SELECT is shown in figure 2.3.

Another important idea 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 becomes clear with questions relating to the closest point on a linestring to a polygon or other geometry. The closest point rarely coincides with any point used to define the linestring.

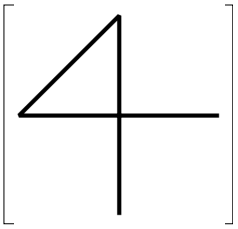


Figure 2.3 A non-simple linestring tested for simplicity

2.2.4 Polygons

Now things get a little more interesting. We process from familiar geometries to form polygons. We start with a simple triangle: Take a closed linestring with at least three distinct points. This linestring takes the shape of a triangle, as shown in figure 2.4. 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, it's the *exterior ring*.

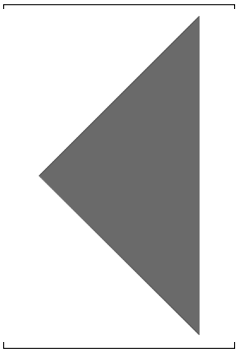


Figure 2.4 Triangle-shaped polygon

```
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))'));
```

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, as shown in figure 2.5. This is why we need the seemingly redundant set of parentheses 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.

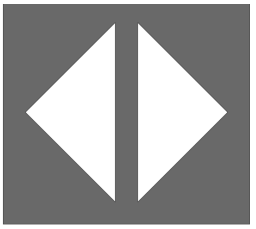


Figure 2.5 Polygon with interior rings (holes)

```
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))'));
```

Always add the extra set of parentheses 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 parentheses without complaining, but not PostGIS.

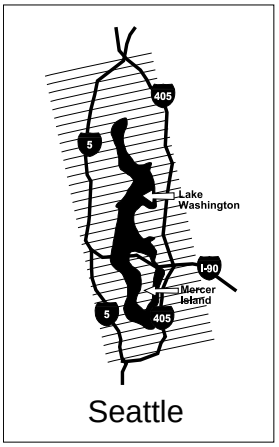


Figure 2.6 We model the Seattle area as a polygon with two rings. Lake Washington fills up the hole. We're also overlooking the existence of Mercer Island in the lake, which would make this a multipolygon.

In the real world, multiringed 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 could start by outlining a big polygon bounded by Interstate 5 on the west and Interstate 405 on the east, as shown in figure 2.6. We could then start to pin down starting and terminal points of popular bus lines and let the computer choose the shortest path within the polygon. Soon enough, we'd realize that most of those popular routes are over water, Lake Washington to be specific. To have the computer pick routes correctly, our polygon of greater Seattle would need an inner ring outlining the shape of Lake Washington. This way, if we were to run a query asking for the shortest path between two points on the polygon and completely within the polygon, we wouldn't end up with buses driving into the water.

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 can't overlap each other and that two rings can't share a common boundary. A polygon whose inner rings partly lie outside its exterior ring is also invalid. Figure 2.7 shows an example of a single polygon with self-intersections. (Visually, you can't discern that it's an invalid geometry because such a visual can be created with two valid polygons or one valid multipolygon that happens to be touching at a point.)

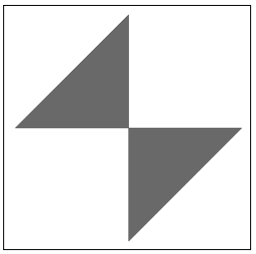


Figure 2.7 Example of a self-intersecting polygon with text representation of `POLYGON((2 0,0 0,1 1,1 -1, 2 0))'`. This is an example of an invalid polygon, but with the naked eye it's impossible to see that it's one invalid polygon and not one valid multipolygon or two valid polygons.

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 don't serve any purpose in real-world modeling.

2.2.5 Collection geometries

To demonstrate the concept of collection geometries, we ask you to mentally picture the 50 states of the United States as polygons. Interior rings allow us to handle states with large bodies of water within their boundaries, such as Utah (the Great Salt Lake), Florida (Lake Okeechobee), and Minnesota with its 10,000-plus lakes. There's at least one state that our polygon has trouble handling: Hawaii. Hawaii comes in at least five big pieces. We could conceivably model Hawaii as five separate polygons, but this complicates our storage. For example, if we wanted to create a table of states, we'd 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. We'd 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 50 states becomes a collection of polygons—a multipolygon.

States as multipolygons

To give you a flavor of real-world GIS, we'll look at state polygon data from the U.S. Census Bureau's TIGER (Topologically Integrated Geographic Encoding and Referencing) data set. In the TIGER data set, only the following states are modeled as multipolygons: Alaska, California, Hawaii, Florida, Kentucky, New York, and Rhode Island.

In reality more states are really multipolygons based on geography alone. Almost all states border large bodies of water and have detached islands. Because the census data is more concerned with people living in the state 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 stretch for a few miles into oceans. These more encompassing boundaries eliminate most states as multipolygons, leaving only the seven as multipolygons.

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 are nothing more than collections of points. Figure 2.8 shows an example of a multipoint.

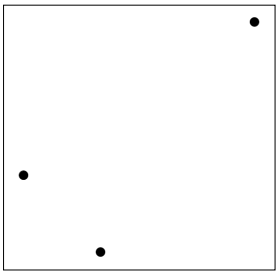


Figure 2.8 A single multipoint geometry—not three distinct points!

In order to represent a multipoint in WKT syntax, you'd use one of the following. If you have only X, Y coordinates for a multipoint, each comma-delimited value would have two coordinates:

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

For a 3DM multipoints, those having X, Y, Z, and M, you'd have four coordinates:

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

For a regular 3D multipoint composed of (X, Y, Z), you'd have the following:

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

For a multipoint where each point is composed of X, Y, M, you'd use MULTIPOINTM to distinguish it from an 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 parentheses to separate each point, for example: `MULTIPOINT ((-1 1), (0 0), (2 3))`. PostGIS will accept this format as input but will output the non-parenthetical version in the ST_AsText and ST_AsEWKT functions.

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

MULTILINESTRINGS

Of no surprise, a multilinestring is a collection of linestrings. Be mindful of the extra sets of parentheses in the WKT representation of a multilinestring that separate each individual linestring in the set. The following examples of multilinestring are shown in figure 2.9.

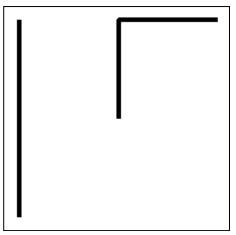


Figure 2.9 Multilinestrings generated with WKT of inline examples

```
MULTILINESTRING((0 0,0 1,1 1),(-1 1,-1 -1))
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))
```

Before moving on to multipolygons, let's return to the concept of simplicity. In section 2.2.3, we tested a linestring for simplicity. Simplicity is relevant for all one-dimensional

linestring type geometries. We consider multilinestrings simple if all constituent linestrings are simple and the collective set of linestrings doesn't intersect each other at any point except boundary points. For example, if we create a multilinestring with two intersecting simple linestrings, the resultant multilinestring isn't simple.

MULTIPOLYGONS

Be careful! The WKT of multipolygons has even more parentheses than its singular counterpart. Because we use parentheses to represent each ring of a polygon, we'll need another set of outer parentheses to represent multipolygons. With multipolygons, we highly recommend that you follow the PostGIS conventions and don't omit any inner parentheses for single-ringed polygons. Following are some examples of multipolygons, the first of which is shown in figure 2.10:

```
MULTIPOLYGON(((2.25 0,1.25 1,1.25 -1,2.25 0)),
 ((1 -1,1 1,0 0,1 -1))) (pictured in figure 2.10)

MULTIPOLYGON(((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)) )
```

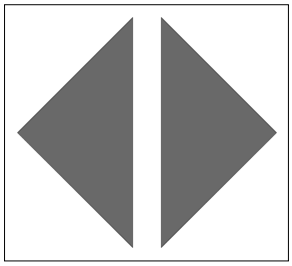


Figure 2.10 Multipolygon generated with WKT of `MULTIPOLYGON(((2.25 0,1.25 1,1.25 -1,2.25 0)), ((1 -1,1 1,0 0,1 -1)))`

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

- Each constituent polygon must be valid in its own right.
- Constituent polygons can't overlap. We'll define overlap more rigorously in chapter 4, but even without that we think you get the picture: Once you lay down a polygon, subsequent polygons can't be put on top and can share boundaries only at finite points.

GEOMETRYCOLLECTION

Geometrycollection is a PostGIS data type that can contain heterogeneous geometries. Unlike multigeometries where the constituent geometries must be of the same type, the geometrycollection data type can include points, linestrings, polygons, and their

collection counterparts. It can even contain other geometrycollections. In short, you can cram every geometry type known to PostGIS into a geometrycollection.

In listing 2.3, 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 functions won't make any sense when used with these data types. For example, you can ask what the area is of a multipolygon, but you can't ask what the area is of a geometrycollection that has linestrings and points in addition to polygons. Geometrycollections generally originate 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. The next listing shows you the result of union queries that generate geometrycollections.

Listing 2.3 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)))

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)))
```

Pictured in figure 2.11

The output of the first geometrycollection of listing 2.3 is shown in figure 2.11. The output of the multim geometry would look the same, except that the M coordinate has no visual representation.

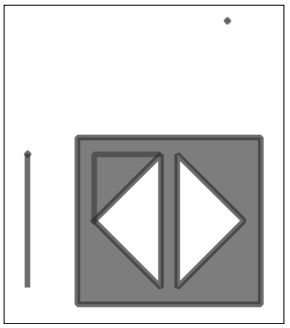


Figure 2.11 geometrycollection formed from code in listing 2.3

In this example we use ST_AsEWKT and ST_GeomFromEWKT to define our geometrycollection 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 aren't designed for anything above 2D, whereas ST_GeomFromEWKT and ST_AsEWKT are PostGIS constructs and 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's 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's 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's defined in the specs.

Curved geometries aren't as mature as other geometries and aren't widely supported. Natural terrestrial features rarely manifest themselves as curved geometries. Manmade structures and boundaries do have curves, but for many modeling cases, these structures can be adequately approximated with lines. Aeronautical charts are full of them because the sweep of radar is circular. Dams, dikes, and breakwaters are other curved macro structures that come to mind. Some highways segments come close to being curves, but linestrings are often more appropriate in modeling them, certainly when processing speed is more important than accuracy. Because of lack of support, we offer the following caveats before you decide to go down the path of using curved geometries:

- Few third-party tools, either open source or commercial, support curved geometries.
- The advanced spatial library called GEOS that PostGIS uses for much of its functionality such as performing intersections, containments checks, and other spatial relation checks doesn't support curved geometries. As a workaround, 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.

- Many native PostGIS functions don't support curved geometries. You can find a full list of functions that do support curved geometries in the PostGIS reference manual. Again, for cases where you need to use functions that don't support curved geometries, you can use the ST_CurveToLine function, apply the function, and then apply the ST_LineToCurve function to convert back if needed.
- Curved geometries have not been supported for as long as the other geometries in PostGIS, so you're 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 support curved geometries.

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

- You can represent a truly curved geometry with fewer points.
- More tools will inaugurate support for curved geometries. The 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.
- PostGIS is increasing support for curved geometries. The complete MM set for curved geometries should be available in PostGIS 2.0. You should also start enjoying complete support of intersections and relation checks in later versions.
- Curved geometries are particularly important when modeling manmade structures such as buildings and bridges, where true curved objects are commonly found.
- 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 a regular polygon using the ST_CurveToLine function.

Let's now take a closer look at the large variety 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* because 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. Figure 2.12 is a diagram of a five-point circularstring.

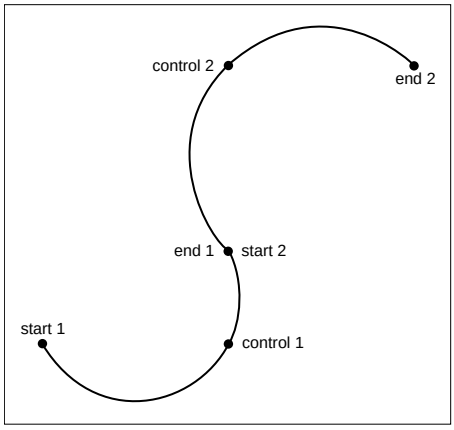


Figure 2.12 A simple five-point circular string, the WKT `CIRCULARSTRING (0 0,2 0, 2 1, 2 3, 4 3)`. The control points are `POINT(2 0)` and `POINT(2 3)`.

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 has been talk of introducing such support in later versions once the basic SQL-MM circularly interpolated curved geometry support is complete. For complete support, PostGIS must be able to instantiate all the different types defined in SQL-MM Part 3 and most of the relation and processing functions that can work with them.

The circularstring is the simplest of all curved geometries and contains only arcs. The following listing contains more examples of circularstrings and how you'd register them in the database.

Listing 2.4 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)'));
```

The output of listing 2.4 is shown in figure 2.13.

You'll discover that not all rendering tools can handle curve geometries. In these instances, the ST_CurveToLine function comes in handy for approximating curved geometries with linestrings. As the following code and table 2.1 demonstrate, to achieve a reasonable degree of curvature, the linestring contains 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;
```

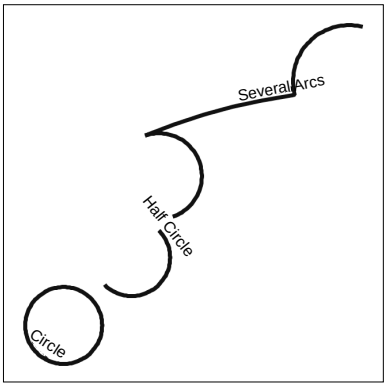


Figure 2.13 Three circular strings generated from the code in listing 2.4

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

Name	cnpoints	lnpoints
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 four circular strings and four straight linestrings. A compoundcurve is a geometry composed of both a circularstring and regular linestring segments, 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). Following is an example of a compoundcurve composed of an arc sandwiched between two linestrings. The output is shown in figure 2.14.

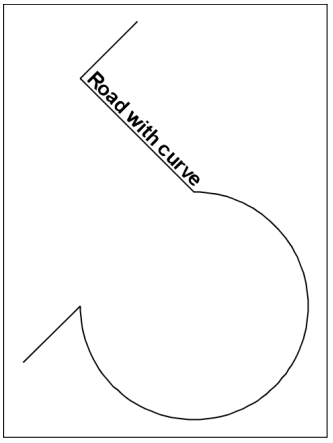
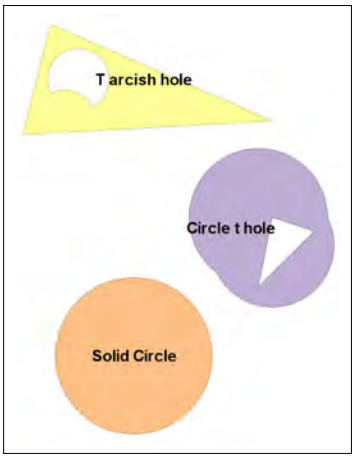


Figure 2.14 A compound curve generated from the previous code

```
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))'));
```



CURVEPOLYGON

A curvepolygon is a polygon that has an exterior or inner ring with circularstrings. In pre-1.4 PostGIS versions, compondcurves can't be used to form rings, even though SQL-MM specs allow for such curvepolygons. The following listing and figure 2.15 show some examples of curvepolygons.

Figure 2.15 Curvepolygons generated from the code in listing 2.5

Listing 2.5 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))'));
```

PostGIS 1.4 introduced support for compoundcurves as rings in a curvepolygon. Following 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))
```

The output of the curve polygon is shown in figure 2.16.

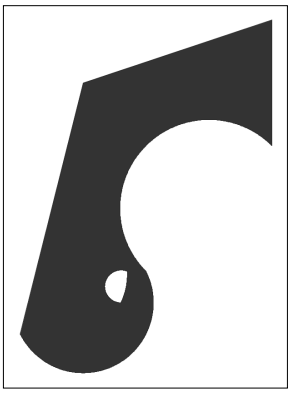


Figure 2.16 Compoundcurve in a curvepolygon with a circularstring hole

2.2.7 3D geometries

PostGIS recognizes and stores 3D geometries, but full support is still spotty. As we've 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're 2D objects sitting in 3D space (as such, they're sometimes referred to as 2.5D). Be mindful of the following caveats when working with 3D geometries:

- PostGIS and the underlying GEOS library have minimal support for 3D geometries. For example, all the relationship operators check only whether 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 because they occupy the same 2D coordinates.
- 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 don't need to be precise, say on a hiking trail, but in general this is unwanted behavior.

- Spatial operators achieve their amazing speed through use of bounding box indexes. Unfortunately, these indexes currently don't extend into the third 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 metatable. Every PostGIS-enabled database has this metatable to catalog information about geometry columns in the database. We then introduced five functions that are commonly used to interact with the geometry_columns metatable, with AddGeometryColumn being the most prominent of these functions. We advised strongly against editing the geometry_columns table directly.

We then moved on to 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 creates collection geometries. We covered multipoints, multilinestrings, multipolygons, and geometrycollections.

We finished the chapter with discussions of curved and 3D geometries. Because these two families of geometries are much more complex and less supported in real-world GIS modeling, they don't 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 to use metatables in PostGIS to centralize data type management. You should have a good understanding of single geometries and the purpose of geometry collections. In the next chapter, we'll apply what you learned in this chapter by exploring the various approaches to organize your data in a PostGIS database.

Finally, we want to remind you not to think too much about rigorous definitions when modeling. If it looks, feels, and smells like a polygon, treat it like a polygon; don't worry about inner rings and outer rings unless you need to. Although PostGIS tries to follow standards and the standards try to follow solid mathematical foundations, you'll invariably run into definitional inconsistencies. Don't be put off by these; instead, focus on what you're trying to accomplish using PostGIS.

3 Organizing spatial data

This chapter covers

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

In the last chapter we did a walkthrough 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'll move on to the various options for designing your database to store these types. As with any database design, there's 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. Unlike databases with 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 your lifetime. It also goes without saying that many factors can't 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 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'll explain how you'd 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'll 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 thing, 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 outnumber those for geography types. Finally, geometry types are inherently faster for most spatial computations. All this may change as geography data types mature. On the other hand, you'll find the general concepts we cover in this section applicable to geography types too.

3.1.1 Heterogeneous geometry columns

This approach doesn't 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're more interested in geographically partitioning the city. For example, Washington, D.C., as well as many other planned cities, is 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 the generic geometry type, the column can store polygons for the many polygonal-shaped government edifices in D.C., linestrings to represent major thoroughfares, and points for metro stations.

There are varying degrees of the heterogeneous approach. Using a generic geometry column doesn't 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 that the spatial reference system and the coordinate dimension of the input geometries be the same.

PROS

Following are the pros of the heterogeneous column approach:

- It allows you to run a single query of several features of interest without giving up the luxury of modeling them with the most appropriate geometry type.
- It's simple. You could conceivably cram all your geometries into one table if their non-spatial attributes are more or less the same.
- Table creation is quick because you can 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 the AddGeometryColumn function. This is particularly handy if you're trying to iteratively load data from a large number of tables and you don't care or know what geometry types each contains.

CONS

And here are the cons of the heterogeneous column approach:

- You run the risk of having someone insert an inappropriate geometry for an object. For example, if you've 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 don't 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 can't 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 a geometry type index or ensure that your queries select only a single type of geometry type from the heterogeneous column.
- For cases where you need only to extract a certain kind of geometry, you'll 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'll need to perform a self-join on this table. Not only are self-joins taxing for the processor, they're also taxing for the mind. Imagine a scenario where you have 100 POIs and two neighborhoods, for a total of 102 records. Determining 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 their own table, you'd only be joining a table of 100 rows with a table of just two rows.

With the disadvantages of heterogeneous storage approach fresh in your mind, let's move on to 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, and so on. This means that each geometry type must reside in its own column at the least, but it's also common to break up different geometry types into entirely separate tables.

If in our D.C. example we care more about the type of the features than the quadrant each feature is located in, we'll employ the homogeneous columns design. One possible table structure would be to define a features table with a name column and three geometry columns. We'd 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'd populate the point column, leaving the other two columns null; if it's linestring data, we'd populate the linestring column only, and so on. We don't 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 homogeneous columns approach.

PROS

The homogeneous geometry columns approach offers the following benefits:

- It 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 supports only one geometry per record, so you'd 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 having large geometries and few records with tables having smaller geometries and 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 a separate table.
- Should you be working with monstrous datasets, separate tables also allow you to reap the benefits from placing your data on separate physical disks for each table by means of 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 a separate physical disk, and distribute your tables across different tablespaces to achieve maximum disk I/O. One common practice is to group rarely used tables into their own tablespace and place them on a slow disk. In PostgreSQL 9.0

(continued)

tablespaces were enhanced to allow you to set random_page_cost and seq_page_cost settings for each tablespace. In older versions you could set these only at the server or database level. The query planner uses these two parameters to determine how to rate query paths based on whether they utilize data running on slower disks or faster disks.

CONS

On the con side, by choosing the homogeneous geometry columns approach, you may face the following obstacles:

- 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 D.C. example involve querying by quadrant only, stick with the heterogeneous approach.
- If you choose the homogenous approach but choose to have multiple geometry columns per table, you may run into performance issues. Multiple geometry columns in a single table means wider, fatter rows. Fatter rows make for slower queries, on both selects and updates.

3.1.3 Table inheritance

The final design approach we offer 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 doesn't 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 doesn't 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, end users can query from the parent table and see all the child data or query from each child table when they need only data from the child tables or child-specific columns. For our D.C. 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 option 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 aren't in an inheritance hierarchy or the query isn't 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're using table inheritance, 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

Here are the benefits of using table inheritance:

- You can 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 didn't exist. This can yield a significant speed boost when you have large numbers of records.
- Inheritance can be set and unset 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 reinherit the child table. This prevents the 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

And here are the disadvantages:

- Table inheritance isn't supported by other major databases. Should you ever need to switch away from PostgreSQL to another, your application code may not be portable. This isn't as big a problem as it may initially appear because most database drivers will 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 don't pass to child tables, though check constraints do. In our D.C. example, if we place a primary key constraint on the parent feature table, dictating that each place name must be unique, we can't expect the child tables to abide by the constraint. Even if we were to assign primary key constraints to the children, we still couldn't guarantee unique results when querying multiple child tables or querying the parent table together with its child tables.
- To maintain hierarchy when adding data, you must take extra steps 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 generally means having to create rules or triggers to insert into a child table when inserting into a parent table or vice versa. We'll 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. Updates to move data from one child to another need to be managed 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'll face an initial performance hit when the query is executed for the 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 the inheritance hierarchy. This should boost performance when querying against inherited tables.
- Listing 3.1 demonstrates how you'd 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'll populate only 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);

CREATE TABLE roads_NE(CONSTRAINT pk PRIMARY KEY (gid))
INHERITS (roads);

ALTER TABLE roads_NE
ADD CONSTRAINT chk CHECK (state
   IN ('MA', 'ME', 'NH', 'VT', 'CT', 'RI'));

CREATE TABLE roads_SW(CONSTRAINT pk PRIMARY KEY (gid))
INHERITS (roads);

ALTER TABLE roads_SW
ADD CONSTRAINT chk CHECK (state IN ('AZ', 'NM', 'NV'));

SELECT gid, road_name, geom FROM roads WHERE state = 'MA';
```

1 Child table

2 Constraints

3 Demonstrate constraint exclusion

In 1, we create a child table to our roads table. 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 the roads_NE table is skipped if the requested state isn't in MA, ME, NH, VT, CT, or RI.

In 3, we write a simple SELECT to pull all roads in Massachusetts. With constraint exclusion, 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've 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 you learned in the previous section by exploring various ways to model a real city. We abandon the quadrants of D.C. and states of the United States 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 20 administrative districts, known as arrondissements. Unlike people in other major cities, Parisians are keenly aware of the presence of administrative districts. It's not unusual for someone to say that they live in the *n*th 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's not

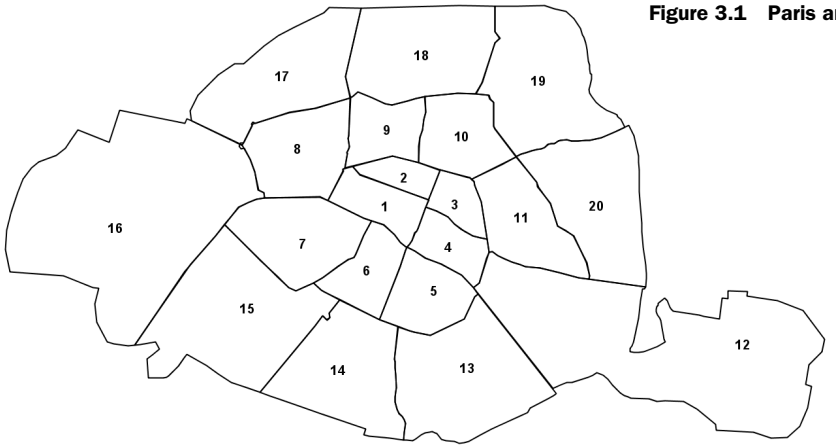


Figure 3.1 Paris arrondissements

often that the exact geographical subdivision of a city has 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 us extra happy.

The basic Paris arrondissements geography is illustrated in figure 3.1.

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 OpenStreetMap (OSM). We transformed all the data to SRID 32631 (WGS 84 UTM Zone 31). All of Paris fits into this UTM zone. Because 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 all of them into a table called arrondissements. The table contains exactly 20 rows. Not only will this table serve as our base layer, but we'll also 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 regard to the type of feature, we can employ a single geometry column to store all of our data. Let's create this table:

```
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 (st_ndims(geom) = 2),
  CONSTRAINT enforce_srid_geom CHECK (st_srid(geom) = 32631)
);
```

Notice how a constraint restricting the type of geometry is 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 two-dimensional geometries and SRID of 32631.

You'll 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 don't 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 PostgreSQL 8.2 and above. To enable this module, 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 also to 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 isn't 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 beforehand means we can query against an integer instead of having to constantly perform spatial intersections later on.

Listing 3.2 demonstrates how to intersect arrondissements with OSM data to yield an ar_num value.

Listing 3.2 Region tagging and clipping data to a specific arrondissement

```
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));
```

1 **Insert data; clip to specific arrondissement**

```
-- repeat for planet_osm_polygon, planet_osm_point
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 **Add indexes and update statistics**

In 1 we load in all the OSM data we downloaded. We listed only the insert from the planet_osm_line table, but you'll need to repeat this for OSM points and OSM polygons. Easier yet, download the code. Features such as long linestrings and large polygons will 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 fifth, sixth, and seventh 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. ❷ We perform our usual index and update statistics after the bulk load.

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

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

This yields the following answer:

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

We should mention that for our example, we extracted from OSM only the area of Paris surrounding the Arc de Triomphe. 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. Figure 3.2 shows a quick map we generated in OpenJUMP by overlaying our OSM dataset atop the arrondissement polygons.

The main advantage of using hstore to hold miscellaneous attribute data is that you don't 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'd like to promote to columns as your needs grow. Using hstore also means that you can add and remove attributes without fussing with the data structure. The drawback is when you do need attributes to be full-fledged columns. You can't query inside an 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

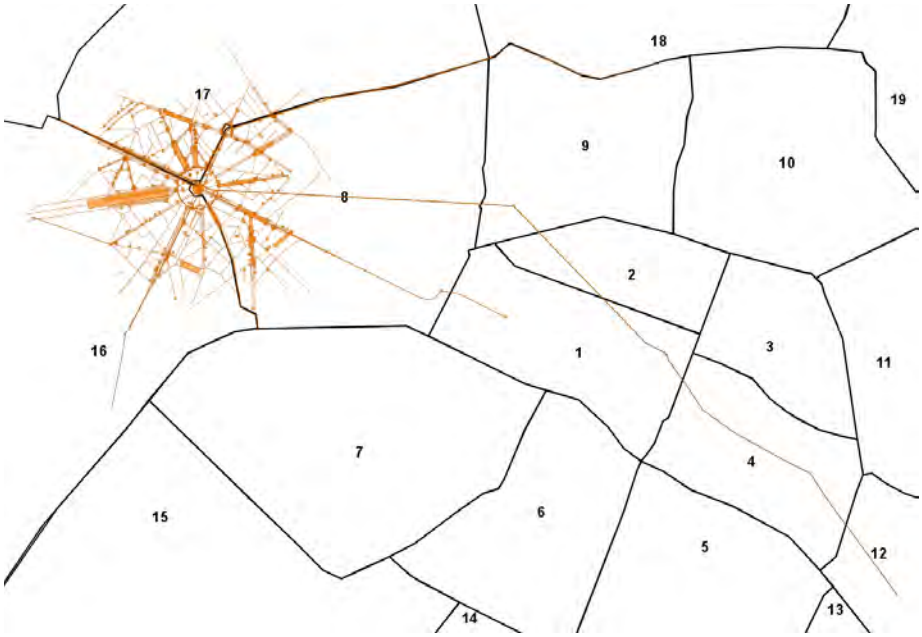


Figure 3.2 Our dataset overlaid on the arrondissements without caring about geometry type

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 an hstore column into virtual data columns, as shown here:

```
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';
```

In this snippet, we create a view that promotes the two tags, name and tourism, into two text data columns. Don't 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 it's 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 on to the homogeneous architecture.

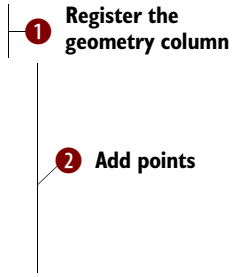
3.2.2 Modeling using homogeneous geometry columns

With the homogeneous columns approach, we want to store each geometry type in its own column or table, as shown in listing 3.3. This style of storage is more common

than the heterogeneous approach. It's 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 3.3 Breaking our data into separate tables with 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));
SELECT AddGeometryColumn('ch03', 'paris_points', 'geom',
   32631,'POINT', 2);
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';
```



We start by creating a table to store point geometry types. ❶ We register the geometry column, effectively constraining the column to store only points. Having a geometry type constraint is the essence of the homogeneous approach. ❷ 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'd 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 shown here:

```
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
UNON ALL
SELECT ar_num, osm_id FROM paris_linestrings
) As X
GROUP BY ar_num;
```

UNION versus UNION ALL

When performing union operations you generally want to use UNION ALL rather than UNION. UNION has an implicit DISTINCT clause built in, which automatically eliminates duplicate rows. If you know that the sets you're unioning can't or need not be deduped in the process, opt for UNION ALL. It will be faster for sure.

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

3.2.3 Modeling using inheritance

Table inheritance is a feature that's fairly unique to PostgreSQL. We gave you a quick overview in section 3.1.3. Now we'll apply it to our Paris example. We begin by creating an abstract parent table to store attributes that all of its children will share, as shown in the following code:

```
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 went to 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 can't 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'll 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'll create only the child table for storing polygons, as shown in the next listing.

Listing 3.4 Creating a child table

```
CREATE TABLE ch03.paris_polygons(tags hstore,
    CONSTRAINT paris_polygons_pk PRIMARY KEY (gid)
)
INHERITS (ch03.paris);

ALTER TABLE ch03.paris_polygons NO INHERIT ch03.paris;
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';
SELECT populate_geometry_columns('ch03.paris_polygons'::regclass);
ALTER TABLE ch03.paris_polygons INHERIT ch03.paris;
```

1 Create an inherited table
2 Disinherit from parent
3 Load
4 Reinherit

In 1 we create a polygon table and declare it as inheriting from our paris table; we need only add the additional columns (in this case tags) beyond what are already defined in the parent. We also add a primary key to the child table because primary

keys don't automatically inherit. In ❷ we disinherit the child from the parent. Disinheriting doesn't remove inherited columns. Once a child table inherits from a parent table, the structure of the parent is passed down permanently. The disinheritance disengages the child from the parent so that queries against the parent don't 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's in the process of being loaded. In ❸, 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 ❹ we reinherit from the parent.

In listing 3.5 we'll repeat the same code for linestrings, but we'll omit the loading of data and the adding of the tag column. Because we aren't populating the table immediately, we constrain 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 3.5 Adding another child and additional constraints

```
CREATE TABLE ch03.paris_linestrings(
    CONSTRAINT paris_linestrings_pk PRIMARY KEY (gid)
) INHERITS (ch03.paris);

ALTER TABLE ch03.paris_linestrings
    ADD CONSTRAINT enforce_geotype_geom
    CHECK (geometrytype(geom) = 'LINESTRING'::text);

SELECT populate_geometry_columns('ch03.paris_linestrings'::regclass);
```

❶ Create child table

❷ Add geometry type constraint

❸ Register geometry column

As we did with polygons ❶ we create a table that inherits from paris to store our linestrings. We aren't ready to load data yet, but we want to constrain the table to just linestrings, so in ❷ we add a geometry type constraint. We don't 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. Because we have a geometry constraint, ❸ 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 care about only the counts there:

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

As you can see, inheritance requires an extra step or two to set up properly, but the advantage is that we're 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 afterthought rather than as part of the initial table design. As an example, say we already set up a paris_points table to store point geometry and have gone to great lengths to populate the table with data. We wouldn't want to drop our points table and re-create it in order for it be a legitimate child of the paris table. In the following listing we demonstrate how to make an existing table a child of paris.

Listing 3.6 Adopt an orphan

```
ALTER TABLE ch03.paris_points DROP COLUMN gid;
ALTER TABLE ch03.paris_points ADD COLUMN gid integer
   PRIMARY KEY NOT NULL DEFAULT nextval
   ➥ ('ch03.paris_gid_seq');
ALTER TABLE ch03.paris_points INHERIT ch03.paris;
CREATE INDEX idx_paris_points_geom
     ON ch03.paris_points USING gist (geom);
```

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. The new parent must not have any columns not found in the child. Though it's not an absolute necessity, it's 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 reassign the gid of our points table to use the family sequence, we drop the gid columns 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. In 3, we make paris_points a child of paris. Finally in 4, we add a spatial index for good measure. If you're 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's try adding tags to the paris table:

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

When we do this, we'll get a notice:

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

This informs us that the child paris_polygons already has this column. And remember how we purposely omitted the tags 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're 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 mechanisms 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 paris, paris_points, paris_linestrings, and paris_polygons. We'll 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 mechanisms will help you decide.

RULES

A rule in PostgreSQL is 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 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 rewritten 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, you're free to use views far beyond a simple SELECT statement. You can have your views manipulate data. You're 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 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 aren't met;
- does something instead of the requested INSERT, UPDATE, or DELETE; or
- 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 won't 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 doesn't satisfy a designated condition. This improves the performance of triggers, because 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 can't simply string together a series of SQL statements. Triggers must be standalone 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 because they're 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 is negligible.

In certain situations only rules can be used. Should you need to bind to a view, only rules will work for versions of PostgreSQL prior to PostgreSQL 9.1.

Triggers on views in PostgreSQL 9.1

In PostgreSQL 9.1, one of the features introduced is the ability to define an *instead of* trigger on a view to more closely follow the ANSI SQL:2003 standard. Example of usage can be found in http://www.depesz.com/index.php/2010/10/16/waiting-for-9-1-triggers-on-views/.

Then there are situations when you can't use a rule and have to implement a trigger. Should you need the logic or specialized functions of procedural languages, these are available only through triggers. Rules are always written in plain SQL. You'd also need a trigger if you needed to execute SQL commands such as CREATE, ALTER, or DROP. You can't run Data Definition Language statements with rules. You also don't 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
- Doing bulk loads
- Binding to views

Use triggers when

- Redirecting inserts from parent tables to child tables
- Preprocessing logic such as converting lon lat to geometry/geographies or geotagging data
- Doing complex validation or with procedural languages
- Needing to run create table or other DDL statements in response to changes 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.

3.3.2 Using rules

Before we provide you with some example uses of rules, you need 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 onto 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 a query broken into subparts is *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 use only DO INSTEAD rules.

In the next listing, we'll create a simple view and then add rules so that we can use the view to perform INSERT, UPDATE, and DELETE operations.

Listing 3.7 Making views updateable with rules

```
CREATE OR REPLACE VIEW ch03.stations
AS
SELECT gid, osm_id, ar_num, feature_name, geom
   FROM ch03.paris_points
WHERE feature_type = 'station';
CREATE OR REPLACE RULE 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);
CREATE OR REPLACE RULE rule_stations_delete AS
   ON DELETE TO ch03.stations
   DO INSTEAD
DELETE FROM ch03.paris_points
WHERE gid = OLD.gid AND feature_type = 'station';
CREATE OR REPLACE RULE 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';
```

1 Read-only view

2 Insertable view

3 Deleteable 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 delete only stations. This ensures that even if we forget a filtering WHERE clause when performing the deletion, the worst we 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.

(continued)

Both NEW and OLD represent exactly one record and take on the column structure of the triggering table. 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 an OLD.

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 counterparts are tables consisting of all the records to be inserted or deleted.

Let's 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 this 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're 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, and 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'd 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'd 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 reuse 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'll now move on to a series of examples showcasing how you can use triggers in a variety of situations to fortify your 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 frontend application.

REDIRECTING INSERTS WITH BEFORE TRIGGERS

For our first trigger example, we'll 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're inserting into the parent table, but we don't 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 don't fit, we toss them into a rejects table created using the following:

```
CREATE TABLE ch03.paris_rejects
(
  gid integer NOT NULL PRIMARY KEY,
  osm_id integer,
  ar_num integer,
  feature_name varchar(200),
  feature_type varchar(50),
  geom geometry, tags hstore);
```

The before insert trigger is shown in the following listing.

Listing 3.8 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
    var_geomtype := geometrytype(NEW.geom);
    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
        FROM (SELECT NEW.*) As foo;
    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
        FROM (SELECT NEW.*) As foo;
    ELSE
        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;
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE 'plpgsql' VOLATILE;
```

1 Using temporary variables

2 NEW is alias for table with new record

3 Nonstandard 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 we 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 to. Normally when you finish 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. **3** But in our case, we want to halt the INSERT into the parent table altogether, so we return NULL instead of NEW. **4** Returning the NEW record is usually used only in a BEFORE trigger, because in an AFTER trigger, there's no hope of being able to change the record being inserted or updated because 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'll often see the use of NEW.* as shorthand to pick up all the columns of the record being inserted. The single-row NEW not only has the same structure, but it also has the exact column order as the triggering table. This often allows you to use the following insert syntax without worrying about listing out each column:

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

For our example we've chosen not to use this syntax. Although this syntax is extremely powerful because you can use it in any trigger function without knowing beforehand what the columns are or will be, it's not without danger.

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

Remember that trigger functions do us no good unless they're bound to a table event. To bind the previous 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's take our new trigger for a test drive. Before we do, we'll 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 types. Because 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 can't 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'll demonstrate with our next example. Suppose, we want to partition our Paris data by arrondissements in addition to geometry type, but we're too lazy to create all 60 arrondissement tables beforehand. 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 doesn't exist, our trigger function will create it.

For the few arrondissements we have or if we know the tables we need beforehand, our dynamic creation in a trigger is overkill. It's 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'd like to break out into weekly tables. If you anticipate your database being in use for 10 years, you'll have to prepare 520 tables at the start. Not only that, on the first day of the eleventh year, your database will fail. The following listing shows a trigger that creates tables as needed.

Listing 3.9 Trigger that dynamically creates tables as needed

```
CREATE OR REPLACE FUNCTION ch03.trigger_paris_child_insert() RETURNS TRIGGER
     AS $$
DECLARE
    var_sql text;
    var_tbl text;
BEGIN
    var_tbl :=  TG_TABLE_NAME || '_ar'                          (1) Assign destination table
      || lpad(NEW.ar_num::text, 2, '0');                            name to variable
    IF NOT EXISTS(SELECT * FROM information_schema.tables       (2) Check if destination
              WHERE table_schema = TG_TABLE_SCHEMA                  table exists
            AND table_name = var_tbl) THEN
        var_sql := 'CREATE TABLE ' … [See Code Listing]         (3) Create destination
        EXECUTE var_sql;                                            table if absent
    END IF;
    var_sql := 'INSERT INTO ' || TG_TABLE_SCHEMA
         || '.' || var_tbl
         || '(gid, osm_id, ar_num, feature_name,
            feature_type, geom, tags)                           (4) Prepare and execute
          VALUES($1, $2, $3, $4, $5, $6, $7)';                      insert SQL
    EXECUTE var_sql USING NEW.gid, NEW.osm_id,
      NEW.ar_num, NEW.feature_name, NEW.feature_type,
        NEW.geom, NEW.tags;
    RETURN NULL;                                                (5) Cancel original
END;                                                                insert
$$ language plpgsql;
```

Before we do anything, we must settle on a naming convention for all of our geometry-arrondissement grandchild tables. We choose paris_points_ar01 through paris_points_ar20, paris_linestrings_ar01 through paris_linestrings_ar20, and paris_polygons_ar01 through 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'd have to further test the geometry type of the new record to figure the destination table. In (2), we check to see if the destination table is present. If not, we create it in (3). By (4), we're 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, and paris_polygons, as follows.

Listing 3.10 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 grandchild tables. If the grandchild is missing, we create it on the fly. To test our new trigger, we delete all the data we've 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, ar_num)
SELECT osm_id, geom, tags, ar_num FROM ch03.paris_hetero;
```

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

Before bringing the discussion of rules and triggers to an end, let's 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;
```

The graphical explain plan output of this query is shown in figure 3.3.

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

3.4 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 the PostgreSQL custom key value hstore data type for

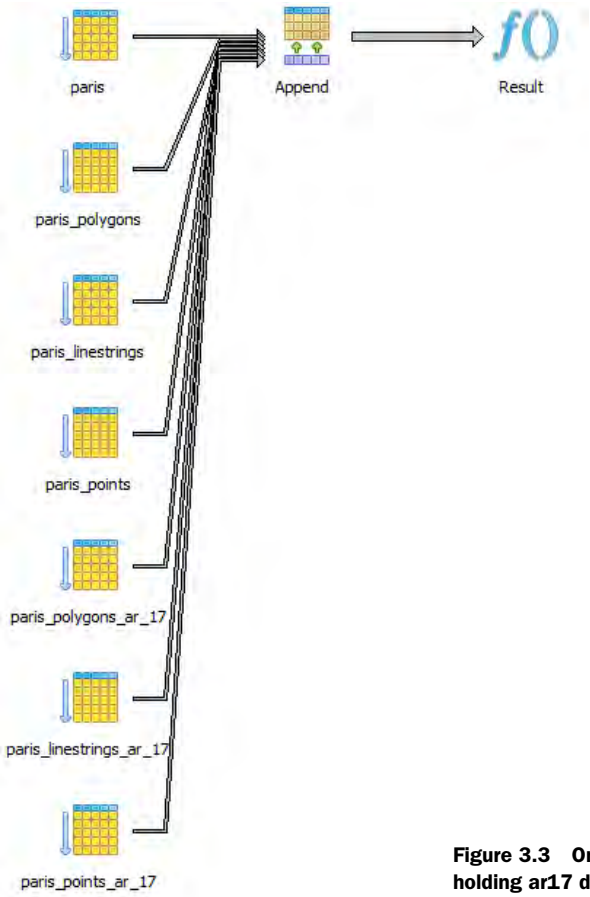


Figure 3.3 Only empty parent tables and child tables holding ar17 data are searched.

implementing schema-less models. We followed up by applying these approaches to storing Parisian data.

We demonstrated in this chapter that PostGIS geometry columns are like other columns you'll find in relational tables. They have indexes and are stored along with other related data such as text columns. 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'll explore in later chapters. In the next chapter and chapters that follow, we'll explore PostGIS functions in more depth. We'll first focus on dealing with single geometries and the more common properties, various functions you can use to inspect and modify geometries. After looking at single geometry functions, we'll focus on functions that involve interplay between two 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'll introduce the core set of functions that work with geometries. This chapter will concentrate on functions that tend to work with single geometries. In the next, we'll work with functions that relate two or more geometries.

PostGIS offers well over 300 functions and operands. To get an overview, we've developed a taxonomy that's driven by intent of use. This is by no means a rigorous classification nor one that will neatly sort each function into a unique classification without ambiguities. Grouping functions by the types of tasks that we're trying to accomplish has been the handiest approach in our experience. Before delving into the functions themselves, let's go through our classification scheme:

- *Constructor functions*—Use these functions to create PostGIS geometries from either a well-known text (WKT) or a 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 extract geometries from an input geometry.
- *Composition functions*—Use these functions to stitch, splice, or group together geometries.
- *Measurement functions*—These functions return scalar measurements of a geometry.
- *Simplification functions*—Sometimes you don't need the full resolution of a geometry. These functions simplify a geometry by removing points or linestrings or by rounding the coordinates. The resultant geometry will still have the basic look and feel of the original but will contain 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'll introduce a dozen functions that are commonly used. You can find an exhaustive listing of all functions and their usage in the official PostGIS manual.

Why ST?

You'll notice that almost all functions 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.

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

We'll start with constructors.

4.1 Constructors

As the name implies, constructor functions create geometries. There are two common ways to create new geometries. The first 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 form new ones. In this section, we start with the first approach. We'll go through the list of common representations of geometric data and the functions used to transform them into bona fide PostGIS geometry objects. Following that we'll introduce some handy functions that create new geometries from existing ones.

4.1.1 Creating geometries from well-known text and well-known binary representations

These indispensable functions will output geometries when you feed them various text or binary representations. They are especially useful for quick viewing of geometries in

various desktop tools. In tools that understand only geometries, the use of these functions becomes almost perfunctory.

ST_GEOMFROMTEXT

Recall from chapter 1 that a common way to represent geometries is through well-known text representations. PostGIS provides a function called ST_GeomFromText that can be used to build 2D geometries. This function is an SQL/MM standard function that can be found in other SQL/MM–compliant spatial databases. It supports only 2D because the SQL/MM–released specs for this function don't support M and Z coordinates. Following are examples of its use:

```
SELECT * INTO table1
FROM ( 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))' ) ) ) As
     foo(geom);
```

ST_GEOMFROMEWKT

PostGIS provides another function called ST_GeomFromEWKT. This is a PostGIS-only function and accepts input from a PostGIS-only format—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 here. Note that EWKT explicitly prepends the SRID of the geometry.

```
SELECT * INTO table2
FROM ( 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))' ) ) ) As
     foo(geom);
```

ST_GeomFromEWKT can accept geometries in plain WKT format as well, so it's often preferred when SQL/MM compliance isn't a concern.

ST_GEOMFROMWKB AND ST_GEOMFROMEWKB

On many occasions, you'll find yourself needing to import data from a client application where geometries are already stored in binary representations. This is where 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 SRID encoding and support for 3D and 4D geometries. These two functions accept byte arrays instead of text strings. One advantage of byte arrays is that they're exact, whereas the ST_GeomFromText and ST_GeomFromEWKT functions truncate at about the fifteenth digit after the decimal point. Following is an example of using ST_GeomFromWKB:

```
SELECT
     ST_GeomFromWKB(E'\\001\\001\\000\\000\\000\\321\\256B\\312O\\304Q\\300\\
     347\\030\\220\\275\\336%E@',4326);
```

Observe that if you were to output the well-known binary of this function,

```
SELECT
ST_AsBinary(ST_GeomFromWKB(E'\\001\\001\\000\\000\\000\\321\\256B\\312O\\304Q
    \\300\\347\\030\\220\\275\\336%E@',4326));
```

it would look like this in pre-PostgreSQL 9.0, but it may look different in newer versions depending on your PostgreSQL bytea_output setting:

```
\001\001\000\000\000\321\256B\312O\304Q\300\347\030\220\275\336%E@
```

The extra slashes we put in when feeding in the value are to escape out the "\" in the string. This is needed only if your database has standard_conforming_strings=off, which is the default for PostgreSQL versions older than PostgreSQL 9.0.

Following is an example if you have standard_conforming_strings=on:

```
set standard_conforming_strings = on;
SELECT
ST_GeomFromWKB('\001\001\000\000\000\321\256B\312O\304Q\300\347\030\220\275\3
    36%E@');
```

Canonical representation

Try doing a simple select statement from a geometry column, unadorned with any functions, and you'll 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, and so on. Our advice, as far as using PostGIS is concerned, is to stay away from them and stick with ST_GeomFromText, ST_Point, ST_MakePoint, ST_GeomFromWKB, and the like. 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, or other type and to nullify it if it isn't. There's no need for such checking if your tables are set up correctly and accept only specific geometry types. These extra functions add unnecessary overhead to your inserts and updates.

4.1.2 Autocasting in PostgreSQL/PostGIS

You'll encounter instances where someone might take a text representation of a geometry and uses it as a parameter to a function. Although this is convenient, you should exercise caution when you do this. Here's a demonstration of such a practice:

```
SELECT ST_Centroid('LINESTRING(1 2,3 4)');
```

To see the output, we do this:

```
SELECT ST_AsText(ST_Centroid('LINESTRING(1 2,3 4)'));
```

which returns this:

```
POINT(2 3)
```

This practice makes you forget that a centroid works on a geometry, not a string. It works because an autocast is 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)'));
```

A problem can arise, however, when you have two functions that take different data types and both data types have an autocast built in. In that case you could end up with an ambiguity error. Here's 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, but after autocasting the text representation to a geometry, PostgreSQL no longer knows whether you intended to pass in a box or a geometry. Here's another example that will fail. ST_XMin is a function defined only for Box3D. This one will fail because there is no autocast that will convert a text representation of a geometry directly into a Box3D, although 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 following error:

```
ERROR:  BOX3D parser - does not start with BOX3D;
```

Bypass the autocasting with the following query:

```
SELECT ST_XMin('LINESTRING(1 2, 3 4)'::geometry::box3d);
```

In the next section we'll 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 requesting 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'll summarize the output formats available, give general use scenarios, and discuss the PostGIS functions to output them. We'll 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 won't go into detail about the various formats.

Finally, we advise that you use good judgment rather than memorize the intricacies of each function when it comes to determining whether the output makes sense for your particular geometry types. 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 and well-known binary

Well-known text is the most common OGC standard format for geometries. We've already used this format quite extensively in the book to show the output of queries because it provides a clear text representation of the underlying geometry.

Two functions that output geometries in this format are ST_AsText and ST_AsEWKT. Recall from earlier discussions that the ST_AsEWKT function is a PostGIS-specific extension loosely based on the OGC-MM WKT standard, but it isn't considered OGC compliant. The OGC-compliant function is ST_AsText, but this function won't output the SRID or the M or Z coordinate. This could change in the future because draft MM standards already propose the addition. Finally, textual representation will always lack the precision of binary representation and will preserve only about 15 significant digits.

Well-known binary is an OGC standard format. Two functions that output geometries in this format are ST_AsBinary and ST_AsEWKB. The ST_AsEWKB function is a PostGIS-specific extension loosely based on the standard, but it's not OGC compliant. ST_AsBinary won't output the SRID or the M or Z coordinate, but ST_AsEWKB will. Unlike text representation, binary format maintains precision. You can be assured that what is stored in your database is what you're outputting, and that what you export can be read back into the database with the inverse functions ST_GeomFromWKB and ST_GeomFromEWKB.

4.2.2 Keyhole Markup Language

Keyhole Markup Language (KML) is an XML-based format created by the Keyhole Corporation to render its applications. KML gained enormous popularity after Google acquired Keyhole and integrated KML into its 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 four 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 lon lat (SRID 4326). As long as your geometry is in a known SRID (via membership in the spatial_ref_sys metatable), ST_AsKML functions will automatically convert it 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 return NUL for unsupported geometry types. Also keep in mind that although the ST_AsKML functions will accept geometries containing an M coordinate, they won't output the M coordinate.

4.2.3 Geography Markup Language

Geography Markup Language (GML) is an XML-based format and an OGC-defined transport format. It's 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, five variants of this function allow you to vary the target GML versions and precisions. Supported GML versions are 2.1.2 (pass in as 2) and 3.1.1 (pass in as 3). If no version parameter is passed in, then 2.1.2 is assumed. Two additional parameters control the number of significant digits and a bit field indicating whether to use short CRS (Coordinate Reference Systems).

ST_AsGML supports 2D and 3D for both geometries and geometry collections. If a geometry has an M coordinate, the M is dropped. Passing in curved geometries will throw an error in PostGIS versions 1.4 and above and return NULL in older versions.

4.2.4 Geometry JavaScript Object Notation

Geometry JavaScript Object Notation (GeoJSON) is a recently developed format based on JavaScript Object Notation (JSON). GeoJSON is geared toward 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 six 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 whether 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 Scalable Vector Graphics

Scalable Vector Graphics (SVG) has been around for a while and is 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's 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 support it, either natively or via an installable plug-in.

The PostGIS function for exporting to SVG is ST_AsSVG. As of PostGIS 1.4, this function outputs only 2D geometries without SRIDs or Z or M coordinates and also doesn't output curved geometries. Three variants of the function indicate whether

the output points are relative to an origin or relative to the coordinate system and indicate the level of precision desired.

4.2.6 Geohash

Geohash is a lossy geocoding system for longitudes and latitudes. It's meant more as a tool for the easy exchange of coordinates than for visual presentation. You can explore its details at http://geohash.org.

PostGIS outputs to Geohash via the ST_Geohash function. Naturally, ST_GeoHash always outputs lon lat (WGS 84) coordinates. 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 their Z and M coordinates. Keep in mind that Geohash is point based, so if you output anything other than points, ST_GeoHash will output only an interpolated point within the bounding box. If you use it to output an area that's too big, it will refuse to proceed.

4.2.7 Examples of output functions

It's now time to present a grand example that brings all the output functions together. We'll be asking our functions to output a line string in SRID 4326 to a precision of five significant digits. (The linestring originates in northern France and terminates in southern England.)

```
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;
```

The results are shown in table 4.1.

Table 4.1 Results of the preceding code

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]]}`
SVG_Absolute	`M 2 -48 L 1 -51`
SVG_Relative	`M 2 4 L -1 -3`
Geohash	`u`

Before moving on to the next section, remember that the output functions we covered export only the geometry fragments necessary to create a fully functional data value in the various formats. Many formats have associated scalar attribute data, but the PostGIS functions will ignore these. For example, KML and JSON often embed scalar data

within JSONed and KMLed wrappers, and these will be lost in translation. In the next section, we'll cover the scalar setter and accessor functions that will be useful for exchanging the non-geometric aspects of geometries.

4.3 Accessor functions: getters and setters

If you're experienced with any object-oriented language, accessor functions come as nothing new. The term comes from OO programming to mean any function that gets or sets intrinsic properties of a geometry. Because quite a large number of functions fall under this classification, we decided to use the term only for functions that return entities that aren't geometries. For example, if we have a square polygon, we'd 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'll call decomposition functions and save for a later section. We also don't consider measurement functions such as those for computing length, area, and perimeter as getters.

A few defining characteristics of geometries are important to know when you're using spatial accessor functions:

- Spatial reference system (SRS) 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, multicurve, and so on.
- 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. (There are many more rigorous definitions, but we stick with something intuitive.) In PostGIS, geometry dimensions can be 0 (points), 1 (linestrings), or 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 every locational application involving measurements, the concept of a spatial reference system and the choice of the appropriate base spatial reference system are of utmost importance. Spatial reference systems allow meaningful measurements and make it possible to share data.

In PostGIS, ST_SRID retrieves the spatial reference system of a geometry. You'll find this OGC SQL/MM standard function in most spatial databases. The companion setter function is ST_SetSRID(), also an SQL/MM standard. This setter function will replace the spatial reference metadata embedded within a geometry. Remember that all geometries must have an SRID, even if it's the unknown SRID (-1). Let's take a look at uses of this accessor in the following listing:

Listing 4.1 Example use of ST_SRID

```
SELECT ST_SRID(
    ST_GeomFromText('POLYGON((1 1, 2 2, 2 0, 1 1))', 4326));
SELECT ST_SRID(geom) As srid, COUNT(*) As number_of_geoms
FROM sometable
GROUP BY ST_SRID(geom);
```

Using ST_SetSRID to change SRID

```
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);
```

If you set up your production tables properly, your geometries should contain only SRIDs found in the spatial_ref_sys metatable. Although nothing in the OGC specification requires SRIDs to have any real-world significance, PostGIS prepopulates the spatial_ref_sys metatable with only the EPSG-approved SRIDs. You're free to invent your own SRIDs and add them to the metatable. People commonly add SRIDs defined by ESRI because PostGIS databases are often used to export, import, or directly service ESRI tools.

4.3.2 Transform to a different spatial reference

No discussion of spatial reference can be complete without introducing the 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 you can take meaningful measurements of the geometry of interest. Following 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 this code snippet is

```
SRID=32618;LINESTRING(668207.88519421 4540683.52927698,
       748464.920715711 4654130.89132385)
```

Now that we've 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 doesn't change the 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 import your geometries as WGS 84 lon lat (SRID 4326), and you later realize they were defined using NAD 27 lon lat coordinates (SRID 4267), ST_SetSRID will correct the mistake.

The ST_Transform function changes the coordinates of each point of a geometry from the geometry's stated SRID to a new SRID using the spatial_ref_sys table to derive

a conversion formula to transform coordinates from the original spatial reference to the target spatial reference and changes the SRID metadata as well. Keep in mind that ST_Transform needs to know the current SRID, because it has to compute mathematically the reprojection of all the points and so needs to read this information from the geometry structure header, whereas ST_SetSRID only needs to know the new SRID, because it will do nothing to the points in the geometry but only write this new SRID value to the geometry's structure header, ignoring whatever was there before. If you started with a wrong SRID (a common mistake), transforming it to another spatial reference system will invariably give the wrong results. The problem is that, in general, you won't get an error message but an empty map, because the coordinates are transformed to a completely different part of the world. The most common beginner's question in GIS, "Why don't I see anything?" is almost always caused by a wrong SRID.

Just because ST_Transform is so versatile, it doesn't 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. You're lucky, though, if you get an error message, because otherwise you're on your own to discover the folly of what you've just done.

Despite its power, ST_Transform isn't all that computationally intensive, but if you have a 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 doesn't 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're keenly aware of the geometry types you're working with, but when importing data containing heterogeneous geometry columns, you'll need the two functions that PostGIS offers to identify geometry types: GeometryType and ST_GeometryType. We've mentioned that functions without the ST prefix in PostGIS are deprecated functions, but in the case of GeometryType versus ST_GeometryType, not only are they different from each other, but both are very much in use.

The GeometryType function is the older function of the two. It's part of the OGC Simple Features for SQL. It returns the geometry types that you're familiar with in all uppercase. 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. The following listing demonstrates the differences between the two and their output.

Listing 4.2 Differences 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);
```

The results are shown in table 4.2.

Table 4.2 Results of code in listing 4.2

new_name	old_name
ST_Polygon	POLYGON
ST_Point	POINT
ST_LineString	LINESTRING
ST_Geometry	GEOMETRYCOLLECTION
ST_CurvePolygon	CURVEPOLYGON
ST_CircularString	CIRCULARSTRING
ST_MultiCurve	MULTICURVE

Determining the geometry type is particularly useful when various functions have to be applied to a heterogeneous geometry column. Remember that some functions accept only certain geometry types or may behave differently for different geometry types. For example, asking for the area of a line is pointless, ditto 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's an example:

```
SELECT CASE WHEN GeometryType(geom) = 'POLYGON' THEN ST_Area(geom)
WHEN GeometryType(geom) = 'LINESTRING' THEN ST_Length(geom) ELSE NULL
END As measure FROM sometable;
```

4.3.4 Coordinate and geometry dimensions

Two kinds of dimensions are relevant when talking about geometries. The coordinate dimension is the dimension of the space that the geometry lives in, and the geometry dimension is the smallest dimensional space that will fully contain the geometry. The coordinate dimension is always greater than or equal to the geometry 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 to a variety of geometries.

Listing 4.3 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);
```

The output of this query is shown in table 4.3.

Table 4.3 Results of the code in listing 4.3

item_name	gdim	cdim
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 table 4.3: A point or a multipoint always has a geometry dimension of 0, a line or multiline always 1, and a polygon or multipolygon always 2.

4.3.5 Geometry validity

We introduced the concept of validity in chapter 2. Pathological geometries such as polygons with self-intersections and polygons with holes 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, and as of PostGIS 1.4, ST_IsValidReason can provide a brief description about why a geometry isn't

valid. ST_IsValidReason will offer up a description for only the first offense encountered, so if your geometry is invalid for multiple reasons, you'll see only the first reason. If a geometry is valid, it will return the string "Valid Geometry".

Enhancements in PostGIS 2.0

Introduced in PostGIS 2.0 is ST_IsValidDetail, which returns a set of valid_detail objects, each containing a reason and location for each validity violation. Also introduced in PostGIS 2.0 is an ST_MakeValid function, which tries to deal with common invalidities and correct them. Both of these functions require compilation with GEOS 3.3.0 or above.

We remind you again that it's important to make sure your geometries are valid. Don't even try to work with geometries unless they're valid. Many of the GEOS-based functions in PostGIS will behave unpredictably on encountering invalid geometries.

4.3.6 Number of points that define a geometry

ST_NPoints is a function that returns the number of points defining a geometry. It works for all geometries. It's a PostGIS creation and so isn't guaranteed to be found in other OGC-compliant spatial databases. Many people make the mistake of using the function ST_NumPoints instead of ST_NPoints. By PostGIS 2.0, these two functions may become interchangeable. Prior to PostGIS 2.0, ST_NumPoints works only when applied to linestrings as dictated by the OGC specification. When used with multilinestrings, only the first linestring in the collection is 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, offer functions that adhere 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. The following listing demonstrates the difference between ST_NPoints and ST_NumPoints.

Listing 4.4 Example of ST_NPoints and ST_NumPoints

```
SELECT type, 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);
```

The results are shown in table 4.4.

Table 4.4 Output results of the code in listing 4.4

type	npoints	numpoints
LinestringM	4	4
Circularstring	3	3
Polygon (Triangle)	4	
Multilinestring	5	3
Collection	5	

Table 4.4 demonstrates that ST_NPoints works for all geometries, whereas ST_NumPoints works only for linestrings and circularstrings. For multilinestrings, ST_NumPoints will count only the vertices in the first linestring.

4.4 Measurement functions

Before taking any measurements in GIS, you must concern yourself with the scale of what you're measuring. This goes back to the fact that you live on a spheroid called earth and that you're measuring something on its surface. When your measurements cover a small area, where the curvature of the earth doesn't come into play, it's perfectly fine to assume a planar model with the earth treated as essentially flat. What distances should be considered *small* depend on the accuracy of the measure you're trying to achieve. We've found that planar measurements are often the first choice, even across very long distances, for example, distances covering an entire continent. People prefer the simplicity and intuitiveness that comes with planar measurement even at the expense of accuracy. Planar measurements generally are in units of meters or feet. Planar models are better supported by GIS tools and are faster to process.

Once distances start to cross continents and oceans, planar measures deteriorate rapidly. You'll 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 backseat because PostGIS supported only planar geometries. With PostGIS 1.5 came the new geography data type. This new data type is always in SRID 4326, and PostGIS functions automatically apply geodetic calculations when using measurement functions against geography data. PostGIS does have dedicated functions that work only on spheroids and can be used with the geometry type. These are used when your application requires you to keep your data in the geometry type, but 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 doesn't make sense. To change a measurement, you have to change the geometry itself.

4.4.1 Planar measures for geometry types

All the planar measurement functions we're about to discuss are in the same units as the spatial reference system that's defined for the geometry. If your spatial reference system is in feet, then the lengths and the areas are square feet. These functions are ST_Length, ST_Length3D, ST_Area, and ST_Perimeter. If your spatial reference system is in degrees of longitude and latitude (spherical coordinates), then your units of measure will be in degrees after PostGIS naïvely maps longitude to X coordinate values and latitude to Y coordinate values. This may only be okay for small areas where earth curvature doesn't matter and you have data with enough significant digits.

For PostGIS 1.5 and below, ST_Length3D is the only one of these measurement functions that considers the Z coordinate. Other measurement functions ignore any Z coordinate in the input instead of throwing an error.

ST_Length and ST_Length3D apply only to linestrings and multilinestrings. ST_Length3D considers the Z coordinate when measuring length, whereas ST_Length ignores the Z coordinate. For PostGIS 1.5 and below, there's no distance function for calculating distance between two points in 3D coordinate space. ST_Length3D is applied in series. A typical workaround is to apply ST_Length3D in series with ST_MakeLine.

> **3D measurement enhancements in PostGIS 2.0**
>
> In PostGIS 2.0 more 3D measurement functions were added: ST_3DClosestPoint, ST_3DDistance, ST_3DIntersects, ST_3DShortestLine, and ST_3DLongestLine. These functions support 3D points, linestrings, polygons, basic collections, and polyhedral surfaces (a new geometry type in PostGIS 2.0).

Following is an example demonstrating the 2D and 3D lengths of a 3D linestring. As demonstrated here, the length returned by ST_Length and ST_Length3D is the same for a linestring in 2D coordinate space:

```
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);
```

The results are shown in table 4.5.

Table 4.5 Result of the preceding code comparing 3D and 2D lengths

Length2D	Length3D
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 only with valid polygons and multipolygons. For multiringed polygons, ST_Perimeter calculates 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.

4.4.2 Geodetic measurement for geometry types

All the measurements we discussed thus far apply to geometries in a Cartesian coordinate systems. Because the earth as a whole isn't flat, a more appropriate coordinate system to use when looking at large parts of the planet is the spherical coordinate system. *Geodetic* is a fancier-sounding term for *spherical*. Spherical coordinates literally throw a curve into our common-sense grasp of lengths, areas, and perimeters. Take the simple question of what is the length between Mumbai and Chicago. The only straight line would pass through the center of the earth. Along the surface of the earth, an infinite number of curved lines connect the two cities. Even if you should always take the shortest curve, there's no guarantee that it will be unique. Try drawing the shortest line between the two geographic poles. You end up not with one but infinitely many.

As of PostGIS 1.4, the only geodetic measurement functions available are ST_Length_Spheroid (also known as ST_Length3D_Spheroid), ST_Distance_Sphere, and ST_Distance_ Spheroid. These functions always return distance in meters. Should you have a Z coordinate value as well to represent elevation, you'll need to make sure the units are in meters. Before using these functions, double-check that your geometries are in some type of degree-based spatial reference system; SRID 4326 is by far the most popularly used.

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 lon 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 defined for geometry Cartesian type were limited. The Length_Spheroid functions of PostGIS 1.4 and below worked only with linestring geometries and multilinestrings, and the ST_Distance_Spheroid and ST_Distance_Sphere functions worked only with points. In 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, whereas Spheroid functions use a named spheroid. If you're using a spheroid, make sure your lon lat are measured along that spheroid model. In later versions of PostGIS it's 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.

When choosing between the geometry and geography type for data storage, you should consider what you'll 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'll 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 need support for only a limited area such as a state, a town, or a small country, then you're better off with the geometry type. If you also do a lot of geometric processing such as unioning geometries, simplifying, line interpolation, and the like, geometry will provide that out of the box, whereas geography has to be cast to geometry, transformed, processed, and cast back to geography.

In listing 4.5, we'll contrast and compare calculating the length of a multilinestring with different spheroids versus calculating the length using a state plane. All linestrings are in Massachusetts. The spheroid calculations from PostGIS 1.5 use the same underlying functions as the geography datatype.

Listing 4.5 Calculating the length of a multilinestring with different spheroids

```
SELECT sp_name, geom_name,                    <-- 1 Geometries
    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))') )

) As g(geom_name, geom)
CROSS JOIN                                    <-- 2 Spheroids
  (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 In this example we compute the lengths of a 2D and a 3D multilinestring first by the spheroid function using both our spheroids. Then we transform our lon lat coordinates to Massachusetts state plane projection and use the regular length 3D function. We repeat the transform exercise but use the U.S. National Atlas projection. 2 The spheroid object is another PostGIS object—the name is arbitrary, but the semi major axis (6378137 for both) and inverse flattening (298…) are relevant. In terms of accuracy, the state plane is the most accurate followed by both spheroids. The U.S. National Atlas is usually accurate within 10 meters (depending on length/distance), but it has the advantage that it covers all of continental United States and can be used in all PostGIS planar operations.

The output of the listing 4.5 is shown in table 4.6. Note that the spheroid (sp3d_length) for 2D geometries is most similar to the Massachusetts state plane. For 3D geometries, the sp3d_length is a bit larger because it takes into consideration the Z coordinate.

Table 4.6 Results of query in listing 4.5

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 of ST_Length_Spheroid functions

They use the Z coordinate (elevation), assumed to be in meters.

They work only with linestrings and multilinestrings.

Units returned are always in meters.

Coordinates of the geometry are always assumed to be lon lat for PostGIS 1.5 and below.

Although there exists no ST_Perimeter_Spheroid function, it's easy enough to simulate one by taking the ST_Boundary of a polygon and then the ST_Length_Spheroid of it. This works only for 2D polygons because ST_Boundary ignores the Z coordinate.

Next we'll look at measurements with geography types in mind.

4.4.3 Measurement with geography type

All measurements based on geography type presume a geodetic model. In addition, all measurements return meters, but all coordinates are stored as WGS 84 lon lat degrees.

Aside from that, the measurement functions you'll find for geography, for the most part, parallel those for geometry. ST_Length, ST_Area, ST_Distance, and ST_DWithin work as 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 the spheroid, but the difference is generally negligible. The measurements don't 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 the 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.

To demonstrate, in the following listing we'll create the same types of objects we had for geometry data types except we're using geography data types, and we'll compare the spheroid against the sphere solutions.

Listing 4.6 Comparing spheroid and sphere calculations in geography

```
SELECT name, ST_Length(geog) As sp3d_lengthspheroid,
         ST_Length(geog, false) As sp3d_lengthsphere
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 shown in table 4.7.

Table 4.7 Results of the query in listing 4.6 demonstrating sphere versus spheroid lengths

geom_name	length_spheroid	length_sphere
2d line	220.337435990337	220.080539442185
3d line	220.337435990337	220.080539442185

As you can see here, 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.

Although the geography type has a fairly complete set of measurement functions, the other functions you'll 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 an ST_Intersects, ST_Intersection, and ST_Buffer. It also has ST_Covers and ST_CoveredBy. The covers family of geography functions in PostGIS 1.5.1 and below support only polygon/point, point/polygon pairs.

4.5 Decomposition

You'll 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 call functions that extract and return one or more geometries *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 question can be answered much faster for the bounding boxes of the geometries than for the geometries themselves. By encasing disparate and complicated geometries in bounding boxes, you 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 two-dimensional box that fully encloses the geometry. (PostGIS also has another kind of box called box3D, but this is rarely used and doesn't serve the same purpose as box2D.) All geometries have boxes, even points! Boxes aren't 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, horizontal lines, or multipoints along a horizontal or vertical. The syntax for a 2D box is

```
BOX(p1,p2)
```

where p1 and p2 are points of any two opposite vertices.

PostGIS functions that create bounding boxes are ST_Box2D. The following listing shows some examples of these in action and the corresponding output.

Listing 4.7 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 (
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 this query are shown in table 4.8. Vertical lines and single points produce degenerate boxes, and the geometry cast produces the same boxes as the geometry itself.

Table 4.8 Results of listing 4.7

name	box	box_casted_as_geometry
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 aren't geometries in their own right. If you need to obtain the geometry of the smallest rectangular box enclosing your geometry, use the ST_Envelope function to return the envelope. In cases where the underlying geometry has no width (such as a vertical linestring), no height (such as a horizontal linestring), or no width and no height (a point), ST_Envelope will simplify the geometry to either

linestrings or points. In the following listing we revisit the previous example, but this time we include the ST_Envelope function.

Listing 4.8 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 query. Observe that for degenerate cases such as a vertical linestring and point, the envelope is the same as the input geometry.

Table 4.9 Results of the code in listing 4.8

name	box	env
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))

Next we'll look at coordinates.

4.5.2 Coordinates

ST_X and ST_Y are a pair of functions that you can use to return the underlying coordinates of points. They're 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.

> **ST_Xm functions are box3D functions**
>
> The ST_Xm functions are defined only for box3D objects, but because there's an autocast in place that converts a geometry to a box3D, you can use it directly on geometries. However, you can't use it on the text representation of geometries, as demonstrated in our discussion on autocasts.

They're rarely used alone but are in general combined with each other to arrive at the pseudo width, height, and so forth of a geometry. We'll demonstrate their use when we talk about translation.

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 doesn't work with geometry collections or curved geometries. The following listing shows some examples of ST_Boundary in action.

Listing 4.9 Examples 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 -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);
```

The output of listing 4.9 is shown in table 4.10 and figure 4.1.

Table 4.10 Output of listing 4.9

name	WKT
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 -1.2 5,-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 4.1 Simple linestring, polygon, and polygon with holes overlaid with their boundaries from the code in listing 4.9

Looking at the query and its output, you can surmise the following behavior of ST_Boundary:

- An open linestring, either simple or 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

A more specialized cousin of ST_Boundary is ST_ExteriorRing. This function accepts only polygons and returns the exterior ring. If you're trying to find the outer boundary of a polygon, ST_ExteriorRing will perform faster than ST_Boundary, but as its name suggests it won't return the inner rings. You can use ST_InteriorRingN to grab individual interior rings.

4.5.4 Point marker for a geometry: centroid, point on surface, and nth point

We've all seen maps where small geometries are reduced to a single point to unclutter the visual representation. Most maps use a star to indicate capital cities rather than the city boundaries. Should you zoom in enough on any online map, for example, 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 won't see any of the details you expect but just a dot telling you that it's a place the government doesn't want you to ever visit.

In PostGIS, ST_Centroid and to a lesser extent ST_PointOnSurface are often used to provide a point marker for polygons. You should think of the centroid of a geometry as the center of gravity as 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 but not curved geometries. For 3D geometries, it ignores the Z coordinate.

ST_Centroid sometimes produces undesirable visual results when the point isn't on the geometry itself. Take the island nation of FSM (Federated States of Micronesia); its ST_Centroid is most likely somewhere in the Pacific Ocean. If you provide a mapping service, you probably don't want people sailing to FSM and failing to end up on dry

land. For this situation ST_PointOnSurface comes to the rescue. It always returns the same point for a given geometry. 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's usually one used to define the geometry. For polygons, it cuts out the M and Z coordinates.

In the following listing, we compare the output of ST_Centroid with that of ST_PointOnSurface for various geometries.

Listing 4.10 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 in listing 4.10 outputs both the centroid and the point on the surface of various geometries. Although the centroid may not always be part of the geometry, the point on the surface is.

Figure 4.2 shows the centroid overlaid with the original geometry that's listed in table 4.11.

Table 4.11 Output of query in listing 4.10

name	centroid	point_on_surface
Multipoint	POINT(0.333333333333333 1.33333333333333)	POINT(-1 1)
Multipoint 3D	POINT(0.333333333333333 1.33333333333333)	POINT(-1 1 1)
Multilinestring	POINT(-0.375 0.375)	POINT(0 1)
Polygon	POINT(1.125 0)	POINT(-0.125 0)

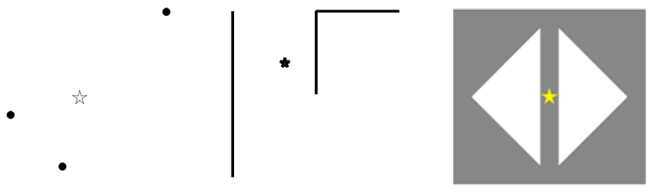


Figure 4.2 Geometries and centroids (denoted by stars) generated from the code in listing 4.10. Observe that the centroid isn't always a point on the geometry.

Figure 4.3 shows the original geometries in listing 4.10 with the point on the surface overlaid, as listed in table 4.11.

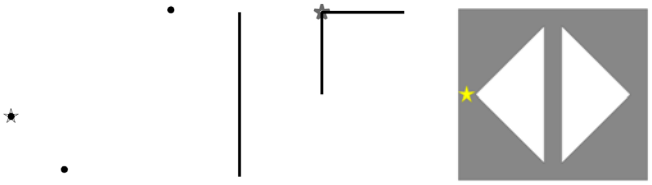


Figure 4.3 Geometries and stars representing the point on the surface generated from code in listing 4.10

ST_Centroid and ST_PointOnSurface in other spatial databases

ST_Centroid and ST_PointOnSurface are both OGC/MM spatial functions, but the specification applied these functions only to surfaces geometries, such as polygons and multipolygons. They can be conveniently extended to other geometry types as many 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. Oracle Spatial supports them only for surface 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 starting at 1. Here's a quick example:

```
SELECT ST_AsText(
   ST_PointN(
     ST_GeomFromText('LINESTRING(1 2, 3 4, 5 8)'),
    2)
);
```

This returns

```
POINT(3 4)
```

Helpful, isn't it?

4.5.5 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 don't quite return the same answer, with the main difference being that ST_Dump recursively dumps all geometries in multi and collection, whereas ST_GeometryN goes down only a single level.

Strictly speaking, ST_Dump returns not a geometry but rows of geometry_dump objects. The 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 at which the extracted geometry was found. The numbering scheme is 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 property. This contains the exploded geometry for that given path. The path is useful if you ever need to reconstitute the original geometry. 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, whereas ST_GeometryN can only explode multicurves, curved geometries, and other standard multi types.

PostGIS 1.5 ST_DumpPoints

PostGIS 1.5 introduced a new function called ST_DumpPoints, which works much like ST_Dump except it recursively dumps out all the points of a geometry collection or non-collection geometry. We have a demonstration of this in chapter 10 of our R example and use it to form a spatial dataframe in R.

Following is a demonstration of 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 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);
```

You can see the results in table 4.12.

Table 4.12 Results of the previous code

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))
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, doesn't recurse, and doesn't 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'll suffer severe performance penalties. The following listing 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. The results are shown in table 4.13.

Listing 4.11 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 4.13 Results of code in listing 4.11

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 used than ST_Dump but is invaluable for breaking up multiringed polygons into smaller polygons. Unlike the 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 alternative is to dump each ring using ST_InteriorRingN and then use ST_BuildArea to form the polygon.

Because 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 that follows, we 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. The results are shown in table 4.14.

```
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 1, 0 0, 1 -1))')) WHERE path[1] IN(0,1);
```

Table 4.14 Results of query in previous code

exterior_ring_polygon	interior_ring1_polygon
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 functions return the rings as linestrings. The results are shown in table 4.15.

```
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 1,0 0,1 -1))') As geom;
```

Table 4.15 Result of query in previous code

exterior_ring	interior_ring1
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 you know how to take geometries apart, you need to know how to put geometries together. We'll move on to composition functions in the next section.

4.6 Composition

We already covered how to create geometries from non-geometry data, either text or binary. In this section, we'll show you how to put together geometries from other geometries.

4.6.1 Making points

Points are the most elementary geometries. Points can be created from X-Y coordinates with two functions: ST_Point and ST_MakePoint. Coordinates aren't geometries, but we feel they're 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. Because these functions don't take SRID as an argument, you need to combine them with ST_SetSRID to denote a spatial reference system.

You may ask yourself what these two additional functions offer beyond the common ST_GeomFromText besides a different import format. To put it concisely: speed and precision. Creating a handful or even a few hundred points doesn't take much time, but loading files with millions of point data with many significant digits (a common task when working with data collected via instrumentation) is a different matter, and you'll certainly come to prefer ST_Point or ST_MakePoint over ST_GeomFromText. To illustrate these two functions, in listing 4.12 we'll simulate reading data points from tracking devices attached to gray 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 an important consideration for import.

Listing 4.12 Point constructor functions

```
SELECT whale, ST_AsEWKT(spot) As spot
FROM
(VALUES
('Mr. Whale', ST_SetSRID(ST_Point(-100.499, 28.7015), 4326)),
 ('Mr. Whale with M as time',
  ST_SetSRID(ST_MakePointM(-100.499, 28.7015, 5), 4326)),
 ('Mr. Whale with Z as depth',
  ST_SetSRID(ST_MakePoint(-100.499, 28.7015, 0.5), 4326)),
 ('Mr. Whale with M and Z',
  ST_SetSRID(ST_MakePoint(-100.499, 28.7015, 0.5, 5), 4326))
) As foo(whale, spot);
```

❶ Whale with time

❷ Whale with depth

❸ Whale with time and depth

This code demonstrates various overloads to the ST_Point and ST_MakePoint functions. In ❶, we employ an extra unit M to store time as a serial. For example, if we take readings every 5 hours, then M=1 would mean this reading was taken 5 hours from the start time, M=2, 10 hours, and so on. If you're keeping data as individual points, this isn't terribly useful, but if you later decide to stitch them together into a LINESTRINGM, then the time slots are encoded in the line and there's only one record for each whale instead of a separate array for the timings. ❷ We may be interested in knowing how far Mr. Whale dove before coming to surface for air, so we use the Z coordinate to store the depth. SRID 4326 is unprojected data, and ST_Transform currently returns the Z coordinate unchanged. ❸ We include both Z and M. M is an additional measurement that you can use to store anything and can mean time or distance from the starting point.

The output of listing 4.12 is shown in table 4.16. Note that all except for the whale with M use POINT but have varying number of coordinates.

Table 4.16 Output from query in listing 4.12

whale	spot
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)

Next, we'll make polygons.

4.6.2 Making polygons

ST_MakePolygon, ST_BuildArea, and ST_Polygonize all build polygons.

ST_MAKEPOLYGON

ST_MakePolygon builds a polygon from a closed linestring representing the exterior ring. Optionally, it can accept as a second argument an array of closed linestrings for interior rings. ST_MakePolygon doesn't validate the input linestrings in any way. This means that if you aren't careful, and you pass in open linestrings or linestrings that can't form polygons, you could end up with an error or fairly goofy polygon, such as polygons with holes outside the exterior ring or interior rings 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 won't ignore Z and M coordinates. ST_MakePolygon accepts only closed linestrings as input—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 organize what you've offered into valid polygons.

ST_BuildArea will accept linestrings, multilinestrings, polygons, multipolygons, and geometrycollections. You don't have to worry about the order or the validity of the geometries that you feed into ST_BuildArea. It will check the validity of each input geometry, determine which geometries should be interior rings and which one should be the exterior ring, and finally reshuffle them to output polygons or multipolygons. ST_BuildArea won't work with arrays. But this shortcoming is mitigated by the fact that it will accept multilinestrings and geometrycollection geometries. If you intend to feed the function an assortment of linestrings and polygons, perform an ST_Collect first to gather all the loose pieces into a single geometry.

All this neatness comes at a price: You sacrifice performance. If you've 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 makes sense only against an 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's 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 next listing.

Listing 4.13 ST_Polygonize , ST_BuildArea, ST_MakePolygon

```
SELECT geom
INTO example
FROM (
(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))'))) ) As e(geom);

SELECT 'ST_MakePolygon (1)' As function,
  ST_AsEWKT(
  ST_MakePolygon(geom)) As polygon
FROM example
WHERE ST_GeometryType(geom) = 'ST_LineString'
UNION ALL
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'
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

FROM example;
```

❶ Make example table

❷ Polygon with no holes

❸ Polygons with holes

First we create the example table. ❶ ST_MakePolygon has two variants. ❷ The simpler version takes an outer ring and forms a polygon without holes. In the second version ❸, we're using the ST_MakePolygon (outer ring, array of inner rings) to form a polygon with holes. We're 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 because there are only two linestrings in our multilinestring example and the first is reserved for the exterior ring. We then use this to extract the second linestring. (If there were more linestrings or more multilinestrings, then the generate series could be 2 to 3 or 2 to 4 and so on.) We then use the array[..] constructor in PostgreSQL, which can take a list of elements, or an SQL statement to populate the array (in our case we're 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. The output is shown in table 4.17.

Table 4.17 Results of query in listing 4.13

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 listing 4.13, 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's incapable of creating polygons with holes, so every ring in the multilinestring becomes a polygon in its own right.

4.6.3 Promoting single to multi geometries

The ST_Multi function is used quite often in PostGIS, mostly to promote points, linestrings, and polygons to their multi counterparts even if they have only a 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 because it's a tiny, landlocked enclave, whereas Indonesia will come in as a multipolygon. To keep your column consistent, you'd promote Lesotho to a multipolygon.

In the next section, we'll cover how to simplify our geometries.

4.7 Simplification

For this section we'll cover the three functions ST_SnapToGrid, ST_Simplify, and ST_SimplifyPreserveTopology. These 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 become important when passing geometries across the internet. Despite recent advances, bandwidth is still a precious commodity, especially with wireless devices. With a tiny, black and white, 200 x 300 resolution GPS screen, transmitting geometries with thousands of vertices or coordinates with a monstrous number of significant digits is certainly overkill.

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 the number of vertices.

There are four variants of this function. The most common one takes one argument for tolerance and rounds the X and Y coordinates while leaving Z and M intact. ST_SnapToGrid doesn't 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 those extra floating-point decimals introduced by ST_Transform. Those extra digits can degrade performance and are generally a nuisance if the precision isn't needed. Another use of ST_SnapToGrid is to group distinct nearby points into a single representational point. For example, if you obtained point data for every school in the country but care only about the location of school districts, then collapsing all the schools down to a single point would be the way to go, especially with data on a national scale.

As with most simplifying operations, you should exercise restraint. Too ambitious rounding can inadvertently turn a valid polygon into an invalid one.

```
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;
```

You can see the results in table 4.18.

Table 4.18 Results of the query in the previous code

tolerance	simplified_geometry
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 of 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 of the geometry, using some variant of the Douglas-Peucker algorithm. The ST_SimplifyPreserveTopology function is newer than ST_Simplify and has safeguards against oversimplification. In extreme cases of oversimplification, the geometry could very well vanish, as shown previously, or become invalid. ST_SimplifyPreserveTopology is generally preferred over the older ST_Simplify even though it's a bit slower.

Both ST_Simplify and ST_SimplifyPreserveTopology take a second argument, which we'll term *tolerance*. This can be roughly treated as the unit of length between the vertices at which you'd 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'll experience more simplification. Putting it another way, the more tolerant you are of losing vertices, the more simplification you can achieve.

These two simplifying functions, unlike ST_SnapToGrid, don't preserve M and Z coordinates and will even remove them if present. They also work only for linestrings, multilinestrings, polygons, multipolygons, and geometry collections containing these geometries. For multipoints they return the same input geometry without any simplification. The reason for this is that ST_Simplify and ST_SimplifyPreserveTopology require edges (lines between vertices) to achieve simplification. Multipoints don't have edges.

> **Don't call ST_Simplify functions with lon lat data**
>
> ST_Simplify and ST_SimplifyPreserveTopology assume planar coordinates. Should you use these functions with lon lat data (SRID 4326), the resultant geometry can range from slightly askew to completely goofy. First transform your lon lat to a planar coordinate, apply ST_Simplify, and then transform back to lon lat.

The following code compares the two functions:

```
SELECT pow(2, n) as tolerance,
  ST_AsText(ST_Simplify(geom, pow(2, n))) As 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 4.19 shows the results of our comparison.

Table 4.19 Results of query in previous code (split into two sections for readability)

tolerance	ST_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
tolerance	**ST_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))

Notice that once you reach a tolerance of 16 with ST_Simplify, the geometry vanishes. But ST_SimplifyPreserveTopology reduces the eight-sided polygon to a four-sided polygon and stops there regardless of the tolerance. Figure 4.4 demonstrates the difference between ST_Simplify and ST_SimplifyPreserveToplogy for the eight-sided version.

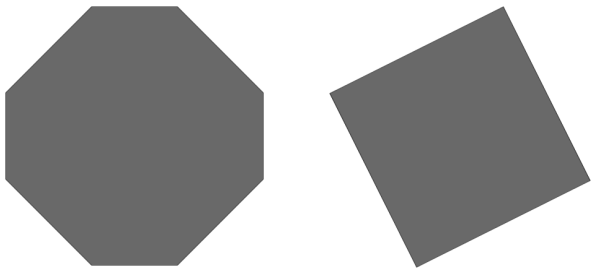


Figure 4.4 ST_Simplify and ST_SimplifyPreserveTopology, going from an eight-sided polygon to a four-sided polygon

This simplification stops there, regardless of how high you raise the tolerance.

4.8 Summary

In this chapter we've started to cover the most commonly used functions in PostGIS. For the moment we concentrated on functions with a single geometry as argument. We developed a loose classification scheme to organize the myriad of unary functions in PostGIS. Starting with constructors, we then moved on to getters and setters, followed by decomposition and composition functions. We ended the chapter with simplification functions. These popular functions constitute but a small subset of all the unary functions available in PostGIS. We highly recommend you to peruse the

official PostGIS documentation to see all that are available. You may find the number of functions overwhelming at first, but on closer examination, you'll find that many functions are closely related and fit nicely into our taxonomy. We also advise you to refer to the documentation before using any of the functions we described.

In the next chapter we'll continue our exploration of PostGIS functions by covering functions that take two or more geometries as input: binary functions. You'll find binary functions to be far more useful, and perhaps more interesting, for answering questions regarding your data, but don't disdain the unaries. Geometries for binary functions almost always have to be prepared by some type of unary function.

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 for geometries and various functions to measure, morph, or transform single geometries. From this chapter forward, we'll 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. 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 gets involved, and 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 aren't limited to geographic systems. In this chapter we'll 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 a potential use case for a spatial database. For these exercises we'll be using the unknown spatial reference system. Later on we'll delve into spatial reference systems when loading geographic data and cover 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 interacts through various mathematical or string operations. When one number meets another number, you can add, subtract, multiply, divide, or do some combination thereof. When one string meets another, you can concatenate, or "substring," one against the other. In spatial databases, when one geometry meets another, things heat up quite a bit. PostGIS has many ways in which the relationship can be consummated. This chapter explores the most commonly used of these relationships. We'll 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 entails 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

Functionally speaking

To avoid muddling meaning while speaking about functions, we suggest that instead of saying `ST_SomeRelationship(A,B)` say `A SomeRelationship B`. For example,

```
ST_Contains(geom1,geom2)
```

would be read as

```
geom1 contains geom2
```

you want to see if one geometry encloses another, you could use ST_Contains. If you want to see if two geometries rub up against each other, you could use ST_Touches.

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 exactly the opposite answer. The exception is invalid geometries, which often return false regardless of the spatial relationship.

When using spatial relationship functions, the two geometries being compared must both be in either the same spatial reference system or in the unknown spatial reference system. If they aren't, the function may return an error. Keep in mind that all spatial relationship functions for the geometry data type presuppose a planar projection (Cartesian coordinates), so when using lon lat data (spherical coordinates), use the geography type instead, especially when comparing large areas. The geography data type will model the spatial relationships using a true geodetic model, but it's not as rich in the number of spatial relationship functions you can use with it. For small areas, the Cartesian spatial relationship assumptions will generally work fine, but as the degree differences increase or you approach the poles, treating lon lat as flat is no longer correct, and you may end up with incorrect results.

Relationship and output functions support for curved and 3D geometries

Although 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 and fourth dimensions, whereas the GEOS relationship functions reject curved geometries.

However, the geometry relationship output functions like ST_Intersection, ST_Difference, and ST_SymDifference don't completely ignore the Z coordinate, but they apply the Z coordinate after doing a 2D relationship process. The results are sometimes less than desirable.

As a workaround 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(b.the_geom, 100))`, where 100 is the number of segments to approximate a quarter circle; the default is 32. The 3D issue is harder to compensate for, but PostGIS 2.0 offers new relationship functions specifically designed for 3D.

We start with intersections, because this is by far the most commonly used relationship between two geometries.

5.2 Intersections

The idea of intersection encompasses a wide range of ways in which geometries can interact. We'll delve into the nuances in time, but let's start with the basic definition of *intersection*: Two geometries intersect when they share space.

PostGIS has two functions that work with intersections. The first is ST_Intersects. It 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 don't intersect, then the intersection is an empty geometry. Both functions are defined in the OGC/SQL-MM specs and so can be found in most databases that follow the ISO SQL-MM model.

What's an empty geometry?

An empty geometry is a geometry with no points, but it isn't NULL. You can create an empty geometry by this command: `ST_GeomFromText('GEOMETRYCOLLECTION EMPTY');`. Although you can have an empty polygon, PostGIS silently converts it to an empty geometry collection. In version PostGIS 2.0, an empty polygon will return an empty polygon instead of an empty geometry collection.

We'll 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 linestring 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 whether 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 show only a simple example, you can imagine how useful this can be if you have all the private properties in a city and want to 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.

Figure 5.1 shows a planned roadway (linestring), our land mass (polygon), and the resulting intersection geometry—the portion we need to take over by eminent domain.

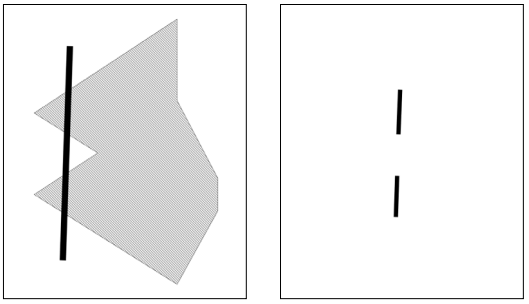


Figure 5.1 The first image is the POLYGON (our property) overlaid with the LINESTRING (our planned road), and the second is the intersection of the two. This results in a MULTILINESTRING that represents the portion of the property we need to take over.

The code to generate these images is as follows:

```
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;
```

In this code we end up with a MULTILINESTRING, as shown in table 5.1, because the line is cut by the polygon.

Table 5.1 Result of query in previous code

they_intersect	intersect_geom_type
t	MULTILINESTRING

Should you be unimpressed by the previous example, the next one ought to change your mind.

5.2.2 Clipping polygons with polygons

One of the most common uses of the ST_Intersection function is to clip 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 had a dozen sales representatives on your staff, you could clip the polygon of the city into 12 sales regions, one for each representative. 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, you can clip the original geometry so as to query against a smaller geometry. For example, if you're working with data covering the entire island of Hispaniola but only need to report on Haiti, you could clip the island using a linestring and so 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 section 5.2.1) into square regions, as shown in figure 5.2.

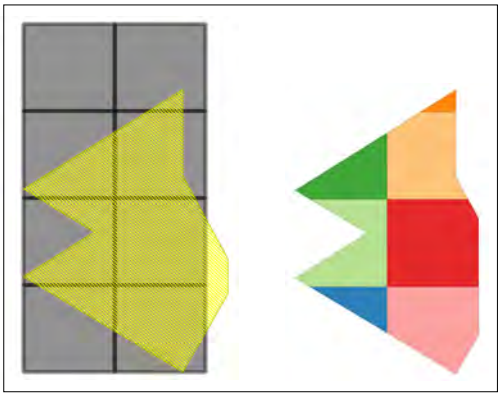


Figure 5.2 Result of code in listing 5.1. The first image shows a region overlaid against square tiles. The second shows the result of intersection of square tiles with the region.

In order to perform this trick we take a rectangle, break it into eight equal-size cells, and then intersect with our starting polygon, as shown in the following listing.

Listing 5.1 Return our sales region diced up

```
SELECT x || ' ' || y As grid_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;

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;

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 Squares to dice

2 Region

3 Dicing yields multiple records

We 1 use the PostgreSQL generate_series to generate two series from min/max x and min to max y and skip every two steps so each square is two units wide and high. 2 This represents the region we want to cut. 3 We combine the two and take the intersection, which results in our geometry being diced.

The well-known text representation of each slice is shown in table 5.2.

Table 5.2 The result of the last query in listing 5.1

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))

This example shows how intersections can be useful for partitioning a single geometry into separate records. Notice that the cutting squares don't need to completely cover the polygon. In this case 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's guaranteed to be of equal or lower dimension than the lowest dimension geometry. For example, if you have two polygons that share an edge, then the intersection of the two will be the linestring representing the shared edge. Similarly, if you intersect a road with a parcel of land, then the intersection would be possibly a linestring that represents the portion of the road that runs through the parcel of land.

To summarize, if A and B are the input geometries to ST_Intersection, then the following are true:

1. ST_Intersection returns the portion shared by A and B.
2. ST_Intersection and ST_Intersects are both commutative—meaning that 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 dimensioned of both input geometries.
5. If A and B don't intersect, then the intersection is an empty geometry.

Now that we've 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, you have at your disposal many PostGIS functions that focus on the properties of the intersection. Again, these functions rely on the fact that the spatial reference system (SRS) of both geometries is the same, though their geometric dimensions need not be the same. The geometries should also be valid; otherwise, the results can't 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 the interior, exterior, and boundary of the second geometry. In the case of intersection of these three parts, the geometric dimension of the resulting geometry is also important. Will the intersection result in a geometry of zero, one, or two dimensions?

We'll cover these concepts in more detail in a later section of this chapter. For brevity, this is what the terms mean:

- *Interior*—That portion of a geometry that's inside the geometry and not on the boundary.
- *Exterior*—The coordinate space outside a geometry but not including the boundary.

- *Boundary*—The coordinate space neither interior nor exterior to the geometry. It's the space that separates the interior from the exterior (the rest of the coordinate space). Recall that we covered in the prior section ST_Boundary, which tells the boundary geometry 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 the coordinate space that's 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. 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 nine pairs can be non-dimensional (no intersection), zero dimensional (finite points), one dimensional (lineal), two dimensional (areal polygons), or a combination thereof in which the dimension is the dimension of the highest dimension element in the collection.

These intersect classes of functions should be used only with valid geometries. The reason 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.

The official PostGIS manual has diagrams of these relationships. We'll try to focus on the corner cases where people have a hard time comprehending the relationships.

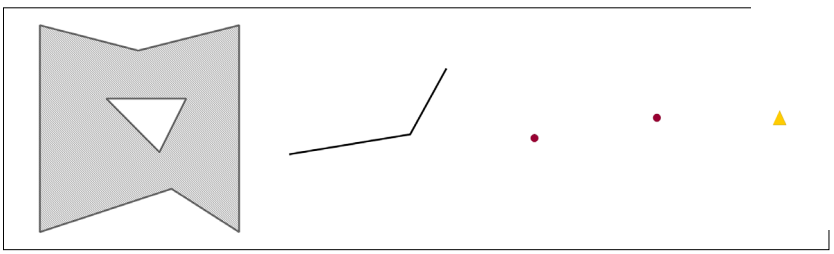


Figure 5.3 A shaded polygon with a hole (a house with a courtyard), a linestring (a walkway), a multipoint represented as two dots (two greeters: front door and courtyard greeter), and a point as a triangle (a door) generated from the code in listing 5.2

We'll use the house example shown in figure 5.3 for explanation of specific intersection relationships because it's a simple corner case that exercises the subtleties of all these relationships.

When these geometries are viewed together on the same grid, the overlay looks like figure 5.4.

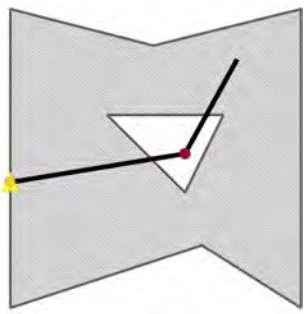


Figure 5.4 The polygon with a hole (a house with a courtyard), the linestring (a walkway), the multipoint (two greeters), and the point as a triangle (a door) seen together. All are generated from the code in listing 5.2.

In this example, we have four geometries that 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 part of the house. The courtyard is thus geometrically a part of the house's exterior.
- A linestring—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 dots—This can be two greeters who are stationed at the front door to greet incoming visitors and at the courtyard to seat guests.

At a particular moment in time, we see all these objects at these particular positions.

We create this particular dataset with the following piece of code.

Listing 5.2 Create sample geometries to exercise intersect relationships

```
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)')) ;
```

Next we'll look at Contains and Within.

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 they can be found in other spatial databases. They have more or less the same meaning 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't be B. In other words, B can't sit entirely on the boundary of A. A geometry doesn't contain its boundary, but a geometry always contains itself.

With the following 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?

```
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 this code is shown in table 5.3.

Table 5.3 Result of the query: 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
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 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 because they as a whole would not intersect the house if they were both in the courtyard).

- Because 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 that's in the courtyard or that sticks out of the house.
- If geometry B sits wholly on the boundary of geometry A, geometry A doesn't contain geometry B. Because the point (door) intersects the house but the house doesn't contain the door, we know the door must be on the boundary of the house. The door intersects the linestring (walkway) but the walkway doesn't 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.
- All geometries contain themselves (a multipoint (the greeters) contains itself, a linestring (walkway) contains itself, and so on).

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 CoveredBy

As you've 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's an often-desired feature. This is why PostGIS introduced the concept of Covers and CoveredBy to satisfy this need that interestingly enough also exist 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. Here are a couple of 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 won't have these functions available.
- ST_Covers is exactly like ST_Contains except it will also return true in the case where a geometry lies completely in the boundary of the other. ST_CoveredBy is to ST_Covers as ST_Contains is to ST_Within.

The following listing and table 5.4 demonstrate situations where ST_Covers covers a geometry but does not contain it.

Listing 5.3 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 this code we're going to list only those geometries where the ST_Covers answer is different from the ST_Contains answer. Table 5.4 shows the result of this query.

Table 5.4 Result of query in listing 5.3: List all where the answer of Covers is different from that of 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

Because we limited our result to the case where the answer produced by ST_Covers is different from that of ST_Contains, 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 don't contain the door.

5.3.4 ContainsProperly

ContainsProperly is a concept that's more stringent than the Contains or Covers relationships. Its main benefit is that it's 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, it's 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.

ContainsProperly will give you the same result as Contains except in the case where any part of geometry B sits on the boundary of A. In listing 5.4 we repeat the same exercise, except that we list only the ContainsProperly options where ST_ContainsProperly gives a different answer from ST_Contains. Here are a couple of caveats to consider:

- It's not an OGC/SQL-MM–defined function; it's a PostGIS-specific function.
- It was introduced in PostGIS 1.4.
- It requires GEOS 3.1 or above, so if you're running PostGIS 1.4 with, say, GEOS 3.0.3, this function won't be available to you.

Listing 5.4 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));
```

Table 5.5 shows the result of this query. Observe that ST_ContainsProperly gives an identical answer to ST_Contains except in the case of an areal geometry, 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 consist of

a finite number of points and therefore have no boundary to speak of. A point can never be sitting partly on its nonexistent boundary.

Table 5.5 Result of query in listing 5.4: geometries where Contains and ContainsProperly 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

You can see from this that the only cases where the ST_ContainsProperly answer is different from that of ST_Contains are the cases of a polygon against itself, a linestring against itself, or 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 because the walkway doesn't properly contain both people, but it does contain both people.

5.3.5 Overlapping geometries

Two geometries overlap when they're the same geometric dimension (points, areal, linestring), they intersect, and one is not 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 used previously, comparing if each one overlaps the other, we'd find that none overlap. The reasons for that are as follows:

- The linestring (walkway) as we've modeled it is not of the same dimension as the polygon (house), so it can't overlap, and the same holds true with point/polygon (door/house) and 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, and 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 don't lie in the interior of both geometries. The function that supports this relationship is ST_Touches, and it's an OGC/SQL-MM–defined function. Revisiting our 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?

```
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 this question is listed in table 5.6.

Table 5.6 Result of latest query: 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
A multipoint	A polygon with hole	t	f

There are a couple of important points to glean from these 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. A point can never touch a point or a multipoint because the shared points would always be interior to both.
- 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 that because we modeled the courtyard as a hole, the courtyard is part of the exterior of the house and not the interior. So even though one greeter is basking in the courtyard and the other is at the door, as a pair they 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's an OGC/SQL-MM–defined function.

Revisiting our 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?

```
SELECT A.ex_name As a_name, B.ex_name As b_name,
    ST_Crosses(A.the_geom, B.the_geom) As a_cr_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_Crosses(A.the_geom, B.the_geom) ;
```

The result of this question is listed in table 5.7.

Table 5.7 Result of this query: geometries that cross each other

a_name	b_name	a_cr_b	a_co_b
A polygon with hole	A linestring	t	f
A linestring	A polygon with hole	t	f

We can glean a few things from this example:

- We have only one pair of geometries that cross each other: the linestring (walkway) and the house. The reason these cross is that they don't 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 within 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't cross.
- It's 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's possible for the ST_Intersects and ST_Disjoint functions to both return false. 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've covered all the many facets of intersects and intersection, in the next section we'll 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

Two output relationship functions are very closely related to Intersection and Intersects. These are Difference and Symmetric Difference. These are much less commonly used than ST_Intersection and return the remainder of an intersection. ST_Difference is a non-commutative function whereas Symmetric Difference is, as the name implies, commutative.

Symmetric Difference is the dark twin of the intersection. It will return the simplest geometric representation of what's 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's not shared with B.

Here's one way to think about it:

```
ST_SymDifference(A,B) =  Union(A,B) - Intersection(A,B)
ST_Difference(A,B) = A - Intersection(A,B)
```

In the following listing we repeat a similar exercise to the one we did with Intersection except we're computing a Difference instead of an Intersection.

Listing 5.5 What's left of the polygon and line after clipping

```
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;
```

❶ Results in polygon

```
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;
```

❷ Results in multilinestring

```
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;
```

❸ Results in geometry collection

❶ This results in a polygon, which is pretty much the same polygon we started out with. This may be surprising to some; you'd expect a linestring would split it. ❷ This results in a multilinestring composed of three linestrings where the polygon cuts through. ❸ Finally, this results in a geometry collection as expected, composed of a multilinestring and a polygon.

Figure 5.5 is a diagram of the results.

As you can see, the linestring doesn't bisect the polygon, though this is a common desire in many cases. Why? If you think of 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,

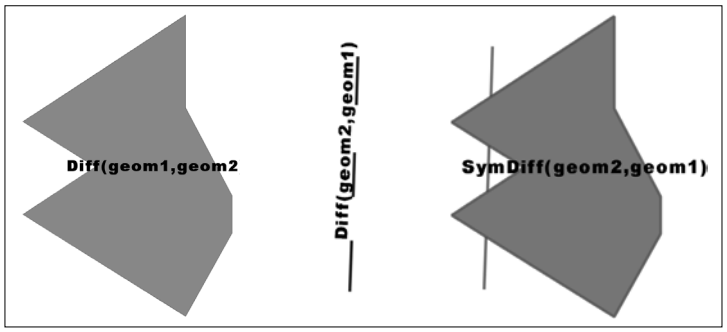


Figure 5.5 Queries from listing 5.5 output. Observe that the difference of the polygon and the line is pretty much the polygon we started out with.

which is more or less the original polygon, not a geometry collection of two polygons or a multipolygon. It can't be a multipolygon because a valid multipolygon can't 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 it has one or two point differences at the boundaries where the line cuts through. 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 won't intersect, and then gluing back the leftover slivers onto one of the resulting polygons. This often is good enough in many cases. The following listing is such a demonstration minus the glue. Table 5.8 and figure 5.6 show the result.

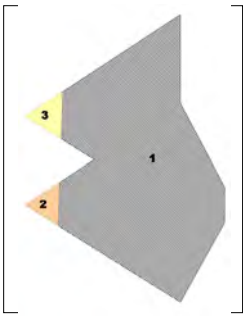


Figure 5.6 Result of the knife trick in listing 5.6

Listing 5.6 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 5.6: The polygon is cut into three 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))

As you can see in this example, we buffer the line to make it a very thin polygon we can cut with. This cuts the polygon into three, resulting in a multipolygon, which we

then break apart into three polygons using the ST_Dump function you learned about in the previous chapter. In later sections we'll delve into more precise but advanced options of cutting geometries.

> **PostGIS 2.0 ST_Split**
>
> PostGIS 2.0 added the ST_Split function, which allows for splitting a line by a point, line by line, or polygon by line with a single statement. All these can be done much simpler and more exact using ST_Split.

In the coming sections we'll explore the ST_DWithin relationship function that you've seen in earlier examples. It's 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'll 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 shorthand for `ST_Expand(A,X) && B AND ST_Expand(B,X) && A AND ST_Distance(A,B) < X`.

> **Why we use two expands in ST_DWithin**
>
> These expand calls may seem redundant: 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 how you expand you may end up using an index or not, and SQL planners don't process statements in order. They first try to process statements in order of cost when computing costs are not more costly than sequential. At runtime one of those checks is generally far more costly than another because one is generally a constant geometry (that's the one you'd 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 way to process the 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, and so returns a box. You can think of ST_Expand as the lazy sibling of ST_Buffer.

From PostGIS 1.3.5 on, additional short-circuit logic is built in.

The ST_DWithin is not an MM/SQL function, but it's pseudo standards compliant in that the many Web Feature Services (WFS) such as GeoServer, MapServer, MapInfo WFS, and Deegree have a DWithin filter operator that does the same thing. Oracle also has a function called SDO_GEOM.WITHIN_DISTANCE that operates the same way,

except the Oracle version accepts a unit of measure as argument, whereas 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

Another side use of this function is as a substitute for the ST_Intersects function. This usage we term as doing an *intersects with tolerance*.

Here are a few reasons why you may choose to use this function instead of 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's very small.
- ST_DWithin doesn't choke on invalid geometries as ST_Intersects often does. With ST_Intersects you'll often get a topology intersects exception error if your geometry has self-intersecting regions. ST_DWithin doesn't care because it doesn't rely on an intersection matrix.
- Although ST_DWithin doesn't work with curved geometries as of PostGIS 1.5, you can expect it to do so 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.

When used in this manner, the distance parameter must be set to very small to represent the maximum distance error to consider two geometries as intersecting. For example, you may consider the linestring and a point to be intersecting if they're 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
 ) ;
```

Next we'll look at finding the N closest objects.

5.5.2 Finding N closest objects

There are many ways of doing a nearest neighbor search. The classic example is finding, for example, the nearest five roads to a particular location that are within 10 kilometers of the location:

```
SELECT r.name, ST_Distance(r.geom, loc.geom)/1000 As dist_km
    FROM ch01.roads As r INNER JOIN
    (SELECT ST_Transform(
```

```
       ST_SetSRID(ST_Point(-118.42494, 37.31942), 4326),
         2163) As geom) As loc
  ON ST_DWithin(r.geom, loc.geom, 10000)
ORDER BY ST_Distance(r.geom, loc.geom)
    LIMIT 5;
```

In this example we're concerned only with finding roads near a single point. The location needn't be a point; it can just as well be a lake, a river, or a building that's represented as a linestring or a polygon or even a geometry collection. Here we're assuming our table geometries are stored in US National Atlas Equal Area meter projection (2163) and our point of interest is in lon lat (4326), so 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 five closest?

As you'll note, with ST_DWithin you need to provide a limit distance. If you wanted to find the five closest roads, doing a simple order by ST_Distance(....) limit 5 without the ST_DWithin, it would be very slow because the query would resort to what's called a table scan—calculating the distance of loc to every geometry in the table and returning the five closest. ST_DWithin reduces the set of false positives significantly. Thus it requires that you know something about your data—that you know all the top five 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 because it will grab more false positives to inspect. Some of this will change in PostgreSQL 9.1/9.2 and PostGIS 2.0+ because k nearest neighbors (kNN) scanning will be added to GIST indexes in PostgreSQL 9.1 or PostgreSQL 9.2 that may make this exercise much easier and efficient.

Another common use case is to find the closest from some other geometry. For this kind of query the DISTINCT ON custom SQL addition of PostgreSQL in conjunction with ST_DWithin comes in very handy. Listing 5.7 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 previously, except we do it for all locations of interest and return the closest road to each location of interest.

Listing 5.7 Find the closest road to each location; search 10 kilometers out

```
SELECT DISTINCT ON(loc.loc_name, loc.loc_type) loc.loc_name,
    loc.loc_type, r.road_name,
        ST_Distance(r.the_geom, loc.geom)/1000 As dist_km
FROM ch05.loc LEFT JOIN
        ch05.road As r
ON ST_DWithin(r.the_geom, loc.geom, 10000)
```

```
ORDER BY loc.loc_name, loc.loc_type,
  ST_Distance(r.the_geom, loc.geom) ;
```

There are three important things about using DISTINCT ON:

- The ON(…) can have 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.
- In the ORDER BY you can add additional columns in addition to the DISTINCT ON ones, to control which record gets picked as the unique one.
- Observe that we changed our query to be a LEFT JOIN; this is so that all loc_name, loc_city combos are returned even if there's no road within 10 kilometers. If you didn't care or were sure all would have roads that close, you could get 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; sadly neither DISTINCT ON nor the LIMIT clause will let you do this in one query. Luckily PostgreSQL 8.4 has what are called Window functions. Window functions are an ANSI SQL 2003 standard piece you'll find in all the high-end enterprise-class commercial databases (Oracle, IBM DB2, and SQL Server 2005/2008). The windowing support in PostgreSQL is more feature rich than in SQL Server 2005/2008, but it's not quite as feature rich as what you'll 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 following example requires PostgreSQL 8.4. It will return the top five closest roads for each location of interest that are within 20 kilometers.

Listing 5.8 Find the closest two 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
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 ch05.land As loc
LEFT JOIN
        ch05.road As r
ON ST_DWithin(r.the_geom, loc.the_geom, 1000)
  WHERE loc.land_type = 'police station') As foo
  WHERE foo.row_num < 3
ORDER BY pid, row_num;
```

1 Sequentially number by proximity

2 Left join includes no matches

3 Order by pid, proximity

This query is run against the fictitious poorly designed town we demonstrated at PostgreSQL Conference 2009 (PGCON2009). The code to build the town is available for download from http://www.postgis.us/presentations. The dataset already built is included in the chapter 5 code data download.

Table 5.9 shows a sampling of the results. There are three 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're 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're using a left join, which guarantees that all records in our WHERE will be included—in this case all police stations—but only roads within one kilometer of each police station will be considered.
3. We then 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 5.8

Pid	land_type	row_num	road_name	dist_km
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

Table 5.9 lists partial results from our query. Observe that for police stations farther than one 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 one kilometer (for example, 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've covered the basics of proximity analysis, we'll 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 box-based short-circuit comparisons. Bounding boxes and their geometry operators are used by spatial indexes to reduce the set of records that need to be more closely scrutinized by the exact comparison operators. We'll 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. Now that we've covered the basics of proximity analysis, we'll move on to bounding boxes and basic geometry comparators.

5.6.1 The bounding box

Let's demonstrate with a quick example. Suppose we have two multipolygons, one representing the state of Washington and one representing the state of Florida. If the bounding box of Washington is strictly above and to the left of (northwest of) Florida, then we 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, we'd 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 could we conclude that all of Washington is northwest of all of Florida. The bounding box methodology shortcuts the point-by-point checking by first drawing rectangular boxes around each state and then asking if the box enclosing Washington is above and to the left of the box enclosing Florida. We can 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 always assumes that you'd take advantage of them, freeing you from having to worry about explicitly calculating bounding boxes and creating indexes out of them.[1] Certainly, we can foresee many instances where bounding boxes won't 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 can't shortcut ourselves to an answer by simply looking at the bounding box of the two states. The next listing contains examples of geometries with their bounding boxes wrapped around them.

[1] The only exception is when you use deprecated functions.

Listing 5.9 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);
```

Figure 5.7 illustrates the output of this query, showing the geometries encased in their bounding boxes.



Figure 5.7 Various geometries and their bounding boxes from geometries in listing 5.9

Bounding box is not the smallest box.

For brevity we stated that the bounding box is the smallest box you can wrap around a geometry. Strictly speaking, it's not the smallest. PostGIS will often expand a bounding box to ensure that the coordinates can be represented with float4 numbers. 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 (0,2.72). The size limit of the box coordinates may change in future versions.

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 uses 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 precheck for spatial relationships.

Table 5.10 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. B-tree indexes are possible only if geometry objects are relatively small, such as points or small polygons and lines. If you have no B-tree index, then an operator that works

only with B-tree won't use an index. You can have both a GIST and a B-tree index on the same geometry field; B-tree indexes on geometries are rare and of minimal utility.

Table 5.10 PostGIS operators that can be applied to geometries

Operator	What is 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	B-tree
>>	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
|>>	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; superseded by ST_OrderingEquals	gist

Bounding box or geometry comparators?

Although 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've covered the basics of bounding boxes, which are used extensively as a precheck for other relationships, we'll explore the most fundamental relationship: equality and its multifaceted meaning.

5.7 The many faces of equality

In conventional databases, you probably never gave the equality comparator (=) a second thought before using it. This unambiguity doesn't carry over to spatial databases. When we compare two geometries, *equality* is a multifaceted notion. We can ask

whether geometries occupy the same space. We can ask whether they are represented by the same points. We can ask whether their bounding boxes are the same.

Three basic kinds of equality are specific to geometries in PostGIS:

- Spatial equality (occupying the same space)
- Geometric equality (same space, more or less same points and same point order, although 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'll find in many spatial databases and is based on what's called an *intersection matrix* model .

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 has only that single polygon in its list. It's an equality that doesn't care about the vector direction of the line segments or point ordering in the geometry.

5.7.2 Geometric equality

Geometric equality is even more strict than spatial equality. When two valid geometries are geometrically equal, not only must they 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're traveling on, the interstate is signed as north versus south or east versus west. Although it's 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 not geometrically equal.

PostGIS offers the ST_OrderingEquals function for geometric equality. In versions pre PostGIS 1.4, the ~= meant the same thing, but in newer installs it's just bounding box equality that uses a spatial index. To be safe, stay away from using ~= and stick with ST_OrderingEquals. In listing 5.10 we demonstrate the difference between ST_OrderEquals and ST_Equals.

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.

Listing 5.10 ST_OrderingEquals equality versus ST_Equals

```
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);
```

The results are shown in table 5.11.

Table 5.11 Results of listing 5.10: Even in an invalid polygon is ordering equal to itself in PostGIS.

ex_name	g_oeq_g	g_oeq_rev	g_oeq_m	g_seq_g	g_seq_m
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 that the multigeometry variant is not geometrically equal to the singular version, but in the case of spatial equality, because they occupy the same space, they're equal.

ST_OrderingEquals versus 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 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 doesn't 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 on whether you did a soft upgrade or a 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 a spatial indexable =. We suggest you don't rely on ~= anymore and use ST_OrderingEquals instead.

5.7.3 Bounding box equality

In PostGIS, the one-and-only equality comparator (=) is reserved for bounding box equality. If you 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 doesn't mean that A is B. Here's an example illustrating the difference:

```
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. You may ask, how is this possible? It's because their bounding boxes are equal although 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 examples in the following listing demonstrate this anomaly.

Listing 5.11 DISTINCT is not always DISTINCT

```
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);

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);

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);

SELECT DISTINCT ST_AsText(the_geom)
FROM (SELECT ST_GeomFromEWKT('LINESTRING (3 5, 3.4 4.5, 4 5)')
```

1 Two records

2 One record

3 One record

4 Two records

```
UNION ALL
SELECT ST_GeomFromText('POLYGON ((3 5, 3 4.5, 4 4.5, 3 5))')

) As foo(the_geom);
```

These examples demonstrate the oddity that is the bounding box = operator. Because = for geometries is mapped to = of the bounding box and SQL uses = for DISTINCT checks, you end up with somewhat strange situations, as demonstrated in the listing. In ❶ we get two records back because we're doing a UNION ALL, and a UNION ALL by definition returns all records in the union. In ❷ we get one record back, which is the first one that's hit (the linestring), because our subquery has a UNION, and UNION without ALL puts in an implicit DISTINCT. Because the bounding boxes of the geometries are equal, they are seen as equal. In ❸ we get one record back for the same reason as ❷. In ❹ we get two records back because the output of ST_AsText isn't 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 people get bitten the most. Here's a demonstration of this tragedy.

Listing 5.12 A count DISTINCT is not always a DISTINCT count.

```
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);
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);

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;

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;
```

❶ Gives 1 as answer

❷ Gives 2 as answer

❸ Returns one geometry

❹ Returns two geometries

As you can see in ❶ the DISTINCT count gives an answer of 1, because both geometries share the same bounding box and therefore there's only one distinct bounding box. In ❷ we get an answer of 2 because we changed the linestring slightly so that the bounding box is different from the polygon. In ❸ we get only one geometry back,

because the group by sees the geometries as the same although they are different. In ❹ 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? We're not sure. One theory is it's efficient and isn't that much of an issue. It's 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's rare that we're only doing a DISTINCT or GROUP BY on a geometry and that our geometries have exactly the same bounding boxes, so the uniqueness of the other fields and the rareness of non-dupes having exactly the same bounding box counterbalances this behavior.

STEPS YOU CAN TAKE TO AVOID THE = TRAP

The bounding box equality issue comes into play in several common SQL constructs:

- 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're 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) is illustrated in listing 5.13.
- If you want to test spatial equality, use ST_Equals. If you want to test true geometric equality, use ST_OrderingEquals instead of =.

Listing 5.13 Guaranteeing unique geometries

```
CREATE TABLE mygeom_unique(the_geom geometry);
INSERT INTO mygeom_unique(the_geom)
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 in this example we're 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 code into the table, PostgreSQL silently casts it to a geometry for you.

Support for curved geometries and 3D geometries in operators

All the operators work for curved geometries except for ~=. For 3D geometries (geometries with a Z coordinate), the Z coordinate is ignored for the bounding box operators but considered with ~=.

Next we'll look at the underpinnings of relationship functions.

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'll 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's a mathematical approach that defines the pair-wise intersection and geometric dimension of the resulting intersection of the three regions of a geometry. It's a 3 x 3 matrix consisting of interior, boundary, and 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 nine cell slots. When used to output the most constraining relationship between predefined geometries A and B, it can contain only F, 0, 1, or 2 in the cell slots. The reason for that is that an intersection must always have a corresponding dimensionality, and with F (no intersection) there's no dimensionality. Not only do there exist quite a number of possible matrices, but you can construct more complex statements by chaining intersection matrices 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 whether geometries A and B satisfy the specified relationship matrix. The second variant denotes the most constraining relationship matrix satisfied by the two geometries.

The three quadrants of the intersection matrix are listed here as well as what each means:

- *Interior*—The portion of a geometry that's inside the geometry and not on the boundary.
- *Exterior*—The coordinate space outside a geometry but not including the boundary.
- *Boundary*—The space neither interior nor exterior to the geometry; it's the space that separates the interior from the exterior.

Each cell of the matrix can hold one of the values shown in table 5.12.

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 doesn't matter if an intersection exists or not.
0	An intersection must exist, and the 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 figure 5.8 we show 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 three matrix statements. In DE-9IM notation it's easier to use proof by contradiction (assuming you're dealing with valid geometries)—state that geometry A intersects geometry B if they are not Disjoint, thus reducing the three matrix statements to a NOT 1 matrix.

		B		
		Interior	Boundary	Exterior
A	Interior	F	F	*
	Boundary	F	F	*
	Exterior	*	*	*

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 two geometries with totally different points or reversed points, and as long as the resulting geometry occupies the same space, they're 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 shown in figure 5.9 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*FFFF*)**

Observe in the chart that interiors must intersect; exterior/interior and exterior/boundary never intersect. The reason for that is that for a given geometry there's 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 say anything about the intersection relation of the boundary with its interior.

The following listing is a simple example for various geometries and the accompanying ST_Relate matrix.

Listing 5.14 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 this query, shown in table 5.13, a given geometry is generally equal to itself and its reverse (same geometry with coordinate points reversed), and it's also equal to its multigeometry counterpart.

Table 5.13 Results of query in listing 5.14

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)	0FFFFFFF2	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	

This model of a geometry that's equal to itself can break down if you have an invalid geometry. The DEM-9IM relation matrix of all satisfies the T*F**FFF* rule except for our bowtie self-intersecting polygon from chapter 2. It fails the DE-9IM test because its interior intersects with its exterior. In other words, the area it defines is ambiguous.

5.8.3 Using the intersection matrix with ST_Relate

The most generic of all relationship functions 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 aren't because the core relationship functions have numerous shortcuts imbedded in them that take advantage of the kind of geometric type each geometry is and how many geometries and so forth. Most of the other relationship functions such as ST_Contains and ST_Touches also take advantage of spatial indexes because their bounding boxes are required to intersect. ST_Relate doesn't take advantage of spatial indexes automagically, because it can be used to express both Intersect relation types as well as non-intersecting relationships.

In some cases, the various permutations that can be allowed by the intersection matrix are more than can be achieved with the functions we've described. Although ST_Relate is rarely used, it's still good to understand it to get a better grasp of what the other relationship functions mean because many can be unequivocally expressed in DE-9IM geeky notation.

Listing 5.15 is an example that exercises both ST_Relate functions. The example 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 5.15 ST_Relate In action

```
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)
)
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;
```

CTE example_set 1

Relate sample set 2

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 to relate to each other row in the set. The output of this query is shown in figure 5.10.

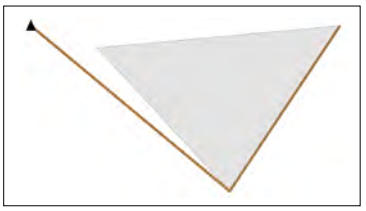


Figure 5.10 The geometries from the query in listing 5.15

The results of our query are shown in table 5.14.

Table 5.14 Results from query in listing 5.15

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
A 2d line	A triangle	F11F00212	t	f	t
A point	A 2d line	FF0FFF102	t	f	t
A point	A point	0FFFFFFF2	t	f	t
A point	A triangle	FF0FFF212	f	t	f
A triangle	A 2d line	FF2101102	t	f	t
A triangle	A point	FF2FF10F2	f	t	f
A triangle	A triangle	2FFF1FFF2	t	f	t

Observe in this example that the result of the not disjoint DE-9-IM statement is equivalent to the answer we get with intersects. Note that the relationship of the triangle with the 2d line is FF2101102 and the 2d line with the triangle is F11F00212. The DE-9IM notation of this relationship is shown in figure 5.11.

		2d line		
		Interior	Boundary	Exterior
triangle	Interior	F	F	2
	Boundary	1	0	1
	Exterior	1	0	2

		triangle		
		Interior	Boundary	Exterior
2d line	Interior	F	1	1
	Boundary	F	0	0
	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. Here are some other important observations:

- The triangle and the lines interiors don't intersect, as you can sort of see from the image of the geometries.
- The interior of the triangle does not intersect with the boundary of the line (recall that the boundary of a line is the start and end points), 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 2D coordinate space that's not the line (so it's areal).

For a more in-depth explanation of the DE-9-IM model refer to http://docs.codehaus.org/display/GEOTDOC/Point+Set+Theory+and+the+DE-9IM+Matrix#PointSetTheoryandtheDE-9IMMatrix-9IntersectionMatrix.

Figure 5.12 shows 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*FF***)**

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't fall outside B (it can't intersect with the exterior of B), and the boundary can't fall outside B (the boundary can't intersect with the exterior of B). The boundaries, however, are free to intersect or not to intersect.

5.9 Summary

In this chapter we covered a fair amount of territory involving spatial relationships. Hopefully we provided you insight into the subtleties of these not-quite-obvious relationships. Now that you understand the foundations of spatial databases, we'll look at their application in more real-world examples. You'll 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 haven't covered; many will be ones we've already explored but that we'll combine with other functions in thought-provoking ways.

Two subjects we haven't yet delved into too deeply are spatial aggregates and geometric processing functions. In the coming chapters we'll demonstrate these.

6 Spatial reference system considerations

This chapter covers

- Characteristics of spatial reference systems
- How to determine and select spatial reference systems

Up to this point we've been working mostly with fictitious data and only glimpsed at real-world data. Using sample data to learn the basics of PostGIS is an excellent beginning. You're immediately rewarded with results without facing the distractions and the obstacles of real-world data. From this chapter forward, we're not going to shield you any more.

We start this chapter with coverage of spatial reference systems. We 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 have been around since the antiquities. Geodetics is the science of measuring and modeling the earth. Cartography is the science of representing the earth on flat 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's of utmost importance to GIS: the spatial reference system (SRS).

In this chapter, we're not going to take the easy way out by accepting SRS without understanding it. We'll 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.

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 to its unglamorous nature compared to other areas of GIS. If GIS is Disneyland, think of SRS as the bookkeeping necessary to keep the Disneyland operation afloat.

Take any two paper maps from your collection having one point in common and overlay one atop of the other using as a reference the point they have in common. Both maps represent the whole or a part of earth, but unless you're 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 don't overlay well because they don't have the same spatial reference system. The main reason for the GIS data consumer to become acquainted with SRS is to bring in data from disparate sources in different SRSes and be able to overlay one atop another. Many standards exist to make this task easy without having to delve into the nuances of SRS. The most common one is the European Petroleum Survey Group (EPSG) numbering system. Take any two sources of data with the same EPSG number, and they'll overlay perfectly. EPSG is a fairly recent SRS numbering system. If you uncover data from a few decades ago, you'll not find an EPSG number. You'll have no choice but to delve into the constituent pieces that form a spatial reference system. So what is 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 actually comprises mountain ranges, deep canyons, and ocean trenches. The surface of the earth with all its nooks and crannies resembles a slightly charred English muffin much more than a lustrous marble. Even the idea of the earth being spherical isn't accurate, because 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 and somewhat squashed orange under our feet, what are we going to do? With our new GPS toys we could conceivably represent every square meter on earth as a satellite map, assigning it a spherical 3D coordinate, and be done with it. This is the approach taken by many digital elevation models. Though this brute force computation method 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 doing 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 because you'll never exercise their points of failure. Until the time when we can afford to carry around portable holograms of the earth, we need several cheap models.

A starting point for any 3D model is the choice of definition of the surface of the earth. Do you 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 can't really go out and set up a standard of measurement that's applicable around the entire world. Take the notion of sea level, for instance. Someone in Cardiff, England, can say that her house is 50 meters above the sea during low tide and use this as a reference against her neighbor's house. Suppose a fellow in Pago Pago has a small house and 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. And even the notion of center of the earth is ambiguous.

Along comes Gauss, who, with the help of a crude pendulum, determined in the early nineteenth century 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 and then use this constant gravitational force to map out an equigravitational 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. You must not forget that the core of the earth isn't homogenous. Mass is distributed unevenly, giving rise to bulges and craters that rival those found on the lunar surface. The advent of the geoid didn't 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.

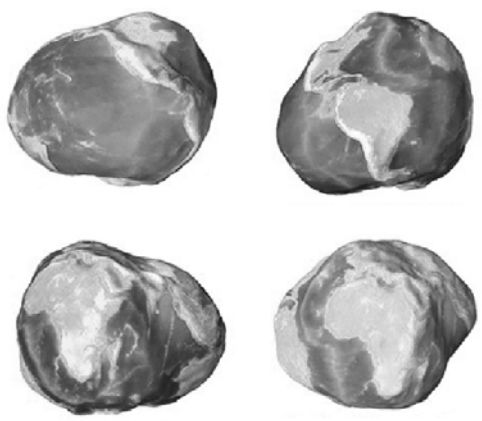


Figure 6.1 A geoid seen from different angles

Although the geoid is rarely talked about in GIS, it's the foundation of both planar and geodetic models. In the next section, we'll discuss the more commonly used ellipsoids, which are simplifications of the geoid and are generally good enough for most geographic modeling needs.

6.1.2 Ellipsoids

Since ancient times, the 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 three radii: a and b are equatorial radii (along the X and Y axes), and c is the polar radius (along the Z axis). In geodesy only two axes are considered: semi major and semi minor. 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 axes on the ellipsoid, you 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 discovery 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.

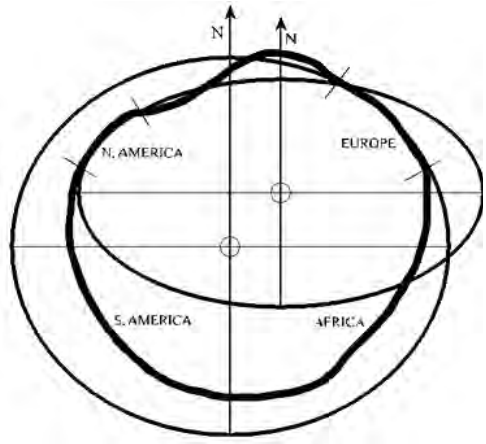


Figure 6.2 The geoid and the ellipsoid seen together

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 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 need for scientists and governments to collaborate and the rise of oil surveying and aviation. Fortunately, today the world is settling on the World Geodetic System (WGS 84) and GRS 80 ellipsoids, with WGS 84 becoming the standard of choice. WGS 84 is what all GPS systems are based on. To call WGS 84 simply an

ellipsoid isn't quite accurate. The WGS 84 GPS systems we use have a geoid component as well. The present WGS 84 system uses the 1996 Earth Gravitational Model (EGM96) geoid and is the best-fitting ellipsoid to the geoid model for the selected survey points in the set.

Common ellipsoids used today are:

- GRS 80
- WGS 84 (more common nowadays and the standard for GPS data)

The 80 and 84 stand for 1980 and 1984, when the standards came out, and they're very similar.

Many ellipsoids have been used over the years, and some continue to be used because of their better fit for a particular region. All historical data is still referenced against other ellipsoids. Table 6.1 shows 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	Time (World)

One common old ellipsoid is the Clarke 1866 (this is so close to what is called the NAD 27 ellipsoid that they're synonymous for most purposes). So even though these old data points are measured in longitude and latitude, they aren't the same longitude and latitude we use today, and they also use different grounding points. They're shifted.

Lon lat which ellipsoid?

This is why it's important to not just call things lon lat. You can have NAD 27 lon lat, NAD 80 lon lat, and WGS 84 lon lat, and each will be subtly different. As a rule, when people nowadays refer to lon lat, they mean WGS 84 datum and WGS 84 spheroid in lon lat units. NAD 27 is the most different because it was done a long time ago. (Note that datum is the shift of a spheroid. See the next section.)

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, you need to anchor it should you ever need to use it for real-world navigation. Every ellipsoid that's not a perfect sphere has 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 the 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 somewhere in Canada.

To anchor an ellipsoid to a point on earth, you need two types of datum: 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 (NAD 27) is anchored at Meades Ranch in Kansas because it's close to the geographical centroid of the United States. NAD 27 is both a horizontal and a vertical datum. Here are some commonly used datums:

- NAD 83 (North American 1983, which is often accompanied with the GRS 80 spheroid)
- NAD 27 (North American 1927, which is generally accompanied by the Clarke 1866/NAD 27 ellipsoid)
- European Datum 1950
- Australian Geodetic System 1984

6.1.4 Coordinate reference system

Many people confuse coordinate reference systems (CRS) 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, you 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're 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've only seen geographical coordinate systems used on a globe, the concept applies to any reference ellipsoid. For that matter, it applies to anything resembling an ellipsoid. For instance, a watermelon has nice longitudinal bands on its surface.

6.1.5 Projection

Let's summarize what we discussed thus far about spatial reference systems:

- We start by modeling the earth using some variant of a reference ellipsoid, which should be the ellipsoid that deviates least from the geoid for the regions on earth we care about.

- We use a datum to pin the ellipsoid to an actual place on earth, and we assign a coordinate reference system to the ellipsoid so we can identify every point on the surface. For example, the zero milestone in Washington, D.C. 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 NAD 27 datum, Clarke 1866 ellipsoid, this would be W -77.03685, N 38.8950.

We can quit at this point, because we have all the elements necessary to tag every spot on earth. We can even develop transformation algorithms to convert coordinates based on one ellipsoid in relation to another. Many sources of geographic data do stop at this point and don't go on to the next step, projections. We term this data *unprojected* data. All data served up in the form of latitudes and longitudes is unprojected. You can do quite a bit with unprojected data, such as by using the great circle distance formula, you can get distances between any two points. You can also 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. Because geodetic and 3D globes are ellipsoidal, they by definition don't 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 eminently practical: You can't carry a huge globe everywhere you go. Less obvious but more relevant is the mathematical and visual simplicity that comes with planar (Euclidean) geometry.

As we have repeated many times, PostGIS works for the most part on a Cartesian plane, and most of the powerful functions assume a Cartesian model. Your brain and the quite different brain of PostGIS can perform area and distance calculations quickly on a Cartesian plane. On a plane, the area of a square is its side squared. Distance is nothing more than applying the Pythagorean theorem. A planar model fits nicely on a piece of paper. Calculating the area of a square directly on the surface of an ellipsoid becomes quite a challenge, not the least aspect of which is deciding what constitutes a square on an ellipsoid in the first place.

PostGIS 1.5 supports geodetic data

PostGIS 1.5 introduced support for geodetic data using the new datatype *geography*, similar in concept to SQL Server 2008 geography types. All spatial functions work for geometry data, with only a few functions and operators also for geography, such as distance functions.

How exactly you'd squash an ellipsoidal earth on a flat surface is controlled by several classes of rules we'll loosely refer to as the *classes of Cartesian coordinate systems*. Each class of rules tries to optimize for a set of features, each specific instance of a coordinate system is bounded by a particular region on earth, and each uses a particular unit (usually meters or feet).

Needless to say, you try to balance four conflicting features. The importance you place on each will dictate the choice of coordinate system and eventually of the spatial reference system(s):

- *Measurement*
- *Shape*—How accurately it represents angles
- *Direction*—Is north really north?
- *Range of area supported*

The general tradeoff is 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 among them. The larger your area, the less accurate and potentially grossly unusable your measurements will be. If you try to optimize for shape and to cover a large range, your measurements may be off, perhaps way off.

There are a few flavors of projections (squashing) you can do to optimize for different things. These are listed here:

- *Cylindrical projections*—Imagine a piece of paper rolled around the globe and imprinting 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 bottom of the rolled cylinder parallel to the equator. This results in great distortion at the polar regions, whereas measurement accuracy is best the closer you are to the equator, because there the approximation of flat is most accurate.
- *Conic projections*—These are sort of like the cylindrical projection except you wrap a cone around the globe, take the imprint of the globe on the cone, and then roll it out.
- *Azimuthal projections*—You project a spherical surface onto a plane tangent to the spheroid.

Within these three kinds of projections you must also consider the orientation of the paper you roll around the globe. These are the possibilities:

- *Oblique*—Neither parallel nor perpendicular to the equator; some other angle
- *Equatorial*—Perpendicular to the plane of the equator
- *Transverse*—Parallel along the equator

Combinations of these categories form the main classes of planar coordinate systems:

- *Lambert Azimuthal Equal Area (LAEA)*—These are reasonably good for measurement and can cover some large areas but are not great for shape. The one we like most when dealing with United States data and when we're concerned with somewhat 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 six-degree longitudinal strips. If you need to cover the whole globe and you use one of these, you'll have to maintain about 60 spatial ref IDs. You cannot use them for the polar regions.
- *Mercator*—These are good for maintaining shape and direction and span the globe, but they're not good for measurement, and they make the regions near the poles look huge. The measurements you get from them are nothing less than cartoonish, depending on where you are. The most common Mercator projections in use are variants of World Mercator (SRID:3395) or Spherical Mercator (aka Google Mercator (SRID:900913), which is now an EPSG standard with EPSG:3785. This last one is fairly new, so you may not find it in your spatial_ref_sys table if your PostGIS version is older. They're common favorites for web map display because you only have to maintain one SRID, and they look good to most people.
- *National Grid Systems*—These are generally a variant of UTM or LAEA but are used to define a restricted region such as a country. As mentioned, US National Atlas (SRID:2163, US National Atlas Equal Area) is common for the United States. These are generally decent for measurement (but not super accurate), don't always maintain good shape, but cover a fair amount of area, which is in many cases the national area you care about.
- *State Plane*—These are U.S. spatial reference systems. They're usually designed for a specific state, and most are derived from UTM. Generally there are two for a state—one measured in meters and one measured in feet—although some larger states have four or more. Optimal for measurement, these are commonly used by state/city land surveyors but, as we said, they can deal with only a single state.
- *Geodetic*—PostGIS can store WGS 84 lon lat (4326) as a geometry data type, but more often than not you'll want to transform it to another spatial reference system or store it in the geography data type for it to be usable. You can sometimes get away with using it as a geometry data type for small distances along the same longitude and when two things intersect, but keep in mind that when you use it, PostGIS is really projecting it. It squashes it on a flat surface, treating longitude as X and latitude as Y, so even though it looks unprojected, in reality it's projected and 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 show how to select 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) is appropriate for their data. The answer is, it depends.

Table 6.2 lists the most commonly used spatial reference systems and their PostGIS/EPSG SRIDs. PostGIS SRIDs follow the EPSG numberings, so you can assume for sake of argument they're the same. This isn't necessarily true for other spatial databases, so keep in mind that a spatial reference system can have several different IDs. Although EPSG is the most common authority on spatial reference systems, it isn't the only one. Many people, for example, load up their tables with ESRI definitions, which are sometimes identical to EPSG definitions, but under an SRID code that's more ArcGIS friendly.

Table 6.2 Common spatial reference systems and their fitness for purpose

EPSG/PostGIS SRID	Colloquial name	Range	Measurement	Shape
4326	WGS 84 lon lat	Excellent	Bad	Bad
3785/900913 (old number)	Spherical Mercator	Good	Bad	Good
900913 (deprecated)	Google Mercator	Good	Bad	Good
32601-32760	UTM WGS 84 Zones	Medium	Fairly good	Good
2163	US National Atlas EA	All U.S.	Medium	Medium
State Planes	US State Planes	Medium	Good	Good

If you deal with mostly regional data, say for a country or state, then it's 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 because PostGIS 1.4 and lower support only Cartesian coordinate systems, you may have to use several if you need to span large areas and maintain measurement accuracy.

6.2.1 Pros and cons of using EPSG:4326

The most common spatial reference system people use is WGS 84 lon lat (EPSG:4326). Aside from the common reason, that people just don't know any better, the reasons why knowledgeable people use this system are:

- It covers the whole globe and is the most common transport spatial reference system. For example, all GPS data is stored in this SRS. If you need to cover the world, dish out data to lots of people, and also deal with lots of GPS data, this isn't a bad choice.
- Most commercial mapping toolkits, although they use some variant of Mercator for display, expect the data to be fed in WGS 84 lon lat. ST_Transform also introduces some rounding errors as you retransform data, so it's best to transform

only once from the source format. ST_Transform is a fairly cheap process, so it's okay to run it for each geometry if you keep functional indexes on the transformations you use for distance checking.

Reasons not to use it:

- It's bad for measurement. If measurement is something you do often, especially when you're concerned about only small regions such as a country or state, you'll 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_Spheroid/Sphere hack doesn't work for PostGIS 1.4 and below.
- Things like intersects, intersection, and union generally work fine for small geometries but fall apart for large geometries, like continents or long fault lines.
- It's bad for shape. It also doesn't look good on a map. It's all squashed because we're showing longitude and latitude, which are meant to be measured around an ellipsoid, and we're showing it on a planar axis we call X and Y.

6.2.2 Geography data type for EPSG:4326

If you'll 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 1.5. The key benefit it provides over the geometry EPSG:4326 is that it's ideal for measurement because it's not projected and measurements are always in meters. Pros are as follows:

- It 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 distance, area, and length measurements, this is probably the best.
- Most web mapping layers such as Google, Virtual Earth (Bing), and the like expect data to be fed to them in WGS 84 so geography will work fine out of the box.

So if 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 an ST_Buffer. But these are just wrappers around the geometry implementation that perform behind the scenes a transformation to a suitable planar projection, so it's not too hard to roll your own functions.
- Although you can piggyback on the geometry functions for processing by casting and transforming to geometry and casting back, the ST_Transform operation isn't a lossless operation. ST_Transform introduces some floating-point errors that can quickly accumulate if you do a fair amount of geometry processing.

- If you're dealing with regional data, WGS 84 is generally not quite as accurate for measurement as regional spatial reference systems.
- If you're building your own mapping app, you'll still need to learn how to transform your data to other spatial reference systems if you want them to look good on a map, and although the transformation process is fairly cheap, it can quickly become taxing the more data you pull, the more users hitting your database, or the greater number of points you have in a geometry.
- Not as many tools support geography. In theory, any tool that just uses the ST_AsBinary and other output functions of PostGIS geometries will work fine with geography without any change.

6.2.3 Mapping just for presentation

Although the basic Mercator projections are horrible for measurement calculations, especially far from the equator, they're a favorite for web mappers because they look good on a map. The advantage of Google Mercator, for example, is that the whole globe is covered with just one spatial ref.

So if your primary concern is looking good on a map and overlaying on Google Maps with something like OpenLayers, Mercator isn't a bad option for native storage of data. If you're concerned with distances and areas, it depends on the accuracy you need. Table 6.3 (generated from code in chapter 8) lists the distances between city pairs measured using various spatial reference systems.

Table 6.3 Results of distance calculations in kilometers

city1	city2	sp	spwgs84	wm
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

Table 6.3 Results of distance calculations in kilometers *(continued)*

city1	city2	sp	spwgs84	wm
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

In this table are various city point pairs and their distances measured in WGS 84 sphere (sp), WGS 84 spheroid (spwgs84), and Web Mercator (wm). As you can see, Web Mercator distance precision is much worse than the others and gets worse the farther away two cities are from each other or for regions farther from equator. The computed distance between, for example, Beijing and Philadelphia is really poor with Mercator. The sphere calculations are pretty good for long-range/short-range rule-of-thumb calculations.

This table covers distance, but what about the areas of geometries? How bad is the story there? Again, this depends where you are on the globe, but in general the situation is bad. Table 6.4 shows the areas of 10-meter buffers around the globe, generated from code in chapter 8.

Table 6.4 List 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 314 sq m?

It isn't. If you do your calculation, a perfect 10-meter buffer will give you an area of 10*10*pi(), which is around 314 sq m. The default buffer in PostGIS is a 32-sided polygon (eight 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're in the unfortunate predicament of needing to cover the whole globe with good measurements and shape accuracy, then most likely a single spatial reference system isn't going to cut it. A common favorite is the UTM family of SRIDs. There are about 60 UTM SRIDs for WGS 84, each covering six-degree longitudinal strips. There is also a series of UTMs for NAD 83, but the WGS 84 one is more common.

You'll need to figure out the UTM WGS 84 SRID for your particular dataset. There is a function for that in the PostGIS wiki at http://trac.osgeo.org/postgis. The following listing shows 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 Determing WGS 84 UTM SRID of a geometry

```
CREATE OR REPLACE FUNCTION upgis_utmzone_wgs84(geometry) RETURNS integer AS
$$
DECLARE
    geomgeog geometry;
    zone int;
    pref int;
BEGIN
geomgeog:=ST_Transform(ST_Centroid($1),4326);

IF (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;
```

1 Convert to a lon lat point

2 Determine UTM start and sub

1 We convert our geometry to a point and then transform it to WGS 84 lon 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 whether latitude is positive or negative: UTM EPSG numbers start with 32600 and increment every six degrees. Negative latitude, or 0, starts at 32700. So the final SRID is between these numbers.

If you need to maintain multiple SRIDs, you have 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 about the correct way to go, and none is right or wrong. For our cases, we've found that keeping one SRID (usually 4326) and transforming as needed works best, provided we maintain functional indexes on transforms used for distance calculations. We also like using views as an abstraction layer where the view contains the calculated transform. PostgreSQL supports not only functional indexes but also partial ones. A partial index, for example, allows you to index only part of your data. So in general you should only apply an ST_Transform function for the region defined for a given UTM; otherwise you'll run into coordinate bounds issues. Generally speaking, it's best to partition your data using table inheritance and use different transform indexes for each table separately. The following listing 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_Tranform(the_geom,32611) As the_geom_utm
    FROM feature_data;
```

In this view, we're transforming our native data to SRID 32611, which is one of the UTM SRIDs for a region of California in the United States.

Functional indexes on ST_Transform

Putting functional indexes on ST_Transform is something we do when building a view on our data with the transformed version of the data. It's a gray zone, in the sense that we're exploiting a small violation of treating ST_Transform as an immutable function, when technically it isn't. 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,

(continued)

it's a bit of lie that it's immutable, because it relies on entries in the spatial_ref_sys table. If you happen to change the entry for your transform in the table, you'll need to reindex your data, otherwise it will be wrong, but then again so would be the 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's rare that you'd 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 you make sure to set the search_path of the ST_Transform function to include the schema the spatial_ref_sys resides in (supported only 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's a bit of a no-no? The other alternative is to keep a geometry field for your alternative spatial references. This is annoying for two reasons: (1) You have to ensure it's 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 create a copy of a record during update. It probably slows down selects too because you have a fatter row to contend with. Using ST_Transform on the fly is cheap, but doing an index search on this calculated call isn't possible without a GIST index on this transformed data.

Often you'll have to load spatial data into your database that you didn't 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 will be wrong. In the next section we'll cover how to determine the spatial reference system of a data source.

6.3 Determining the spatial reference system of source data

In this section, we'll go through some exercises to determine the spatial reference system of source data. This will prepare you for the next chapter, where we finally start loading real data. Before being able to do that, you need to know where you 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 not. Sometimes a site just tells you the EPSG code for its data, and your work is done. Often, it 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'll need to match up the description with a record in the spatial_ref_sys table.

With newer ESRI shapefiles there often is a file with a .prj extension giving the spatial reference system information in WKT SRS notation. This file is often used by third-party tools to derive the projection for the case where different layers need to be transformed to the same spatial reference system to be overlaid on a map. In the following exercises, we'll demonstrate some SRS text descriptions and demonstrate how you can match these with an SRID in the spatial_ref_sys table. In some cases your task may be hard, especially when the record you're looking for doesn't exist and you'll need to add it. We'll go over that too.

More shockingly, some data comes with no spatial reference information or (even worse) the wrong information. The easiest way to determine this is to overlay a map where you suspect this to be the case on top of a layer for the same region that you know the spatial reference system for and reproject to the suspected projection. Common errors are, for example, using NAD 27 data in a NAD 83 spatial reference system. In these cases you'll see Doppler-like shifts when you overlay the two. 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. This is the cause for a well-known beginner's FAQ: "Why don't I see anything?"

6.3.1 Guessing at a spatial reference system

We'll go over some 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. But for this particular set, the site gave us a states020.txt file, which gives us spatial reference information as well as lots of details about how the dataset was made and its licensing.

If you scroll down far enough in the file, you'll 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
```

This is an important piece of information. It tells us that the data is in decimal degrees, and uses ellipsoid GRS1980 and datum North American Datum of 1983. These are the three ingredients you need to know about every data source you have:

- Unit: degrees
- Ellipsoid: grs1980
- Datum: nad1983

If you're dealing with projected data (non-degree data), there are some other fuzzy pieces you'll need to know. One is the projection, and depending on the projection, each type of projection has additional parameters:

- Projection: (degree is longlat), eaea, utm, tmerc, lcc, stere

Once you've figured out these pieces, the next thing 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're seeking isn't in the table and you'll need to add it. Living without one is only an option if you know your data is planar, you know the units, and all data you'll be getting is from the same source and was made using the same spatial reference system. In this case, you're using the unknown SRID, which is -1 currently in PostGIS but 0 in the OGC standard.

Two fields of information in the spatial_ref_sys table can help you guess at the projection. For the previous 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 will return one record with SRID 4269. It's generally easier to query the proj4text field for matches because the proj4text field is much shorter and more consistent than the srtext field.

EXERCISE 2: SAN FRANCISCO DATA (READING FROM .PRJ FILES)

For this second exercise we grabbed a zip file with Bay Area bridges. The file includes a .prj file, which has projection information: http://gispub02.sfgov.org/website/sfshare/catalog/bayarea_bridges.zip.

The .prj contents look like this:

```
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",6561666.666666666],
PARAMETER["False_Northing",1640416.666666667],
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, it's NAD83 datum, and the projection is 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%';
```

This query yields six records. When we look at the srtext field of each, each has something of the form `NAD83 / California zone 1 (ftUS)`, where the number ranges from 1 to 6. Remembering our Roman numeral lessons from grade school, we recall that III is the Roman numeral for 3. So our answer must be SRID 2227, which has an srtext field that looks like this:

```
"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 you have a small grasp of how to match an SRS to one in your table, what do you do if there isn't one in the table?

EXAMPLE 3: IF YOU GUESS WRONG

Let's imagine 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's a maintenance function in PostGIS to help you out in this situation called UpdateGeometrySRID, which will correct the mistake.

```
SELECT UpdateGeometrySRID('sf', 'bridges', 'the_geom', 2227);
```

Let's imagine that we brought our San Francisco data in an unknown with -1 SRID or some wrong spatial reference. This would become quite apparent if we tried to transform our data. If we did and the data was wrong, we'd 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 doesn't have the spatial reference you're 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're looking at. The best place to go at that point is http://spatialreference.org.

The spatialreference.org site contains thousands of user-contributed spatial reference systems in addition to the standard ones. Best of all, if the record you're looking for can't be found and you happen to have a .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 into your spatial_ref_sys table.

SpatialReference.org uses the auth_srid field instead of SRID

The spatial reference site by default assigns an SRID starting with 9 to denote it was grabbed from the spatialreference.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 won't accidentally insert a record into spatial_ref_sys that's already in the table.

Although it's possible to create your own custom spatial reference system to suit your specific needs, such a topic is beyond the scope of this book. PostGIS uses the PROJ.4 library to underpin its projection support. For those interested in how to do this, the links to articles in appendix A on spatial reference systems and POJ.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 their importance, as well as the general rules of thumb for selecting one and determining which ones your source data is using.

In the next chapter we'll continue our journey into the real world by loading real geographic data. We'll 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'll go over the pros and cons of each as well as provide examples of 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'll cover how to load real data and export it. You can find free geographic data to load in your database in numerous locations. Geographic data that covers large areas on 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 determine which ones are suitable for your use case. We hope the fundamentals of these we provided in the last chapter are sufficient to help you handle working with real data.

Before we begin our exercises, you need to know where you can get free data to play with. Locations for free data can be found in appendix A. In the next section, we'll 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

Many tools are available for getting data into and out of PostgreSQL/PostGIS. For this chapter, we'll focus on only the more common free and open source tools.

7.1.1 PostgreSQL built-in tools

PostgreSQL has some built-in command-line tools that are useful for getting data into and out of PostgreSQL.

- *psql*—PostgreSQL psql is a command-line tool that has both a non-interactive and an interactive interface.
- In the interactive connection you can use the `\copy` command to load comma and tab-delimited data. The built-in `copy` in psql copies from/to the client's file system (the machine from which psql was launched).
- The non-interactive mode allows you to load data in batch mode and also to run .sql scripts. The PostGIS shp2pgsql tool we'll cover shortly also relies on psql to silently execute the generated SQL.
- *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 run only on machines with a graphical interface, such as Mac OS X, Windows, Linux/BSD, Unix with Gnome, or KDE. It has similar functionality as psql but does not support a client-side `\copy` command. You can only use the SQL `COPY` command. Via the pgAdmin interface, you can launch a psql session, and it will automatically fill in the credentials of the database you're connected to.
- *pg_dump/pg_dumpall/pg_restore*—If you ever want to distribute large amounts of data, or you just need to do simple backups and restores to other databases, then these are the tools you'll want to use. They'll dump out data, even that of a spatial nature, and can even run in a compressed format to save space. You can then use pg_restore to restore these tables, functions, and so on 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 three 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 command-line tool is used to import both plain dBase files as well as ESRI shapefiles. It has good support for converting dBase datatypes to PostgreSQL ones, but it lacks transformation capabilities. It can also output to an intermediary .sql file for processing at a later step. It gives you the option of either

maintaining the case of field names or lowercasing them to remain within the PostgreSQL standard, where all fields are lowercased. You also have the option of just importing DBF files or the DBF file part of an ESRI shp/dbf combination.

- *shp2pgsql-gui*—This is the graphical wizard version of the shp2pgsql tool, introduced in PostGIS 1.4 as a separate download. As of PostGIS 1.5, it's also available for Windows users as part of the StackBuilder installer. You can use it to import both plain dBase files and ESRI shape files. It's 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 a command line. The downside is that it isn't scriptable. You can also install it as a plug-in in pgAdmin III by editing the plugin.ini file. Check our instructions for details: http://www.postgresonline.com/journal/index.php?/archives/145-PgAdminShapefilePlugin.html. shp2pgsql-gui is a GTK-based tool and isn't always included in the package.
- *pgsql2shp*—This is a command-line tool to output PostGIS spatial data to ESRI shapefile format. It also outputs the .prj format (spatial reference system file) as of PostGIS 1.3.6. It can output an ad hoc query as well as a view or table. The ad hoc queries can be of any complexity. While it's designed for spatial data, it can also be used to output non-spatial data to dBase DBF if no geometry field is included. It's fairly lightweight, so it's handy for creating web apps that can output queries in ESRI shapefile format. We'll cover this usage 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.bqg.

7.1.3 OGR2OGR: all-purpose vector data loader

OGR2OGR is the Swiss army knife for vector data. Its companion is the Geospatial Data Abstraction Library (GDAL), which is used for loading and outputting raster data. OGR is also often referred to as GDAL/OGR because OGR is nowadays 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/Mac OS X.
- 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 (including spatial), SQL Server (including spatial), 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).
- Fairly lightweight though not as light as shp2pgsql or pgsql2shp. Although there is an install, you can for the most part use it just by copying the needed

binaries and so on into a folder and running from the folder. This allows it to be called from web applications and anywhere else without an install.

- Supports transformation. If you're not content with the native spatial reference of a data source, you can transform it into a different one as part of the import process.
- Can do ad hoc SQL queries, though when doing ad hoc SQL, the field format is even more impoverished: Not even the data field lengths of text fields are respected. So again, for outputting to ESRI shapefile format, pgsql2shp is usually a better choice.
- Can perform batch importing by specifying a folder or a database.
- Can often guess at spatial reference from the .prj file or for MapInfo from the built-in projection information.

Weaknesses:

- Not packaged with PostGIS so it's not always available, though you can download the source or the precompiled binaries. There's a nice install for Windows users. The Linux precompiled binaries tend to be a bit out of date, but the Windows binaries are almost always up to date.
- Not as lightweight as pgsql2shp or shp2pgsql, so if you just need to output ESRI shapefiles in a web application, 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 OGR2OGR (as of this writing at least). OGR2OGR seems to like to bring in dBase fields as character instead of varchar (which technically they are, but it's far from ideal in most use cases).
- Even for non-spatial data, OGR2OGR insists on registering the table in geometry_columns.
- The options it offers are overwhelming and sometimes hard to figure out.
- Kind of weak with text encodings, though this varies from driver to driver.

GDAL export PostGIS Raster

The latest version of GDAL and the Windows FWTools 2.4.6+ binaries come packaged with a driver called PostGIS WKT Raster (PostGIS Raster in later versions). This allows you to export data from the PostGIS raster type into other raster formats using the gdal_translate executable. We'll be covering this in the PostGIS Raster chapter.

One tool that looks promising is *ogr2gui*. This tool offers a GUI to guide you through OGR2OGR and generates the appropriate command-line statement at the end. This tool is available at http://www.ogr2gui.ca. It doesn't currently support all formats

OGR2OGR supports, but the latest version as of this writing (0.7) supports PostGIS, Oracle, and SQLite. If you are using the precompiled Windows binaries, you need to download both the binary executable and the ogr2gui_dll and put the files in the same folder. For other OSes, you need to compile it yourself.

7.1.4 Quantum GIS Shapefile to PostGIS Import Tool

Quantum GIS is a common free desktop tool used for viewing and editing geometry and raster data. It has lots of Python scripting capabilities and good integration with other tools such as Geographic Resources Analysis Support System (GRASS). It works on Windows/Linux/Unix and Mac OS X. We'll cover it in our later section on desktop tools. In addition to its other features, it has an easy-to-use GUI import plug-in, called *Shapefile to PostGIS Import Tool* (SPIT). The screenshot shown in figure 7.1 is fairly self-explanatory. It's similar in concept to the shp2pgsql-gui tool and probably a good one to use if you're a heavy Quantum GIS user.

In the screenshot we're using SPIT in Quantum GIS 1.5.0 to demonstrate how to import two files simultaneously. The strengths and weakness of SPIT are as follows:

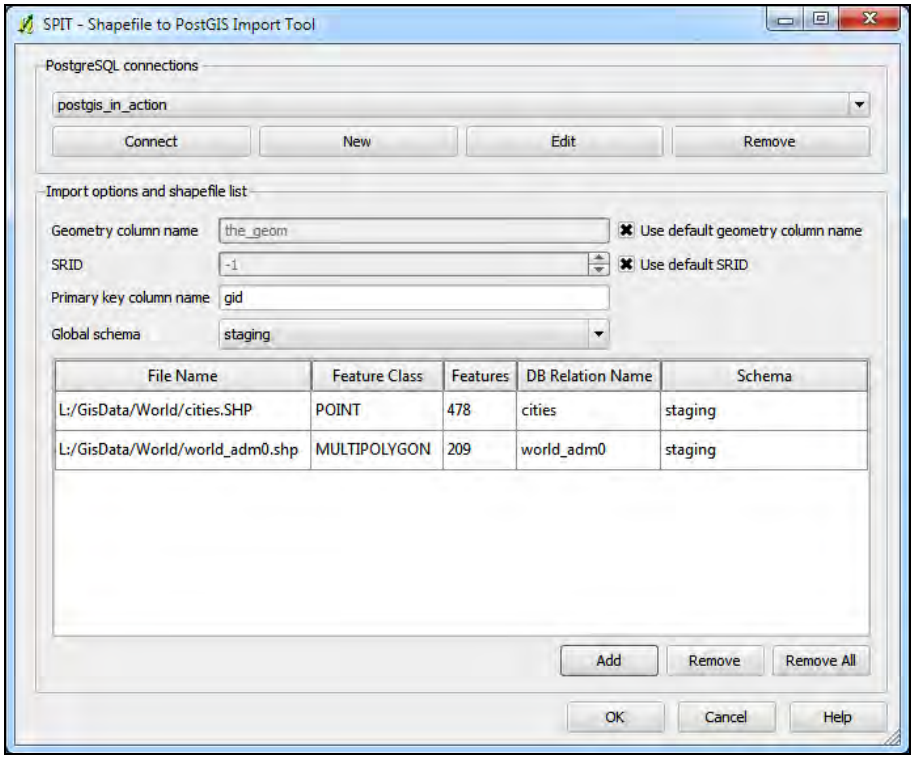


Figure 7.1 Quantum GIS Shapefile to PostGIS Import Tool

Strengths:

- Supported on Windows/Linux/Unix and Mac OS X.
- You can add multiple files at once by clicking the Add button, though all the files you add must have the same spatial ref.
- Gives a brief summary of each file before load, such as geometry count and type.

Weaknesses:

- Not packaged with PostGIS; you need to install QuantumGIS.
- No support for transformations.
- Supports only ESRI shapefiles, and can't guess at SRID from reading the .prj as OGR2OGR can. In many cases, you'll load data in its native form, but often, depending on your use case, you'll want to store your data in a different spatial reference system and do some additional spatial massaging. In either case you'll need to know the spatial reference system of your input data and the pros and cons of keeping it in its original form. See the section on spatial reference systems for how to figure out the SRID of your input data.
- As of this writing, it brings in all field names as uppercase (essentially the way DBF files are set up), and there doesn't seem to be a way to disable this. This is terribly annoying because in PostgreSQL non-lowercase field names need to be quoted when used in SQL statements. Hopefully this will change in future versions because there have been many complaints about it.

  One workaround for this problem is outlined at the following website, though be forewarned the solution involves updating system tables, which is generally a bit risky because you can't be guaranteed that tables won't change from version to version, and you could very well screw up your database: http://workshops.opengeo.org/stack-intro/postgis.html#postgis.

  An alternative and safer approach is to generate Data Definition Language (DDL) to correct the problem, as demonstrated in the following listing.

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 like the following that you can quickly inspect and run:

```
ALTER TABLE staging.statesp020 RENAME "AREA" TO area;
ALTER TABLE staging.statesp020 RENAME "PERIMETER" TO perimeter;
```

7.1.5 osm2pgsql: OpenStreetMap to PostGIS loader

OpenStreetMap (OSM) 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 (Bing)) but also can import 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 (or for speed).

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 a new license, the terms of which are still under discussion, called the *Open Database License*. This new license is more specifically geared toward data sharing. In the next section of this chapter we'll cover how to carve out specific areas of OpenStreetMap data and download them in OSM XML format and how to import this file into your PostGIS-enabled database.

Now that we've covered the more common free options available for loading data, in the next section we'll test these tools.

7.2 Loading data

In this section we'll go over some real use cases with the aforementioned tools and focus on loading data. We'll start off with the built-in PostgreSQL/PostGIS tools and use them to load an ESRI shapefile.

Before starting we'll create a few schemas to hold our data. In real-world scenarios, you'll want to create several schemas to logically partition your data. The PostGIS tables, such as geometry_columns and all the PostGIS functions, can remain in the public schema. These will be shared across all schemas data resides in.

For starters, we'll create some schemas to use later on. The us schema holds our U.S. data, canada holds our Canada data, and staging holds our temporary information. The following exercises assume you've created a spatially enabled database called postgis_in_action, and we'll use that to store all our data.

```
CREATE SCHEMA us;
CREATE SCHEMA canada;
CREATE SCHEMA staging;
ALTER DATABASE postgis_in_action SET search_path=public,"$user",us,canada;
```

Use non-public schemas for your data and custom functions

It's a good idea to get into the habit of using your own custom schemas for your data instead of throwing everything in the public schema. If you build many specific custom functions, it's better to store them in custom schemas or even create a schema to specifically hold functions. By using named schemas for your own custom functions and data, upgrading to newer PostGIS versions will be much easier, and so will restoring selective data and functions to another database.

We think that partitioning your data and functions in logical sections is a good idea; however, we don't think it necessary to schema qualify commonly used things when querying. In fact, often it's even better not to, for ease of use and portability. To avoid qualifying database objects with schema names, make sure to add commonly used schemas to the database search path, as we discussed in the previous `ALTER DATABASE` command. Less-often-used schemas or schemas for temp data like our staging schema should always be schema qualified and left out of the database search path. Let's start by getting and extracting compressed files with WGET and Windows 7-Zip.

7.2.1 Getting and extracting compressed files

For downloading files on all systems, we recommend using the command-line tool Wget. For extraction on Windows, we recommend 7-Zip.

DOWNLOADING FILES

Wget is a command-line tool for grabbing files from the internet that generally comes prepackaged with Linux/Unix systems. It can be downloaded for free for Windows as well from http://gnuwin32.sourceforge.net/packages/wget.htm. Get the binaries and the dependencies, and extract them to same folder.

If you're on Windows or any OS with a GUI, you can also download files using your browser; however, you may still find Wget handy for automating the download of many files.

The following demonstrates use of handy command-line switches for use with Wget. These apply to both Unix/Linux and 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
```

This 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 your internet connection; you can just pick up where you left off.

If you just need to pull down a single file, use the following:

```
wget  http://www2.census.gov/geo/tiger/TIGER2008/tl_2008_us_zcta500.zip
```

This will put the file in your current directory. It won't create subfolders like the aforementioned mirror example. If you're downloading from an FTP site, you can also use a wildcard such as `*zcta500.*` to pull down multiple files with the same command.

EXTRACTING FILES

Most files you'll download are compressed in tar.gz or zip format. Most Linux systems have command-line tools to extract these files. We'll quickly go over the basics for

those new to Linux. Table 7.1 has some common commands to extract the most common types of files.

Table 7.1 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 (two variants).
`tar xvfz somefile.tar.gz`
`gzip -d -c somefile.tar.gz | tar xvf --`

For Windows users, we recommend the 7-Zip extract/compress tool. 7-Zip is free for both personal and commercial use and can extract all the aforementioned formats plus more. For simple .zip files, you can also use the built-in uncompress in Windows. We've found the 7-Zip uncompress/compress to be better than the built-in Windows tool because it can handle compressing/extracting files over 4 gigs and gives you many more compression options such as password protection and level of compression. You can download 7-Zip from http://www.7-zip.org/, and after you install it, you can right-click a file in Windows Explorer and choose to extract it 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 that's useful for automating zip/unzip processes. The command-line interface is the 7z.exe file. To make this portable, you can copy the 7z.exe and 7z.dll files to a floppy disk or USB or folder and use them from anywhere without doing an install. Table 7.2 has some simple tips for using the 7z command-line interface.

Table 7.2 Example uses of the 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`

Table 7.2 Example uses of the 7z command line *(continued)*

Example uses
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`

Next we'll look at tools to load data.

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'll walk through the following tasks:

- Load single ESRI shapefile and DBF files with the shp2pgsql command line
- Quickly demonstrate 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 these:

- `-s`—The spatial reference system. If you're dealing with geographic data, you should always provide this, especially if you're going to be loading data from various sources.
- `-W`—The encoding. The default in pre-2.0 versions is ASCII and in later versions it will be UTF-8. The default will work in many cases, but sometimes it will let you silently lose data. This happens when you see a blip screen going by saying, "Failed encoding incorrect something or other." If your data has Spanish, Italian, French, German, or other diacritical marks, a better choice is LATIN1. LATIN1 tends to work for most data sets we've come across, and even if the data is coded in ASCII, it's generally harmless to specify LATIN1 encoding. Some data really is in UTF-8 and should be imported with that encoding because higher-bit UTF-8 will give errors with any other encoding.
- `-I`—Creates a spatial index on the geometry column. This is useful if you aren't going to append data to this table in the future, and you want to add a spatial index immediately after loading.

For this section we'll load an ESRI shapefile data set into PostgreSQL using the command-line shp2pgsql loader. For this exercise we've chosen the following data source:

state boundaries from the U.S. Geological Survey Earth Science Information center, located at http://edcftp.cr.usgs.gov/pub/data/nationalatlas/statesp020.tar.gz.

1. Extract this file using your tool of choice.
2. You should end up with statesp020.dbf, .shp, .shx, .txt (ESRI shapefiles 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).
3. Figure out the spatial reference system. For this dataset we know the spatial reference system of this data is NAD 83 lon lat, which has an EPSG code and PostGIS SRID number of 4269. How we arrived at this conclusion and how you can determine the same for your data we describe later in this chapter.
4. Next, load up your data using shp2pgsql. We will load the states data into the schema called *us* that we created earlier. For Windows users, you'll 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 this:

   ```
   cd "C:\Program Files\PostgreSQL\8.4\bin" (just for windows users)
   ```

   On Linux installs, the bin folder ends up being in the path, so you can simply use the binaries without specifying the full path. The following code 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
   ```

   - Here we're loading up our statesp020 data that we extracted previously into a folder called C:\GISData and put it in a new table called statesp020 in the staging schema. The shp2pgsql tool will use both the .shp and .dbf files and store the .dbf attributes in common PostgreSQL field types and the .shp geometry information in a field called the_geom_4269, as we indicated with the `-g` switch. If you don't specify a `-g` switch, then the geometry is stored in a column called the_geom. We prefixed our table with the schema that we're loading into. Note that the `shp2pgsql` part of the command simply generates the SQL statements needed to create the table, to add the geometry column and register it in the geometry_columns table, to insert the data into the table, and to add the index. It doesn't really do anything to the database; that work will be done by the `psql` command. For this example, the index is superfluous because we'll be throwing this table out once we're finished with it.
   - The second part beginning with the pipe (`|`) actually does the loading. This part works equally well on Windows as it does on Linux. It pipes the SQL output generated by `shp2pgsql` to the `psql` command for further processing.
   - If you don't want the data loaded but only store the .sql file for future loading, you'd do the following instead:

     ```
     shp2pgsql -s 4269 -g the_geom_4269 -I -W "latin1"
     "C:\GISData\statesp020" staging.statesp020 > C:\GISData\statesp020.sql
     ```

Dumping to an intermediate .sql file is convenient if you have bad data items that will screw up the loading or if your shapefile is so large that the data can't fit into memory before it's piped to psql. When you're finally ready to perform the actual loading, run the following line:

```
psql -h localhost -p 5432 -d postgis_in_action -f C:\GISData\statesp020.sql
```

This raw data format has some issues not suitable for our use case: It's in lon lat, so it's not suitable for measurement; it has over 2,000 records and we'd prefer one for each state; and it's sufficiently dense and more precise than we need. To solve these issues, we take our staged data and dump it into our us schema; see listing 7.2.

It's best to use PostGIS 1.4 and above

You really shouldn't do the exercise shown in listing 7.2 if you're running a version 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're lucky and don't run out of memory. In contrast, in PostGIS 1.4+, because of the significantly improved speed of unioning many polygons, this will take about 26 seconds or less.

Listing 7.2 Converting data from native format to more optimized format

```
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
);

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,
    ST_Multi(
    ST_SimplifyPreserveTopology(
        ST_Union(
            ST_Transform(the_geom_4269, 2163)
            ),
            700
        )
            ) As the_geom
FROM staging.statesp020
GROUP BY state, state_fips, order_adm, month_adm, year_adm;
```

1 Create table

2 Create geometry column

3 Convert to multipolygon

4 Simplify every 700 meters

5 Dissolve boundaries

6 To equal area meters...

7 ...simplify every 700 meters

8 Group non-agg columns

In this code listing, we're ❶ creating a new table to store our United States records. ❷ We add the geometry column in order to give it constraints and to register it in the geometry_columns table. PostGIS 1.4+ does provide additional ways of doing this. ❻ 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 ❺ ST_Union in combination with a ❽ SQL GROUP BY clause to dissolve boundaries between records of the same state so as to end up with one record per state. This will make our resulting table contain only 53 records instead of the original 2,895 records we had in our staging file. After we've finished unioning, we simplify the new geometries ❹. ❼ The value of 700 in this case is in meters (because since simplification is happening after the transform and union operations and our geometry is in SRID 2163 space). We're telling PostGIS to remove vertices until all remaining vertices are at least 700 meters apart. Simplification is useful for many situations, but it 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 the download speed as fast as possible. Moreover, it makes processes such as ST_DWithin and ST_Intersects faster (and in some cases a lot faster) because they have fewer vertices to deal with. It's important to simplify in planar space and not in lon lat space because the same degree changes in different latitudes result in different lengths, which can lead to very choppy simplification. If you want to keep your data in lon lat, you should transform to planar space, simplify, and then retransform back to lon lat space. After that we apply ❸ a PostGIS ST_Multi operation. Recall from prior chapters that this converts a polygon to a multipolygon. We do this because some states after unioning will be polygons and some will be multipolygons, so to maintain consistency we make them all multipolygons. In short, we've found this to be the best order in which to apply our operations to ensure good-quality data. A number of operations even 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's often fast enough to simplify on the fly and base your simplification tolerance on how far the user zooms in to the features. For large datasets and for indexing reasons, you may want to keep prebuilt simplified versions of the data as well as the original high-grained data.

Once you've finished loading the data, it's a good idea to at least put a spatial index on the new data and possibly other indexes you know you'll commonly use in the WHERE clauses of your SQL statements. The exercise in the following listing should be more or less a refresher course from past chapters.

Listing 7.3 Putting in indexing and preparation for querying

```
CREATE INDEX idx_us_states_the_geom
    ON us.states USING gist (the_geom);

ALTER TABLE us.states
  ADD CONSTRAINT pkey_us_states_state_fips PRIMARY KEY(state_fips);

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);

VACUUM ANALYZE us.states;
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're using state_fips because a lot of U.S. data comes with this code. Alternatively, you can use the state code. 3 Here we add a dummy key (which is our serial key) and make it unique. We do this to appease the many GIS tools that require a key to be an integer. You might need to make this a primary key for some tools, which will annoy many database purists because it has no business meaning (you should either explain why it doesn't fit into database theory or leave it out). Next, we add other keys to use in queries. We emphatically suggest you to do this step *only* when you've started doing queries against this table and done some benchmarking about the way those queries work. What we're doing here 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 that a spatial index will always be needed, so we always add it. We can do the btree indexes at an early moment because it's a static table (it 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 can be a bother because processing/IO time is spent updating them with every data change. 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. This will drop the table and remove all entries of it from the geometry_columns table.

QUICK DEMO OF SHP2PGSQL-GUI

Figure 7.2 is a screenshot of what the shp2pgsql-gui looks like when we load the states table, as we did in the command-line version.

PostGIS shp2pgsql-gui in PgAdmin III

The PostGIS shp2pgsql-gui is designed so that it can be deployed as a plug-in 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.

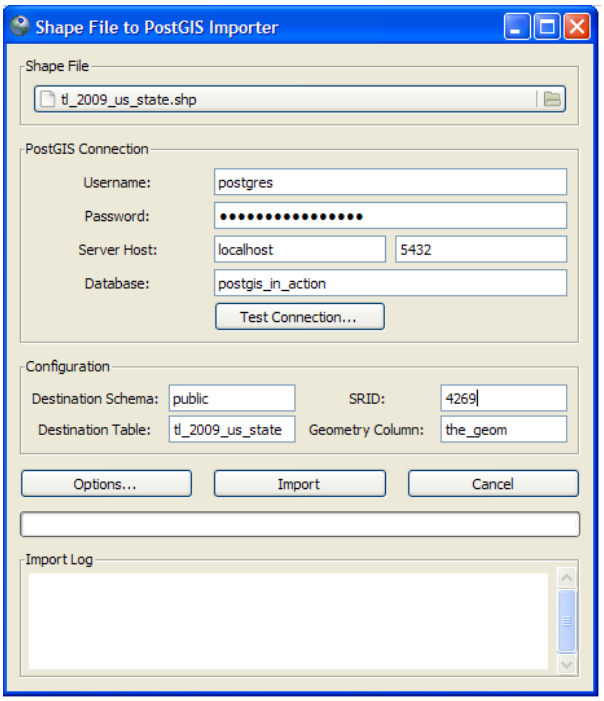
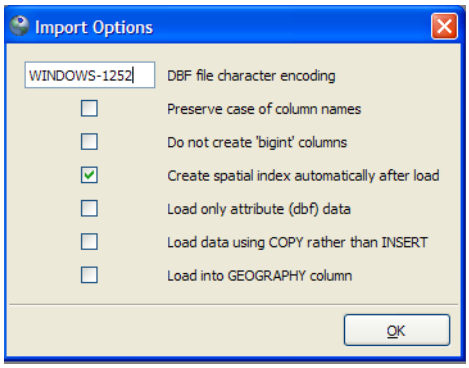


Figure 7.2 Using shp2pgsql-gui to load the states table

Figure 7.3 shp2pgsql-gui Import Options dialog box showing the advanced options



In this figure we've browsed to the states table using the browser icon and filled in the relevant information. Next we'll click the Options button to verify the other settings (figure 7.3). The character encoding of many shapefiles is usually LATIN1 or WINDOWS-1252 (a variant of LATIN1 with additional Windows characters).

> **PostGIS 2.0 shp2pgsql-gui enhancements**
>
> In PostGIS 2.0, the GUI is enhanced to allow loading of multiple files at once, similar to the QGIS SPIT plug-in.

In this section we covered how to load spatial data with the prepackaged tools provided with PostGIS. In the next section, we'll demonstrate using OGR2OGR to load spatial data from various different kinds of spatial data sources.

7.2.3 Loading data with OGR2OGR

As we mentioned in our quick survey, although shp2pgsql is fine for loading ESRI shapefiles and dBase files, that's all you can use it for. If you have MapInfo, GPX, ODBC, MySQL, SQL Server, ESRI Personal GeoDatabase, or AutoCAD files (to mention only a few), then OGR2OGR will do the trick for you.

OGR2OGR is supported on Linux/Unix as well as on Windows and Mac OS X. You can download the version for your particular operating system from http://trac.osgeo.org/gdal/wiki/DownloadingGdalBinaries.

Formats supported are listed here: http://www.gdal.org/ogr/ogr_formats.html. Formats that require proprietary DLLs such as Oracle Spatial, ESRI ArcSDE, and FME are not compiled by default and require you to compile them with the dependent libraries.

A common package that contains OGR2OGR is called FWTools. It offers a version for both Linux and 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 shown in the listing 7.4.

Listing 7.4 OGR2OGR supported formats list

```
"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'll demonstrate loading data from GPX, ESRI Personal GeoDatabase (which is stored as an MS Access database), and MapInfo to PostgreSQL.

OGR2OGR is an extremely rich tool, especially given its small size. We'd have to devote a whole book to it to do it justice. We hope the samplings we picked are the most common use cases you'll 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 installation for Windows: http://www.bostongis.com/PrinterFriendly.aspx?content_name=ogr_cheatsheet

Before we begin our OGR journey, we outline some options that are specific to working with the OGR PostgreSQL driver. They're useful regardless of what data source you're 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 following are copied from the official

OGR2OGR documentation, with some additional comments and some rarely used options left out. Full details can be found here: http://gdal.org/ogr/drv_pg.html.

- `GEOM_TYPE`—The `GEOM_TYPE` layer creation option can be set to either Geometry, BYTEA, or OID to force the type of geometry used for a table. In general there's no need to set this.
- `LAUNDER`—This may be set to 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 lowercase and converts some special characters like - and # to _. If it's set to NO, then the exact names are preserved. The default value is YES. If enabled, the table (layer) name will also be laundered.
- `PRECISION`—This may be set to YES to force new fields to be created with the available width and precision information, using NUMERIC(width, precision) or CHAR(width) types. If set to NO, then the types FLOAT8, INTEGER, and VARCHAR will be used instead. The default is YES.
- `GEOMETRY_NAME`—Sets the name of the geometry column in a new table. If omitted, it defaults to wkb_geometry. Use `-lco` to override, as shown here:

  ```
  -lco GEOMETRY_NAME=the_geom
  ```

- `SCHEMA`—Name of schema for new table. Using the same layer (table) name in different schemas is supported.

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 choosing Control Panel > System > Settings > Advanced and clicking Environment Variables, or you can set it in an import batch script with the `set` command:

```
set PGCLIENTCODING=latin1
```

- `PGCLIENTENCODING`—This is really a PostgreSQL environment variable but it overrides OGR's default of UTF-8. 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 should be set to YES to use the `COPY` command 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've found it not only faster but also 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 grained 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 Lon Lat, which has a 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.

OpenStreetMap is full of user-contributed GPX files that are uploaded by users about every minute. You can find these at http://www.openstreetmap.org/traces. We randomly selected one from Australia titled "A bike trip around Narangba" by going to http://www.openstreetmap.org/traces/tag/australia and downloading the file 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. The following listing shows what we get when we enter the command `ogrinfo 468761.gpx`.

Listing 7.5 Displaying ogrinfo about GPX file

```
ogrinfo 468761.gpx                          <-- Command

Had to open data source read-only.          <-- Results
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)
```

We'll now load this up into our staging schema with the simple OGR2OGR commands shown here.

Listing 7.6 Loading data from GPX

```
ogr2ogr -f "PostgreSQL"                     <-- 1 Single table load
➥ 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"

ogr2ogr -f "PostgreSQL"                     <-- 2 Multi table load
➥ PG:"host=localhost user=postgres port=5432
➥ dbname=postgis_in_action password=mypassword"
➥ 468761.gpx -overwrite -lco GEOMETRY_NAME=the_geom
➥ -lco SCHEMA=staging tracks track_points
```

1 This does a simple load into a new table called staging.aus_biketrip_narangba. This table contains all the layers and so has lots of blank fields to accommodate the attributes

of the various layer types. We also specify the geometry_name column field. If you leave this out, geometries are stored in a field called wkb_geometry. ❷ In the second approach, we're taking the same GPX file but breaking it into separate tables by feature type. We're also pulling only a subset of the layer types available. Many others in this GPX file are empty.

In the next exercise we'll 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 metadata tables added to maintain information about these geometries. 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's reaching obsolescence, however, and is slowly being replaced by ESRI's File Database format, which can handle larger file sizes but is more proprietary (nonpublished standard), and few tools aside from ESRI-made ones know how to do deal with it. OGR2OGR currently supports reading of ESRI's Personal GeoDatabase but not the new File Database format. For this exercise we'll download the Personal GeoDatabase of world administrative boundaries from here: http://www.gadm.org/data/gadm_v1_mdb.zip (this is a 504MB zip file, which extracts as an MDB). To get a catalog of what's in this Access MDB, we use the ogrinfo tool:

```
ogrinfo gadm_v1.mdb
# result below
INFO: Open of 'gadm_v1.mdb'
      using driver 'PGeo' successful.
```

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 the `ogrinfo` clause, as shown in listing 7.7. Because this is a large database, it is doesn't load instantaneously. We're using the `-so` switch to let OGR know we want only summary data; we don't want to see the data in the records. Set `-geom=NO` if you don't want the data to output the geometry column.

Listing 7.7 Use `ogrinfo` to list fields for a Personal GeoDatabase layer

```
ogrinfo gadm_v0dot9.mdb -so -geom=YES gadm

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]],
```

❶ Command

❷ Results

```
    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)
```

From the command ❶ the result ❷ is the names of the fields, their sizes, and also the spatial reference system of the data (WGS 84 Lon Lat, our familiar SRID 4326). It also tells us that the geometry is of a mixed type, not all polygons, linestrings, and so on.

We 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
```

Here we're performing a couple of things: We're selecting just USA boundaries with the `ISO='USA'` where clause, and we're transforming from the native spatial ref of the data EPGS:4326 to our preferred EPSG:2163 for this subset. We're 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 information to guess at the source spatial reference system, so we don't have to provide it. In many cases you may need to provide `-s_srs "EPSG:4326"` or whatever the native is so OGR can transform correctly.

If we had tried to load the full dataset, we might have run into errors because of the various languages the text are in. We'd 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. Therefore, we have to fiddle with environment variables or set our database to client_encoding LATIN1 or some other relevant encoding while we're loading.

EXERCISE 3: LOADING A MAPINFO TAB FILE AND FOLDER OF MAPINFO FILES

Another popular format is the MapInfo tab file format, which has spatial reference info built into the file format. Unlike the ESRI shape format, it can have multiple kinds of geometry types in the same file, and field names can be upper/lower/mixed case and aren't limited in length to 10 characters as in DBF files. It also allows storage of a lot of cartographic formatting (which is ignored by OGR2OGR). For this exercise we'll pull a file from Statistics Canada, http://www.statcan.gc.ca/mgeo/boundary-limite-eng.htm, download its Population Ecumene Census Division Cartographic

Boundary File, and choose MapInfo tab format. This zip file contains several tab files. For this exercise we'll demonstrate loading a whole folder of files, which is one of the beautiful features of 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 example we explicitly specified the source spatial reference. If we hadn't done this for this particular file, then OGR2OGR would create 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 something to do with the MapInfo driver. If you look at the proj4text entry, you'll see it's identical to the entry for 4269, which is why we must force it.

7.2.4 Importing OpenStreetMap data with osm2pgsql

The OpenStreetMap export format 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, available at http://www.openstreetmap.org/export/, to carve out a section of space and export just that section. Various other command-line tools are available for working with OSM data. One that's specific for working with road networks and the PostGIS pgRouting add-on is osm2pgrouting, which you can download from http://pgrouting.postlbs.org/wiki/tools/osm2pgrouting.

In our chapter 3 Paris example, we used this interface to export regions of Paris to load into our database. To export a region of space, follow these steps:

1. Go to http://www.openstreetmap.org/export/ and type in the lon lat block you want or draw a box on the map.
2. Select a region encompassing the Arc de Triomphe.
3. Choose the following BBOX: 2.28568,48.87957,2.30371,48.8676.
4. Select as export format OpenStreetMap XML Data. We called ours arctriump.osm.

Once you have a .osm-formatted file, you can load it 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.

Osm2pgsql has numerous other options we won't explore, such as on-the-fly projection using the `-E` switch, importing as lon lat with the `-ll` switch, and so forth. You can get a listing of all options by calling

```
osm2pgsql -h
```

The rest of the steps are more or less the same regardless of which OS you're on. Note that psql is located in the bin folder of your PostgreSQL install.

Load the 900913 (Web Mercator) spatial reference into your database with the following command using psql:

```
psql -f 900913.sql -d mydb -U postgres -p 5432
```

If this spatial reference system is in your database, you'll 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 that work on it have been added to the mix as well.

To install you need to run the hstore.sql file that's located in your PostgreSQL /share/contrib folder:

```
psql -f hstore.sql -d mydb -U postgres -p 5432
```

Now you're ready to load your .osm-formatted data into PostgreSQL. The commands for user name, PostgreSQL port, and database are pretty much the same as for psql, except that for port (which is only really needed if installing in a PostgreSQL database that's not on the standard port), the switch is uppercase `P` instead of lowercase `p`.

```
osm2pgsql arctriump.osm -d postgis_in_action
➥ -U postgres -P 5432 -S default.style --hstore
```

Once you've finished, you should see a bunch of tables created in the public schema that start with planet_osm.

> **Appending versus overwriting with osm2pgsql**
>
> By default, OSM will overwrite the tables and create them fresh. If you're appending multiple OSM files at different points in time, you'll want to use the `--append` switch to switch to append mode. Note that you can process multiple files at once by separating the filenames with spaces, for example, file1.osm file2.osm ....

READING HSTORE TAGS

If you used the `--hstore` flag as we did previously, each table should have a column called tags that 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 schema-less 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, 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. You'd write a query such as the following:

```
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 the columns osm_id, key, and value. You'd get a record for each key-value pair. So if you had 10 entries in each tags column, you'd 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'd create against PostGIS geometry/geography/raster columns.

As we've demonstrated, there are numerous open source tools freely available for getting data into your PostgreSQL/PostGIS tools. Many of these tools grew up alongside PostGIS, and so the PostGIS free importer tools are often more tested and functional than what you'll find for other spatial databases. While it's easy to get data into your PostgreSQL/PostGIS database, it's just as easy to export data into various formats. In the next section, we'll demonstrate how to export data from your spatial database into formats consumable by various GIS desktop tools and other spatial databases.

7.3 Exporting data from PostGIS

A database is only as good as the information you can get out of it and the data you can share with others. You can use various tools 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 is at your disposal:

- PostgreSQL has a myriad of `copy` commands. One that's part of its SQL offering will dump data to a server file in a location that the postgres process has access too. Only the super admin can use this command. Another `copy` command is part of the psql command-line interactive client: It will dump the file to the client workstation and doesn't require admin rights. In addition, psql has some handy features to create text and HTML reports for regular tabular data. Because this is a book about spatial data, we won't focus on this, although it's useful for 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 attributes) as ESRI shapefiles or plain DBF files. It is fairly powerful and lightweight. We have demonstrated this tool.

- As we've discussed already, OGR2OGR can also import PostGIS and regular PostgreSQL attribute data to various formats. To use it for export, reverse the procedure and make PostgreSQL the from source instead of the to source.

In this section we'll 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 lightweight candy dispenser. It needs only a couple of files to be functional (libpq plus libpq dependencies and pgsql2shp binary) and can output any spatial query to an ESRI shapefile format. It's 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, pgsql2shp does the following:

- It outputs the following related ESRI files (.dbf, .shp, .shx, .prj; you can see all related files by using the `-f` option, for example, `-f streets` outputs streets.shp, streets.dbf, streets.prj, and streets.shx). The .prj file is output only if the projection is known, for example, if you didn't use an SRID of -1 or 0 and all the geometries you're outputting are of the same spatial ref.
- If you output a table or query with no geometry column, it outputs a .dbf file.
- It truncates field names that are too long (greater than 10 characters per DBF standard) and numbers any duplicates.
- It truncates large text fields greater than 255 characters because the 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 common use case.

This first snippet exports a whole table called zips in the ca schema of the database gisdb to a file called cazips.*. (It will create cazips.shp, cazips.dbf, and cazips.shx, and for PostGIS 1.3.6+ packaged pgsql2shp, it 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. Further flags are needed if your PostgreSQL server doesn't run on the standard port or you want to authenticate as a specific user.

```
pgsql2shp -f /gisdata/cazips –h localhost –u pguser
➥ –P somepassword -p 5432 gisdb ca.zips
```

Although exporting whole tables is a common need, for large tables you may want to export only a portion of a table or even a complex query.

EXERCISE 2: EXPORTING AN AD HOC QUERY

The next example demonstrates outputting the results of ad hoc queries:

```
pgsql2shp -f boszips –h localhost –u postgres gisdb
➥ "SELECT * FROM ma.zips WHERE city = 'Boston'"

pgsql2shp –f boszips localhost –u postgres gisdb
➥ "SELECT zip5, ST_Transform(the_geom, 4326) As the_geom
➥ FROM ma.zips WHERE city = 'Boston'"
```

We demonstrate here how to use pgsql2shp to output database queries. The first is a very basic query. The second includes an ST_Transform call. The first query will output a projection that's in the native format of the data. The second query will reproject the data and output it in WGS 84 Lon Lat, which is the most common of distribution spatial reference systems. 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 need to output data in other formats without programming, pgsql2shp won't do that for you. OGR2OGR, however, will in most cases be a good, free, lightweight tool for that purpose. It's not quite as lightweight as pgsql2shp, but it makes up for that by providing many more formats to choose from.

Like pgsql2shp, OGR2OGR allows you to output spatial queries as well as tables and views. The `-sql` switch seems much more finicky and not as robust as the pgsql2shp query, even with the PostgreSQL driver. It's hard to use randomly complex queries with it as you can with pgsql2shp. The `-sql` switch seems to have trouble figuring out data types, so in many cases you'd be better off creating a temporary view and outputting the view. We'll show various common formats in the following examples. One unique thing about OGR2OGR that's quite nice is that you can use it to output multiple spatial tables at once.

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 that is often combined with the `-select` switch.
- `-sql`—Useful if you want to output a more complex query than what the `-select` and `-where` combo offer, but the output column data types may not reflect the data types of your query columns.
- `-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. But if you have your data in unknown projection (SRID -1 or 0 or something not in the proj list of your OGR2OGR, then you'll need to specify the `-s_srs` switch, 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. It's useful if you have a nightly scheduled dump where you're constantly overwriting the same files.

EXERCISE 1: EXPORT TO KML

In the example shown in listing 7.8 we'll demonstrate how to output both a table and a query in the Keyhole Markup Language (KML) format using OGR2OGR. If you want fine granular control, you'll probably want to write your own export logic because KML is fairly easy to code 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.

The most important data-creation option in the KML driver is the NameField. This determines which field in your KML output is used for the KML title for each object. Note also 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 7.8 Export PostGIS table and query to KML

```
ogr2ogr -f "KML" /gisdata/us_adminbd.kml                          <-- 1 Simple export
➥ PG:"host=localhost user=postgres port=5432 dbname=postgis_in_action
➥ password=mypassword" us.admin_boundaries -dsco NameField=name_2

ogr2ogr -f "KML"                                                  <-- 2 Export of filtered set
➥ /gisdata/biketrip.kml PG:"host=localhost user=postgres port=5432
➥ dbname=postgis_in_action password=mypassword" -dsco NameField=time
➥ ñselect "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 Export as multiple tables
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
```

In this KML exercise, we demonstrate three approaches for using OGR2OGR to export to KML format: ❶ simple table/view export, ❷ filtered export that uses the `-sql` and `-where` switches of OGR2OGR, ❸ 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 ❸ in Google Earth, you'll see two 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'll demonstrate outputting to MapInfo tab format. Unlike the KML format, which is always in WGS 84 Lon Lat, MapInfo data can be in any spatial reference system. In many cases the spatial reference system the data is stored in is not the one you want to use to distribute the data. In the exercise in listing 7.9 we demonstrate how to

make OGR2OGR transform data. Although the MapInfo tab format isn't as popular as the ESRI shapefile format, it has a couple of advantages: It isn't constrained by field name lengths, and it can store more than one geometry type in a single tab file. An ESRI shapefile can have only one type (POLYGON/MULTIPOLYGON, POINT/MULTIPOINT, and so on), so you can't dump a mixed-geometry type table in that format. Fieldnames in ESRI shapefiles are also limited to 10 characters, as dictated by the DBF standard. This means that many of your field names may get truncated.

Listing 7.9 Export PostGIS table and query to MapInfo tab format

```
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

ogr2ogr -f "MapInfo file"
➥ /gisdata/biketrip.tab
➥ PG:"host=localhost user=postgres port=5432
➥ dbname=postgis_in_action password=mypassword"
➥ ñselect 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

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
```

❶ Export with transform

❷ Export with filter

❸ Export multifile

We performed similar exercises for MapInfo tab as we did for KML. ❶ For our U.S. admin boundaries output, we store the data in a national atlas projection, but we want to export it to WGS 84 Lon Lat. In ❷ we are outputting only a subset of the records and columns using the `-select` and `-where` switches. In ❸ we output two 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). This also creates the folder tab_files to store the files in. If this folder exists, the command will fail unless you add `-dsco overwrite=YES`.

7.4 Summary

In this chapter we demonstrated the use of the 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 and hope these exercises will provide you with the base knowledge to load and export your own data. In the chapters that follow, we'll focus on using PostGIS to solve real-world problems.

Part 2

Putting PostGIS to work

In part 1 of *PostGIS in Action*, we covered all the building blocks you'll need to solve spatial problems. By now you should be able to set up a PostGIS database, populate it with data, and be able to transform between disparate spatial reference systems. You should also be comfortable using the most common functions in PostGIS and be able to take advantage of their prowess when writing SQL. In part 2, we'll put the pieces together to solve real problems. The important lessons we want you to take away from part 2 entail how we tackle each problem, starting with building a correct formulation, setting up an appropriate structure to support the analysis, choosing the most appropriate PostGIS functions, and putting it all together using SQL. Chapter 8 covers various common problems that you'll 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 then dive into building PostgreSQL functions. For some problems, we'll demonstrate more than one approach to arriving at a solution.

Chapter 9 is about performance. Now that you're able to put together complex queries, you need to make sure they'll finish running in your lifetime. We'll teach you how to speed up queries and caution you against common SQL pitfalls. We'll cover the finer points of employing both spatial and non-spatial indexes and fine-tuning PostgreSQL settings. In addition, we'll demonstrate the often-neglected tactic of simplifying geometries to arrive at "good enough" answers to problems quickly rather than overly precise answers slowly.

8 Techniques to solve spatial problems

This chapter covers

- Using joins with spatial functions
- SQL aggregates and spatial aggregates
- 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'll 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 different 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'll tackle this chapter using prebuilt data combined with ad hoc generated data sets. If SQL is new to you, you may want to read appendix C, "SQL primer," which discusses the fundamentals of SQL. The SQL primer demonstrates SQL constructs applicable to many relational databases.

In this chapter as well as in the rest of the book, we'll no longer be shy about combining the strength of relational databases with spatial functions. We won't satisfy ourselves with asking if something X is related to something Y. We're going to query entire tables at a time and use the expressiveness of SQL joins with wild abandon. You'll learn the fundamentals of doing proximity analysis as well as various methods for processing geometries. Being able to combine and convert simple geometries into more complex or less complex ones is useful both for map rendering as well as a starting point for more analytical spatial operations. We'll also start to explore compartmentalizing 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 you can locate places using a set of coordinates, questions such as the following arise, which always involve some kind of distance calculation: How far is my house from the nearest expressway? How many pizza joints are within a mile drive? What's 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 as easily as you can relate things by numbers, dates, and strings. In this section we'll explore proximity relationships and demonstrate how you can use these to derive relationships that conventional SQL joins can't accomplish.

8.1.1 Check for intersections and measuring distances

We'll 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 almost any geometry. You can throw points, linestrings, or 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 data types of numbers and strings, and you'll be stuck.

> **ST_Intersects and geometry collections**
>
> The ST_Intersects geometry function currently doesn't work with generic geometry collections, but ST_DWithin does. To use it with geometry collections, you need to explode them with (ST_Dump(the_geom)).geom or just use ST_DWithin. Note that for PostGIS 1.5 geography, the ST_Intersects operator does work with geography collections because the geography implementation is distance based rather than intersection matrix based and also doesn't rely on GEOS.

We'll start with some common examples. For our exercises, we pulled some freely available data for the San Francisco Bay Area from DataSF.org. We'll start with an

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_nam, ST_Distance(c.the_geom, b.the_geom) As dist_ft
FROM sf.cities AS c CROSS JOIN sf.bridges As b;
```

We chose this example to specifically demonstrate that ST_Distance always returns the minimum distance between two geometries. You see this when both the 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+

The ST_MaxDistance function was introduced in PostGIS 1.5, along with various other distance functions such as ST_DFullyWithin and ST_ClosestPoint. This is all thanks to the work of Nicklas Avén. ST_MaxDistance is useful if you want to know the distance between the farthest point from the city to the farthest 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 of the geometries, and the geometries have to have the same spatial reference. Because our SF data came in feet, the resulting distances are all in feet.

ST_Distance for the geography data type

For PostGIS 1.5+, the geography data type also has an ST_Distance function, and although geography data is stored in WGS 84 lon 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 in prior versions).

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_nam
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, although useful and fast, simply gives true or false as an answer; it only checks to see if the minimum distance between two geometries is zero or positive. If you want objects that fall within a certain radius of another, then ST_DWithin is much more useful.

In the example in listing 8.1 we'll locate all bridges that are within 1000 feet of San Francisco.

Our San Francisco data is in feet, therefore our unadulterated distance check is done in feet. The ST_Distance function is an often-used companion function to ST_DWithin.

Listing 8.1 Basic ST_DWithin query

```
SELECT DISTINCT c.city, b.bridge_nam,
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';

SELECT ST_Buffer(ST_Union(c.the_geom),1000) As sf_1000ft
FROM sf.cities As c
WHERE c.city = 'SAN FRANCISCO';
```

❶ ST_DWithin query

❷ Buffer for visualization

In ❶ we have the standard query we'd use for tabular reporting. In ❷ we show the buffer we'd create to visualize our region. Note that in ❷ we're using the aggregate ST_Union function, which will group all those San Francisco records into a single geometry record and then buffer them.

ST_DWithin should be the function of choice when it comes to finding geometries within a desired distance because it's extremely efficient. Novice PostGIS users often resort to two common, considerably slower alternatives to determine whether two geometries are within a certain distance of each other. The first is to pair up all the geometries and find the distance between each two, then order them from closest to farthest, and finally pick off the set that's within 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. In that case, 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 avoid having to calculate exact distances for every pairing. This in many cases makes it orders of magnitude faster. Buffering has the additional disadvantage of needing to first create a derivative geometry using buffering, which introduces inexactitudes of its own because buffers are always approximations of a true buffer. A buffer is still useful, particularly for visualization of the affected area.

ST_DWithin for the geography data type

The ST_DWithin function for the geography data type is the indexable companion to ST_Distance for geography. The distinction between geometry and geography is that the ST_DWithin for geography tolerance is always in meters, whereas with geometry it's in the units of the spatial reference system of the input geometries.

8.1.2 Convert to different units of measurement

Although 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. We like to keep our functions and function-related data in a separate schema for easier manageability, so we create a new schema to house the new functions and function helper data.

```
CREATE SCHEMA utility;
```

We then include this new schema in our database search path so we don't need to prefix the functions with the schema when we want to use them unless we want to.

```
ALTER DATABASE postgis_in_action set search_path=public,utility;
```

The following listing is a quick script to generate the units table.

Listing 8.2 Create a simple units conversion table

```
set search_path=utility,public;
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 these, we now do the following.

Listing 8.3 Which bridges are within a half-mile of San Francisco?

```
SELECT DISTINCT c.city, b.bridge_nam,
    ST_Distance(c.the_geom, b.the_geom) As dist_ft,
ST_Distance(c.the_geom, b.the_geom)*u.convfactor As dist_miles
FROM (
    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
```

1 Convert feet to miles

2 Conversion table

```
                INNER JOIN sf.bridges As b ON (
    ST_DWithin(c.the_geom, b.the_geom,0.5/u.convfactor))
WHERE c.city = 'SAN FRANCISCO'
ORDER BY dist_miles;
```

3 Convert 0.5 mile to 2640 feet

2 We use our conversion table twice: to grab the units of measure in feet and 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 native feet to miles and in 3 our ST_DWithin match to convert our 0.5 miles to 2640 feet.

Having to include this cross join in every query can become a bit tedious. We can black-box the conversion in an SQL function, as shown in listing 8.4.

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
$$
    WITH u(unit, unit_to_meters)  AS
    (VALUES ('mile', 1609.3400),
            ('kilometer', 1000),
            ('meter',1),
            ('feet', 0.3048)
        )
    SELECT ufrom.unit_to_meters/uto.unit_to_meters*$3
            FROM
         u As ufrom CROSS JOIN u As uto
               WHERE ufrom.unit = $1 and uto.unit = $2;
$$
  LANGUAGE 'sql' IMMUTABLE STRICT
  COST 10;
```

In listing 8.4 we create a function using SQL that implicitly converts all measurements to meters so that we only have to keep conversion table based on the meter unit of measurement. We use a CTE (PostgreSQL 8.4+) to simplify our SQL.

Using tables in functions

This function 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 advantage of inlining the table is that we can make the function IMMUTABLE instead of just STABLE because it doesn't rely on a table, which provides for better caching. It also makes the function self-standing. The advantage of using a CTE here instead of a subquery is that we need to define the table only once though we use it twice, but we lose backward compatibility with older versions of PostgreSQL. The downside of using this approach instead of using the lu_units table is that it's 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 knowledgeable about new units.

Listing 8.5 Using the unit conversion function

```
SELECT DISTINCT c.city, b.bridge_nam,
    ST_Distance(c.the_geom, b.the_geom) As dist_ft,
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 (
        ST_DWithin(c.the_geom, b.the_geom,
  units_from_to('mile','feet',0.5) ) )
WHERE c.city = 'SAN FRANCISCO'
ORDER BY dist_miles;
```

❶ Feet to mile

❷ 0.5 mile (mile to feet)

We now use our black-boxed function to convert feet to miles and miles to feet. ❷ We want to express our within distance in terms of miles, but because our geometry units are in feet, this 0.5 miles needs to be converted to our geometry units of feet. ❶ 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 doesn't come into play. If you attempt to use functions like ST_Distance on a global scale, you'll need to take the earth's curvature into consideration. To obtain sensible results, you need to make sure that you've transformed your measurements into a spatial reference system using some distance-preserving projections before applying the ST_Distance function. For instance, the popular Web Mercator spatial reference system looks great on maps because it conserves directions, 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 a spatial reference system 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're unable to find a distance-preserving spatial reference system and you only need to measure distance between point geometries, PostGIS offers two functions that take the 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.

Users of PostGIS version 1.5 or higher 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 lon lat units). Even though the WGS 84 spheroid serves as the basis for all geography objects, and

those objects are referenced in lon 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

This data type parallels the geometry data type but presupposes a spheroidal surface and a fixed SRID of 4326. Geography expects all data to be in WGS 84 lon 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 in the future.

The distance/area calculations for geography default to using a WGS 84 earth spheroid but also support the faster but less accurate sphere model with a 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 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 listing 8.6, we'll compare Web Mercator distance in meters with measurements in spatial reference systems designed for a specific region and later the distance spheroid functions. We'll compare and see just how badly or how well 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,
  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,
CAST( ST_Distance( ST_Transform(c.the_geom,3785),
   ST_Transform(b.the_geom,3785)
     ) As numeric(10,2) )  As wm_m,
CAST( ST_Distance(
    geography(ST_Transform(c.the_geom,4326)),
    geography(ST_Transform(b.the_geom,4326))) 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;
```

1 Feet to meters

2 Transform to natea (2163)

3 Transform to Web Mercator (3785)

4 geometry to geography dist spheroid

This example requires PostGIS 1.5+ because we're using the geography data type. In this example we're comparing the distance between bridges and cities. We perform

quite a bit of casting to numeric with two places after the decimal point because ST_Distance will return a double-precision number with many digits. ❶ We begin by converting our native feet distance to meters because we'll be using meters to compare all the final measurements. In ❷ we're transforming to National Atlas Meters SRS, ❸ Web Mercator. In ❹ we're transforming to WGS 84 lon lat and then casting our geometry to the geography data type. Once cast to the geography data type, ST_Distance will automatically use a spheroid-based calculation and return all answers in meters, as shown in table 8.1.

Table 8.1 Sample records of distances between cities generated by listing 8.6

city	bridge_nam	ca_m	natea_m	wm_m	geog_spheroid_m
Sausalito	Golden Gate Bridge	83.52	84.10	106.04	83.52
San Francisco	Golden Gate Bridge	16.29	16.35	20.62	16.30
San Francisco	Third Street Bridge	16.14	16.27	20.47	16.14

Table 8.1 provides sample output from the distance query. Assuming that the CA State Plane represents the most accurate measurement, we can easily see that even for areas within a half mile, Web Mercator gives significantly exaggerated distances.

Also as expected, National Atlas Equal Area (NATEA), because it's 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 CA State Plane. The geography datatype gives roughly the same answers as our State Plane feet measurements converted to meters. Both Web Mercator and NATEA have the advantage of being presentable on a map, whereas geography isn't as supported and may require transformation to look good on a map. NATEA has the disadvantage unlike Mercator of not being able to cover the globe; it covers just North America.

In the next listing we'll only do point distance checks but for a much larger range. Because we're 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 this query can be seen in table 6.3 of chapter 6. Mercator, although much worse than before, may be suitable for some close-proximity rule-of-thumb calculations and has the advantage of working on older PostGIS installs. It's 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 because it's a native geometry type it enjoys all the power of the GEOS geometric processing functions (though care must be taken) and that it has extensive support by third-party tools.

- If your distance ranges are small, say covering a country, state, or county, choose the good-for-measurement planar spatial reference system 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 you can transform on the map as needed to the most suitable spatial reference system for the zoomed-in area. Note that for many use cases like Google Maps and Microsoft Bing, which require data in WGS 84 lon lat, this transformation step isn't necessary; just use the standard output functions ST_AsText, ST_AsGML, ST_AsKML, and so on accordingly or convert back to geometry and then transform to Google Web Mercator. Bing, and so on.

8.1.4 Choose spatial reference systems when measuring area

The considerations for area are a bit different from those for distance. With area, local measurements have to be accurate, and the region you're calculating the area for is generally smaller than the region for which you need to measure distance. In the next listing 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 compared the smallest and the largest measurements in our set of data and also included the percent difference in measurement between our State Plane data and alternatives. The results are shown in table 8.2.

Table 8.2 Comparing area of cities around San Francisco in different spatial reference systems

city	casp	geog	naea	wm	pgeog	pnaea	pwm
San Quentin	1233	1233	1232	1986	-0.01	0.05	-61.08
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

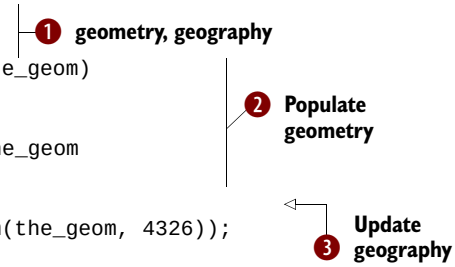
The pgeog, pnaea, and pwm fields are the percentage differences from the California State Plane measurements. As you can see, the National Atlas is about 0.06% off (note that if you use ST_Area(geog,false), the geography measurement will be using sphere, which is close to NAEA numbers), and geography (using spheroid) is about 0.01% off or less from those numbers, whereas the Web Mercator is a whopping 38% off. City polygons are huge objects, so they might not reflect the more common case of small objects, such as buildings in different cites. In the next exercise we'll draw a 10-meter radius patch of land around several city centroids using the UTM spatial reference system for that region so our units are all in square meters.

In this section we'll break one of the common best practices we noted in previous chapters; we're going to use one table to store records with different spatial reference systems. We do this to easily compare the more preferable good-for-distance UTM with our geography (web sphere/spheroid model) and Web Mercator. In listing 8.9 we'll create a new table of circles that have a 10-meter radius around key cities. We're doing this buffering in the UTM zone for that region because in lon lat units, the circles will be lopsided. To figure out the UTM SRID for each city point, we're going to use the utmzone contrib function in the PostGIS wiki: http://trac.osgeo.org/postgis/wiki/UsersWikiplpgsqlfunctionsDistance.

In the next listing we'll also exercise the new geography datatype as a storage type instead of casting to it as we have in prior exercises.

Listing 8.9 Using multiple spatial reference ids

```
CREATE TABLE world.city_buffers(city varchar(150) PRIMARY KEY,
    the_geom geometry,
    the_geog geography(POLYGON,4326));
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;
UPDATE world.city_buffers
  SET the_geog = geography(ST_Transform(the_geom, 4326));
```



In listing 8.9 we're ❶ creating a table with a geometry data type column and a geography data type column. Because we're using a generic geometry data type, we can stuff any kind of geometry with any kind of SRID in it. We're 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 there's no need for AddGeometryColumn. ❷ We now populate our table by buffering in UTM zone (comprising about 60 SRIDs); for the geometry type we keep the respective UTM zone spatial ref. ❸ We then update our the_geog field with the transformed 4326 cast to geography. Note that we could have used the ST_Buffer function of geography, but that 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).

In listing 8.10, we'll compare the areas of the buffers we created using a native geography area function with areas when transformed to UTM and Web Mercator.

Geography ST_Buffer

The geography datatype in PostGIS 1.5 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 the areas of our little patches of land and compare the UTM answers to the geography and Web Mercator. You 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 that Web Mercator is only reasonably accurate for area computations if you're living near the equator. If you live way up north in Murmansk or Helsinki, then your property size will be inflated.

Next, we'll look at data tagging.

8.2 Data tagging

Data tagging refers to a class of spatial techniques where we try to situate points located within the context of another geometry. Region tagging and linear referencing are two common tasks for GIS practitioners, because they're preparatory steps for all statistical analysis. For example, if you have rainfall data from various collection stations, unless you group the stations into regions, you won't be able to arrive at any conclusions.

8.2.1 Techniques for generating dummy data

Although generating data may seem a pointless exercise, generating random data or data that fits a certain pattern is useful for testing to determine how robust your queries will be when dealing with immense amounts of data or for testing general theoretical models against reality. Normally when you start off, the amount of data is miniscule, and it's only as it grows that you'll discover errors in your query or speed bottlenecks. This is especially true with data collected by instruments. For example, many flight-tracking software programs take FAA data of plane locations during flight and superimpose the positions on a map. In the United States, at any given moment, several thousand commercial aircrafts could be in the air. If you collect position data every 5 minutes, in a few hours you could have over 100,000 records. You don't want to end up in a situation where your common queries take minutes to run; for web mapping, any query that takes longer than 10 seconds to output a result is considered too long by average web visitors. To this end, simulated data is critical to test the load of your queries prior to deployment in a production environment.

The set of examples in the following listing requires PostgreSQL 8.4+ because of our use of array unnest. The already generated data is included in the source code/data download.

Listing 8.11 Create dummy observation data

```
CREATE TABLE us.observations (
    obsid serial PRIMARY KEY,
    obs_name varchar(50),
    obs_date date,
    state_fips varchar(2),
    state varchar(20));
SELECT
  AddGeometryColumn('us','observations','the_geom','4326','POINT','2');

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)
    CROSS JOIN generate_series(1,2000, 1) As i;

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;

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

This demonstrates how you can generate mock-observational data. In 1 we generate the observations table without a geometry column, and add in state_fips and state columns to hold the state where each of the observations occurs. In 2 we add in the geometry column. Note in 3 we use PostgreSQL 8.4 to quickly convert an array to a table. In 4 we add more kinds of creature observations and set our random generator to favor some states more than others. 5 Finally, we remove data that doesn't fall within the U.S. state boundaries.

We'll update these fields in the next example.

8.2.2 Tag data to a specific region

One of the more common uses for spatial databases is to tag regions. Following are the classic steps:

1. You have named regions of space divided into polygons or multipolygons. These could be political districts within a city, sales territories, states, and so on.
2. You have geocoded data with lon lat coordinates, and you need to figure out which region this geocoded data falls in.
3. 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'll use the state boundaries we created in the previous chapter as the region, and we'll make up some random WGS 84 lon lat points for the observation data. We'll then determine which state each observation point belongs to. Keep in mind that this can be applied to any region or set of points.

```
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 this example we tag each observation with a state, and we transform our lon lat degrees to US National Atlas coordinates because that's what we stored our us.states table in.

Notice also that we denormalized our data by updating a column with information that depends on other tables. If you're wondering why we didn't simply create a view that can tag the data on the fly, there are at least two reasons. The first is speed. Data tagging is something that needs to be done only once. Once we've placed a point within a particular region, this information doesn't change. In order not to have to repeat our tagging computation, which can become quite elaborate over large datasets, we store our results. The second reason is expediency. Often we need to export our data to non-relational applications. This requires that we flatten their structure. For better or for worse, much spatial work still involves spoon feeding to more rudimentary systems.

Another kind of tagging is tagging data to a location on a linestring such as a road. In the next section we'll describe 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, find what points fall on what line and where on the line they fall. This happens, for example, if you're 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 exercise is derived from Paul Ramsey's "Snapping Points in PostGIS" http://blog.cleverelephant.ca/2008/04/snapping-points-in-postgis.html.

EXERCISE 2: SNAP ALL POINTS WITHIN 10 UNITS OF A LINE 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.

The basic approach to the solution is as follows:

1. First, we find which line each point is closest to. We do that with the combination of a PostgreSQL DISTINCT ON and PostGIS ST_Distance function, which return each point once and its closest linestring. We also use ST_DWithin as a fast distance filter to quickly reject a combination point and line that are too far from each other.
2. Next, given this line and point, we interpolate the point on the line to figure out the closest point on the line on which it falls.

Listing 8.12 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.gid)
     ln.the_geom AS ln_geom,
     pt.the_geom AS pt_geom, ln.gid AS ln_id, pt.gid AS pt_id
FROM
    ch08.sites AS pt INNER JOIN
    ch08.roads AS ln
ON
    ST_DWithin(pt.the_geom, ln.the_geom, 10.0)
ORDER BY
    pt.gid,ST_Distance(ln.the_geom, pt.the_geom)
) AS subquery;
```

1 Closest point on line to point
2 Return point once
3 Limit search 10 units
4 Linestring closest to each point
5 Alias as subquery

1 First we locate the closest point on a line to a point. In 5 we return a subquery that returns a linestring and point pair where each point is at most 3 10 units away from the line, and if a point has multiple lines that are within 10 units of it, we use a 2 DISTINCT ON clause to ensure the point is selected only once and 4 pick the linestring that's closest to the point as the pair.

This can be written without a subquery as shown in the next listing, but the version without the subquery tends to be slower.

Listing 8.13 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
   ST_DWithin(pt.the_geom, ln.the_geom, 10.0)
ORDER BY
  pt.id,ST_Distance(ln.the_geom, pt.the_geom);
```

Figure 8.1 is a pictorial view of the original points and the snapped points.

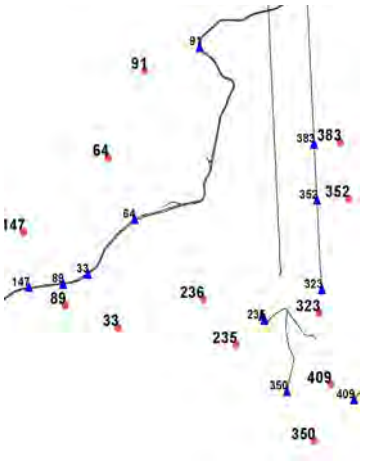


Figure 8.1 Snapping points to a line using the query from listings 8.12 and 8.13. The red dots are the original points and the blue triangles are the snapped ones.

ST_ClosestPoint in PostGIS 1.5+ a better line interpolate locate point

If you have PostGIS 1.5+, it's more efficient and shorter to use the ST_ClosestPoint function. Plus you can use it for more than lines, and it's 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 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

Oftentimes 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 the stclines_streets file (Street Centerlines for San Francisco area) from http://gispub02.sfgov.org/website/sfshare/index2.asp. Note that these units are in the same planar State Plane feet as the other sf sources we've been working with. We'll also use a made-up table of addresses. Note that the example in listing 8.14 is simplified in that our address names are well formed and normalized. In most cases you'll

Be careful when using lon lat (degrees)

For best results, you should stick with planar units like meters or feet because these functions are designed to work on a plane. But if you're talking about relatively small streets, the error introduced by lon lat isn't too bad. The assumption of planar, though wrong in those cases, isn't too far off at very small scales. This is why with the TIGER data we cover in chapter 10, the fact that the data is in lon lat is okay, because the line segments are small enough that the error introduced isn't significant.

have to use a combination of soundex, prefix matching, and the like to get raw input addresses into a normalized form suitable for geocoding.

Listing 8.14 Geocode an address to a point

```
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
        st_pos = CASE WHEN MOD(sc.lf_fadd,2) =
                  MOD(sf.test_addresses.st_num,2)
            THEN 'L'
        ELSE 'R'
        END,
        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
      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

In this example we first 1 create our dummy table of addresses and insert the data without any geometry. In the next part we determine on which segment in our 4 street centerlines the address is and on 2 which side of the street. We then 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 distance from start along the line a point lies. This we determine by taking the difference between (street number and start of street)/(street number range). We use a couple of built-in PostgreSQL helper functions to achieve this 3; the least and greatest functions in PostgreSQL take an infinite number of arguments and return the smallest or the largest in the argument set. We're also using the PostGIS ST_LineMerge function to convert our multilinestring to a linestring. This is necessary if the street centerlines are all stored as multilinestrings but are contiguous and can be stitched together to form a single linestring. The ST_Line_Interpolate_Point function works only with linestrings.

8.3 Slicing and splicing linestrings

In this section and next we'll 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'll 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 enthusiasts spent their leisure time visiting points of interest, taking GPS readings, and offering them to the general public via popular mapping sites. Some of the more common venues are local taverns, eateries, fishing holes, and filling stations with the lowest prices. A common follow-up task after gathering the raw positions of the various places is to connect them to form a course.

In this exercise we'll use the 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.

OBSERVATION LINE PATHS FROM GPS POINTS

For the exercise in listing 8.15 we want to create a new linestring for every 15 minutes of movement. You can use any grouping you want, such as the GPS course if that's filled in. In this case we're using the 15-minute mark for grouping because our other fields are blank and also the date and 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 8.15 Create line path from point observations

```
SELECT t.track_period, MIN(time) As t_start,
    Max(time) As t_end,
    ST_MakeLine(the_geom) As the_geom
INTO ch08.aussie_run
FROM (
    SELECT p.time, p.the_geom,
        DATE_TRUNC('minute', p.time)
        - CAST(
          MOD(CAST(DATE_PART('minute', p.time) As integer),15) ||
                        ' minutes' As interval) As track_period
    FROM ch08.aussie_track_points As p
    ORDER BY (DATE_TRUNC('minute', p.time)
        - CAST(
          MOD(CAST(DATE_PART('minute', p.time) As integer),15) ||
                        ' minutes' As interval) ) , p.time) As t
GROUP BY
    t.track_period
HAVING COUNT(time) > 2
ORDER BY t.track_period;
```

1 Return columns

2 Snap time to closest track period

2 Snap time to closest track period

3 Order by track period

4 Group by track period

```
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 ch08.aussie_run;
```

❺ **Calculate length, time per period**

In listing 8.15 we're ❷ creating a subquery 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 transfer each time point to the 15-minute slot that comes on or before it. ❸ In order for our points to be ordered by track_period and time, we order our subquery by those two fields. In this case, the order by p.track_period is redundant because the p.time guarantees that, but if you were going to group by course, for example, then this field would be needed. In ❹ we group by the calculated track_period so that each record will be for a 15-minute mark period, and in the SELECT ❶ we define our outputs to return track_period, min, max, and the line created when we stitch the points together. Then we dump it into a new table called work.aussie_run. ❺ Finally we do a query against our new table to see the results of our handiwork. In this query, we use the ST_Length_Spheroid instead of ST_Length, because our GPS data is in lon lat; a simple ST_Length would give us distance in degrees instead of meters, which isn't terribly useful. We also do a lot of casting to strip off time zones and floating points that ruin the presentation.

Table 8.3 is a sampling of our query in ❺.

Table 8.3 Output of query in listing 8.15

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	33	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 you can see, not all of our time increments are even or 15 minutes in duration. Who knows what was happening—perhaps the runner took a rest or our GPS got hiccups.

8.3.2 Break linestrings into smaller segments

In this section we'll go through a couple of exercises for breaking up lines. How you break the lines depends on what you're trying to do and the approach you take.

CREATING TWO-POINT LINES FROM MANY-POINT LINESTRINGS

One common task is taking a linestring with various points and breaking it into smaller linestrings, each with two points. In many cases, a line with just a start point and an end point is easier to work with, for example, when you want to group together edges shared by polygons. The example in the following listing takes generated GPS tracks (those imported from GPS format with OGR2OGR) and converts them to two-point lines.

Listing 8.16 Make two-point lines from linestrings

```
SELECT ogc_fid, n As pt_id,
    ST_MakeLine(
    ST_PointN(the_geom,n),
    ST_PointN(the_geom,n + 1)
) As the_geom
FROM ch08.aussie_tracks
    CROSS JOIN generate_series(1,10000) As n
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

Here we use the 1 non-aggregate version of the ST_MakeLine function that takes in two-point geometries and makes a simple two-point line. ST_PointN is a point iterator function that for a linestring (non-MULTI) will return the nth point. 2 We use the powerful built-in PostgreSQL generate_series function to do a cross join to generate an iterator between 1 and 10,000. This works only if each linestring has fewer than 10,000 points. Then we use 3 the `WHERE n < number of points` condition to limit the number of records to the number of points for each line. This example will work in lower versions of PostgreSQL as well as versions of PostGIS 1.3+.

This use of generate_series is a common idiom in SQL to simulate a procedural *for loop* . It's especially common in PostGIS because many geometry processes involve iterating over geometries. As mentioned, it works only with linestrings, so what if you have multilinestrings? This is one occasion where the ST_Dump() function comes in handy. Watch closely. The next example is more expensive but will handle multilinestrings as well.

Listing 8.17 Make two-point lines from multilinestrings or linestrings

```
SELECT ogc_fid, n As pt_id,(sl.g).path[1] As nline,
    ST_MakeLine(
    ST_PointN((sl.g).geom,n),
    ST_PointN((sl.g).geom,n + 1)
) As the_geom
```

1 Get path and make two-point line

```
FROM (SELECT ogc_fid, ST_Dump(the_geom) As g
FROM ch08.aussie_tracks) As sl
    CROSS JOIN generate_series(1,10000) As n
WHERE n < ST_NPoints((sl.g).geom)
ORDER by ogc_fid, nline, pt_id;
```

❷ Subselect explode multi to linestring

❸ Limit loop to # of points

In listing 8.17 we're using ST_MakeLine again ❶, but note that we're using a field called geom instead of the_geom. This is because the ❷ ST_Dump function you learned about in earlier chapters returns a set of geometry_dump objects consisting of two fields, geom and path—the (sl.g) is how we reference the composite object g in subquery sl. Because ST_Dump is a set-returning function, it explodes the number of rows we have so that we'll have one row for each linestring in our multilinestring/linestring. The path object is a one-dimensional array consisting of the path position of the subgeom within the geometry. In the case of multilinestrings, there's only one element in the array: the position of the linestring in the multilinestring. For a nested geometry collection, the path info becomes more interesting. In ❸ we use a WHERE clause to limit expansion to the number of points.

PostGIS 1.5 ST_DumpPoints

If you're using PostGIS 1.5 or above, 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'll demonstrate how you would go about, given a table of points and a table of linestrings, splitting the lines at the intersecting points. This happens, for example, if you need to put up road posts and need your roads split at these post points. Although this exercise does make great use of linear referencing functions, people generally don't think of it as a linear referencing activity.

The basic steps for this exercise are as follows:

1. Figure out which points intersect with line.
2. For each point intersection use ST_Line_Locate_Point to figure out the percentage along the line where 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.

To start, there are too many steps to put everything in one SQL statement, so it's best to wrap everything in a stored function. Also keep in mind that for these activities it's useful to keep status information in variables and that there isn't much use of spatial indexes within the overall body of an SQL statement. This means we want to choose a language that allows us to define variables and can be seen as a black-box query (rather than an inlined SQL query). Because of these two desired criteria, we'll choose to write our function in PL/PgSQL instead of as an SQL function.

Our function in listing 8.18 will take three inputs: a multilinestring/linestring, a multipoint/point, and a distance tolerance in units of the spatial reference system of the geometries. It 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'll allow a point on the line. So, for example, if we set our tolerance to be 1 foot, then our function will consider any point within 1 foot of the line to be on the line and snap it to the closest point on the line.

Listing 8.18 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
    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
  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
```

1 **Explode geometries**

2 **Loop through points**

```
            var_lset[j] := ST_Collect(var_sline, var_eline);

          END IF;
           ELSE
            var_lset[j] :=
                upgis_cutlineatpoints(var_lset[j], var_pset[i]);
           END IF;
         END IF;
       END LOOP;
     END LOOP;

  RETURN ST_Union(var_lset);

  END;
$$
  LANGUAGE 'plpgsql' IMMUTABLE STRICT;
```

Create a multilinestring cut (3)

Recursively cut (4)

(1) We use our favorite pattern of exploding a multigeom into single geom pieces and then collapsing those into a geometry array for easier processing. This will turn our multilinestring/linestring into an array of linestrings, and multipoints into an array of points. (2) Next we step through each point, and for each line that intersects the point but isn't on the boundary we perform the four-step process outlined earlier. The matches will result in a (3) multilinestring composed of two linestrings, respectively the start and the end segment, and these we put in place of our original linestring. Note that if a line is cut multiple times (the multilinestring can expand to more than two pieces), we use the power of recursion (4) to repeat the whole process so that at any point in time we're 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'll cut San Francisco streets that fall within 100 feet of our desired point, and we'll use ST_Dump again to explode our multilinestring into individual linestrings.

PostGIS 2.0 ST_Split

In PostGIS 2.0 there's a function called ST_Split, which will allow you to split a linestring by a point or a polygon by a line. Using that function would be more efficient and shorter to write, but it doesn't currently handle more complex cases.

```
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);
```

This query will return the ID of the street that was cut and the individual pieces as separate rows. We also include the original for comparison. A pictorial view of this is shown in figure 8.2.

In the section that follows, we'll talk about other useful things you can do with PostGIS affine family of functions.

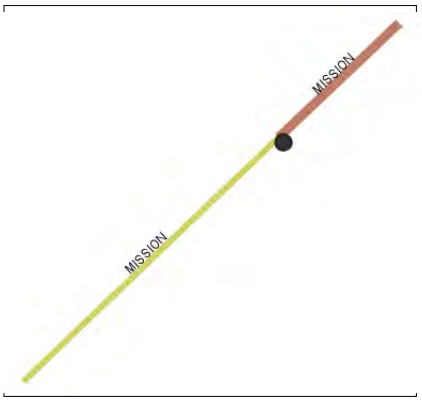


Figure 8.2 Using our upgis_cutlineatpoints function, we cut Mission street with a point that's within 100 feet of the line.

8.4 Slicing and splicing polygons

In the previous exercises you learned how to use basic geometries to do distance computations and areal analyses. Before you can even get to the point of doing interesting things with space, your data needs to be geocoded. Once you have geocoded data, 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, and so on. This is all made possible through the power of aggregation.

Aggregation is the concept of rolling up several records into one by grouping by like values or using a function that takes a set of values and returns one value. Functions that take a set of values and return one are called aggregate functions. We go into a little more detail about the concept of aggregation in appendix C and cover the various parts of GROUP BY, HAVING, and the SELECT SQL clauses used as well as common aggregate functions.

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, geometry aggregation functions are added to the mix: 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 split your geometries by political boundaries that 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't form a valid multipolygon out of them. You must either store them as a generic geometry collection or lose the shared boundaries by unioning them. You also learned in prior chapters that generic geometry collections are pesky creatures because many functions don't work with them. For our

study of proximity, we'd prefer each city to be represented as a single record even with the realization that we may lose political boundaries. To do that, we'll combine them using the OGC ST_Union function and group by city name.

DISSOLVE BOUNDARIES WITH ST_UNION

The following code tells us for each city that will be collapsed how many records we'll be collapsing and the number of polygons in the new record.

Listing 8.19 Analysis pre-dissolving records

```
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_afte
FROM sf.cities
GROUP BY city
HAVING COUNT(city) > 1;
```

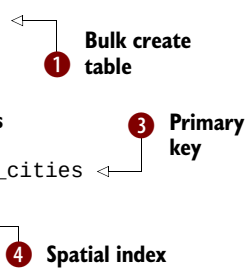
From the code in listing 8.19 we know that we'll be collapsing 10 cities, but of those only Brisbane and San Francisco will actually dissolve boundaries, because only these cities have fewer polygons per geometry than we started out with.

In the following listing, we use an SQL bulk insert statement that unions and groups same-named city records in order to guarantee one record per city.

Listing 8.20 Create one record per city

```
SELECT city, ST_Multi(ST_Union(the_geom)) As the_geom
INTO sf.distinct_cities
FROM sf.cities  GROUP BY city;

SELECT populate_geometry_columns
➥ ('sf.distinct_cities'::regclass);
ALTER TABLE sf.distinct_cities ADD CONSTRAINT pk_distinct_cities
    PRIMARY KEY(city);
CREATE INDEX idx_distinct_cities_the_geom ON
    sf.distinct_cities USING gist(the_geom);
```



In 1 we 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 add the usual 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 needed constraints by inspecting the table. 3 4 For good measure we put in a primary key and a spatial index.

8.4.2 Tessellate areas

It's commonly useful for statistics to divide your areas by spatial area size or by population such that each region has approximately the same area or the same population.

For these exercises, we'll explore both of approaches. We'll first break our data into equal areas and then create what we'll call an observation_tract, such that each observation tract contains approximately the same number of things. We'll demonstrate the power of SQL by bringing several concepts together:

- Revisit our dicing routine we created in earlier chapters to divide the United States into areal units.
- Use PostgreSQL 8.4+ Window functions in combo with some simple mathematics to group our areas into a collection of area with the total area close to our desired area.
- Use the powerful ST_Union spatial aggregate function we saw earlier to union these areas into single areas.
- Use the new common table expressions feature introduced in PostgreSQL 8.4 to combine all these SQL statements into a single query to create the final table.

EXERCISE 4: CREATE A GRID AND SLICE YOUR TABLE GEOMETRIES WITH THE GRID

In this exercise we'll cut our states into smaller units using a grid. You saw this exercise before, but we'll add a couple of twists to it. This is useful in a few scenarios:

- It improves spatial searches.
- It divides an area into smaller units that are more suited for heat maps.
- It reallocates areas by first dividing and then putting them back together differently.

Here we divide the U.S. data into smaller quadrants so that we can collect it later. When we've finished we'll have a throwaway_grid that looks like the one shown in figure 8.3.



Figure 8.3 Our throwaway_grid

Let's see the code behind this figure.

Listing 8.21 Divide the United States into quadrants.

```
WITH usext AS
(
 SELECT ST_SetSRID(
    CAST(ST_Extent(the_geom) As geometry),2163) As the_geom_ext,
    60 as x_gridcnt, 40 as y_gridcnt
 FROM us.states As s
  ),
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
)
SELECT grid.x, grid.y, state, state_fips,
    ST_Intersection(s.the_geom, grid_geom) As the_geom
    INTO us.grid_throwaway
    FROM us.states As s
   INNER JOIN grid
      ON (ST_Intersects(s.the_geom, grid.grid_geom));

    CREATE INDEX idx_us_grid_throwaway_the_geom
        ON us.grid_throwaway USING gist(the_geom);
```

Define constants

Create a painting tile

Divide extent into rectangles

Cut grid by state boundary

Index

This exercise uses a grid of 60 cells along X and 40 along Y of the extent of the United States to dice up our state boundaries such that no two states fall in the same region. Note that because of the way the tile cuts through the United States, the tiles have various shapes and sizes. This isn't 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 dice without paying much attention to the size of each piece. In most scenarios, we require fewer cuts but ones of equal size.

CREATE A SINGLE LINE CUT THAT BEST BISECTS INTO EQUAL HALVES

To create equal-area slices, the first strategy we'll employ is one of convergence toward a solution. In the following listing we start with a trial cut through our area and measure the area of the cut. If it's 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 8.22 Bisect the state of Idaho

```
WITH RECURSIVE
  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'
   ),
  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
) ,
bi(n) AS
 (SELECT n
    FROM T
    ORDER BY n DESC LIMIT 1)
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 Define variables

2 Recursive iterator

3 CTE returns how far in x dir to cut

4 Return both parts of Idaho

1 We first create a CTE to store variables we'll be using in subsequent CTEs: the reference geometry we want to cut in the X direction, the envelope of the reference geometry, the number of meters (nit) we'll be moving per iteration, and our target area, which is half the area of the state. Note that the units are in meters because our us.states table is in National Atlas Equal Area meter units. 2 This is a recursive iterator

that keeps moving the extent box across until it hits our target area. ❸ We care about only the last record of two, so we hold it in the CTE called *bi*. bi represents the number of meters to move the bounding box of the reference geometry to bisect the reference geometry. ❹ Our final query returns the two halves of our geometry, as shown in figure 8.4.

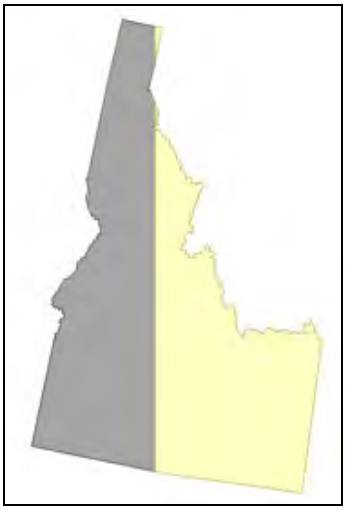


Figure 8.4 State of Idaho bisected

In our query, we use a CTE to perform the iteration. This is more to illustrate the capabilities of PostgreSQL than anything else. For clarity and portability, we advise you to create a function that performs the cut and then call the function as often as needed to reach 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 by looping multiple times through the cutter. For slicing into fourths, you perform the cut twice. For slices that are multiple of two, you can use another layer of recursion to further bisect your resultant areas until you have the number of total slices you want. You can even combine vertical cuts with horizontal cuts by iterating through the Y axis simultaneously to divide an area into quadrants.

CREATING EQUAL AREAS BY DISINTEGRATION AND INTEGRATION

In the next approach, we'll use a similar grid cut 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'll create a new bucket of shards. We'll then union them together by the bucket they're in. We'll 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 although we're 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 features of those points to achieve other equalities such as equality of population, trees, and so on. In that case, each resulting bucket would represent, for example, an equal population. This is basically what the U.S. Census does; all tracts are equal in population, not area.

When we've finished, we'll have a function that takes a geometry and the number of sections as input. We can call it 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;
```

This example would break Oklahoma into four equal regions, looking like the diagram in figure 8.5.

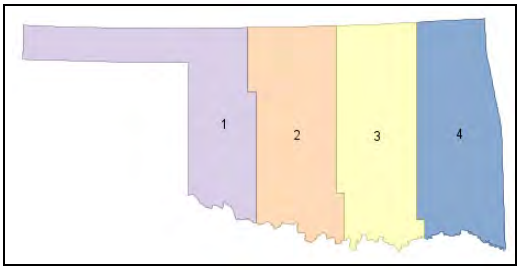


Figure 8.5 The state of Oklahoma broken into four equal quadrants

We can verify our work by taking the resulting area from our tabular, which looks like table 8.4.

Table 8.4 Area of Oklahoma cut into four equal quadrants

bucket	the_area
4	45407005343.697
2	45287294131.5032
1	45267841092.5329
3	45214837721.5997

In the following listing we'll walk through the function and see how it's constructed.

Listing 8.23 upgis_slicegeometry—cuts a geometry into equal areas

```
CREATE OR REPLACE FUNCTION utility.upgis_slicegeometry(ageom geometry
     ,numsections integer, OUT bucket integer, OUT the_geom geometry)
RETURNS SETOF record AS
$$
WITH RECURSIVE

    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)
            )
```

1 **Define constants**

```
        ),
        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
),
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),
     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
    ),
    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,
        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)
```

1 **Define constants**

2 **Start position squares**

3 **Window translate dice**

4 **Bucket shards**

```
        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
        )
   SELECT bucket, ST_Union(the_geom) As the_geom
    FROM T
    GROUP BY T.bucket, T.targ_area
```

4 **Bucket shards**

5 **Union buckets**

```
$$
LANGUAGE 'sql' IMMUTABLE;
```

We're using a recursive CTE construct with several subtable expressions, some of which are recursive and some of which are not. ❶ 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. Then we cut 15* numsections along the X and Y axes. ❷ We do a recursive query to return X/Y starting positions for each square. We could use generate_series instead, but this is slightly shorter. ❸ We use a SQL Window ROW_NUMBER() and ST_Translate query to cut up our geometry. The row_num column will return sequential unique numbers ordered by our OVER(ORDER …. Note that the ROW_NUMBER OVER(ORDER BY x,y) controls our cut. If we wanted our cuts going down instead of across, we'd order by y and then x. We can make intricate cuts by using other functions like ST_SnapToGrid or even sinusoidal functions. ❹ We use a recursive query to throw our shards into buckets. The trick here is our SQL CASE statement: We keep on adding to the existing bucket until the desired area is exceeded or the remaining area is less than one-fourth of the target area. The one-fourth is arbitrary. ❺ We union the shards in each bucket. Our resulting table will have fields called bucket and the_geom because that's what the output parameters are called. In PostgreSQL all arguments to a function are assumed to be only input-only parameters unless you explicitly put in OUT or INOUT.

In this section we've tasted a little bit of what translating geometries can do for us. In the section that follows, we'll explore a few more tricks with translate and other affine functions.

8.5 Translating, scaling, and rotating geometries

Do you remember what you learned during your first linear algebra course? (We don't.) Namely that 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 the umbrella function ST_Affine that lets you

explicitly specify the transformation matrix. We'll not go into detail about the ST_Affine function because it's rarely used directly.

Though you may think of shapes on a map as mostly static objects that don't get repositioned much, these handy functions intrude more often than expected. We've encountered the following common uses and are sure that more creative uses abound:

- Creating grids to divide larger geometries into smaller pieces or to use as an overlay on maps or to produce heat maps with color variation based on number of geometries and the size of geometries in each tile
- Simulating movement along a road
- Simulating position change of objects
- Correcting coordinates of a geometry when someone gave 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 switch planes of comparison

8.5.1 Move a geometry along X, Y, Z

The most popular transformations function is ST_Translate, which shifts a geometry. ST_Translate has many overloads; we'll demonstrate the ST_Translate(geom,x,y) variant, which is the most predominantly used.

A common and unexpected use for ST_Translate is to create grids by using one geometry to paint across and down a region. This artificial graticule can then be intersected with a reference geometry to divide it into rectangular regions or other shapes. We call this spatial design pattern the *cookie cutter grid strategy*. Take any paper road map, and you can see this. The map is often divided into alpha and numeric rectangles so that you can find a street in the index and then find it on the map itself. Another common use of an artificial grid is to summarize geometries within each tile to produce what are called heat maps. Now we'll show you how to create artificial cells on a map. The examples in listing 8.24 will create a honeycomb and rectangular grids somewhere in the middle of the United States. Again, we won't hold back on using SQL because the awesome power of PostGIS becomes visible only when we make liberal use of SQL's bulk processing capabilities. In this example we'll use the PostgreSQL 8.4 Common Table Expression feature again. If you're using 8.3 and below, you'll need to repeat the CTEs where we use them or create temporary tables to hold the CTE expressions. Better yet, upgrade!

Listing 8.24 Generate a rectangle and a hexagonal grid centered in the United States.

```
WITH center_point(x,y) AS
(
   SELECT -288499, -2718
),
paintbrush(the_hex, the_rect) AS
(
SELECT ST_SetSRID(ST_Translate(
```

1 CTE center_point

2 CTE hex/square dual paintbrush

```
        ST_GeomFromText('POLYGON((0 0,64 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
)
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;
```

2 CTE hex/square dual paintbrush

Start from center paint 3

The example generates a hexagonal and a rectangular grid consisting of 10,201 records in a couple of seconds using a two-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 one-row table where we'll position our paintbrush. This is where we'll move the center of our paintbrush heads. 2 We create our paintbrush CTE with two heads: the_hex representing the hexagonal head and the_rect representing the rectangular head. We're using the ST_MakeBox2D function to create our rectangular head because it's simpler to express squares and rectangles with boxes. But because a box isn't a geometry, we have to convert it 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 two geometries: one representing each rectangular tile and one representing each hexagonal tile. Observe that our iterators are multiplied by the width and height of each brush to ensure that the tiles don't overlap each other. We could have used the step variant of generate_series instead of generate_series(start, end, step) as is done in the wiki example. Because we're doing a cross join, our final output will consist of 10,201 records ((50 + 1 + 50) * (50 + 1 + 50)). The results are shown in figure 8.6.

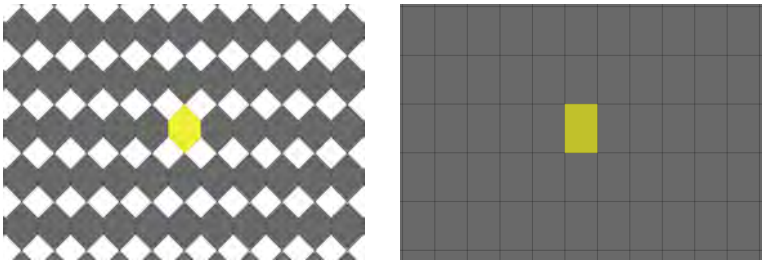


Figure 8.6 The center portions of the hexagonal and rectangular grids generated with code from listing 8.24. The highlighted center tiles are the location of our paintbrush CTE.

8.5.2 Increase and decrease size of geometry

The scaling family of functions comes in two versions: ST_Scale(geometry, xfactor, yfactor) and ST_Scale(geometry, xfactor, yfactor, zfactor). Both preserve the dimension of the geometry. If you pass in a 3D geometry, you'll get back a 3D geometry.

Scaling takes every coordinate and multiplies it by the factor parameters. If you pass in a factor between 1 and -1, then you'll shrink the geometry. If you pass in negative factors, the geometry will be flipped in addition to any scaling. The following listing shows an example of scaling a hexagon.

Listing 8.25 Example of scaling a hexagon to different sizes

```
SELECT xfactor, yfactor, ST_Scale(hex.the_geom, xfactor, 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;
```

❶ Hex to scale

❷ Scaling values

In this example we start with a ❶ hexagonal polygon and shrink and expand the geometry in the x and y directions from ❷ 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 one-half for each step. The results are shown in figure 8.7.

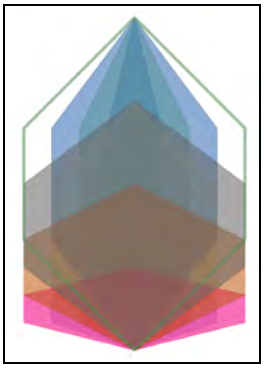


Figure 8.7 Diagram of query in listing 8.25. The dark area is the original hexagon (xfactor: 1, yfactor: 1), and the larger area is the hexagon scaled to twice its size (xfactor: 2, yfactor: 2).

This diagram multiplies our coordinates, and because our hexagon starts at the origin, all resulting geometries have their base at the origin. Normally when you scale, you want to maintain the centroid constant, so you would use a combination of scaling and translation, as shown in the next listing.

Listing 8.26 Combining scaling and translation to maintain the 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 the previous one. Here we're scaling a hexagon from half its size to twice its size in x and y directions. We're then translating the resulting scaled geometry so that the new centroid is where the original hexagon centroid was; see figure 8.8.

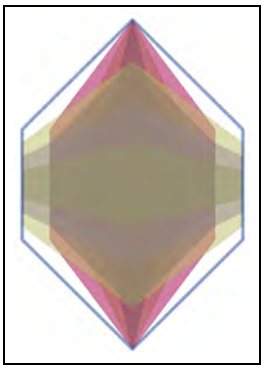


Figure 8.8 Result of query in listing 8.26. This demonstrates scaling and then translating to maintain the original centroid position. The darkened black-bordered geometry is our original hexagon, and the outermost hexagon is our hexagon scaled to twice its size in all directions. The various spectral colors are incremental scalings ranging from 0.5 to 2 in steps of 0.5.

8.5.3 Rotate a geometry

ST_Rotate, ST_RotateX, ST_RotateY, and ST_RotateZ are used to rotate a geometry around the X, Y, or Z axis in radian units. ST_Rotate and ST_RotateZ are exactly the same because the default axis rotation is Z for 2D applications. These functions are rarely used in isolation because their default behavior is to rotate the geometry around the origin rather than about the centroid. ST_Rotate is almost always combined with two translations to achieve rotation about the centroid; an example is shown in the following listing, and the results are diagrammed in figure 8.9.

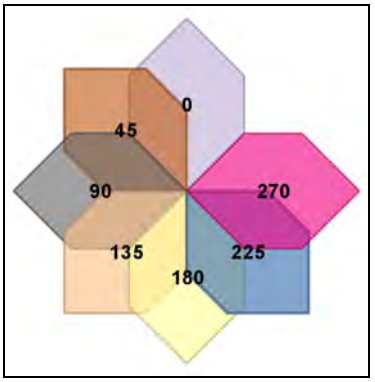


Figure 8.9 Result of query in listing 8.27 of rotating a hexagon from 0 to 270 in 45-degree increments

Listing 8.27 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;
```

As you can see in figure 8.9, our original polygon, which happens to be based at 0,0 (the origin) is rotated around that base. If the polygon is far away from the origin, it will be rotated over a much greater distance.

Almost always a rotation is desired around the centroid or some other point on the geometry, which requires a combination of translation and rotation.

A simple function called RotateAtPoint is available in the PostGIS wiki, http://trac.osgeo.org/postgis/wiki/UsersWikiplpgsqlfunctions, which allows you to rotate a geometry about any point. In the following example we use this function to rotate a geometry about its centroid.

Listing 8.28 ST_Rotate in combination with ST_Translate to rotate about a centroid

```
CREATE OR REPLACE FUNCTION RotateAtPoint(the_geom geometry,
    pt_x double precision, pt_y double precision,
        rotrads double precision)
 RETURNS geometry AS
$$
SELECT ST_Translate(ST_Rotate(
         ST_Translate($1,-1*$2,-1*$3),$4),$2,$3)
$$
   LANGUAGE 'sql';
```

❶ RotateAtPoint function

```
SELECT rotrad/pi()*180 As deg,
     RotateAtPoint(hex.the_geom,ST_X(ST_Centroid(hex.the_geom)),
       ST_Y(ST_Centroid(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(0,1) As x) As xf;
```

❷ Rotate hexagon about centroid

In this example ❶ we've installed the RotateAtPoint function from the PostGIS wiki, which we use in ❷ to generate two hexagons, the original and one rotated 45 degrees. The results are shown in figure 8.10.

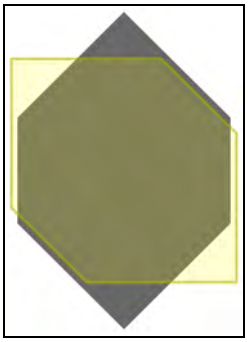


Figure 8.10 Result of query in listing 8.28. The gray area is our original hexagon, and the shaded geometry is our hexagon rotated 45 degrees.

8.6 Summary

In this chapter you learned a lot about leveraging the power of SQL with the power of space. We didn't focus on optimizing our queries for speed, and given the limited amount of data, speed was a non-issue. In more realistic scenarios, you'll have lots of data, and hopefully lots of people consulting the database, and then you'll need to make your application run as fast as possible. Designing snappy queries requires paying attention to things like indexes, function costing, and simplifying geometries. In the next chapter, we'll focus more on performance, how to analyze performance, and how to improve performance.

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—tuning queries, tables, and geometries becomes a major consideration. The way you express your queries is also important because the same logic expressed with two different SQL statements can have vastly different performance. The complexity of geometries, memory allocation, and even storage affect performance.

The query planner has many options to choose from, especially 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 scans, sequential scans, index scans, hash joins, and the like) it will use. All these play a role in the speed and efficiency of the queries.

It's only partly true that SQL is a declarative language that allows you to state a request without worrying about the way it's eventually implemented. 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, the way you state your question can greatly influence the way the planner answers it, and that answer has a great impact on speed. This is why all high-end databases provide "explain plans" or "show plans" to give you a glimpse of the planner's strategy. SQL allows you only to ask questions and not define explicit steps, but you should still take care in how you ask your questions.

9.1 The query planner

All relational databases employ a query planner to digest the raw SQL statement prior to executing the query. The most important thing to keep in mind when writing a query is that the planner isn't perfect and can optimize some SQL statements better than others. The query planner breaks down an SQL query into execution steps and decides which indexes, if any, and which navigation strategies will be used. It bases its plans on various heuristics and on its knowledge of the data distribution. It knows something that you often don't know: how your data is distributed at any point in time. It won't, however, relieve you of having to write efficient queries.

The spatial world of PostGIS offers some classic examples:

- One such example is asking for the top five closest objects, which we covered in chapter 5. (If you ask this, you force the PostgreSQL planner to do a table scan of all records and rank them all by distance and pick the top five). Or you can ask for the top five closest within 10 miles. In the second case, the planner can use a spatial index to throw out all objects that aren't within 10 miles and then scan the remaining. You don't care about the 10 miles and don't 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 here:

  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 slow but more obvious way:

```
SELECT restaurant_name
FROM restaurants
ORDER BY ST_Distance(ST_GeomFromText(…), restaurants.the_geom) LIMIT 5;
```

- Another example 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 leads to inefficient planner strategies. So instead, you ask to collect all objects that meet a criterion that's capable of using an index as well as the ones that don't and throw out the meeting criteria. Surprisingly, this

happens to work pretty well for most relational databases, whereas the straightforward question isn't as often converted to an efficient strategy.

KNN GIST and planner strategies

The planner and indexes are two core facilities constantly under improvement in PostgreSQL. As such, it's important to keep an eye on these enhancements, especially when upgrading. Although this is the case for PostgreSQL 8.3-9.0 and PostGIS 1.3-1.5, future versions may be smart enough to not require such a low-distance span to get the top five. For example, currently in the works is a KNN GIST feature that allows for deeper inspection of GIST indexes. This will make things like point in large polygon proximity queries much faster and will also reduce the need for specifying a spanning distance or allow you to specify a much larger distance without penalty. Some of this work you may see in operation in PostgreSQL 9.1 or 9.2 and PostGIS 2.0/2.1.

Also note that nearest neighbor (NN) queries are different from spatial database to spatial database. Oracle has some functions, such as SDO_NN, specifically for optimizing these queries. SQL Server 2011 (code-named Denali) has enhanced its spatial indexes to provide better NN logic with a simple `STDistance(g1,g2) > x` syntax. SQL Server indexes have always been based on a gridding model as opposed to the R-Tree approach that both PostGIS and Oracle use. Therefore, tricks for optimizing these kinds of queries aren't as portable between the different platforms as other tasks.

We'll delve into these and other planner topics, such as PostgreSQL settings, as we examine real case scenarios.

9.1.1 Planner statistics

The planner uses data statistics as well as various server configurations (allocated memory, shared buffers, seq costs, and the like) to make its decision.

Most relational databases use planner statistics as input to 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, from 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, it's best to do a vacuum analyze of the table after a large load rather than waiting for PostgreSQL's vacuum run.

You can also do a plain

```
analyze sometable verbose;
```

if you want to update the statistics without getting rid of dead tuples and want to see progress of the analyze.

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 the 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 this query is shown in table 9.1.

Table 9.1 Result of planner statistics query

colname	n_distinct	common_vals	dist_freq
gid	-1		
state	-1	-1	
state_fips	-1		
order_adm	-0.962264	0	0.0566038
month_adm	-0.226415	December	0.169811
day_adm	0		
year_adm	-0.660377	1788	0.150943
the_geom	-1		

Having -1 in the n_distinct column means that the values are more or less unique across the table in that column. A number less than 1 tells you the percentage of records that are unique. If you see a number greater than 1 in the n_distinct column, then that's usually the exact number of distinct records found. The common_vals column lists the most commonly observed values. For example, month_adm tells us that December is the most common month, and because our n_distinct number is relatively high, about 70% of the records will fall in the common_vals section for that column. This is useful to the planner, because it can use this information to decide the order in which to navigate tables and apply indexes as well as plan the strategy. It can also guess whether a nested loop is more efficient than a hash by looking at the where and join conditions of a query and estimating the number of results from each table. In the next section we'll look into the mind of the planner and investigate how it reasons about the queries it has to assess.

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 the size of the table and the default_statistics_target. Note that you can also set planner statistics separately for each column in a table if you want more or fewer records to be sampled. You do this using `ALTER TABLE ALTER COLUMN somecolumn SET STATISTICS somevalue`. We cover this in more detail in appendix D.

9.2 Using explain to diagnose problems

There are a few items you should look for when troubleshooting query performance:

- What indexes, if any, are being used?
- What is the order of function evaluation?
- In what order are the indexes being applied?
- What strategies are used, for example, nested loop, hash join, merge join, bitmap?
- What are the calculated versus the actual costs?
- How many rows are 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 both actual and planned execution plans.

Explain in other relational databases

If you're 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 explain plan (built into Enterprise Manager, Studio, or Studio Express) similar to pgAdmin's graphical explain as well as a text explain plan similar to the PostgreSQL raw explain. The graphical explain in SQL Server is much more popular than the following text explain option:

```
SET SHOWPLAN_ALL ON
GO
sql_goes_here
```

There are three levels of explain plans in PostgreSQL:

- *EXPLAIN*—This doesn't try to run the query but provides the general approach that will be taken without extensive analysis.
- *EXPLAIN ANALYZE*—This 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 takes at least the amount of time needed to run the query (minus network effects of returning the data). In addition to the rows estimated it provides actual row counts and timings for each step. In 8.4+ it also provides the amount of memory used. Comparing the actual times against the estimated ones is a good way of telling if your planner statistics are out of date.
- *EXPLAIN ANALYZE VERBOSE*—This does an in-depth plan analysis, and for PostgreSQL 8.4+ it 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 can alert you, when you using the evil SELECT *, how costly it is. EXPLAIN ANALYZE also displays memory utilization.

The following exercises use some of our pre-generated as well as our loaded data.

9.2.1 Text explain versus pgAdmin III graphical explain

There are two 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 generally we find the graphical explain easier to scan, more visually appealing, and as a rule of thumb a good place to focus our efforts. In this section we'll experiment with both. There are many PostgreSQL tools that provide a graphical explain plan and a textual explain plan, all different in look, ability to print, and so on. For this study we'll focus on the pgAdmin III graphical explain plan packaged with PostgreSQL and the native raw text explain plan output by PostgreSQL.

WHAT IS A TEXT EXPLAIN?

A text 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 in content than that of most databases. The text explain in PostgreSQL is presented as indented text to demonstrate the ordering of operations and the nesting of suboperations. You can output it using psql or pgAdmin III. For outputting nicely formatted text 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; it's available at http://explain.depesz.com/help.

Planner changes in PostgreSQL 9.0

One of the latest changes in PostgreSQL 9.0 is the ability to output the text explain in XML, JSON, and YAML formats. This should provide more options for analyzing and viewing explain plans. We have an example of prettifying and making the JSON plan interactive using JQuery at http://www.postgresonline.com/journal/archives/174-pgexplain90formats_part2.html.

It generally provides more information than a graphical explain plan, which we'll cover shortly, but tends to be harder to read and even sometimes provides too much information.

WHAT IS A GRAPHICAL EXPLAIN?

A graphical explain plan is a beautiful thing. It shows a diagram of how the planner has navigated the data, what functions it processed, and what strategies it has used—all this in bright and beautiful glowing icons and colors. The pgAdmin III graphical explain plan is quite attractive to look at. It has cute little icons for window aggs, hash joins, bitmap scans, and CTEs and provides tool tips as you mouse-over the diagram. In addition, the thickness of the lines gives a sense of the cost of a step. Thicker lines mean more costly steps. It's similar in flavor to the Microsoft SQL Server show plan. In pgAdmin III 1.10 and above you can save the plan as an image.

In the next set of exercises we'll look at some sample plans of queries and describe what each is telling us. We'll look at each in its raw intimidating text explain form and its user-friendly cute pgAdmin III graphical presentation.

9.2.2 The plan without an index

We purposely didn't index our tables so that we could demonstrate what a plan without an index looks like.

EXAMPLE SAN FRANCISCO BRIDGES AGAIN

In this example we look at the simple intersects query from the last chapter. We'll demonstrate the three text plans EXPLAIN, EXPLAIN ANALYZE, and EXPLAIN ANALYZE VERBOSE to see what further level of analysis each provides. We'll then follow up with the graphical explain.

```
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 this explain is as follows:

```
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 tells us the strategy the database would take to answer this question and also provides 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 previous code you see the ST_Intersects function as two functions: an && operator that does a bounding box intersect check and the _st_intersects that does the more intensive intersect checking. You'll only see this behavior with functions written in SQL because SQL functions are often inlined in queries and so are transparent to the planner. This allows the planner to reorder the function, even splitting it into two and evaluating the parts out of order. An inlined function is generally a good feature, but it can be bad too if it distracts the planner from more important analysis or encourages it to use an index where not using an index is more efficient.

In this next example, we repeat the same SQL but use 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 this:

```
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)
  (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
```

You can see that EXPLAIN ANALYZE provides more information. In addition to the plan, it provides us with actual timing, total time, and the number of rows being traversed. You can see, for example, that the slowest part of our query is the nested loop. Nested loops tend to be the real bottlenecks, but in many cases they're 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_nam
FROM sf.cities AS c INNER JOIN
  sf.bridges As b ON ST_Intersects(c.the_geom, b.the_geom);
```

The result with VERBOSE is shown here:

```
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_nam
   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_nam, 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_nam, b.the_geom
 Total runtime: 40.118 ms
```

Here you see the output of the VERBOSE variant. The VERBOSE version tells us also what columns are being output. Also notice that the time this one takes is much lower than for our EXPLAIN ANALYZE. Because both an EXPLAIN ANALYZE and an EXPLAIN ANALYZE VERBOSE run the query, the database already knows how to plan this query and may have the plan cached and some portion of the function calls cached in short-term memory. It doesn't always cache function answers, but if it decides to do so, the function is marked as immutable and the cost of caching is cheaper than recalculating the answer. Now we'll take a look at the friendly sibling, the graphical explain.

To summon the graphical explain in pgAdmin III, you highlight the SQL statement in the query window and click the Explain Query icon; you could also check the Analyze option if you want a more in-depth tool tip with real analysis, as shown in figure 9.1. The graphical explain, however, can't handle VERBOSE, so checking Verbose would force a plain-text explain.

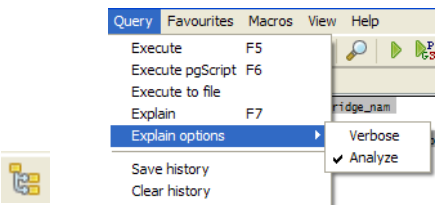


Figure 9.1 Graphical explain controls

Our pretty sibling looks like figure 9.2.

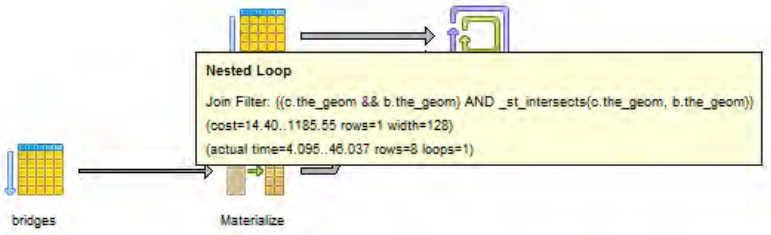


Figure 9.2 Graphical explain analyze of our bridge and city intersects query

It's a bit easier to see the order of operation with the graphical explain. The graphical explain always reads from left to right whereas with the text explain the time generally goes from bottom to top. Note that the nested loop is the last operation to happen. The other thing that's 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 the one you see going to the nested loop in figure 9.2, means a more costly step. The tool tip feature is both good and bad. It's good in the sense that it allows you to focus on one area; this is especially useful with complex queries. The downside is you can't see all the detail at a glance as you can with the text version.

So you know now that we're doing sequential scans on two tables and creating a work table (*materialize* means that we're creating a temp table). For functions that are costly to calculate and are reused in the query and return few rows, you want the planner to materialize and may need to employ tricks to force that.

In the next section you'll see what a plan with indexes looks like by rerunning the same query after adding spatial indexes and vacuum analyzing 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'll 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 B-tree or some other index access method as well.

The main families of indexes used in PostgreSQL are B-tree, GIST, and GIN; certain kinds of objects such as PostGIS are designed to take advantage of a certain index because of the way their data is structured. PostGIS, pgSphere, and Full Text Search objects can all use GIST indexes. Full Text Search can also use another index called a Generalized Inverted Tree (GIN) index, which is basically an R-tree (implemented with GIST) flipped upside down. Almost everything else works best with or can only use a B-tree index. Hash indexes are rarely used and are considered deprecated these days because they take longer to build and aren't any faster than a B-tree or GIST, and for most applications that used to use them, they're worse than a GIST index.

9.3.1 The plan with a spatial index scan

In the previous example you saw what our planner does without the help of indexes. In this listing, we'll help the planner out a bit by adding spatial indexes to our tables. Observe how the planner reacts to this change of events.

Listing 9.1 Index, vacuum, explain

```
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);
```



```
vacuum analyze sf.bridges;
vacuum analyze sf.cities;
```

❷ Update stats

```
EXPLAIN ANALYZE VERBOSE 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);
```

❸ Show explain

In this example we've indexed the tables ❶ and then ❷ updated the statistics by vacuum analyzing. An analyze would have been sufficient, but as a general rule we vacuum because vacuuming doesn't add much more time if there are no dead tuples (no new updates) and reduces the records that need to be scanned for future queries when there are dead tuples. ❸ Finally we do a full ANALYZE VERBOSE of our query. The associated graphical explain looks like this:

```
QUERY PLAN
------------------------------------------------------------
 Nested Loop  (cost=0.00..22.17 rows=4 width=35)
   (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
```

What's interesting about the query plan is that although we don't have c.area and so on as part of the SELECT output, the planner is still dragging them along for the index and sequential scan. There's a penalty with wide tables even if you aren't selecting those columns, but the bigger penalty comes when you update even wider tables.

In figure 9.3 you see the graphical representation twice: one with the tool tip opened to the index and one opened on the nested loop.

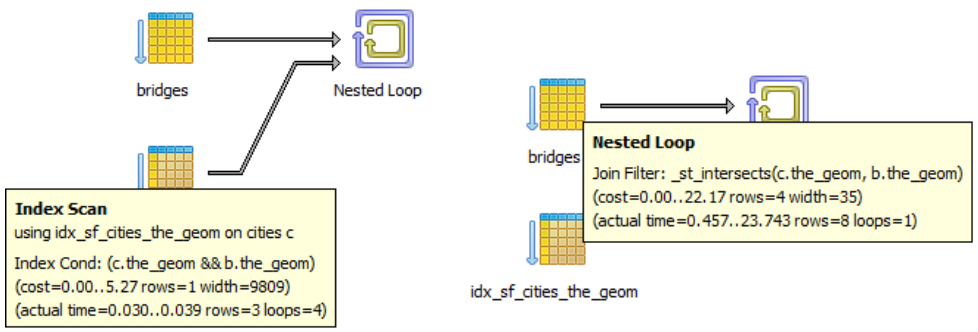


Figure 9.3 Graphical explain after the addition of the index, first with the index tool tip and then with the nested loop tool tip

As you can see, the plan has changed simply by adding in some indexes. The main differences in this plan are as follows:

- The materialized table is gone.
- The index on sf cities is being used and an index scan is happening instead of a sequential scan.
- The ST_Intersects call has been split in order so the && part, which uses the spatial index scan, happens first, and the more costly _ST_Intersects call is the only one left in the nested loop. This is an 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 even after the _ST_Intersects, or happens before a cheaper index scan.
- Most likely as a result of updated stats, our actual and estimated row numbers are closer. The closer the numbers the better, because closer numbers mean the planner's estimates of the data distribution are more accurate.
- These tables are relatively small so our speed improvement isn't significant after cache.

Planner short-circuiting

The planner employs a common programming tactic called *short-circuiting*. Short-circuiting occurs when a program processes only one part of a compound condition if processing the second part doesn't change the answer. For example, if the first part of the logical condition "A and B" returns false, the planner knows it doesn't have to evaluate the second part because the compound answer will always be false. This is a common behavior of relational databases and of many programming languages. But unlike many programming systems that implement short-circuiting, relational databases (PostgreSQL included) generally don't check A and B in sequence. They first check the one they consider the cheapest. In the previous example with the index, you can see that the planner now considers the && cheaper than the _ST_Intersects, and so it processes that one first. Only after that will it process the _ST_Intersects for those records where geomA && geomB is true. For cost analysis, it looks at several things; with AND conditions, it often looks at the cost of a function and uses that to forecast 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 expensive, so in those cases it simply processes the conditions in order. So even though the query planner may not process conditions in the order in which they're stated, it's 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 you can see at a glance

how the ST_Intersects is broken up. In the graphical one, you need to mouse-over and click to see what's happening.

9.3.2 Options for defining indexes

In addition to the various index types, you can also control which records are indexed with filter conditions or indexed on an expression instead of a column.

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 as follows:

- It creates a smaller index so there's less storage.
- The index is faster because it's lighter and can better fit in memory.
- Sometimes it forces a more desirable strategy. For example, if your data is the same in 90% of the rows and different in only 10%, and you expect this to be the case for future data, it's more efficient for the planner to scan only the index in the 10% and do a table scan for the other 90%. The planner has an easier time deciding whether to use an index.

These are the main cons:

- 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 is that a plan with a parameter can't assume anything about the value of that parameter; therefore it can't determine whether to use the index. This is described in Hubert Lubaczewski's Prepared Statements Gotcha at http://www.depesz.com/index.php/2008/05/10/prepared-statements-gotcha/.
- You can't cluster on a partial index.

Here's an example of a partial index:

```
CREATE INDEX idx_sometable_active_type
  ON sometable
  USING btree
  (type) WHERE active = true;
```

This situation presents itself where we normally query only active records for type and don't care about type when pulling inactive records.

COMPOUND INDEX

PostgreSQL, like most other relational databases, gives you the option of indexes composed of more than one table column or calculated columns but that use the same

index access method. You can combine this with the aforementioned partial and the functional (aka expression) index, which we'll discuss shortly.

What is a compound index?

A compound index is an index that's based on more than one column. In PostgreSQL 8.1, the bitmap index plan strategy was introduced, which allowed using multiple indexes at the same time in a plan. The introduction of the bitmap index strategy made the compound index much less necessary, because you could combine several single-column indexes to achieve the same result. Still, in some cases, for example, 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, because a simple index scan is somewhat less expensive than a bitmap index scan strategy. Note that unlike some other databases such as SQL Server and MySQL that can take advantage of compound indexes (for some storage engines) to satisfy a query select (often called a covering index), PostgreSQL always fetches the rows from disk or cache because PostgreSQL indexes aren't MVCC aware (they may contain indexes of dead records). There have been discussions to improve on this to implement true covering index behavior similar to what SQL Server offers. We'll probably not see this enhancement until PostgreSQL 9.2. For a spatial index, it will always have to go to disk anyway because 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 needed to satisfy a query, and there's no need to retrieve extra data from disk.

Because most common data types don't have operators for GIST access 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," indexes a calculated value. It has two restrictions:

- The arguments to the function need to be fields in the same table, although the function can take as many columns or constants as arguments as it likes. You can't go across tables or use aggregates and so on.
- The function must be marked immutable, which means that the same input always returns the same output and that no other tables are involved.

When immutable functions change

If you change the definition of an immutable function and this function is used in an index, you should reindex your table; otherwise, you'll run into potentially bizarre results.

Functional indexes are pretty useful, especially for spatial queries. As mentioned earlier, we use ST_Transform functional indexes of the following form, which is a bit of a no-no:

```
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 column. 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're using third-party rendering tools where you can't completely control the behavior of the generated query. We didn't explore this approach and only throw it out as food for thought. Here's an example (requires PostGIS 1.5 or above):

We can create a geography functional index by doing this:

```
CREATE INDEX idx_sometable_the_geom_geography
  ON sometable
  USING gist
  (geography(ST_Transform(the_geom,4326)));
```

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 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're using geography as a filter in the WHERE clause, such as shown here, 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.geog, s2.geog, 5000));
```

For this query, the spatial index might not be used if the costs on some functions are set too low. In that case, the planner has been given bad information and underestimates the cost of calculation versus the cost of reading from the index. We'll revisit this when we talk about computing the cost of functions.

Other common uses for functional indexes in the spatial world are indexes on calculations with ST_Area or ST_Length, if you don't want to store them physically, or putting a trigger on a table that's filtered a lot where the filtering should be indexed. For geocoding, soundex is a common favorite. Following is an example of a soundex index and how it can be used.

Using soundex

The soundex function isn't 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 distance algorithm (which returns the Levenshtein distance between two strings—the least number of character edits you need to convert the first string to the second one).

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's 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 misspellings. The previous query would return "DEVONSHIRE" whether or not you have a soundex index in place. Note that because soundex doesn't care about string casing, you can use it without changing your case to match your data. The previous 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 because the speed is already pretty good. It does add a lot of speed if you're dealing with a huge number of records.

PRIMARY KEYS, UNIQUE INDEXES, AND FOREIGN KEYS

There's a lot of debate as to whether foreign keys are good as opposed to primary and unique keys, which we think most database specialists would consider 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're using inherited tables, because a primary key on the parent (though kind of meaningless), fools the planner into thinking it's unique and it can stop checking once it hits a child with the requested key. There isn't much difference between primary keys and a unique index except for the following:

- Only one primary key can exist per table, although you can have multiple unique indexes. Both primary keys and unique indexes can contain more than one column.
- Only primary keys can take part in foreign key relationships as the one side of a one-to-many relationship.

- A primary key can't contain NULLs, but a unique key can and can even have many, but remember that NULLs are ignored when considering uniqueness.

Both have the side benefit of ensuring uniqueness (except in the 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 impact performance only during updates/inserts and in most cases negligibly unless you're constantly updating the key fields. Although foreign keys in and of themselves don't improve performance, they help in three indirect ways:

- They ensure you don't have orphans, which generally means fewer records for the planner to scan through, wasting time, and in addition with CASCADING delete/update actions, they're maintained by the database.
- They're self-documenting; another database user can look at a foreign key relationship and know exactly how the two tables are supposed to be joined and that they're related.
- Lots of third-party tools GUI query builders take advantage of them. So when an unsuspecting user drags and drops two tables in a query designer, the builder automatically joins the related fields.

Sometimes in a query, the planner refuses to use an index that you expected it to use. There are two main reasons for this:

- It can't use the index because you didn't set it up right. A common example of this is the B-tree index; how B-tree indexes behave changes from version to version. We'll go more into detail about this in appendix D.
- The second reason is that a table scan is more efficient. If you think of an index as a thin table, scanning an index isn't a completely free option. It costs additional reads, so the planner has to make the decision whether the effort of reading the index to determine the location of records to pull is less costly than scanning the raw data. For small tables or tables with many search hits, scanning the table is often faster.

Now that we've given you some fat to chew about deciding on indexes, we'll explore how you can write your queries to change the planner's behavior.

9.4 Common SQL patterns and how they affect performance

You know how to force a change of plan by adding in indexes. There are more complex join cases where you can control the plan by stating your query in different ways. In this section, we'll demonstrate some of the common approaches for doing that.

Following are some general rules of thumb we'll 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're a bit slower than INNER JOINs. If you use one, make sure you really need it.

We'll start off 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, WHERE, or FROM part of a query. For large numbers of rows to be returned, you're much better off not using a subselect in the SELECT or WHERE clause, because it forces a query for each row, particularly if it's a correlated subquery. For small numbers of records to return it varies, but it's generally easier to read to put the query in the FROM clause. Sometimes you're better off not using a subselect at all. If speed becomes problematic, you may have to test various ways of writing the same statement.

> **Correlated subquery**
>
> A correlated subquery is a query that can't 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 it often results in slow queries.

The first exercise we'll 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'll ask this question with two vastly different queries. One is the naïve way in which people new to relational databases approach this problem, where they put the subselect in the SELECT. In some cases, counterintuitive to most database folk, this performs better or the same as the conventional JOIN approach. The second approach doesn't 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 for this data set.

```
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 as follows:

```
 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'll 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;
```

Here's 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
```

Using a join is generally faster than the SELECT 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's not asked for. In these cases, it's 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: You have no idea what complicated formula could be stuffed into a column.

EXERCISE: HOW MANY CITIES HAVE STREETS, HOW MANY STREETS, AND HOW MANY ARE LONGER THAN 1000 FEET?

In this exercise, we'll demonstrate the danger of subselects. When you see yourself having multiple subselects in your SELECT clause, ask yourself if they're really necessary. Once again we'll demonstrate this query using two different approaches, shown in the following listing. One is the naïve subselect way, and one is the JOIN way with CASE WHEN statements.

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;
```

The result of this query are shown in the following listing.

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 stclines_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, this query looks impressive because we've used complex constructs such as subselects, exists, and aggregates all in one query. In addition, 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 slow and long-winded query. The graphical explain plan, shown in figure 9.4, looks particularly beautiful, I think.

Even though this does look convoluted, it has its place. It's 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's not a bad way to go. It's an 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 isn't asked for.

PostgreSQL 9.0 join removal optimization

PostgreSQL 9.0 introduced an enhancement to the planner that allows it to remove unnecessary joins. This feature will make queries using complex views that have lots of joins but the query selects few of these fields comparable in speed to (or faster than) the subselect approach.

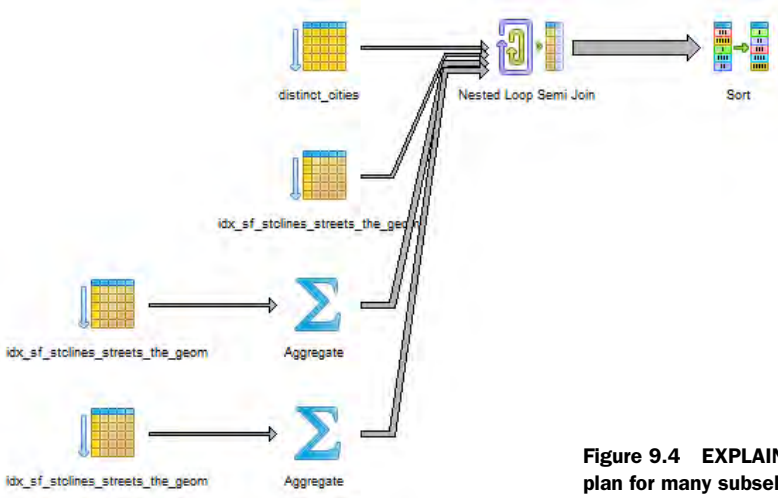


Figure 9.4 EXPLAIN ANALYZE graphical plan for many subselect queries

The following is the same exercise solved with a CASE statement instead of subselect. 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 this looks like the following.

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)
   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 you can see, this query is not only shorter but also faster. Also observe that in this case we're doing an INNER JOIN instead of a LEFT JOIN as we had done much earlier. This is because we care only about cities with streets. The results are shown in figure 9.5.

As you can see from the diagram, the graphical explain plan is simpler but not as much fun to look at. It does have an exciting HashAggregate that combines both aggregates into a single call as a result of our CASE statement.

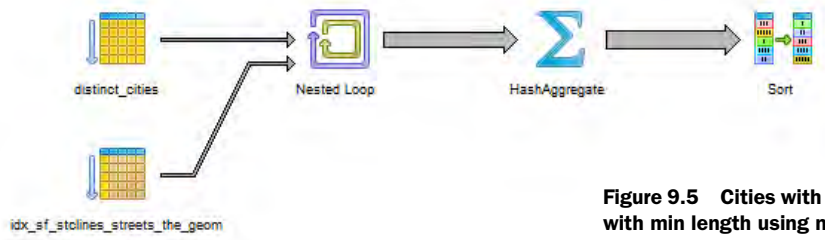


Figure 9.5 Cities with streets and count with min length using no subselects

9.4.2 FROM subselects and basic common table expressions

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 that can be used elsewhere in your statement. In PostgreSQL 8.4 ANSI standard common table expressions, there's a little twist added to the subselect that you know and love. The benefit of the common table expression is that it can reuse the same subselect in as many places as you want in your SQL statement without repeating its definition.

A couple of things about subselects used in FROM and in CTEs aren't entirely obvious, even to those with extensive SQL backgrounds:

- Although you write a subselect in a FROM as if it's a distinct entity, it's often not. It often gets collapsed in (rewritten, if you will). It's not always materialized, and the order of its processing isn't even guaranteed.
- For PostgreSQL CTE incarnation, although the ANSI specs don't require it, a CTE always seems to result in a materialization of the work table, although you can't tell this from the plan because it shows a CTE strategy.

CTE gotcha

Be careful with CTEs because as stated in PostgreSQL 9.0 and below they always result in a materialization of the table expressions. Try to avoid table expressions within your overall CTE that return a lot of records unless you'll be returning all those records in your final output. If your subquery returns many records and can be compartmentalized in the FROM clause, then you'd generally be better off with a subselect in FROM rather than a CTE.

For small subselect tables with complex function calculations such as spatial function calculations, you generally want the subselect to be materialized, although for large data sets you don't generally want subselects to be materialized. You can't directly tell PostgreSQL this, but you can write your queries in such a fashion as to sway it in one direction or another. For PostgreSQL 8.4+ you'd write those expressions as CTE subexpressions to force a materialization. For prior versions of PostgreSQL you can throw in an OFFSET 0, which tricks the planner into thinking there's a costly sort and makes it more likely that it will materialize or preprocess the costly subselect function calls. An example of using OFFSET follows. Note, however, that this isn't guaranteed to cache. We recommend this kludge only if you're observing significant performance issues. In those cases it doesn't hurt to compare the timings to see which gives you better performance. For most queries it doesn't make much of a difference, but for some, it can be fairly significant.

Here's 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 caching of the distance calculation by making the subselect look more expensive. Because distance is a fairly costly calculation, if you'll use it in multiple locations, you'll prefer it to be materialized.

We describe examples of where this situation arises at http://www.postgresonline.com/journal/archives/127-PostgresQL-8.4-Common-Table-Expressions-CTE,-performance-improvement,-precalculated-functions-revisited.html; Andrew Dustan also demonstrates this at http://people.planetpostgresql.org/andrew/index.php?/archives/49-Well-use-the-old-offset-0-trick,-99..html.

Now that we've covered the use of subselects and CTES, we'll explore Window functions and self-joins.

9.4.3 Window functions and self-joins

The Window function support introduced in PostgreSQL 8.4 is 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 that still can't be mimicked by a window in PostgreSQL. In PostgreSQL 9.0 the window functionality was enhanced, further minimizing the need of a self-join. For cases where you can use a window, and you aren't concerned about backward compatibility with prior versions of PostgreSQL, then using a window frame approach is generally much more efficient and results in shorter code as well. The next listing demonstrates the same spatial query: one with a window and one with a self-join (the 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
(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) )
  OFFSET 0
) As main
    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 this example, we employ lots of techniques in unison. We ❶ use a subselect to define a virtual worktable that will be used extensively to determine what cities are within 500 feet of ALBANY. ❷ We use the OFFSET hack described previously to encourage caching. Without the OFFSET our query takes 2 seconds, and with the OFFSET the timing is reduced to 719 ms, a fairly significant improvement. This is because the costly distance check isn't recalculated. ❸ We do 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`; that way our RANK will count at least our reference geom even if there's no closer object.

This 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 same query written using the RANK() Window function.

Listing 9.6 Using window frame to number results—PostgreSQL 8.4+

```
SELECT RANK() OVER w_dist As rank,
    p2.city As city_2, ST_Distance(p1.the_geom, p2.the_geom) As dist
FROM sf.cities As p1 INNER JOIN sf.cities As p2
 ON (p1.gid <> p2.gid AND ST_DWithin(p1.the_geom, p2.the_geom, 500))
 WHERE p1.city = 'ALBANY'
WINDOW w_dist AS (PARTITION BY p1.gid
    ORDER BY ST_Distance(p1.the_geom, p2.the_geom))
 ORDER BY RANK() OVER w_dist, p2.city;
```

❶ Reusable window frame

The equivalent window frame implementation using the RANK function is a bit cleaner looking and also runs much faster. This runs in about 215 ms, and the larger the geometries the more significant the speed differences between the previous RANK hack and the new one. In ❶ you see the declaration of WINDOW—WINDOW naming that doesn't exist in all databases supporting windowing constructs. It allows us to define our partition and order by frame and reuse it across the query instead of repeating it where we need it.

Now that we've covered the various ways you can write the same queries and how each one affects performance, we'll examine what system changes you can make to improve performance.

9.5 System and function settings

Most system variables that affect plan strategy can be set at the server level, session level, or database level. To set them at the server level, 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

```
SET somevariable TO somevalue;
```

To set at the database level use

```
ALTER DATABASE somedatabase SET somevariable=somevalue;
```

Setting at the function level requires PostgreSQL 8.3+:

```
ALTER DATABASE somefunction(argtype1,argtype2,arg…) SET somevariable=somevalue;
```

To see the current value of a parameter use

```
show somevariable;
```

Now let's look at some 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 most affect query speed and efficiency. For many of these, particularly the memory ones, there's no specific right or wrong answer. A lot of the optimal settings depend on whether your server is dedicated to PostgreSQL work, the CPU and amount of motherboard RAM you have, and even whether your loads are more connection intensive versus more query intensive. Do you have more people hitting your database asking for simple queries, or is your database a workhorse dedicated to generating data feeds? 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 under what loads.

CONSTRAINT_EXCLUSION

In order to take advantage of inheritance partitioning effects, this variable should be set to On for PostgreSQL versions prior to 8.4 and set to Partition for PostgreSQL 8.4+. This can be set at the server or database level as well as the function or statement level. It's generally best to set it at the server level so you don't need to remember to do it for each database 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's looking at a table that has children. This saves a few planner cycles over the previous On. The On is still useful, however, with union queries.

MAINTENANCE_WORK_MEM

This variable is the amount of memory to allocate for indexing and vacuum analyze processes. When you're 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 database 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 32 MB, 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 of the service after setting.

WORK_MEM

Work_mem is the maximum 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 doing intensive

things at the same time, this number should be fairly high. This is also a setting you can set conditionally at the function level or connection level, so 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)

The enable strategy options are listed here and all default to True/On. You never should change these settings at the server or database level, but you may find it useful to set them per session or at the function level if you want to discourage a certain plan strategy that's causing query problems. It's rare that you'd ever need to turn these off, and we personally have never had to. Some PostGIS users have 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's useful to turn off because it forces the planner to use an index that it seemingly could use but refuses to. It's a good way of knowing if the planner's costs are wrong in some way or if a table scan is truly better for your particular case or your index is set up incorrectly so the planner can't use it.

In some cases even settings that are turned off won't be abided by. This is because the planner has no other choice of valid options. Setting them off will discourage the planner from using them but won't 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 in figure 9.6.

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 with function settings. Functions also have specific settings relevant only for functions. We'll go over these in the next section.

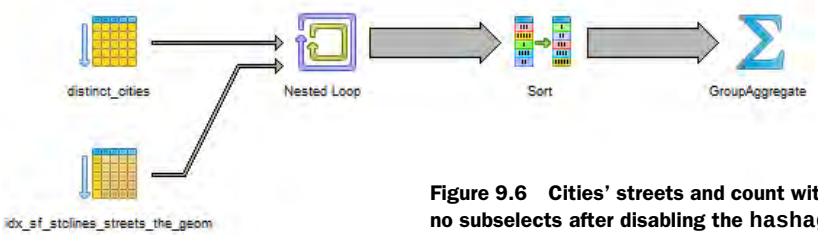


Figure 9.6 Cities' streets and count with min length using no subselects after disabling the hashagg strategy

9.5.2 Function-specific settings

Cost and row settings were introduced in PostgreSQL 8.3. The estimated cost and rows settings are available only to functions. They're part of the definition of the function and not set separately like the other parameters.

The form is

```
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's 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 behaved badly and sometimes the more-costly process ran 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 non-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 well in general, though no extensive benchmarking has been done on these functions to determine optimal settings.

ROWS

This setting is relevant only for set-returning functions. It's an estimate of the number of rows you expect the function to return.

IMMUTABLE, STABLE, VOLATILE

As shown previously where you have IMMUTABLE, when writing 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. Because 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 functions can generally be assumed to produce the same result, but they can't be treated as immutable because they have external dependencies such as dependencies on other tables that could change. As a result they perform 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 the same inputs. Functions that depend on time or some other randomly changing factor or that change data fit into this category because they change state. If you mark a volatile function such as random() non-volatile, then it will run faster but not behave correctly because it will be returning the same value with each subsequent call.

Now that we've covered the various system settings you can employ to impact speed, we'll take a closer look at the geometries themselves. Can you change a geometry so it's still accurate enough for your needs, but the performance of applying spatial predicates and operations is improved?

9.6 Optimizing geometries

Generally speaking, spatial processes and checks on spatial relationships take longer with geometries with more vertices and holes, and they're also either much slower or even impossible with invalid geometries. In this section we'll go over some of the more common techniques to validate, optimize, and simplify your geometries.

9.6.1 Fixing invalid geometries

The main reasons to fix invalid geometries are:

- You can't use GEOS relationship checks and many processing functions that rely on the intersection matrix with invalid geometries. Functions like ST_Intersects, ST_Equals, and so on return false or throw a topology error for certain kinds of invalidity regardless of the true nature of the intersection.
- The same holds true with Union, Intersection, and the powerful GEOS geometry process functions. Many won't 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. A contrib function called cleanGeometry.sql does a fairly good job of this. See http://trac.osgeo.org/postgis/wiki/UsersWikiCleanPolygons.

> **PostGIS 2.0 fixing invalid geometries**
>
> In PostGIS 2.0, a function called ST_MakeValid was introduced that can be used to fix invalid polygons, multipolygons, multilinestrings, and linestrings.

In addition to making sure geometries are valid, you can improve performance by reducing the number of points in each geometry.

9.6.2 Reducing number of vertices with simplification

Reducing the number of vertices by simplifying the geometries has both speed improvement effects and accuracy tradeoffs.

PROS

There are two major advantages of simplifying geometries:

- It makes your geometries lighter in weight, which becomes increasingly important the more you zoom out on a map.
- It makes relationships, distance checks, and geometry processing faster because these functions are generally slower the more vertices you have. You can gain

quite a performance increase by reducing an 80,000-point geometry to 8,000, for example.

CONS

These are the downside:

- Your geometries get less accurate. You're trading precision for speed.
- You often lose colinearity—things that used to share edges no longer do, for example.

Never simplify in WGS 84 lon lat or other lon lat SRIDs

Simplification assumes a planar model, and so applying it to something 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 lon lat after the simplification process.

The following listing is a quick example of simplification, where we simplify our state boundaries and then compare performance before and after.

Listing 9.7 Simplified state versus non-simplified

```
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) ) ;

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;

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) 21,964 ms—222 rows

(3) Simplified: 9,376 ms—222 rows

In listing 9.7 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 with a tolerance of 1500 meters (basically we treat points within 1500 meters as being equal). The units are in meters because our data is 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

In PostGIS 1.5, the ST_DWithin and ST_Distance functions were improved to better handle geometries with more vertices. As a result, the previous simplification isn't as stark in PostGIS 1.5 as it is in prior versions.

In many cases you can get away with simplification on the fly and still achieve about the same performance benefit as with a stored simplification. The only thing you need to be careful of is not to lose the spatial index on the table in the process. To achieve this, we'll create a new ST_DWithin function in the following listing that works against the simplified data 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

```
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;

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) ) ;

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) ) ;
```

❷ 1500 tolerance (14,727 ms—222 rows)

❸ 4000 tolerance (8,408 ms—222 rows)

In the example we ❶ create a new function that behaves like the built-in PostGIS ST_DWithin function, except that it applies a simplification before doing the distance within check. Note that the index check && is applied to the original geometries to utilize the spatial index on the tables. This new function takes in an additional argument compared with the standard PostGIS ST_DWithin: the simplification tolerance. In ❷ we don't get quite as much performance improvement (14.7 seconds) as with our similar stored states_simp1500 (9.8 seconds). However, ❸ by increasing the level of simplification, we get even faster performance (8.5 seconds). The trick is to find the maximum simplification with acceptable loss in accuracy.

In the next section, we'll demonstrate another kind of simplification, and that's removing unnecessarily small features from geometries.

9.6.3 Removing holes

In some situations you may not need holes. Holes generally add more processing time to things like distance checks and intersection. To remove them, you can employ something like the following 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 necessary because the ST_ExteriorRing function works only with polygons. We then convert the exterior ring to a polygon because the exterior ring is the linestring that forms the polygon. 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, only the small ones that don't add much visible or information quality to your geometry but do make other checks and processes slower. The next listing shows a simple method for removing holes of a particular size and is excerpted from the following article: 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
$$
SELECT ST_BuildArea(ST_Collect(b.final_geom)) as filtered_geom
  FROM (SELECT ST_MakePolygon((
    SELECT ST_ExteriorRing(a.the_geom) as outer_ring )
, ARRAY( SELECT ST_ExteriorRing(b.geom) as inner_ring
       FROM (SELECT (ST_DumpRings(a.the_geom)).*) b
      WHERE b.path[1] > 0 /* ie not the outer ring */
        AND ST_Area(b.geom) > $2)
 ) as final_geom
         FROM (SELECT ST_GeometryN(ST_Multi($1),
         generate_series(1,ST_NumGeometries(ST_Multi($1)))
                                  ) as the_geom ) a
       ) b
$$
  LANGUAGE 'sql' IMMUTABLE;
```

Big inner rings

Single to multi

To put this 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, keeping only the holes that are greater than 51,000 square feet.

The next query will tell us which records have been changed by the previous 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. It's needed because the ST_NumInteriorRings returns only the number of holes in the first polygon, so if we're dealing with a multipolygon, we need to expand to polygons and then count the rings. You should encapsulate this into an SQL function if you use this construct often. Once again, this is the explode, process, collapse spatial design pattern at work.

9.6.4 Clustering

In this section, we'll talk about two totally different optimization tricks that sound similar and even use the same terminology but mean different things. We'll refer to the first as index clustering and the second as spatial clustering (bunching). The term *bunching* is more colloquial than industry standard. The index-clustering concept is one that's fairly common and similarly named in other databases.

- *Index clustering*—By clustering we're 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 easy to pick. Your index seeks will be faster because when 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. It's done by taking a set of points, usually close to one another or related by similar attributes and aggregate by collecting them into multipoints. You can imagine in this case you'd be talking about 100,000 rows of multipoints versus 1,000,000 rows of points, which can be both a great space saver as well as a speed enhancer because you need fewer index checks.

CLUSTER ON AN INDEX

We've talked about this before in other chapters, but it's 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 don't 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 doesn't 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 specific to tables is the FILLFACTOR. Those coming from SQL Server will recognize this term. It's 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 adding to a page until it's that percentage full.

For static tables, you want the FILLFACTOR to be really high, like 99 or 100. A higher fill factor generally performs better in queries because 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're 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's no space on the page, then it will need to create a new page, more likely ruining your cluster until you recluster.

FILLFACTOR can be 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 MULTIPOINTS 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're comparing fewer multipoints to multipoints versus more records. The following query 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
   INTO work.observations_bunched
FROM us.observations
GROUP BY ST_SnapToGrid(ST_Transform(the_geom,2163),50000,50000)
    , obs_name;
```

1 Bulk insert

2 Transform and snap

This example takes the observation points we created in WGS 84 lon lat and clusters them into 5000x5000 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. The new dataset has 8,163 records compared to our original 27,316.

9.7 Summary

In this chapter we covered the various ways of improving the performance of spatial queries.

We discussed various approaches for writing spatial queries, how to troubleshoot query performance, how to optimize geometries, and what common settings in PostgreSQL can be changed to improve performance. Although many of these techniques focused on spatial queries, many can be applied to non-spatial queries as well.

PostGIS and PostgreSQL aren't 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. You'll learn not only how to view PostGIS output but also 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 showed you performance tips for getting the most speed out of your spatial queries. PostGIS is a seductive mistress widely courted by both commercial and open source tools. In part 3, we'll cover some of the more common open source tools that are used to complement and enhance PostGIS.

Chapter 10 covers SQL add-ons, such as the 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. You'll learn how to write stored functions in these languages and use them in SQL queries. We'll also cover the TIGER geocoder, which is a package of scripts, SQL functions, and PostgreSQL types that utilizes U.S. Census TIGER data to build geocoders and reverse geocoders. In addition we'll cover pgRouting, which is another package of SQL functions used to build routing applications and do various kinds of traveling-salesperson problems.

In chapter 11 we cover the server-side mapping servers and client-side mapping frameworks that are commonly used to display PostGIS data on the web. You'll learn how to display PostGIS data layered with third-party mapping layers such as OpenStreetMap, Google Maps, and Microsoft Bing. You'll also learn the basics of setting up GeoServer and MapServer and configuring them as WMS/WFS services.

Chapter 12 introduces popular open source GIS desktop tools used to display PostGIS layers. We cover OpenJUMP, uDig, Quantum GIS, and gvSIG.

In chapter 13 we venture into the latest addition to PostGIS—raster support. You'll learn how to use raster data in conjunction with vector data using the new PostGIS raster type.

10 Enhancing SQL with add-ons

This chapter covers

- TIGER geocoder
- pgRouting
- PL/R
- PL/Python

In this chapter, we'll cover common open source add-on tools that are often used to enhance the functionality of PostgreSQL. What makes these tools special is that they unleash the power of SQL, so you can write much more powerful queries than you can with PostGIS and PostgreSQL alone. They also allow for greater abstraction of logic, because you can reuse these same functions and database triggers across all your application queries.

The tools we'll be covering are as follows:

- *Topologically Integrated Geographic Encoding and Referencing (TIGER) geocoder*—A geocoding toolkit with scripts for loading U.S. Census TIGER street data and approximating address locations with this data. It also contains geocoding

functions on top of PostGIS functions for address matching. The benefits of using this toolkit instead of a geocoding web service are that it can be customized any way you like and you won't incur service charges per batch of addresses geocoded.

- *pgRouting*—A library and set of scripts used in conjunction with PostGIS functions. pgRouting includes various algorithms to perform tasks like shortest path along a road network, driving directions, and geographic constrained resource allocation problems (aka traveling salesman).
- *PL/R*—A procedural language handler for PostgreSQL that allows you to write stored database functions using the R statistical language and graphical environment. With this you can generate elegant graphs and leverage a breadth 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 breadth of Python functions for network connectivity, data import, geocoding, and other GIS tasks. You can similarly write aggregate functions in Python.

We expect that after you've finished this chapter, you'll have 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 TIGER U.S. Census data. The TIGER geocoder not only makes it easy to batch geocode data with SQL update statements but also provides geocoding functionality to applications via simple SQL select statements. Although the TIGER geocoder is specific to the TIGER U.S. data structure, its concepts are useful when creating your own custom geocoder for specialized data sets.

What is a geocoder?

A geocoder is a utility that takes a textual representation of an address, such as a street address, and calculates its geographic position using data such as street centerline geometries. The position returned is usually a lon 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 U.S. Census data ESRI shapefile format. For those, therefore, we're 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/downloads/tiger_geocoder_2009.zip. More details on getting Steve's latest code can be found in appendix A.

Another geocoder built for OpenStreetMap utilizes PostgreSQL functions and a C library. This one may be of more interest to people outside the United States or OSM data users, but we didn't have time to investigate and cover it. This one is called Nominatim and can be accessed from http://wiki.openstreetmap.org/wiki/Nominatim.

For our exercises we've taken Steve Frost's newer version and made some minor corrections to support the TIGER Census 2009 data. We've also changed the lookup table script to create skeleton tables we inherit from. We've introduced an additional script file called tiger_loader.sql that generates a loader batch script for Windows or Linux. We'll 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:

- Install the TIGER geocoder .sql files.
- Have Wget and UnZip (for Linux) or 7-Zip 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. You can then call this generated script 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 file packaged with the TIGER geocoder code on the *PostGIS in Action* book site. Now that we've outlined the basic steps, we'll 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, and pre- and post-process steps. The script to populate these configuration tables and the generation function loader_generate_script are in the file tiger_loader.sql. The key tables are as follows:

- *loader_platform*—This table lists OS-specific settings and locations of binaries. We prepopulated it with generic Linux and Windows records. You'll probably want to edit this table to make sure the path settings for your OS are right.
- *loader_variables*—These are variables not specific to the OS, such as where to download the TIGER files, which folder to put them in, which temp directory to extract them to, and the year of the data.
- *loader_lookuptables*—This table of instructions shows 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 function will generate the command-line shell script to download, extract, and load the data. For our example, we downloaded just Washington, D.C., data because 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 need more than one state and for a different OS, you would list the states as follows:

  ```
  SELECT loader_generate_script(ARRAY['DC','RI'], 'linux');
  ```

  The script will generate a separate script record for each state.

  You can then copy and paste the result into a .bat file (remove the start and end quotes if copying from pgAdmin) and then run the shell script.

The generated scripts use Wget, 7-Zip (or UnZip), and the PostGIS-packaged shp2pgsql loader to download the files from the U.S. Census, unzip them, and load the data into a PostGIS-enabled database. For Windows users we highly suggest using 7-Zip and installing Wget for Windows, as we described in chapter 7.

For Linux/Unix/Mac OS X users, Wget and UnZip are generally in the path, so you probably don't need to do anything aside from CD-ing into the folder where you saved your generated script. We use the states_lookup table in the tiger schema to fine-tune which specific states to download data for and generate the download path based on the new TIGER path convention. Then we have a single SQL statement that combines all these to generate either a Windows command-line 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 state's set of tables is prefixed with the state abbreviation.

We use inheritance because it's more efficient for large data sets because it allows for piecemeal loading or reloading of data. It has the side benefit that the TIGER geocoder doesn't need to know about these tables to use them transparently via the skeleton parent tables we've set up. It also gives us the option to easily break these tables into other schemas later any way we care to.

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 because it's too long to include, but you can download the code from the *PostGIS in Action* book site at http://www.postgis.us. Click the Chapter Code Download link, choose the chapter 10/TIGER_geocoder_2009 folder, and read the ReadMe.txt file 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-U.S. 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 in the following listing.

Listing 10.1 Example of the geocode function

```
SELECT   g.rating,
  ST_X(g.geomout) As lon,
  ST_Y(g.geomout) As lat,
  (g.addy).*
FROM geocode('1731 New Hampshire Avenue Northwest, Washington, DC 20010') As g;
```

❷ Lon and lat

❶ Get rating

❸ 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 ❶ rating field. For perfect matches, the 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 ❸ we're exploding addy 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. ❷ The result also includes a field called geomout, which is a PostGIS lon lat point geometry interpolated along the street segments. We display the lon lat components of this point.

If we wanted just individual elements of the addy object and not all of them, then we would write a query something like this:

Listing 10.2 Listing specific elements of addy in geocode results

```
SELECT g.rating,
  round(ST_X(g.geomout)::numeric,5) As lon,
  round(ST_Y(g.geomout)::numeric,5) As lat,
  (g.addy).address As snum,
  (g.addy).streetname || ' ' || (g.addy).streettypeabbrev As street,
  (g.addy).zip
FROM geocode('1021 New Hampshare Avenue, Washington, DC 20010') As g;
```

❶ Rounded lon lat coords

❷ Extract addy elements

In this example we also test 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 Zip Code and

a misspelled street. In this case, we get three possible results, all with slightly different ratings. ❶ We also want to trim down the number of digits of the lon lat, so we round the digits. We first cast them to numeric because the round function expects a numeric number and PostGIS returns double precision. This casting may or may not be needed depending on 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. ❷ We select specific elements out of addy and glue them together for a more appropriate output.

The result of this query is a table like table 10.1.

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 that you shouldn't use lon lat data with PostGIS linear referencing and other Cartesian functions. In this case it's more or less safe to do so because the street segments are generally so short that the approximation of the sphere projected to a flat surface (Plate Carrée projection) doesn't distort the results too much. The longer the street segments, the more erroneous your interpolations will be. It's also rare that street numbers are equally spaced along a street, so the interpolation is still a best guess under the assumption of a perfect distribution.

From the table, we can quickly deduce that the first record with a rating of 10 is most likely the best match. Now when geocoding a table in a batch, we may want just the first and best match plus the rating. The following listing is an update statement that will do that.

Listing 10.3 Batch geocoding with the geocode function

```
set search_path = ch10, public, TIGER;
CREATE TABLE addr_to_geocode(addid serial NOT NULL PRIMARY KEY,
  rating integer,
  address text,
  norm_address text, pt geometry);
INSERT INTO addr_to_geocode(address)
  VALUES ('1000 Huntington Street, DC'),
    ('4758 Reno Road, DC 20017'),
    ('1021 New Hampshare Avenue, Washington, DC 20010');

UPDATE addr_to_geocode
    SET  (rating, norm_address, pt)
    = (g.rating,
      COALESCE ((g.addy).address::text, '')
      || COALESCE(' ' || (g.addy).predirabbrev, '')
      || COALESCE(' ' || (g.addy).streetname,'')
```

❶ Create test data

❷ Batch geocode

❸ Multicolumn update

```
|| ' ' || COALESCE(' ' || (g.addy).streettypeabbrev, '')
  || COALESCE(' ' || (g.addy).location || ', ', '')
  || COALESCE(' ' || (g.addy).stateabbrev, '')
|| COALESCE(' ' || (g.addy).zip, '')
          ,
        ST_SnapToGrid(g.geomout, 0.000001))
FROM (SELECT DISTINCT ON (addid) addid, (g1.geo).*
  FROM (SELECT addid,  (geocode(address)) As geo
FROM addr_to_geocode As ag
  WHERE ag.rating IS NULL ) As g1
ORDER BY addid, rating
) As g
WHERE g.addid = addr_to_geocode.addid;
```

4 Just one rec per addid

5 Pick lowest rating

We 1 first create some dummy addresses to geocode. 2 Then we geocode all in the table. We're using the 3 ANSI SQL multi-column update syntax so we can simultaneously update the rating, norm_address, and pt with a single command. This can be broken out as well. 4 We create a subselect using PostgreSQL's unique DISTINCT ON feature and include only those records that we haven't already geocoded (ag.rating is NULL). Using DISTINCT ON (addid) will guarantee that we get only one record back for each address. 5 We order by addid and then by rating to ensure that we get the addresses 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 the 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 data type object and fed to the various geocode functions. The geocode function first does a normalize_address of the address and then feeds it to geocode_address. Following is a demonstration of normalize_address:

```
SELECT foo.address As orig_addr, (foo.na).*
  FROM (SELECT address, normalize_address(address) As na
    FROM addr_to_geocode) AS foo;
```

Note that 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 this query looks like table 10.2.

Table 10.2 Result of normalizing our test addresses

orig_addr	address	predirabbrev	streetname streettypeabbrev	
1000 Huntington Street, DC	1000	Huntington	St	
4758 Reno Road, DC 20017	4758	Reno	Rd	..
:				

10.1.4 Summary

In this section you learned how to load TIGER data and use the PostGIS TIGER geocoder. We hope we've provided enough detail for you to put it to work.

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 and best routes to take. Weights are arbitrary costs you assign to each road element based on criteria such as whether the road requires paying a toll, whether the road is congested, its capacity for traffic, 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 the traveling salesman problem (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 web services. pgRouting allows you to solve these problems right in the database and to share them across 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 your own. For Windows and Mac users, binaries are currently available for PostgreSQL 8.3 and 8.4. You can download the source and binaries from http://www.pgrouting.org/download.html. 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 that wrap the base functions as PostgreSQL SQL functions. Further instruction can be found in the pgRouting site and in the chapter 10 data download file. For convenience, we've collected them on our companion website at http://www.postgis.us.

In future versions of PostGIS after PostGIS 2.0, there are plans to integrate pgRouting into the PostGIS project similar to what has been done with the raster project. You can expect PostGIS 2.1+ versions to have routing capability as part of the PostGIS core.

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 best 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 have one of the most convoluted patterns we could find of all the major cities in the United States. A truck driver trying to pass through the cities via the shortest route has quite a few choices to make; furthermore, the shortest choice isn't apparent from just looking at the map; see figure 10.1. A driver entering the metropolitan area from the south and wishing to leave via the northwest has quite a few options. We'll 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, which have already been added by the ch10_data.sql script:

```
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 linestring two integer identifiers: one for the starting point and one for the ending point. The function makes sure

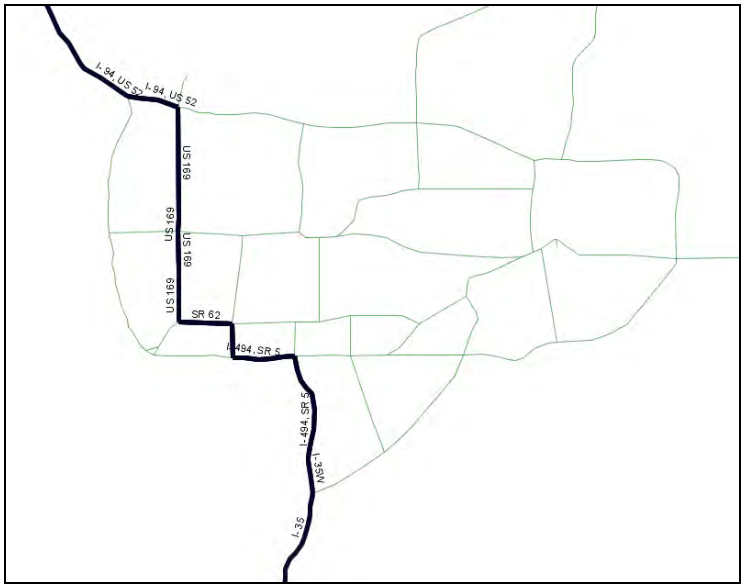


Figure 10.1 We plot the shortest route through the Twin Cities.

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 a cost to each linestring. Because we're looking at distance, we'll take the length of each linestring and fill in the length column:

```
UPDATE twin_cities SET length = ST_Length(the_geom);
```

Although we're not doing it in our example, we can 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 get live feeds of traffic conditions so that routes with major traffic congestion 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:

```
set search_path = public, ch10;
SELECT the_geom INTO ch10.dijkstra_result 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.

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 you try to minimize the cost of achieving an objective by selecting the cheapest solution to the problem. The concept of cost is something that's user defined. Don't limit yourself to solving problems involving time and distance. For example, you can easily download a table of calories from your local McDonald's, group the food items into sandwiches, drinks, and sides, and ask the question of 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've come across the need to find solutions to TSP-related problems. Many times we've given up because nothing was readily available to quickly accomplish that task. Although algorithms in many languages are available, setting up a network and pairing the algorithm with whichever database we were using at the time was much too tedious. We often resorted to suboptimal SQL-based solutions. How often have we hoped that 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 has to visit each city only once, how should he plan his itinerary to minimize total distance traveled?

To demonstrate TSP using pgRouting, we'll pretend that we're a team of inspectors from the International Atomic Energy Agency (IAEA, the United Nations' 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);
```

This table is included as part of the ch10–data.sql script.

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 include an id column and assign 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 an SQL string. This string must return a set of records with the columns source_id, x, and y. 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. The results are shown in figure 10.2.

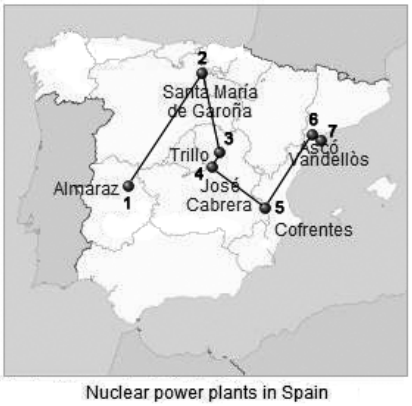


Figure 10.2 Optimized shortest-distance travel path for visiting all nuclear power plants in Spain starting from Almaraz

Like all algorithms in pgRouting, TSP works only on a Cartesian plane. SRID doesn't even come into play for TSP because TSP doesn't require a geometry column. In the interest of computational speed, routing problems rarely demand exact answers. 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. Don't apply the algorithms where your distances cover more than a hemisphere; otherwise, you'd 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'd 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'd have to repeat the process. In the next sections, we'll explore PL languages. PL languages and SQL are another kind of marriage that combines the 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's 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 allow 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, PL/PgSQL, and SQL. Stored functions are directly callable from SQL statements. This means you can do certain tasks much more easily than you would if you had to extract the data, import them into these language environments, and push them back into the database. You can write aggregate functions and triggers and use functions developed for these languages right in your database. These languages are prefixed with PL: PL/Perl, PL/Python, PL/Proxy, PL/R, PL/Sh, PL/Java, and so on. The code you write is pretty much the same as what you'd write in the language except for the 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:

- 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'll use them in—usually by running a `CREATE LANGUAGE` statement

The functionality of a PL extension 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 record 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'll find. PLs are registered in two flavors: trusted and untrusted. PL/Perl can

be registered as both trusted and untrusted. Most of the other PLs offer just the untrusted variant.

What's 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 it 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 care must be taken. It can delete files, execute processes, and do all things that the PostgreSQL daemon/service account has the power to do. Untrusted language functions must run in the context of a superuser, which means you need to create them as a superuser 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 demonstrate PL/Python and PL/R. We've chosen these particular languages because they have the largest offerings of spatial packages. We also think they're pretty cool languages. They're favorites among geostatisticians and GIS programmers. Both languages have only an untrusted flavor.

Python is a dynamically typed, all-purpose procedural language. It has elegant approaches for creating and navigating objects, and it supports functional programming, object-oriented programming, building of classes, meta programming, reflection, map reduce, and all those modern programming paradigms you've probably heard of. R, on the other hand, is more of a domain language. R is specifically designed for statistics, graphing, and data mining. It has a fairly large cult following among research institutions. It has many built-in statistical functions or functions you can download and install via the built-in package manager. Most of the functionality it offers you'll not find anywhere else except possibly in pricey tools such as SAS and MATLAB. You'll find tasks such as applying functions to all items in a list, doing 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 only 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 have them 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 use 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 a change of data in the database without ever touching the application edit code. This feature 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 available only in these languages to summarize the data. Imagine an aggregation function that returns a graph for each grouping of data. You can find some examples of this listed in appendix A.

In the sections that follow, we'll write some stored functions in these languages. These examples will have only a slight GIS bent. Our intent here is to show you how to get started integrating these in your PostgreSQL database and give you a general feel of what's 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 utilize 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 or leverage a large body of statistical packages that R provides. R is a favorite among statisticians and researchers because it makes flipping matrices, aggregation, and applying functions across rows and columns and other data structures almost trivial. It also has a breath of options for data import from various formats and has many contributed packages available for 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* by Robert I. Kabacoff or *Applied Spatial Data Analysis with R* by Roger S. Bivand, Edzer J. Pebesma, and V. Gómez-Rubio. Check out appendix A for other useful R sites and examples.

For the exercises that follow, we'll use R 2.10. Most of these should work on lower versions of R as well.

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 OS X, as well as Windows. Unix/Linux users may need to compile plr. For Windows and Mac OS X users, there are precompiled binaries. Any R from version R 2.5 through R 2.11 should work fine. We've tested it against R 2.5, R 2.6, and R 2.11. You can download source and precompiled binaries of R from http://www.r-project.org/. After the install, check your environment variables to make sure R_HOME is 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're using an installer, this is probably already done for you. If you're running under Linux, R should be configured with the option `-enable-R-shlib`. Note that you must use the version compiled for your

version of PostgreSQL. You can download the binaries and source from http://www.joeconway.com/plr/. You may need to restart the PostgreSQL service before you can use PL/R in a database.

- Run the packaged plr.sql file in the PostgreSQL database in which you'll be writing R stored functions. You need to repeat this step for each database you want to R enable.

If any of this is confusing or you get stuck, check out PL/R wiki installation tips guides at 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 the 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 the default search path. These steps are already done for you if you're using an installer. For Linux/Unix you can set this with an `export R_HOME = ...` and include it as part of your PostgreSQL init script. You may need to restart your postgres services for the new settings to take effect.

After you've installed PL/R in a database, run the following command to verify that your R_HOME is set right: `SELECT * FROM plr_environ();`.

10.4.2 Saving datasets and plotting

Now we'll take PL/R for a test drive. For these exercises, we'll use 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 pull data out of PostgreSQL and save it to R's custom binary format (RData). There are two common reasons to do this:

- It makes it easy to interactively test different plotting styles and other R functions in R's interactive environment against real data before you package them in a PL/R function.
- If you're teaching a course and are using R as a tool for analysis, you may want to provide your datasets in a format that can be easily loaded in R by students. For that reason, you may want to create PL/R functions that dump out data in self-contained problem set nuggets.

For this example we'll use the 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, as shown in listing 10.4. These objects can be dataframes (including spatial dataframes),

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
$$
  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'")
  save(dccounties,dczips, file="C:/Temp/dc.RData")
  return("done")
$$
language 'plr';
```

(1) Store results in R variables

(2) Save variables to R data file

In this example, we (1) create two datasets that contain Washington, D.C., counties and Zip Codes. 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 this save example, we run `SELECT ch10.save_dc_rdata();`.

We can load this data in R by clicking the file or by opening R and running a load call in R. Following are a couple of quick commands to try in the R environment:

To load a file in R:

```
load("C:/temp/dc.RData")
```

To list contents of file in R:

```
ls()
```

To view a data structure in R:

```
summary(dczips)
```

To view data in R, type the name of the data variable:

```
dccounties
```

To view a set of rows in a variable in R:

```
dczips[1:3,]
```

To view columns of data in R variable:

```
dczips$zcta5ce
```

These R commands demonstrate some commonly used constructs in R. Though not demonstrated, if you use the variable `<-` syntax, data gets assigned to an R variable instead of printed to the screen. Figure 10.3 is a snapshot of the commands we described.

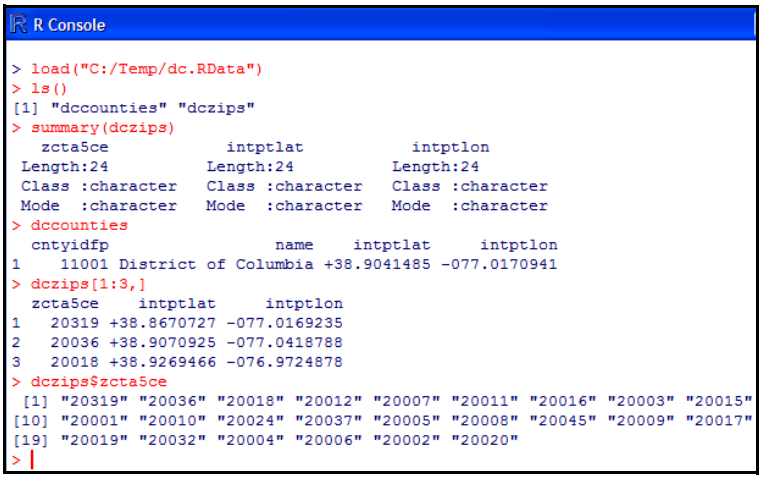


Figure 10.3 Demonstration output of running the previous statements in R

One task that R excels in is drawing plots. Many people, even those who don't care about statistics, are attracted to R because of its sophisticated scriptable plotting and graphing environment. In the next listing, 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
$$
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
            GROUP BY x ORDER BY x")
png('C:/temp/housepercap.png', width=500, height=400)
opar <- par(bg = "white") #set background color
plot(x=randdata$income,y=randdata$avgprice, ann = FALSE, type = "n")
yrange = range(randdata$avgprice)
abline(h=seq(yrange[1],yrange[2],
  (yrange[2] - yrange[1])/10), lty=1, col="grey")
lines(x=randdata$income,y=randdata$avgprice, col = "green4",
lty = "dotted")
points(x=randdata$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)
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 we're creating a stored function written in PL/R that will create a file called housepercap.png on the C:/temp folder of our PostgreSQL server. ❶ We first create random data by running an SQL statement using the PostgreSQL generate_series function and dump this in the randdata R variable. ❷ We then create a PNG file (note other functions such as pdf, jpeg, and the like can be used to create other formats), which all the plotting will be redirected to. ❸ We then draw our plot. The n type means no plot; it just prepares the plot space so that we can then draw ❹ grid lines ❺, lines, and ❻ points on the same grid. ❼ We close writing to the file with dev.off() and then return text saying "done."

Unable to start device devWindows

It's 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 PL/R runs in the context of the postgres service account. Any folder you wish to write to from PL/R must have read/write access from the postgres service/daemon account.

You run the previous function with an SQL statement:

```
SELECT ch10.graph_income_house();
```

When it's done, it will return "done." Running this command generates a PNG file, as shown in figure 10.4.

10.4.3 Using R packages in PL/R

The R environment has a rich gamut of functions, data, and data types you can download and install. All the functions and data structures are distributed in packages that are often referred to as libraries. When a package is installed, it becomes a library folder

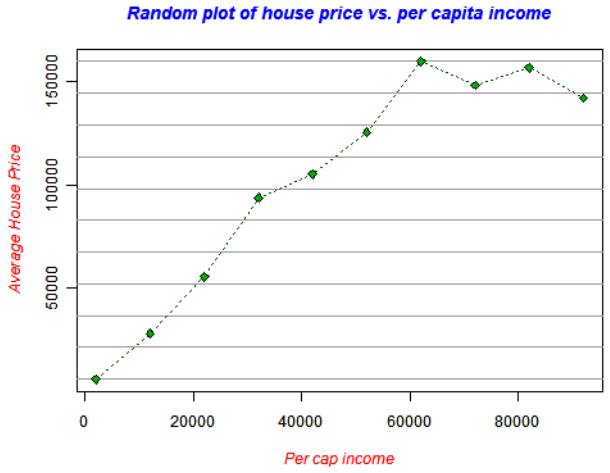


Figure 10.4 Result of SELECT `ch10.graph_income_house()`

you can see in the library folder of your R installation. R makes finding, downloading, and installing additional libraries easy using the Comprehensive R Archive Network (CRAN). Once a package is installed in R, you can then use it in PL/R functions.

What's particularly nice about the R system is that many packages come with demos that show the features of the package. Commands to view these demos are shown in table 10.3. They also often come with something called vignettes. Vignettes are quick tutorials on using a package. Demos and vignettes make R a fun, interactive learning environment. In order to use a vignette or demo, you first have to use the library command to load the library.

Table 10.3 Commands for installing and navigating packages

Command	Description
library()	Gives a list of packages already installed
library(packagename)	Loads a package into memory
update.packages()	Upgrades all packages to latest version
install.packages("packagename")	Installs a new package
available.packages()	Lists packages available in default CRAN
chooseCRANmirror()	Allows you to switch to a different CRAN
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 that you must load lib first)
help(package=somepackagename)	Gives summary help about a package
help(package=packagename, functionname)	Gives detailed help about an item in a package
vignette()	Lists tutorials in packages
vignette("nameofvignette")	Launches a PDF of exercises

In order to test the CRAN install process, we'll install a package called rgdal, which is an implementation of the Geospatial Data Abstraction Library (GDAL) for R. You saw GDAL in 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 there's no need to install sp if you install rgdal. In addition to supporting vector data using OGR commands, GDAL also supports raster data.

To install packages in R, we use the R command line or Rgui:

- At a command line type `R` (or `Rgui`).
- Once in the R environment, type the following:

To launch R:

```
R
```

To load the rgdal library:

```
library(rgdal)
```

If prior command failed, use the next commands to install rgdal and then load it:

```
install.packages("rgdal")
library(rgdal)
```

To get help about the rgdal library:

```
help(package=rgdal)
```

To quit out of the R console:

```
q()
```

> **Complex R packages**
>
> We installed rgdal on Windows. For this particular installation and some more complex R packages such as RGTK2, you may need to exit R environment first and then restart it before you can use the libraries. To use these libraries from PL/R, you also need to restart the PostgreSQL service after the library install in R. These steps aren't necessary for all R packages.
>
> For operating systems other than Windows, there might not be a precompiled 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'll test drive our rgdal installation with a couple of exercises.

10.4.4 Quick primer on rgdal

In order to test our rgdal installation, we'll run the following commands in R and then wrap some of this functionality in a PL/R function.

To load the rgdal library:

```
library(rgdal)
```

To get a list of raster drivers:

```
gdalDrivers()
```

To get list of vector geometry drivers:

```
ogrDrivers()
```

To return structure details about the 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 field-names and lengths in addition to a few other summary details.

Now we'll make the gdalDrivers list queryable from within PostgreSQL by creating a short PL/R function, as shown in the next listing.

Listing 10.6 Function to get list of supported rgdal raster formats

```
CREATE OR REPLACE FUNCTION r_getgdaldrivers() RETURNS SETOF text AS
$$
  library(rgdal)
  return(gdalDrivers()['long_name'])
$$
language 'plr';

SELECT driver
FROM r_getgdaldrivers() As driver
WHERE driver ILIKE '%arc%'
ORDER BY lower(driver);
```

❶ Load gdal lib

❷ List of drivers

We created a function called r_getgdaldrivers(), which returns ❶ just the long name field of the data.frame returned by the gdalDrivers function. Because the return type is a set of text, we can use the function as we would any other one-column table. In ❷ we do just that. The output of ❷ is shown in table 10.4.

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 this example we output a whole column of the data.frame. For large data.frames or other kinds of R data objects, you'll probably output just a subset of rows. For the next example, shown in listing 10.7, we'll create a function that allows us to query if a format is updateable or copyable. In this next example we'll demonstrate the use of the built-in R function subset, which behaves much like a SELECT SQL statement. We'll 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)
  return (subset(
      gdalDrivers(),
      create==param_create & copy==param_copy, select=c(long_name))
```

❶ R select where

```
    )
$$
language 'plr';

SELECT driver
  FROM r_getgdaldrivers(true,true) As driver
  ORDER BY lower(driver);
```

2 Use r function in regular SQL

This example demonstrates both the 1 subset() and c() functions in R. We create an overloaded function that will allow us to select just those drivers that are fitting for our create and copy functions. The subset() function is similar in concept to an SQL WHERE clause. The first argument is the data set we select from (parallel to a table in SQL), the next defines the WHERE condition with & instead of 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 for the function are used by name without any typecasting. PL/R automatically converts from a PostgreSQL boolean to an R boolean. 2 In the second code snippet, we test our new overloaded function r_getgdaldrivers to list only those drivers that support both copy and create.

In the next example, we'll start to use the drivers. We'll create a PL/R function that will read the metadata 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 file extension.

Listing 10.8 Reading metadata from image files with rgdal

```
CREATE TYPE gdalinfo_values AS(key text, value text);

CREATE or REPLACE FUNCTION r_getimageinfo(param_file text)
  returns SETOF gdalinfo_values AS
$$
    library(rgdal)
    tile_summary <-GDALinfo(param_file)
    result_labels <- labels(tile_summary)
    result_values <- tile_summary[result_labels]
   df <- data.frame(key = result_labels, value = result_values);
    return(df)
$$
language 'plr';
```

1 Custom data type

2 Load lib and store variables

3 Coerce into dataframe

In this example we create a PL/R function that will return a set of key-value pairs that represent our metadata. 1 We first create a data type to hold our results. Ideally we would have used OUT parameters for 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 metadata 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 file and use this to index and pick up the elements. 3 To return the data as a set of properties that can be read like a table, we coerce our data into a data.frame, filling the key column with the labels and the value column with the elements. Note that the

naming of the columns of our data.frame mirrors the gdalinfo_values data type. To run the function we do the following:

```
SELECT *  FROM r_getimageinfo('C:/Winter.jpg') As v;
```

The result is shown in table 10.5.

Table 10.5 Result of r_getimageinfo function call

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, and points. It also has spatial polygon, line, and point data frames. Spatial data frames are similar in concept to PostgreSQL tables with geometry columns. Pushing PostGIS data into these spatial data frames and spatial objects is unfortunately not easy at the moment.

There are two approaches you can use to push geometry data into these SpatialPolygons, SpatialLines, and other such 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 precompiled versions of rgdal aren't 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 using it to build an aggregate function, the whole thing would be next to impossible to manage.
- The second approach is to reconstitute an 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 into points. We've chosen the explode approach using the function ST_DumpPoints introduced in PostGIS 1.5 and will demonstrate this. These exercises will work only in PostGIS 1.5+, but if you need to use a lower version of PostGIS, you can implement your own ST_DumpPoints or copy one available in PostGIS 1.5. Note that the PostGIS 1.5 version of ST_DumpPoints is implemented as a PL/PgSQL function, so it's fairly simple to copy to a PostGIS 1.4 database.

For our first example, we'll 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)

  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]")

  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.dijkstra_result ) As r
            ORDER BY path[1]")


  geo.fact <- factor(geodata$gid)
  geo.id <- levels(geo.fact) #get unique list of linestring ids
  ngeom <- length(geo.id)

  geo.xy <- split(geodata[2:3], geo.fact)
  geo.geoms <- list()
  for(k in 1:ngeom ){
    geo.geoms[k] = Lines(list(Line(geo.xy[k])), ID = geo.id[k])
  }

  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)

  plot(sdf,xlim=c(-94, -93),ylim=c(44.5,45.5),axes=TRUE);

  lines(sdf_result, col = "green4", lty = "dashed", type="o")
```

1 SQL query dump points

2 Dijkstra

3 Regroup points into splines

4 Lines to SpatialLineDataFrame

Axes, limit range

```
  title(main= "Travel options to Twin Cities", font.main=4,
     col.main="red", xlab="Longitude", ylab="Latitude")
  dev.off()
  return("done")
$$
  LANGUAGE 'plr' VOLATILE
```

For this example we ❶ run a PostgreSQL query that explodes the street segments into points. We run another query ❷, which gives us the points for the results of Dijkstra pgRouting procedure. Comments in R are denoted by #, similar to Python. ❸ We regroup points into sp lines and then ❹ lines into SpatialLinesDataFrames.

We then load the R data.frame coming out of SpatialLinesDataFrames to plot on the same axis. Figure 10.5 shows the result of running the following query:

```
SELECT ch10.plot_routing_results();
```

Saving to other vector types

Although we didn't demonstrate it, after you've loaded the data into a spatial data frame, you can then use the saveOGR functions of rgdal to save them in 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, and we encourage you to explore it. You will see some fancy plots you can create with spatial data by running the following commands from R GUI console:

```
library(sp)
demo(gallery)
```

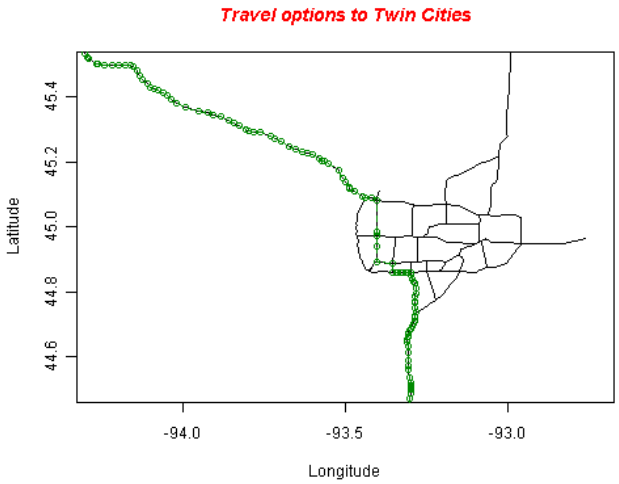


Figure 10.5 pgRouting Twin Cities results plotted with PL/R

10.4.6 Outputting plots as binaries

In all plotting examples we've used, we've saved the plots to a folder on the server. This approach is fine if you're 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. We're aware of three approaches:

- The first uses RGtk2 and a Cairo device. This approach is documented on the PL/R wiki and requires installing both the RGtk2 and Cairo libraries. In this approach you output a graph as a bytea. The problem we've found with this approach is that those two libraries are fairly hefty and require installation of another graphical toolkit called GTK. The other issue is that, as of this writing, it seems to crash in PL/R during the library-load process when PL/R PostgreSQL is running, at least under Windows. The benefit of this approach, however, is that it produces nicer-looking graphics and also prevents temporary file clutter, because it doesn't ever need to save to disk. It's also a one-step process.
- The next approach is to save the file to disk and let PostgreSQL read the file from disk. There's a superuser function in PostgreSQL called pg_read_file(..), but that's limited to 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-generated files to there and then use pg_read_file to get at 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 that 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 language favorite of GIS analysts and programmers. 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 GIS, OpenJUMP, GeoDJango, and even in commercial GIS systems 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. A PL/Python stored function can be called in any SQL statement. You can even create aggregate functions and database triggers with Python. In this section we'll show some of the beauties 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 only negotiates the

messaging between PostgreSQL and the native Python environment. This means that any Python package you install can be accessed from your PL/Python stored functions. Unfortunately, not all database data types to PL/Python object mappings are supported. This means you can't return some 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 doesn't 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 beforehand. In PostgreSQL 9.0 and above PL/Python supports arrays as input arguments, and the SQL to Python type support has been enhanced.

In order to use PL/Python, you must have Python installed on your PostgreSQL server machine. Because 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 precompiled PostgreSQL 8.3 PL/Python libraries packaged with most distros of Windows/Mac/Linux are compiled against Python 2.4, 2.5, or 2.6. These work only with the Python minor version they were compiled against.

Windows one-click installer

For Windows users the one-click installer 8.3 version of PostgreSQL is compiled against Python 2.5, and for PostgreSQL 8.4 it's against Python 2.6. The plpython.dll is already packaged with the one-click installer. In order to use it, you need to spend the five minutes to install the required Python version on your server and enable the language in your database.

If you're using the PostgreSQL Yum repository for the PostgreSQL installation, you can get PL/Python by doing this:

```
yum install postgresql-plpython
```

On most Linux/Unix machines, you can determine which version of Python PL/Python is compiled against by doing this:

```
cd /
locate plpython.so
ldd path/to/plpython.so
```

PostgreSQL 9.0 support for Python 3.0

PostgreSQL 9.0+ has support for PL/Python using Python 3.0. In order to use the newer PL/Python, you must use the plpython30u handler and enable it with plpython3u instead of plpythonu. plpythonu in PostgreSQL 9.0 defaults to a Python 2 major version. You can also use plpython2u in PostgreSQL 9.0+ to be more explicit. You can have both versions installed in a single database, but you can't run stored functions written in both languages in the same session.

Once you have Python and the PL/Python.so/.dll installed on your server, run the following statement to enable the language in your database:

```
CREATE PROCEDURAL LANGUAGE
'plpythonu' HANDLER plpython_call_handler;
```

If you run into problems enabling PL/Python, please refer to our PL/Python help guide links in appendix A. The common issue people face is that the required version of Python isn't installed on the server or the plpython.so/.dll file is missing.

10.5.2 Our 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

```
CREATE OR REPLACE FUNCTION python_addreduce(param_start integer,
    param_end integer)
  RETURNS integer AS
$$
 def add(x,y): return x+y
 return reduce(add, range(param_start, param_end + 1));
$$
  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 PL/Python function that 1 takes start and end numbers to define a range. 2 We first define a simple function 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 this example, we can do the following:

```
SELECT python_addreduce(1,4);
```

This gives us an answer of `10`.

10.5.3 Using Python packages

The Python standard installation comes with only the basics. Much of what makes Python useful is the large range of free packages for things like matrix manipulation,

web service integration, and data import. A good place to find these extra packages is at the Python Cheeseshop package repository.

In order to use these packages, we'll use the Python Cheeseshop package repository and the setup tool called Easy Install.

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 file is located in the C:\Python26\scripts folder for Windows users.

Now we'll move on to installing some packages and creating Python functions using these packages.

IMPORTING AN EXCEL FILE WITH PL/PYTHON

For this example, we'll use 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 in any OS. It doesn't have any additional dependencies, uses the standard setup.py install process, and, for Windows users, has a setup.exe file as well. For this exercise, we'll install it from the command line, which should work for most any OS if you've installed easy_install: `easy_install xlrd`.

We'll test our installation in listing 10.11 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 PL/PgSQL, 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)
RETURNS SETOF ch10.place_lon_lat AS
$$
  import xlrd
  book = xlrd.open_workbook(param_filename)
  sh = book.sheet_by_index(0)

  for rx in range(1,sh.nrows):
```

1 Import package

2 Skip headers

```
    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;
```

❸ **Append to result**

❶ We import the xlrd package so we can use it. We'll assume there's data in only the first spreadsheet. We loop through the rows of the spreadsheet, ❷ skipping the first row and using the Python ❸ 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 creating 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're doing an insert using the Excel file as the FROM source. Because the server is running the Python code and running under the context of the postgres daemon account, the Excel file path has to be accessible by the postgres daemon account.

IMPORTING SEVERAL EXCEL FILES WITH SQL

Now let's imagine you have several Excel files you got from a vendor, all with the same structure. They're all in one folder and you want to import them all at once. Sometimes they even dare to give you duplicated rows that are distinct in each file but repeated in other files. Here's where the real beauty of a PL married with SQL comes in.

First we'll create a Python function that lists all the files in a directory. Then we'll write another query to treat this list like a table to filter, and finally we'll write one SQL function to insert all the data using this list.

We'll create a function that lists the files in a passed-in directory path and returns 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.

PL/Python takes care of converting the Python list object to a PostgreSQL set of text objects.

Then we use the function in a SELECT statement, much like we can do with any table, applying LIKE to the output to further reduce the records we get back:

```
SELECT file
FROM ch10.list_files('/temp') As file
WHERE file LIKE '%.xls';
```

In our next example, we'll 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 PL/PgSQL and PL/Perl, but it 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 this 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 documented 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('/temp/' || file)).*
  FROM ch10.list_files('/temp') AS file
  WHERE file LIKE 'Test%.xls'
  ) As d;
```

This example is similar to our last, except that we're doing three interesting things. For each file in our Temp directory that starts with Test and ends with .xls, we're importing the data into our places table, but we're only importing distinct values across all the files, using the DISTINCT SQL predicate.

10.5.4 Geocoding with PL/Python

If perchance 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, MapQuest, Yahoo Maps, or Bing. You can find numerous Python packages at the Cheeseshop 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 either run the Windows setup if you're on Windows or compile it yourself on Linux. This particular package contains a geocoder, driving directions, and reverse geocoder functionality. Once you have the package installed, you can run the following exercises.

Geocoding web services caveats

Although calling web services with Python is easy, geocoding services tend to cost money or have limits on their 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 did earlier.

First we create a PostgreSQL data type to return our results:

```
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 lon 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 in table 10.6.

Table 10.6 Results of our Google Maps 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 take these steps:

- 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 through the database, retrieve the value for each record, geocode it, and update the database with the computed values.

By packaging our Python code as a stored function, we can reuse this same function easily in every 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 an address when the address changes in the database.

10.6 Summary

In this chapter, we introduced various tools 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'll further explore these tools and discover what other treats they hold in store.

Next 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 following chapter, we'll 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 15 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 wouldn't have been possible via traditional printed media. Only 20 years ago, a GIS practitioner wishing to share his data would have had to print out large maps on oversized printers. And 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 on the receiving end (the client).

In this chapter we'll cover web tools that work with PostGIS. We'll start with two server tools, MapServer and GeoServer, which can read data from PostGIS and serve

images or data according to OGC standards. We'll then move on to the client side of the equation, where we look at OpenLayers, a JavaScript-based tool that greatly enriches the viewing experience for the user. Along with OpenLayers, we'll check out the new GeoExt extension to OpenLayers based on the new ExtJS JavaScript framework.

11.1 GIS and the web

The first question many readers might ask would be why we need anything above and beyond the technologies widely available to produce 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. This section starts by pointing out the limitations of conventional web technologies in serving up dynamically generated maps, both from the server and from 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 extract our map at various zoom levels. Using conventional web server technology, we'd have to limit the user to a fixed set of zoom levels, generate the images beforehand, and serve them as requested. Now consider what would happen if the user would like to see only subsections of the map: We would have to slice up our maps beforehand and restrict users to picking from one of our prepared slices. There are two big problems here: First, we can't possibly predict what portions the user 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 zoom levels, and the problem becomes intractable.

The client side of the picture isn't much rosier. For zoom-level selectors we could use standard HTML combo boxes, but the drop-down list would have to be changed from map to map. If a map has three zoom levels, we'd have to prepopulate our combo box with three values. If the next map has 30 zoom levels, we'd have to have 30 rows in the combo box for the user to pick from. Using various programming technologies now available, such as Python, PHP, and ASP.Net, we can dynamically generate our HTML combo box, but this requires that the mapping person also be a web programmer—and not in just one language. The demands become even more challenging if users have to be able to draw rectangles around subsections to be blown up, add their own markers, or have pop-up description balloons when hovering over certain points of interest. These interface features would all require extensive programming on the client side. If server-side programming didn't already discourage the GIS specialist, the client-side programming surely will. What we need is a suite of client tools with useful controls for map viewing and editing already built. Sure, the suite will dictate the overall appearance and functionality, but this still is preferable to building our

own solution from the ground up. After all, our goal is to disseminate our maps, not to program web servers.

11.1.2 Mapping servers

Mapping servers have one central purpose: to render images for delivery to a client on the fly. As mentioned previously, conventional web servers can't serve up images unless they already exist, but generating and storing 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.

At the time of this writing, four major open source server products dominate the market: MapServer, GeoServer, FeatureServer, and SharpMap.NET. Because mapping servers are rarely the starting point of a GIS project, people generally start from a need to spatially extend existing web applications or to disseminate existing data via the web.

To decide which server products to use, we recommend that you judge how easily each fits 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 the data sources you already have, be they PostGIS, Oracle Spatial/Locator, SQL Server 2008, SpatiaLite, MySQL, shapefiles, raster, or something else?

What are web services?

Loosely speaking, a web service is a non-proprietary standard for function calls across the internet. The service accepts requests from clients usually using HTTP and standard messaging streams (raw get, posts, XML, JSON, SOAP, and the like) and returns the processed 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 written in XML. To consume web services, the requestor application generally creates stub classes to make the web service call indistinguishable from a local function call. Many tools are available to autogenerate stub classes, sparing you the pain of having to write them 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/deserialization of objects into XML, or some other format, so that they can traverse the internet.

PLATFORM CONSIDERATIONS

One of the most important deciding factors for choosing a tool is the platform requirements. If you're on a shared web host, you may not be able to use anything that requires installation. Even if you have complete control over your server, you may shy

away from technologies that require additional installation. Table 11.1 outlines the prerequisites for 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
CGI/Fast-CGI	Yes	No	No	No

MapServer is perhaps the most popular of these tools because it contains a lot of functionality and can run under practically any web server without requiring installation. Just drop the compiled .so/.dlls/.exe into 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 in many flavors, with PHP MapScript, Python MapScript, and C# MapScript being the most common. This allows for more granular control by allowing you to create layers and other map objects from PHP, C#, and Python server-side code. The downside of the MapScript interface is that it also requires writing more code in general than using the Common Gateway Interface (CGI) executable interface.

GeoServer is built on Java servlets. Some binary distributions of GeoServer come packaged with their own mini web server called Jetty. GeoServer requires an existing installation of Java 1.5+ SDK. If you need to run GeoServer under the context of an existing web server service, you'll need to get a servlet plug-in for your web server such as Tomcat and install the Java Web Archive (WAR) version. Unlike the other tools, it comes packaged with a user-friendly web-based administrative interface. This makes GeoServer a popular option for those who prefer GUIs and wizards over configuration scripts.

SharpMap.Net is a popular option for .NET programmers. It comes packaged as a .NET .dll. 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 don't have your own web server. On the downside, SharpMap.Net does a lot less out of the box than MapServer or GeoServer, and you'll 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 serve PostGIS layers in the various formats, you need to have psychopg or psychopg2 installed. It can run as a standalone Python server, as a CGI, or in Apache via mod_python. It can also be run under IIS if you have Python bindings enabled. It's trickier to set up than MapServer or GeoServer, so we won't be covering it. The main features that make it stand out from other web-mapping servers are that it's written purely in Python, it supports both the

OGC WFS as well as a much simpler feature service REST interface for both querying and editing feature data, and it has some kinds of data sources that some of the other mapping servers don't support. FeatureServer supported formats include data sources such as Twitter, SQLite, DBM, OSM, and all OGR (if you have the OGR Python plug-in configured). It can output in KML, GeoRSS, GeoJSON, GML, HTML, and OSM. However, it's a feature server and doesn't handle web image map requests.

OGC WEB SERVICE SUPPORT

You may recall from earlier chapters that OGC is short for the Open Geospatial Consortium, 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 won't limit end users to their particular web or desktop client. All the open source web-mapping clients and the desktop tools that we'll cover in chapter 12 consume OGC web services. Even proprietary desktop applications such as Manifold, Cadcorp, and MapInfo nowadays offer decent support for OGC web-mapping services. The most common of the web services defined by OGC are the following:

- *Web Mapping Service (WMS)*—For rendering vector and raster data as map images in JPEG, PNG, TIFF, or some other raster format. This is suitable if you want to show a map of an area, but downloading and rendering the data would be too processor intensive. For example, if you want to display maps on a mobile device with limited processing power, retrieving ready-made images from a WMS server 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. Geography JavaScript Object Notation (GeoJSON) is another option and is more processor friendly for consumption by JavaScript because it's 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 users need to highlight regions of a map and display attribute info or styling options, without making roundtrips to the server. It's often used in conjunction with WMS, where WMS would be used to show aerials or large zoomed-out regions of a map, and WFS for overlaid key features or more granular control when zoomed in.
- *Web Feature Service Transactional (WFS-T)*—For editing vector data in 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 access to the database.

There are other web services as well, such as Web Tiling Services (WTS) and Web Coverage Services (WCS). Table 11.2 is a brief summary of the key OGC web services and which tools support them.

The REST architecture is a lighter weight interface than WFS and relies on concepts of GETs, PUTs, and DELETEs to update data and output XML streams. A WFS that supports GET requests can be considered for all intents and purposes as a REST service.

Table 11.2 Web services 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*

* This means support via an extra downloadable plug-in or library.

SUPPORTED DATA SOURCES

All maps are derived from data. The WMS/WFS/WFS-T protocols allow various data sources to be accessed via one web interface. They provide an abstract interface for GIS data similar to ODBC and JDBC drivers for databases. All web-mapping server tools support various data formats. Table 11.3 describes which tool supports which format, so you can make an informed choice. They all support PostGIS geometries and ESRI shapefiles out of the box, so we left those out.

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*

* This means support via an extra downloadable plug-in or library.

11.1.3 Mapping clients

Once the web-mapping services have been set up, you need client applications to consume them. Client applications come in two 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 Quantum GIS, uDig, gvSIG, OpenJUMP, and countless others or a proprietary desktop tool such as Manifold, MapInfo, Cadcorp SIS, and ArcGIS desktop, to name a few. We'll cover the open source desktop tools in the next chapter.

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 this is that it gives you the ability to overlay proprietary non-OGC-compliant mapping layers with OGC WMS, WFS, and WFS-T layers.

OpenLayers is often extended to create more advanced or specific toolkits. Two common ones that build on top of OpenLayers are GeoExt, which is used by OpenGeo's GeoExplorer, and MapFish. 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 a user-authentication service among other advanced features.

11.1.4 Proprietary services

We'd be remiss if we failed to mention that the most popular web-mapping services around are still proprietary, such as Google Maps, Bing, and MapQuest. These services package server, client, and data together in a slick, easy-to-use interface and make mapping accessible to the general public. Though these packages are easy to use, each has its own proprietary JavaScript API with limited control over overlaying data. You won't be able to write SQL queries let alone represent anything more complex than points and line segments, at least without extensive effort.

One serious drawback is their proprietary and inflexible nature, even on the data level. You can't remove one core feature. For example, if you wanted to display foliage density over a region instead of the usual streets and places, you can't do so with these popular packages. You also can't suppress the commercial licensing clause of these packages. For recreational use, these packages are in most cases free, but once you start to use them for profit or for non-public websites, you'll find yourself needing to cough up a rather exorbitant licensing fee. Because each has its own custom API that's incompatible with any other one, you'll have to rewrite much of your custom data overlay logic when deciding to swap services.

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 the world the power of dynamic mapping on the internet and continue to lead the way in the development of display technologies. Because this book is devoted to open source solutions, we won't cover these proprietary JavaScript APIs, but we advise you to not lose sight of the important role they play on the World Wide Web today.

Each of these web GIS tools provides a lot of functionality out of the box. They do so by limiting you to certain protocols when you interact with your database and other spatial data. For many solutions that need only light support for maps but heavier

support for data, you may want to forgo web-mapping services altogether and build the logic 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 to 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 vector features would be slow and cumbersome. In this case, it's better 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. For this next example, we'll demonstrate using MapServer's WMS features.

11.2.1 Installing MapServer

MapServer is a mature product, and as such there's little need to compile from scratch unless you want to. There are already precompiled binaries for most any operating system. (See http://mapserver.org/download.html#binaries.)

WINDOWS INSTALL

For MS Windows installs several options are available. The OSGeo4W and MS4W are bulky installations because they include an Apache server and various other GIS open source packages.

We like using the FWTools package because it's much lighter weight, tends to be up to date, and also often contains the latest developer version. It also includes the C# Interop extensions to allow the use of MapScript from an ASP.NET (VB.Net or C#) environment. To deploy on a Windows IIS server as CGI, we usually do the following:

1. Extract the FWTools executable installer file (yes, you can treat it as if it were a zip file).
2. Copy the contents of the $HWNPARENT/bin folder to somewhere on the web server that's marked as allowing executables (this can be a cgi-bin or some folder you create that you mark as allowing executables).
3. Copy the proj_lib folder onto the web server. You'll need to reference the path in your MapServer map file later, but it doesn't need to be web accessible.
4. If you want to use .NET MapScript in VB.NET or C# or some other .NET language, then copy the csharp folder files into the bin folder of your .NET application. There are thread issues, especially in .NET MapScript, so many people prefer to use SharpMap.NET for mapping if tight integration with a .NET application is needed.

SECURITY CONSIDERATIONS

If you're going to have PostGIS layers, you may need to put the password in the map file or in 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 copyright reasons.

There are a couple of safeguards against this. Please do at least one of these:

- Don't put your map file in a folder that's web accessible. Admittedly, we tend to break this rule out of convenience of having everything related together.
- Use the msencrypt executable packaged with MapServer to encrypt the password, and 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 isn't served by a web server. For example, we use the .config extension in IIS because ASP.NET will never serve a file with this extension. Using an INCLUDE for the PostGIS connection string is also convenient, at least if all your PostGIS layers use the same database. This saves you from having to repeat the same information over and over again.
- If you have control of your own web server, you can block dishing out map files by editing your httpd.conf or in IIS mapping the files to 404.dll or some other IIS ISAPI processor.

11.2.2 Creating WMS and WFS services

MapServer supports its own non-OGC API as well as WMS, WFS, WCS, and other web service interfaces. We're 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 map file with WFS/WMS metadata sections, a set of fonts, a symbol set, and proj_lib will do.

For our map files, we like to use INCLUDEs for sections that we reuse repeatedly within the map or reuse across several maps, such as for the PostGIS connection string, or for general configurations like the location of the projection library.

The following listing shows what such a map file looks like.

Listing 11.1 Map with INCLUDEs

```
MAP
  INCLUDE "config.inc.map"
  NAME "POSTGIS_IN_ACTION"  #name to give your map service
  EXTENT 221238 881125 246486 910582
  PROJECTION
   "init=epsg:26986"
  END
  WEB
    MINSCALEDENOM 100
    MAXSCALEDENOM 100000
    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 plug-ins

2 Default projection of map

3 WMS/WFS metadata

❶ We include a file called config.inc.map that contains the paths to our projection library, symbolset, and fontset. All INCLUDEs are relative to the location of the file they're included in. ❷ This defines the default output projection of the map if none is given. Each layer can be in a different projection, but they'll be reprojected to the map projection when the map is called. This projection is often overridden in WMS calls with the SRS parameter. ❸ The metadata section is particularly important, because this makes the map file behave like a true WMS/WFS. The `ows_*` elements are shorthand for WFS and WMS so properties that are the same for both don't have to be specified twice. WFS version 1.0.0 (supported by MapServer 5.6) can have only one SRS. The WMS standard allows many SRSes, and the ones listed are the ones the WMS service will allow as parameters 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's shown in the following 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're on Windows, you can copy the fonts you'll 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 one of the layers in our layers.inc.map file. Note that you can include layers directly in the main map file.

Listing 11.2 Sample layer from layers.inc.map

```
LAYER
 NAME major_roads
 TYPE LINE
 STATUS ON
 DUMP TRUE
 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"
 END
 CLASS
  COLOR 255 0 0
```

❶ Name and type

❷ Database config

```
  LABEL
   TYPE truetype
   FONT arial
   MINDISTANCE 50
   POSITION AUTO
   ANGLE AUTO
   SIZE 6
   COLOR 0 0 0
  END
 END
```



Every map layer starts with ❶ LAYER and has a NAME and TYPE. TYPE for PostGIS layers is usually LINE, POINT, POLYGON, or ANNOTATION. ❷ We include a file called postgis.config and will include this for each of our PostGIS layers to define the connection string to our PostGIS database. ❸ MapServer supports angled text, which is useful for labeling streets. Using ANGLE AUTO, the labels will wrap along the line segments.

The postgis.config file looks something like this:

```
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, but how do we turn this map file into a WMS/WFS service? We call the MapServer CGI with the map file as argument. The following code snippet calls the GetCapabilities request to show what layers and functionality are provided:

```
http://yourserver/cgi-bin/mapserv.exe?map=c:/mapserv/maps/
     postgis_in_action.map&
     ➥ REQUEST=GetCapabilities&SERVICE=WMS&VERSION=1.1.1
```

11.2.3 Calling a mapping service using a reverse proxy

Specifying a map file for each call is often unwanted. Many people prefer to set up either a CGI script or a reverse proxy so that the map file doesn't have to be explicitly named. You can do more with a reverse proxy than with a CGI script.

> **What is a reverse proxy?**
>
> A *reverse proxy* is a server that behaves as a client and has access to other services such as web-mapping servers that a requesting client can't directly access. It's often used for load balancing by accepting requests from a web browser on the outside and funneling them to the least-busy mapping server. In addition, it can call services on other ports on the same machine.

If we use a reverse proxy or a cgi-bin script, our long map URL example can be reduced to

```
http://yourserver/GetPAMap.ashx?REQUEST=GetCapabilities&SERVICE=WMS
➥ &VERSION=1.1.1
```

In listing 11.3, 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're using PHP, you can implement similar logic using curl. It can also be used to set up GeoServer web-mapping services, for example, if you want GeoServer to run on its own Apache or Jetty web server on a local port or even on a separate server in your internal network, while keeping the regular port 80 for a regular Apache or IIS server. The next example deals only with GET requests, which is generally what most WMS servers use. For POST you can do a check on the Request method by looping through the REQUEST and POST variables.

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);
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") {
  sr = new System.IO.StreamReader(
  ➥ WebResponseObject.GetResponseStream());
  context.Response.ContentType = "application/xml";
  context.Response.Write(sr.ReadToEnd());
}
else {
  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**

2 **XML request to MapServer**

3 **Image request to MapServer**

We first ❶ loop through all the arguments received via the client query string. ❷ We then check to see if the OGC request is a GetCapabilities or GetFeatureInfo, and if it is, we assume that the result returned by our internal server is XML. If it isn't, we'll assume it's an image and ❸ process it as such.

In order to overlap our PostGIS MapServer layers using our reverse proxy, we'd use code similar to that in the following listing.

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 map-serving program similar to MapServer.

11.3 Using GeoServer

GeoServer is similar in flavor to MapServer except that it's a bit heftier and comes with an administrative user interface, so there's not as much need for manually configuring files with a text editor, and it supports WFS-T.

11.3.1 Installing GeoServer

GeoServer has several installation packages that can be downloaded from http://geoserver.org/display/GEOS/Stable.

- Setup installers for Windows and Mac guide you through the setup. They come with the mini web server Jetty.
- Java binaries are available for all operating systems. You need only extract them to a folder and manually set the environment variables. Jetty is included too.
- A web application archive (WAR) is available for those who already have a servlet engine installed on their server and just want to run GeoServer as another servlet application. This one doesn't come with Jetty.

We chose the Java binary geoserver-2.0.1 version. To set it up, do the following:

1. Make sure you have Java JDK 1.5+ installed.
2. Extract the folder into the root, for example, C:\geoserver or /usr/local/geoserver.

3 On Windows, set the appropriate system environment variables. JAVA_HOME would be something like C:\Program Files\Java\jdk1.6.0_16 (or whatever JDK you have).
4 `cd` into the geoserver\bin folder and from the command line run startup.bat (for Windows) or startup.sh for Linux/Unix.
5 You then should be able to get to the administrative panel by navigating to the following 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. Follow these steps:

1 From the Admin menu > Data, choose Workspaces and click to add a new workspace. Your New Workspace screen should look something like figure 11.1.
2 From the Admin left navigation menu choose Data >Stores.
3 Click Add New Store and then choose PostGIS from the list of options, as shown in figure 11.2.

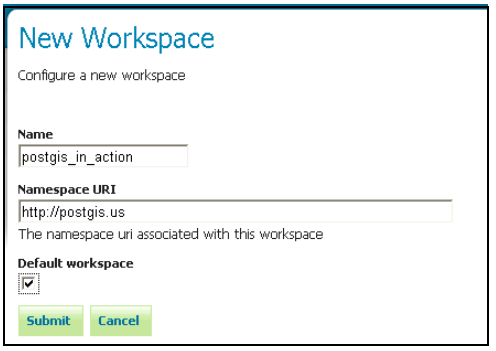


Figure 11.1 Setting up a GeoServer workspace

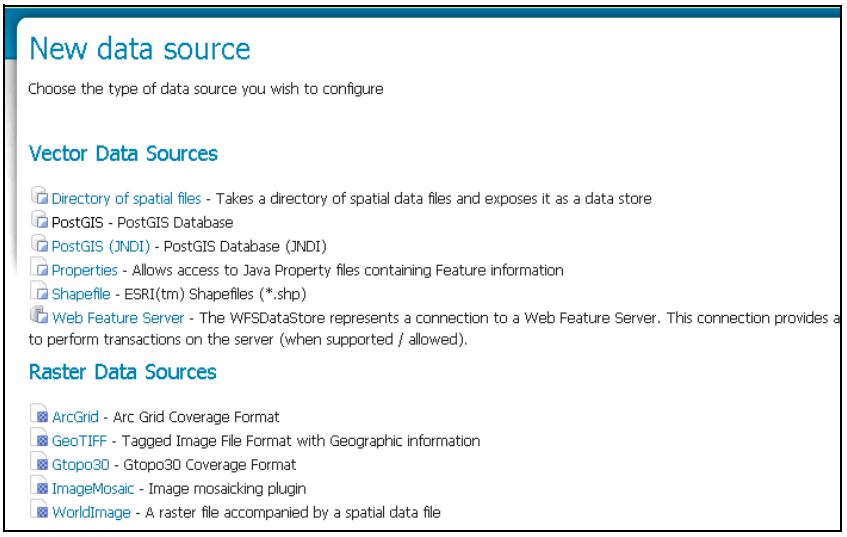


Figure 11.2 Adding a GeoServer PostGIS data source

4. Give the data source a name—`ch11`—and fill in all the credentials asked for. By default GeoServer uses the public schema, which means it will list only layers from that schema. If you want it to list a different schema, like in our case ch11, then replace `public` with `ch11`.
5. Select Layers from the Admin menu, click Add a New Resource, and choose the postgis_in_action store you created previously. Your screen should look something like figure 11.3.



Figure 11.3 Selecting PostGIS layers

GeoServer data stores from other schemas

It's possible to leave the schema setting blank in GeoServer for PostGIS, and the layer chooser will list them all. However, we've found that in that case publishing layers in non-public will throw an error. So be sure to create a different data store for each schema you want to publish.

6. Publish the layer you want. Make sure to choose Compute Bounding Boxes from Data.
7. Click Add New Resource.
8. Repeat steps 6 and 7 for each layer you want to publish.

11.3.3 Accessing PostGIS Layers via GeoServer WMS/WFS

Once you've published your PostGIS layers, you can quickly see them via the Layer Preview menu link. Figure 11.4 shows what that screen looks like. Note that it also shows the OpenLayers code to be used to call the layer. It also shows GeoJSON as a direct WFS output format.

OpenLayers is a popular web-mapping client companion for GeoServer and UMN MapServer. As you saw in the layer preview, GeoServer even autogenerates OpenLayers sample JavaScript code to display each of your layers.



Figure 11.4 Layer preview screen of GeoServer

In the next section, we'll introduce you to using OpenLayers and GeoExt. OpenLayers and GeoExt are 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 data grids and other controls that give the web application more of a desktop feel. These two frameworks are useful for commercial web services, WMS services offered by GeoServer/MapServer/SharpMap.NET, and for scripted applications with languages such as ASP.NET or PHP.

11.4 Basics of OpenLayers and GeoExt

In the beginning, mapping services like Google Maps, Virtual Earth, MapQuest, and Yahoo had their own proprietary JavaScript APIs to access their data. This was a Bad Thing, because if you decided you liked the maps of service A better than the maps of service B, or if usage and pricing became too cumbersome, then you had to rewrite everything. Worst yet, if you wanted to feed your own data via an OGC WMS or ArcGIS/IMS server for your area of interest, it was hard to integrate the 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, or, better yet, to be used together in the same map. OpenLayers started life as an incubation project of MetaCarta (now a part of Nokia), because it needed to create an easy-to-use toolkit for customers to digest its 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 Google Maps, Virtual Earth (Bing), MapQuest, Yahoo, and ArcGIS Rest, using the same interface for all

- Layer classes to access OGC-compliant map servers WMS, WFS, and WFS-T, again using the same fairly consistent map layer creation call
- The ability to overlay all these competing proprietary services in one map
- Various controls to build custom menus, toolbars, and widgets to enable map editing

All these things are wonderful, and that's why OpenLayers has become so popular. Most great things aren't without their tradeoffs though. So what are these tradeoffs?

- It's hard to get to the deep features of a proprietary service, such as the 3D street views provided by Google Maps and Bing. This may change as new OpenLayers layer classes are added to support these features. Note that GeoExt does have controls to get to Google Maps Street View and to synchronize it 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.org. Class documentation is available, although you'll probably find the numerous code samples to be much more useful for getting started. Because OpenLayers is nothing more than a glorified JavaScript file, you can download the file and use it directly from your web server. Alternatively, you can link your code directly with the version on OpenLayers, 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 (Virtual Earth), OpenStreetMap (OSM), MapServer-specific 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're going to use a base layer offered by OpenStreetMap and add a WKT layer with points marking New York City, Los Angeles, Chicago, Houston, and Philadelphia (the five largest cities in the United States at the time of writing). We want you to observe from the code that an OpenLayers HTML file almost always comprises the following sections:

- OpenLayers and other relevant JS includes. For custom layers such as OSM, Google, or Bing, you'd include the script source that points to those sites. In this case we include the custom script source from the OpenStreetMap site that includes the OSM class. Because OpenStreetMap builds its sites on OpenLayers as well, it extends the OpenLayers base classes. For other layers such as Google, you'll 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's defined in JavaScript and is created and initialized in an initialization method.
- The call to the initialization method, either in the body onload event or at the end of the JavaScript section. We don't like the first method, because certain languages like ASP.NET get messy when calls are put in the body load events.

- The full map is available at http://www.postgis.us/demos/chapter_11/osm_newengland1.htm and can be downloaded as part of the chapter 11 download. The following listing shows the OpenLayers general setup.

Listing 11.5 OpenLayers general setup

```
    <script src="js/ol28/OpenLayers.js"></script>
    <script
src="http://www.openstreetmap.org/openlayers/OpenStreetMap.js"></script>

    <script type="text/javascript">
        var lat=43.66596;
        var lon=-73.13868;
        var zoom=7;
        var map; //complex object of type OpenLayers.Map
        function init() {
            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")
            } );

            layerMapnik = new
            ➥ OpenLayers.Layer.OSM.Mapnik("OSM Mapnik");
            map.addLayer(layerMapnik);

            if( ! map.getCenter() ){
                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 OSM classes
3 Declare center and zoom
4 Instantiate OpenLayers map function
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 this script, you should download and reference it locally, as shown in this code. You can also link directly to the version on the OpenLayers website, http://openlayers.org/api/2.8/OpenLayers.js, if you want to get going quickly. 2 We include a link to additional OpenStreepMap classes. In the case of Google Maps or Bing, you'd need to include scripts from those providers to use as map layers. 3 We declare our global variables. The zoom denotes the default zoom level for our PanZoomBar. 4 The init function is central to the setup. It instantiates the OpenLayers map and loads it into a div

called "map" (you can call the div anything you want). We load only a few basic controls. Note that because we set the displayProjection to EPSG:4326, the mouse position is shown in lon lat instead of map units. ❺ We declare the OSM tile class. There are several to choose from: Mapnik, CycleMap, and Osmarender, each of which has a slightly different look and feel and data. ❻ We center the map at the lon lat point we declared in ❸. Note the transformation from lon lat to map units.

In our next part, we'll add the body of our HTML page:

```
<div style="width:100%; height:100%" id="map"></div>
<script type="text/javascript" defer="true">
   init();
</script>
```

We create the div for the map at the position where we want to place the map on the screen. In this case we set its width and height to 100% by 100%, so the map will expand dynamically to fit the page. In many cases you'll set this to a fixed size, expressed in pixels, for example, 500px. We then include the JavaScript with the defer options so it isn't called until the rest of the page has loaded.

After this our map will look like the one in figure 11.5.

In many cases, the user should be able to be able to pick the layers to display from a menu or toggle between several base layers. In order to allow this, we'll add the OpenLayers control called LayerSwitcher as well as another third-party layer. Some layers are always base layers, but to make an included WMS layer a base layer, you need to set the isBaseLayer property of the layer explicitly to true.

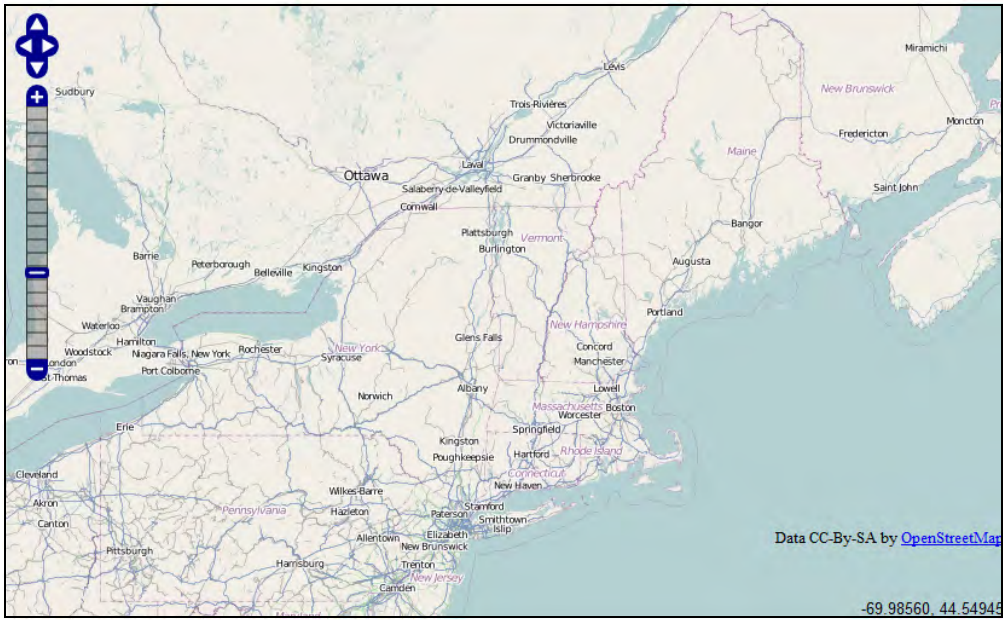


Figure 11.5 Result of our map in listing 11.5

In the following listing, we'll add Yahoo! Maps and the layer switcher control. The resulting map can be seen at http://www.postgis.us/demos/chapter_11/osm_newengland2.htm.

Listing 11.6 Revising the map to have Yahoo as an option

```
<script
  src="http://api.maps.yahoo.com/ajaxymap?v=3.0&appid=postgisus">
</script>                                                          ❶ Include Yahoo! map classes
:
function init() {
      :
var lyryahoohyb = new OpenLayers.Layer.Yahoo(                      ❷ Create Yahoo layer and add
                "Yahoo Hybrid",
                {'type': YAHOO_MAP_HYB, 'sphericalMercator': true}
            );
map.addLayer(lyryahoohyb);

map.addControl( new OpenLayers.Control.LayerSwitcher() );          ❸ Add layer switcher
      :
}
```

In ❶ we add the script source for Yahoo! Maps, which is needed for the OpenLayers Yahoo layer class. This class only acts as a proxy and translates from the OpenLayers API to the Yahoo API. ❷ We then revise our init function to create the Yahoo layer. Note the sphericalMercator setting. Without this, the Yahoo layer can't be overlaid with the OSM layer, which is in a Mercator projection. ❸ We add a layer switcher control so that the user can toggle back and forth between the two layers.

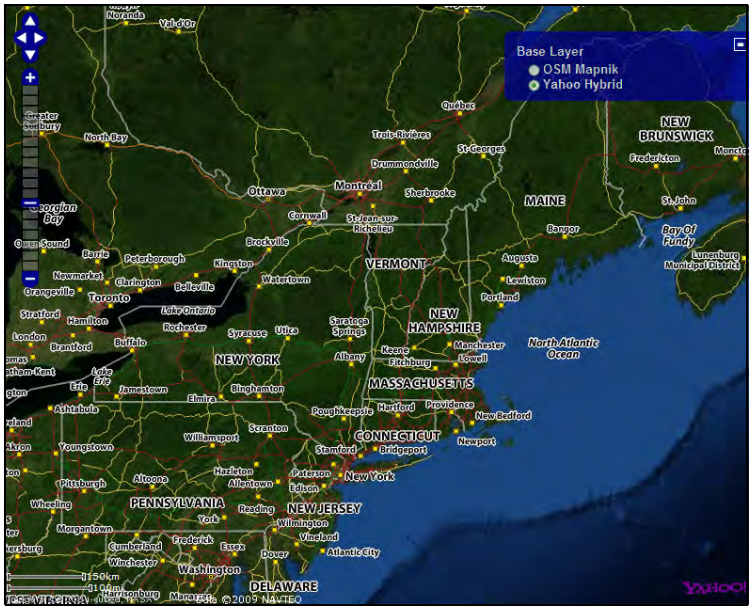


Figure 11.6 Map after adding change in listing 11.6

You can see in figure 11.6 the layer switch control showing the available layers and with the Yahoo layer currently selected.

We've just experimented with base layers. For any map, there can be only one base layer selected. The other type of layer is an overlay layer that can be combined with the base layer. You can select as many overlays as you want to be shown together.

ADDING WMS LAYERS TO A MAP

All open source web-mapping servers support the OGC web-mapping service standard. This means you can add your own layers as well as layers from third parties.

In this next example we'll demonstrate how to add MassGIS layers to our OpenLayers map. MassGIS has a useful page describing how to use its web services as well as detailing the fundamentals of the WMS and WFS standards: http://lyceum.massgis.state.ma.us/wiki/doku.php.

For this example, we're going to use the OpenStreetMap Mapnik map style as a base layer and add MassGIS WMS layers. Note that MassGIS, the primary provider of GIS data for Massachusetts, uses GeoServer. So when you set up GeoServer, the way you overlay your services will be pretty much the same as you see here. This particular example can be viewed at http://www.postgis.us/demos/chapter_11/olmapmassfish.htm.

Listing 11.7 Adding 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="

  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 EEOC</a> Fish',
   'isBaseLayer': false, 'visibility': true, 'buffer': 1,
  'singleTile':false, 'tileSize': new OpenLayers.Size(200,200) })
      );
    $('legend').innerHTML =
    ➥ 'Layers from MassGIS (EEOC)
<br />Lobster <img src="' + massgiswsleg +
➥ 'massgis:MORIS.QUAL_COMM_FISH_LOBSTER" /><br />' +
    'Flounder: <img  src="' + massgiswsleg +
    ➥ 'massgis:MORIS.QUAL_COMM_FISH_WFLOUNDER" /><br />' +
    'Black Sea Bass: <img  src="' + massgiswsleg +
    ➥ 'massgis:MORIS.QUAL_COMM_FISH_BSB" />';
```

1 Add WMS layer

2 Pull legend images from WMS

In this code snippet we add 1 three seafood layers from MassGIS web services. These are overlays by default. Because MassGIS doesn't have a default attribution, we add an attribution text that will appear in the attribution section whenever this layer is

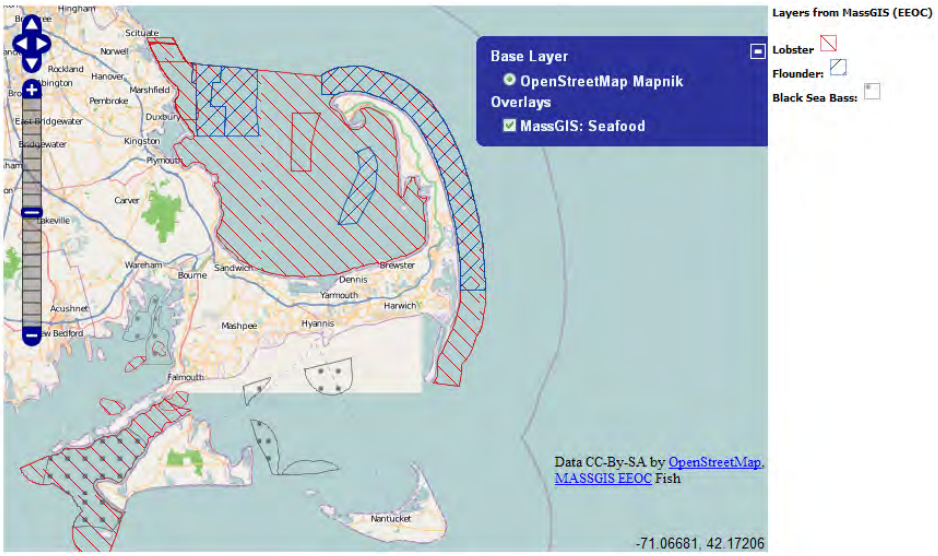


Figure 11.7 Example of overlaying WMS layers on an OpenStreetMap base using the code in listing 11.7

selected. ❷ We also use another feature of WMS, the GetLegendGraphic, to get the graphics for each of the layers and put it in a div element called "legend."

The result of this snippet is shown in figure 11.7.

For our next example, we're going to enhance our OpenLayers experience with GeoExt.

11.4.2 Enhancing OpenLayers with GeoExt

In the beginning, all neogeographers were happy with OpenLayers. With happiness came the realization that we can become happier still. People wanted grids to show attribute data. We wanted to be able to drag and drop things. We wanted to sort tables and have tree menus. We wanted selections on a grid and to use them to reposition the map. We wanted sliding controls, date pickers, and charts. In essence, we wanted to build web-mapping applications that felt more like desktop applications without the need of Flash and Silverlight plug-ins. All these things could be done with OpenLayers if we were willing to do the additional custom JavaScript programming, but a lot of this functionality already existed in ExtJS. ExtJS is a popular JavaScript API for making rich web applications that look like desktop applications. What the GeoExt project did was to combine the OpenLayers mapping system with the ExtJS web application–building interface to create something that would be the best of both toolkits. Haiti Crisis Map, http://haiticrisismap.org/, shown in figure 11.8, is an example of an application built on GeoExt that uses ExtJS accordion panes to enable different features. Many of the GeoServer administrative web interfaces are now also built with GeoExt.

In short, you can use OpenLayers by itself, and most people still do, or you can enrich it with GeoExt. We can't talk about GeoExt without first demonstrating

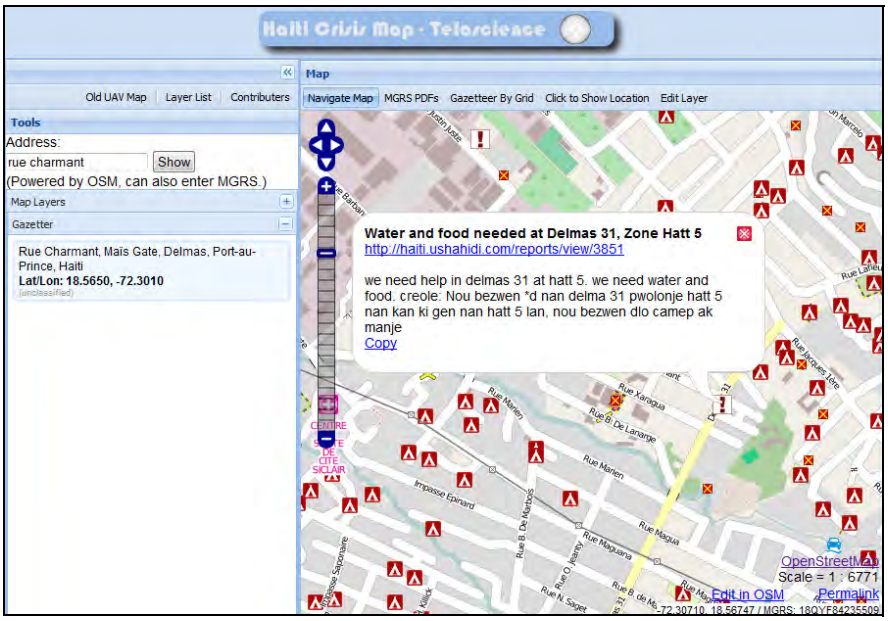


Figure 11.8 Example of a GeoExt application using ExtJS collapsible panels and OpenLayers

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 and commercial-friendly licenses. OpenLayers is under an MIT License, and GeoExt is under a BSD license. But GeoExt also relies on ExtJS, which is under a dual GPL v3 and commercial license. This means that if you need to extend 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 are available 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 you can download ExtJS from http://www.extjs.com/. For our examples, we'll be using ExtJS 3.3.1 and GeoExt 0.6.

For this first example, we create a page that loads an OpenLayers map into an ExtJS window using GeoExt. We divide the code into two parts: the .htm file, shown in the following listing, and the .js file, shown in listing 11.9.

Listing 11.8 geoextnewenglandwin.htm: basic structure of page

```
<html>
<head>
    <title>OpenStreetMap New England States</title>
<script
src="http://api.maps.yahoo.com/ajaxymap?v=3.0&appid=postgisus">
</script>
```

1 Includes two JS APIs

```
    <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" />
    <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>
```

Our GeoExt app as a JS include ❷

In ❶ we include all the JS dependency files we need, in this case the APIs for Yahoo!, OpenLayers, ExtJS, GeoExt, and OpenStreetMap. ❷ We then include the JavaScript file for our custom app. Note that for ❷ we could have put all the JavaScript on the page itself, but it's standard practice to put it in a separate file, especially if there's a fair amount of JavaScript code.

The meat of the application is in the JS, which is shown in the following listing.

Listing 11.9 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()
       ],
    maxResolution: 156543.0399,
    numZoomLevels: 20,
    units: 'm',
    projection: prjmerc,
    displayProjection: prj4326
  } );

lonlat = new
➥ OpenLayers.LonLat(lon, lat).transform(prj4326, prjmerc)
```

Initialize ExtJS

```
Ext.onReady(function() {
  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();

});
```

Create collapsible window with map

This leads to the page shown in figure 11.9. The benefit of putting an OpenLayers map in a separate 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 ports and panels. View ports allow for autostretching within a div. They allow you to position the map alongside grids, collapsible panels, and other form controls and to let the map interact with them.

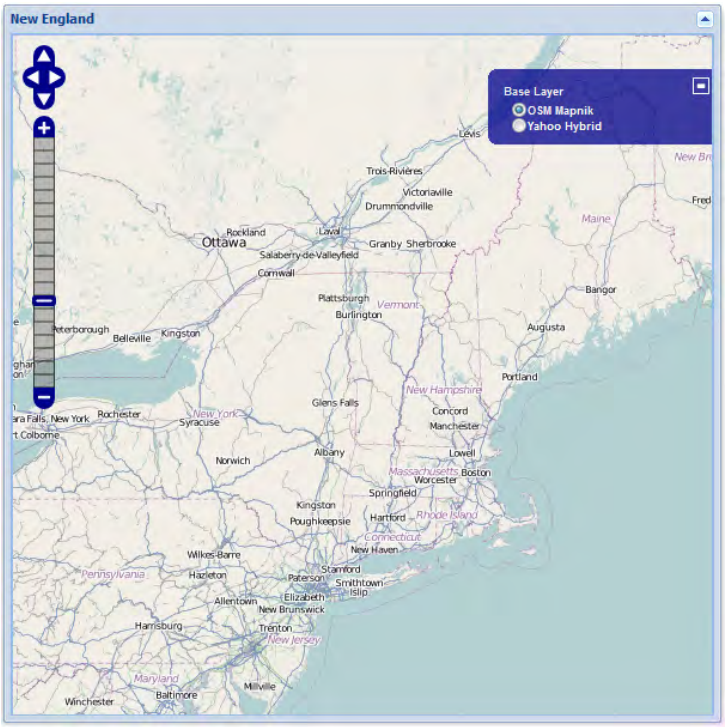


Figure 11.9 OpenLayers map in an ExtJS movable, stretchable, collapsible window using listing 11.9

For our next example, we're 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 server-side web scripting

In this section, we'll demonstrate examples of 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'll demonstrate the following concepts:

- Outputting layers with PostGIS ST_As* output functions
- Consuming the output layers with GeoExt and Google Earth
- Proximity queries with PostGIS geography

11.5.1 Using PostGIS output functions with PHP

For this exercise, we're going to use PHP and the PHP helper libraries Smarty and PHP ADOdb to build a datafeeder.php script file. We'll later use this PHP script file to feed our web mapping and Google Earth client frontends. ADOdb is a database abstraction layer used by many PHP applications to provide a generic interface to all databases. Smarty is a templating engine for PHP that allows the separation of presentation logic from application logic. Both are free and open source with LGPL/BSD–style licenses. You can obtain them from the following 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'll need. Such a 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 the following, 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

The datafeeder.php will accept an argument called *format*, which will be used to determine the type of output format. In our code we've 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 you can just as easily use PHP PEAR. This keeps our data load logic short and also has the benefit of making it more generic. As can be clearly seen, 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 file acts as the controller—reading the request and figuring out which layout and, if applicable, which query to use to satisfy the request.
- We define one Smarty template for each format. Note that formats like KML include styling options, which make using a generic KML handler such as OGR2OGR not ideal in some cases. Using a template allows us to customize this and also to inject database-stored styles in the file.
- We're using PHP classes instead of standard PHP procedures. Because our class inherits from Smarty, we can call all built-in Smarty functions as if they were native.

The core pieces of the page are shown in the following listing.

Listing 11.10 Core pieces of datafeeder.php

```
include_once("app.inc.php");
class _DataFeeder extends Smarty {
  private $db;
  private $supported_formats = array('json'=>'ST_AsGeoJSON', ...);

  function __construct {
    $this->plugins_dir = array('plugins', 'extraplugins');
    $this->left_delimiter = '<!--(';
    $this->right_delimiter = ')-->';
    $this->db = &ADONewConnection(DSN);
      :
    $this->page_load();
  }

  function page_load(){
    $data_template = 'data_json.tpl';
    if (!empty($_REQUEST['format']) ){
      $format = $_REQUEST['format'];
    }
    $convertfunction = $this->supported_formats[$format];
    if ( empty($convertfunction) ){
      $convertfunction = 'ST_AsGeoJSON';
    }
    else {
      $data_template = "data_$format.tpl";
    }
    $sql = "SELECT gid As id, ..
      $convertfunction(the_geom) As geom ...";
    $rsdata = $this->db->Execute($sql)->GetRows();
    $this->assign('rs', $rsdata);
    $this->display($data_template);
```

1 Constructor

2 Handle web request

3 Determine format

4 Display data in right format

```
  }
}
new _DataFeeder;
```

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. In 1 we also redefine the plug-ins to look in our extraplugins folder for additional plug-ins. 2 Then we call the page_load() function to handle a web request. You can put all that logic in the __construct function, but we break it out for clarity. In our page load 3 we look up 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 output the geom field, load this data into a PHP array using the ADOdb GetRows() function, assign it in our template, and then display the merged data and template.

Our JSON Smarty template looks like the following listing.

Listing 11.11 Smarty data_json.tpl file

```
{
"type": "FeatureCollection",
"features": [
<!--(section name=sec loop=$rs)-->
  { "type": "Feature", "properties":
  {"id":<!--($rs[sec].id|json_encode)-->

<!--(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)-->
  }
<!--(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 through our PHP array. 2 We then use a `foreach` to loop through all the fields of each record and output the ones that aren't `id` or `geom`, and then we output the geometry as a separate field. Note the `json_encode`: This is a Smarty modifier that's just a wrapper around the PHP 5.2+ built in Json_encode function that will convert our value into safe JSON text. 3 We need to separate each feature by a comma, except if it's the last feature in our result set.

We put the `json_encode` Smarty modifier in a file: libs/smarty/extraplugins/modifier.json_encode.php. It looks like the following:

```
<?php
function smarty_modifier_json_encode($string){
    return json_encode($string);
}
?>
```

The generated output of our JSON can be seen on the book site; use this URL: http://www.postgis.us/demos/chapter_11/datafeeder.php?format=json.

If we wanted our data to also be accessible from a KML viewer such as Google Earth, we'd offer the KML format by adding in a KML template. The data_kml.tpl template file is shown in the following listing.

Listing 11.12 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 that we use the `escape:html` Smarty modifier to make the fields KML friendly. The escape modifier is included in the Smarty download.

11.5.2 Displaying data in Google Earth

We can use Google Earth to display data output by the datafeeder script using the KML format. In Google Earth we create a network link, http://www.postgis.us/demos/chapter_11/datafeeder.php?format=kml, as shown in figure 11.10.

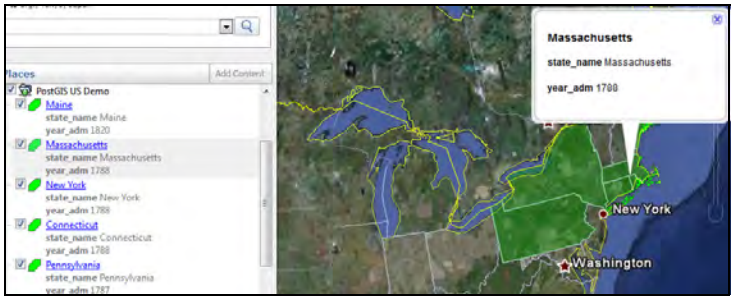


Figure 11.10 Displaying KML layer in Google Earth using template in listing 11.12

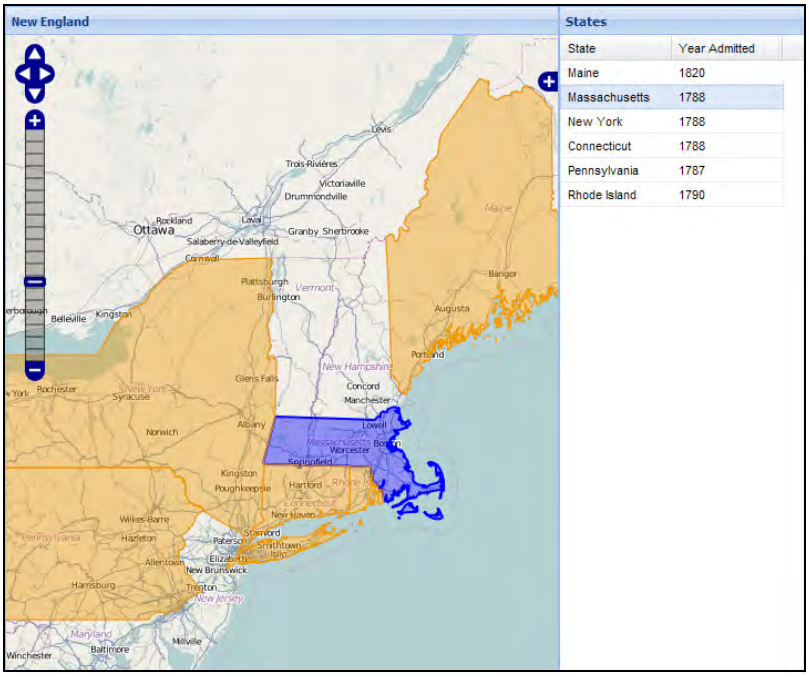


Figure 11.11 Application with feature grid in sync with map using table column layout from the code in listing 11.13

In the next exercise, we'll use the GeoJSON output of our datafeeder.php in a simple GeoExt application.

11.5.3 Loading custom layers with GeoExt

For this exercise, we'll display a sortable grid using GeoExt where the grid data comes from a GeoJSON datastream output by datafeeder.php. This datastream can be created with any web-scripting language such as Python or ASP.NET. When you're finished, your page will look like the one shown in figure 11.11.

For this application we created a new .js file similar to the one shown earlier, but with the following parts added:

Listing 11.13 Adding a feature grid window

```
lyrStates = new OpenLayers.Layer.Vector("States");
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",
```

1 **Blank vector layer**

2 **Popular vector layer**

```
                format: new OpenLayers.Format.GeoJSON()
            })
        }),
        autoLoad: true
    });
    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()
    });

panel = new Ext.Panel({
    id:'main-panel',
    baseCls:'x-plain',
    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
    ]
});
```

❸ Grid panel for attributes

❹ Two-column map panel and grid panel

❺ Output to body of document

In ❶ we define a new OpenLayers layer that will store the features. ❷ We create a GeoExt feature store object that automatically retrieves data from our custom-defined PHP script. ❸ We then 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. ❹ Instead of a movable window we use a two-column table layout, with the first column holding our map (note the use of `gx_mappanel`) and the second column holding our feature grid panel. This panel is the only panel that gets displayed because of the ❺ `renderTo: Ext.getBody()`, and it contains the mapPanel and gridPanel features as child items.

11.5.4 Proximity queries with PostGIS geography

For many use cases, you don't care about maps. You just care about how far away the data elements are 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.

The next listing is a simple PHP snippet of code that demonstrates how to use ADOdb (or some other database abstraction layer) and Smarty.

Listing 11.14 PHP find all roads within one mile of a requested lon lat

```
if(!empty($_REQUEST['lat']) && !empty($_REQUEST['lon']) ) {
  $lat=$_REQUEST['lat'];
  $lon=$_REQUEST['lon'];
  $range = 2;//in miles
  if ( is_numeric($lat) && is_numeric($lon) ){
    $pt = "ST_GeogFromText('SRID=4326;POINT($lon  $lat)')";
    $sql =
    ➥ "SELECT full_name, lfromadd, ltoadd
      FROM roads
    WHERE ST_DWithin(roads.geog, $pt,1609*$range)
 ORDER BY ST_Distance(roads.geog, $pt) ";
    $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

1 We verify that the input is passed in via a GET or POST request. Note that if you wanted to limit to just POST variables, you'd use $_POST[..]. To prevent SQL injection, we verify that the inputs are numeric. We could also validate for range to make sure the values fit in the range -180 to 180. 2 Next, we convert our lon lat to a PostGIS geography POINT expression 3, and we build the SQL statement. Because geography is always in meters, we multiply by 1609*$range to convert miles to meters for ST_DWithin. We also use ST_Distance to sort the results by proximity. 4 We execute the statement, dump the results 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

There's nothing wrong with using proprietary packages such as Google Maps or Microsoft Bing, at least if the data you need to deliver doesn't exceed their limitations and your usage complies with their licensing terms. Face it: You'll never be able to re-create the jaw-dropping, uber-cool features that proprietary packages now include, and as a GIS expert, you shouldn't have to.

If you can't or choose not to take advantage of proprietary packages, you need to worry about the web setup at both 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 create 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's offered in the UI.

Though GIS is a data-centered pursuit, you can't ignore the clients' browser experience when it comes to delivery of GIS data. Web consumers are no longer satisfied with the mere display of a map on the web browser. They want zooming, panning, and the ability to tag, edit, layer, and so on. To give users what they want, we recommend that you use OpenLayers. Its JavaScript base makes it more portable than anything else, and it's 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 that of many proprietary packages available today.

If sharing data via the World Wide Web isn't for you, 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 don't intend to share data via the web, setting up a mapping server may still be a good idea because 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
- Quantum GIS
- uDig
- gvSIG

In this chapter, we'll cover some of the popular open source GIS desktop viewing tools that work with PostGIS. As with proprietary software, you'll find that each has its own strengths and weaknesses and caters to a certain niche of users or tasks. In this chapter we'll start off by providing a brief at-a-glance summary of these tools, liberally ladling out our personal opinions. We hope that once you've completed this chapter you'll have a better understanding of which tools are best for what you're doing and for your particular style of working. We'll focus mostly on the use of these tools to view and query data but will also highlight the features that each has to build custom desktop applications to extend the native feature set via plug-ins and scripting.

12.1 At a glance

For those of you who don't wish to unnecessarily delve into the details of each tool, we start this chapter with a quick summary of features. After reading this section, you may be able to rule out some of the tools altogether and can therefore 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 might have missed. New features are being added to the tools quicker than any book can keep up with. If you've dismissed a tool due to lack of some critical feature a year ago, you may find it now incorporated. Table 12.1 provides a quick overview of the four tools that we'll cover in this chapter.

Table 12.1 Summary of tools based on architecture, language, OS, setup

Feature	OpenJUMP (3)	QGIS	uDig	gvSIG
Current version	1.3.1	1.4.0	1.1.1	1.9
JVM	1.4+	N/A	1.5+/JAI	1.5+/JAI (2)
Plug-in	jars/Jython/beans	Python/Qt	Eclipse	JARs
Scripting	Jython/BeanShell	Python	No	Jython (1)
Download size	11 MB	30 MB	100 MB	70 MB
Extract and go	Yes	No	No	No
Ease of setup (4)	Easy	Moderate	Moderate	Tricky
Ease of use	Easy	Easy	Moderate	Difficult
Mobile (5)	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 (JAI) is an API created by Sun (now Oracle) for supporting advanced imaging in Java. (3) The JUMP unified mapping platform is the platform for OpenJUMP, but some other applications use it as a framework, including the namesake desktop application JUMP and the more CAD-focused SkyJUMP. (4) How easy is it 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've 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 subjective so YMMV.[1]

OPENJUMP

This is our favorite tool because it's lightweight and lets us write raw spatial SQL and immediately view the visual results. OpenJUMP also has nice features for correcting

[1] Your mileage may vary.

and analyzing geometries as well as tools to fix up faulty shapefiles. It's probably best suited to people who aren't afraid of querying directly against the database and don't like cluttered workspaces. For Java and Python/Jython programmers, OpenJUMP easily automates common workflows. On the downside, we wish that OpenJUMP would provide better support for non-PostGIS spatial databases, such as Oracle, SQL Server 2008, and SpatiaLite (the flowering little sister of PostGIS).

QUANTUM GIS

New GIS users tend to gravitate towards Quantum GIS (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 it's the only one of these tools besides OpenJUMP with support for SpatiaLite SQLite extender. The SpatiaLite support is built in, whereas for OpenJUMP it's available via a plug-in. QGIS provides a simple and user-friendly query interface but no facility to write full SQL statements. If you're a DB programmer, you might find its lack of full SQL support a bit of a disappointment.

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 features. It caters to an Eclipse Java audience with heavy emphasis on cartography niceties. Eclipse programmers might find it just what you're looking for. As of this writing, uDig has no Jython/Python scripting framework, though plans are in the works. Therefore, if you're a Python programmer, you'll probably be disappointed with it.

GVSIG

gvSIG has lots of basic support for various databases, OGC services, and non-OGC products such as ESRI ArcIMS. It's also extensible via Java. As far as PostGIS support goes, we found it to be the clunkiest and least intuitive to use of all these products. If you've invested in ESRI and are looking for something to tie into your legacy ESRI stack, this may be your best choice. It's the only one of these products that touts a mobile edition.

12.1.2 Spatial database support

It goes without saying that all these four desktop tools are free and support PostGIS out of the box in one way or another. 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, we'll provide a simple Yes/No purely based on if we see the database listed on the menu or the documentation claims it supports the database. We'll also consider it a Yes* if it isn't part of the core download but you can download it as a separate extension.

Table 12.2 details the depth of spatial support. Following is a list of terms we use that may not be clear to you:

- *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?

- *geometry_columns optional*—Can you view tables that aren't registered in the geometry_columns table?
- *—An asterisk next to a Yes means the feature is supported but via a plug-in to be downloaded separately or via extra database drivers. A No* means that, although it doesn't support that feature, it can emulate it under certain modes or there's an easy workaround.
- *Geography*—Does it support the geography data type?
- *SQL queries*—You can write fully qualified SQL queries and see their output visually.
- *Heterogeneous column*—We mean that 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's an integer in order to render geometries in the table?
- *Save PostGIS*—It has the ability to save as a new PostGIS table.
- *Edit PostGIS*—It has the 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, but we expect that one or all of these tools will support it shortly. Some amount of support is available in the .NET framework GIS Open Source with such things as SharpMap and NTS Topology Suite and in SQL Server 2008 RS Reporting Services, but these are more SDK tools than desktop tools usable 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
PostGIS geography	Yes*	No	No	No
PostGIS raster	No*	Yes*	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

Table 12.2 Spatial database support *(continued)*

Feature	OpenJUMP	QGIS	uDig	gvSIG
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
Views	Yes	Yes*	Yes	Yes*

12.1.3 Format support

In this section we'll cover the various vector, raster, and web service formats supported by each tool; see table 12.3. Note that this list is not comprehensive but tries to cover the more common formats that people expect in a desktop tool.

- Tab is the 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 and PROJ similar to PostGIS for spatial function support. Think of SpatiaLite as a lightweight 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 its new nonpublished file storage format, which no open source software to: our knowledge currently supports. To our knowledge, only ESRI ArcGIS and possibly Safe FME as a commercial tool support this newer file 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

Table 12.3 Vector file data formats *(continued)*

Format	OpenJUMP	QGIS	uDig	gvSIG
MIF/MID	Yes	Yes	No	No
TAB	No*	Yes	No	No
Excel	Yes	Yes	No	No
CSV	Yes	?	No	IX
SVG	Yes	No	No	No

Table 12.4 lists the various raster formats supported by these tools. We didn't investigate the editing and exporting capabilities of these tools, so a Yes means only that 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 service formats and the support each program has for all of them; see table 12.5. Following 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 or menu items.

- *WMS (Web Mapping Service)*—This is the oldest and most common. It allows requests for image data based on layer names and bounding regions using the GetMap method. It also has a simple GetFeatureInfo call, which can retrieve already formatted text information.
- *WFS (Web Feature Service)*—This web service generally returns vector-formatted data based on a web query. The standard format is Geography Markup Language (GML). There do exist WFS service providers 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 formats such as GML or WKT.

- *WPS (Web Processing Service)*—This is the OGC GIS web service protocol for exposing generic work processes. Key parts are DescribeProcess, GetCapabilities, and Execute (Execute takes a named process with arguments and executes it).
- *WCS (Web Coverage Service)*—This is the OGC GIS web service protocol for raster coverages and the like.
- *ArcIMS*—This is a proprietary ESRI web-mapping service framework. It has been superseded by AGS, but many sites still maintain ArcIMS web services.

Table 12.5 Web services support

Format	OpenJUMP	QGIS	uDig	gvSIG
WMS	Yes*	Yes	Yes	Yes
WFS	Yes*	Yes	Yes	Yes
WFS-T	Yes*	No	Yes	No
WPS	Yes*	No	Yes	No*
ArcIMS	Yes*	No	No	Yes
WCS	No	No	No	Yes

Now that you have a basic sense of the possibilities of each program, we'll take all of them for a test drive.

Before we jump into the various desktops, we'd like to note that all tools get the full extent of a layer (in PostGIS terminology a geometry column) by right-clicking the layer and choosing Zoom To Layer. This is pretty consistent across them all.

12.2 OpenJUMP Workbench

OpenJUMP is a Java-based, cross-platform open source GIS analysis and query tool. It's fairly rich in functionality for statistical analysis and geometry processing. It works well with ESRI shapefiles, PostGIS datastores, and many other formats. We've found it to be the best open source tool for ad hoc spatial queries on PostGIS-enabled databases. Its main focuses are spatial analysis and geometry processing. Its cartography offering is adequate, but it's nothing to write home about. Although it's lightning fast for geometry processing tasks such as aggregation and simplification, you'll 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 Engine Open Source (GEOS), on which PostGIS is based. Because JTS is usually some versions ahead of what GEOS offers, you'll find that many new features will appear in OpenJUMP before they become available in PostGIS.

In the sections that follow we'll outline OpenJUMP Workbench's 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's certainly a good try for those in love with Python because it does support a Jython scripting/plug-in 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 and is a feature that the other tools in our discussion lack. These other tools may allow you to pick tables and limit outputs with WHERE field conditions but not more complex SQL. 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. Unlike uDig and gvSIG, OpenJUMP doesn't come packaged with its own JVM. As such the download is much lighter, but you must have a Java JVM installed already for it to work. You can get a 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 quirkier than that of QGIS, it 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 is that you can use the ad hoc tool for geography columns as well because 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/. In contrast with OpenJUMP, SkyJUMP can export to PDF and is integrated with OGR2OGR, with which it comes packaged. This means that it pretty much supports all the different data types QGIS supports, in addition to what JUMP natively supports. It also has a slightly slicker user interface (prettier icons, more right-click menus, and so on) 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 has out of the box are a connector for ESRI SDE and export/import to DXF. Its main focus seems to be the Windows user, because its only setup is a Windows executable. The exercises we'll discuss hereafter should work in SkyJUMP as well.

PLUG-INS

OpenJUMP supports plug-ins, extensions, and registries. A plug-in is a Java archive file (JAR) that you drop in the lib or lib/ext directory of your OpenJUMP install. The plug-

ins could be database drivers, geometry functions, and the like. An extension manages a set of plug-ins to accomplish a certain workflow and manages the installation and configuration of plug-ins. It can be packaged in the form of a JAR file or can be a Python or BeanShell script. These go in the lib/ext folder of your OpenJUMP install. A registry is more vague and is a dictionary of what's available in an extension.

SCRIPTING

In addition to Java JARs, you can add functionality to OpenJUMP using Java BeanShell scripting and Jython scripting. These scripts and Python classes are kept in the lib/ext folder of the OpenJUMP install. You'll see a folder for BeanTools and one for Python scripts called jython.

FORMAT SUPPORT

To load a vector file in a supported format you use the Load Dataset option by right-clicking the workspace or using File > Open. 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 can load the following vector formats out of the box: GML, JML, ESRI shapefile, WKT, and PostGIS. It can also save to the following vector formats: Scalar Vector Graphics (SVG), ESRI shapefile, and GML. It supports loading and saving to the following raster formats: GIF, TIFF, JPG, and PNG. With additional plug-ins, to be downloaded separately, it can support MrSID, MIF, ArcSDE, Oracle, GPX, and XLS. With an extra plug-in, it can also save workspace layers to a new PostGIS table.

In addition to the built-in formats, OpenJUMP supports other formats such as GPS with plug-ins downloaded separately. The key extra plug-ins can be found on http://sourceforge.net/apps/mediawiki/jump-pilot/index.php?title=Plugins_for_OpenJUMP, which is the Plugins for OpenJUMP page.

POSTGIS SUPPORT

OpenJUMP has good support for PostGIS. Its key features are listed here:

- *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 freehand 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 or ST_AsEWKB around the geometry/geography column.

12.2.2 Register data source

OpenJUMP 1.3.1 comes packaged with a PostGIS 1.0 driver and a 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 older drivers with the latest PostGIS and PostgreSQL drivers with these simple steps:

- Download PostGIS's latest JDBC .jar snapshot from http://www.postgis.org/download.
- Copy the PostGIS JDBC into your OpenJUMP/lib folder and remove postgis_1_0_0.jar.
- Download the latest PostgreSQL JDBC3 driver from http://jdbc.postgresql.org/download.html. If you're using PostgreSQL 9.0+, make sure to use the newest JDBC driver because the default binary output format has changed in 9.0, and older drivers will fail.
- Copy the PostgreSQL JDBC driver into your OpenJUMP lib folder and delete the 8.3 version.

OpenJUMP maintains a list of data sources you can connect to. You must register these prior to using them. You register a data source using the OpenJUMP Connection Manager. We'll demonstrate one way to connect to a PostGIS data source. You can get to the Connection Manager via the Layer menu and then selecting Run Data Store Query. Clicking the database icon next to the Connection drop-down list (figure 12.1) brings you to the Connection Manager pop-up (figure 12.2).

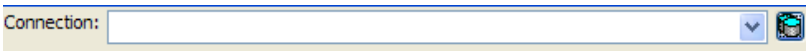


Figure 12.1 OpenJUMP drop-down list for Connection and Link to add a connection

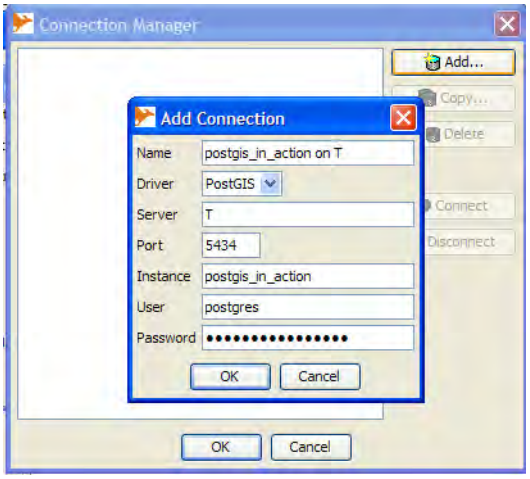


Figure 12.2 Adding a new PostGIS database connection

Enter the database name in the Instance field. Once you've successfully added the connection, you should see a new item in your Connection Manager list with a green dot in front (figure 12.3). Should you end up with a red dot, delete the connection and try again. OpenJUMP doesn't have an edit option.

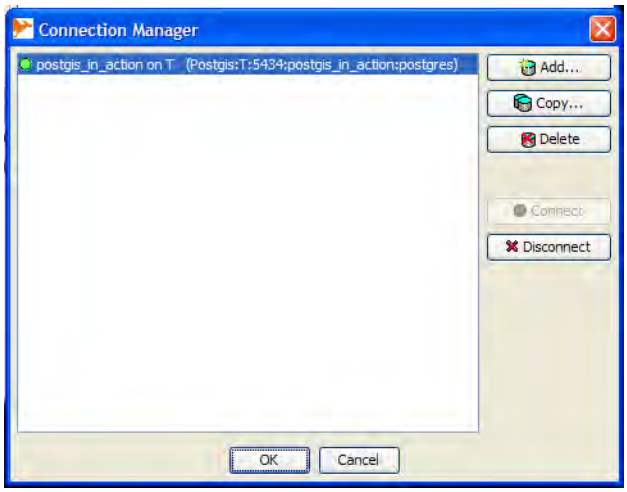


Figure 12.3 OpenJUMP Connection Manager with a new connection

12.2.3 Rendering PostGIS geometry data

The Add DataStore Layer dialog box 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).

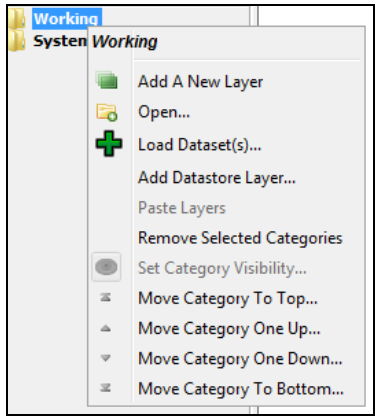


Figure 12.4 Adding a PostGIS table in OpenJUMP

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).

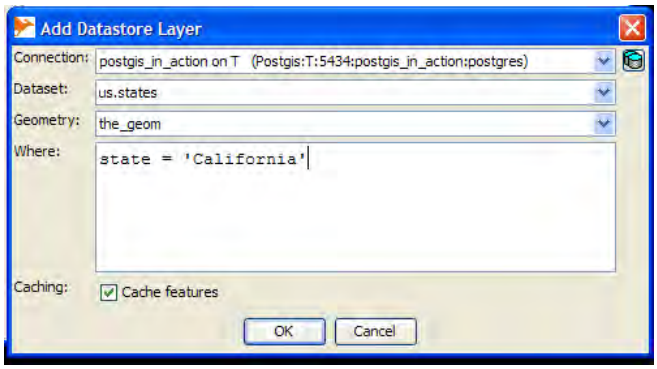


Figure 12.5 Datastore Layer setup in OpenJUMP

If nothing appears on the main display window after you click the OK button, select the layer and choose Zoom To Layer. OpenJUMP 1.3 is a bit finicky in that it considers the extent of the table equal to 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 themes. 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 embed the geography or geometry column within the ST_AsBinary function. You're free to use any SQL statement or access custom objects you've created in the database you're connected to.

The following listing 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 styles in OpenJUMP, right-click a layer and choose Change Styles and then Enable Colour Theming. Figure 12.6 shows the output of one run of the query after applying styles.

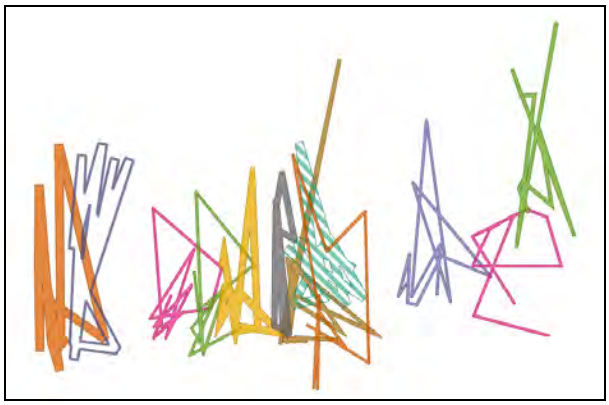


Figure 12.6 Output of SQL art query after applying custom styles

Next you'll learn how to export data using OpenJUMP.

12.2.4 Exporting data

OpenJUMP comes packaged with basic load and export functions that allow you to save files in ESRI, GML, WKT, raster, and SVG formats. By installing additional plug-ins, you can export to AutoCAD DXF and print to PDF. Visit http://sourceforge.net/projects/jump-pilot/files for the plug-ins. To export a layer to ESRI, GML, or WKT, right-click the layer and choose Save Dataset. To save the current view to raster or SVG, choose Save As from the File menu.

You can also export the data within a layer to PostGIS. You may do this after having imported data from a non-PostGIS data source or after having edited the data from a PostGIS table. You'll need to install an additional plug-in called PostGISPlugin. Visit the URL referenced previously. Download and copy 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 and choosing Change SRID. Set it to -1 or some other valid PostGIS SRID or, best yet, the SRID of the dataset. You should take care of a couple other things when saving to a PostGIS table. First, you should use lowercase names. If you want to save to a schema other than public, prefix the table name with the schema. For example, to save to the hydro schema, you'd name the table hydro.rivers. In earlier versions of the plug-in, saving to any other schema but public was not possible.

12.2.5 Summary

In this section, we've given you a taste of what OpenJUMP offers. We encourage you to explore the plug-ins available for it to enhance your experience. You'll find plug-ins 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 plug-ins written in Python 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 Quantum GIS.

12.3 Quantum GIS

Quantum GIS (QGIS) is a free desktop GIS viewing, editing, analysis tool. It's particularly popular among GIS novices, Python programmers, and GRASS users. Among the tools we're covering, it has the best GRASS and Python support. It's also the only one that's not based on Java. Instead, it's built on the QT framework, a C/C++ cross-platform windowing framework.

12.3.1 Feature summary

What makes QGIS stand out from the other tools is its high level of integration with GRASS, its extensive support for raster analysis, its integration with OGR/GDAL family suite, and its native Python scripting framework. Because its Python scripting framework uses Python directly, you're free to use any of the Python libraries available with

the standard Python installs such as those available via Python Cheeseshop. Finally, one of the most appealing features of QGIS is its user-friendly interface. With the other tools, we've encountered places where we needed to second-guess the UI. With QGIS, everything is nicely organized and there's no need to question whether we're missing a key feature simply because we're unfamiliar with its navigation.

INSTALLATION

Installation is a breeze. Because QGIS doesn't rely on Java, you don't have to download 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 put at ease those who are more apprehensive about the rapid development pace associated with open source software. The LTS version doesn't update as frequently as the current editions, but it provides a level of comfort for those who work in situations where they have to provide day-to-day support of installed software.

QGIS is also packaged into OSGeo4W. OSGeo4W is an installer that can quickly install on Microsoft Windows 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 (a la MapInfo style), and it lists the other fields in the table, allowing you to browse the contents and pick them. Table viewing and attribute editing are also implemented in a nice way. It has sorting capabilities and can zoom to the location of the selected row in the map. You can edit attribute data directly. One pet peeve of ours is that it requires a table to have an integer unique/primary key to allow loading or editing, and it doesn't support character primary keys. Most common things are so brain-dead intuitive that you generally can get away without reading any of the 200 pages of the manual to get the basics working.

POSTGIS SUPPORT

QGIS matured alongside PostGIS. Therefore QGIS spatial database support for PostGIS has been time tested more than any of the other spatial databases it supports.

- *Heterogeneous and multiple columns*—QGIS can also handle displaying a heterogeneous column of geometries. It presents the different geometry types as separate layers in the connection list. It can also handle multiple geometry columns in a table, handling them the same way as the heterogeneous columns: by displaying them as separate layers.
- *Ad hoc queries*—QGIS has no mechanism (or at least we couldn't find it), for writing self-standing ad hoc SQL statements like OpenJUMP does. Although the build query tool is nice and inviting, it doesn't let you formulate advanced SQL queries such as those using aggregates and CTEs. As a workaround, you can create a view with the desired SQL and render that. Don't forget to add a column to the view that can serve as a unique identifier.

- *PostGIS direct edit*—We found QGIS's editing of PostGIS geometries and attributes the easiest to use of all. Pity it doesn't have the functionality to save as a new table, as you can with other tools such as OpenJUMP.
- *Viewing of PostGIS raster*—QGIS is the first desktop tool to support PostGIS raster format. This support is currently experimental and available via a QGIS plug-in. More details can be found at http://mapeandoobrasil.blogspot.com/2010/12/postgis-raster-plugin-para-qgis.html, which is developed by Brazilian developer, Mauricio de Paulo.

12.3.2 Adding a PostGIS connection

Adding a PostGIS connection in QGIS is easy and intuitive. Almost everything you need can be found under the Layer menu, shown in figure 12.7. The connection screen gives you the option of searching across the database as well as searching only the geometry_columns table. A geometry_columns–only search is faster than searching across the database, especially large databases.

The geometry doesn't need to be in geometry_ columns to be listed. For rows that contain multiple kinds of geometry types, it shows each as a separate row. Figure 12.8 shows the QGIS PostGIS connection screen.

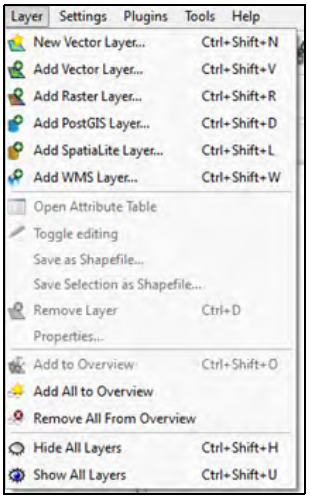


Figure 12.7 Adding a layer and PostGIS connection

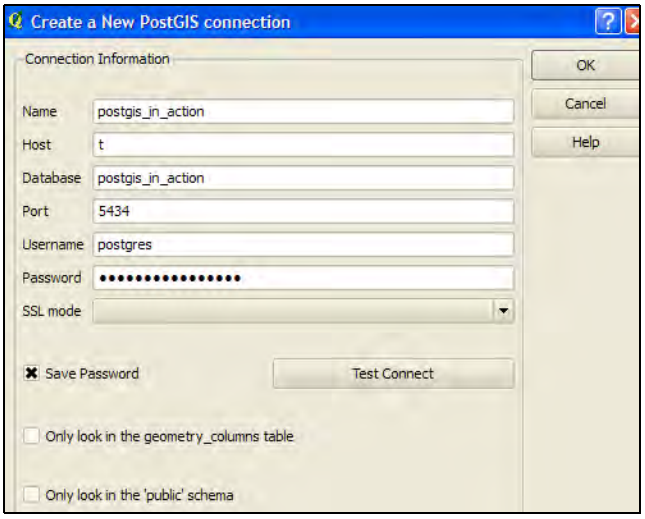


Figure 12.8 The QGIS PostGIS connection screen allows you to specify a search in geometry_columns only.

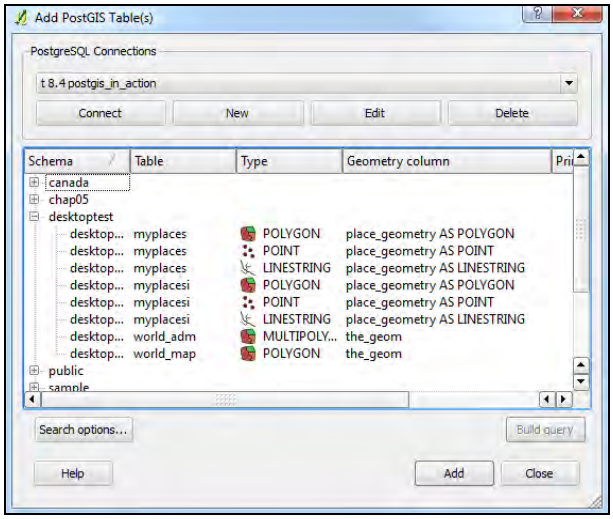


Figure 12.9 QGIS PostGIS Connect tables and schemas; myplaces.place_geometry is a table consisting of different kinds of geometry types. Each type shows as a separate layer option.

12.3.3 Viewing and filtering PostGIS data

The QGIS equivalent of OpenJUMP's Add Data Store Layer displays the type as an icon, unlike OpenJUMP's undecorated drop-down lists. If a geometry field is composed of multiple kinds of geometries, it lists each type as a separate layer, as shown in figure 12.9.

If you select a layer, you can filter the number of records and fields returned with the intuitive Build Query button, again in MapInfo-style layout, as shown in figure 12.10. You can 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.

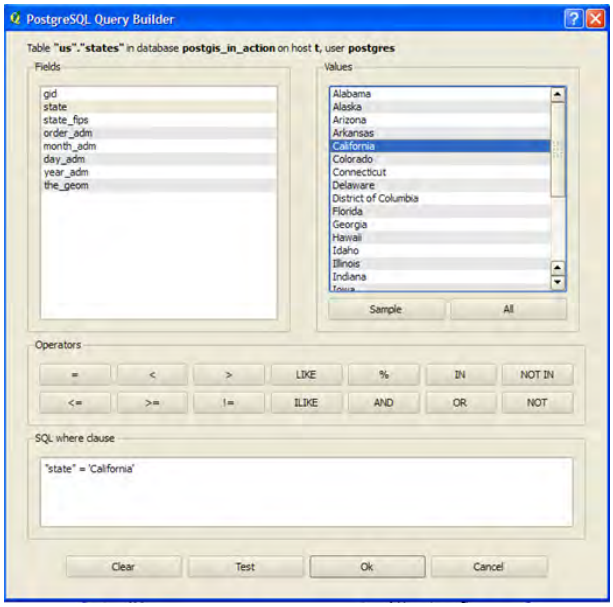


Figure 12.10 QGIS Build Query allows you to sample field data and double-click to drop a value into the Where window.

The QGIS Build Query is launched when you click the Build Query button. In this particular case we selected us.states from the US schema as we had done in OpenJUMP. QGIS lays out the fields in a list; selecting one and clicking Sample or All allows you to get distinct values from that field. You can then position your cursor in the SQL Where Clause window and double-click a value or field to put it into the position of the cursor. Click OK and then Click Add from the layer view to render the layer.

Sadly, QGIS, in all its good showings, didn't allow us the one thing we most cherished: "writing a complete spatial SQL statement and seeing it rendered in bright beautiful colors we could selectively theme" the way OpenJUMP does it. We also found its labeling and theming features much less intuitive than those of OpenJUMP.

12.3.4 Connecting with other spatial databases

As you saw from the menu, QGIS comes prepackaged with support for a database type called SpatiaLite. It's the first of the tools we're discussing that has support for SpatiaLite. OpenJUMP only recently added this support via a plug-in.

QGIS has support for other spatial databases too. For Oracle Spatial it has support for both the OGC SFSQL (vector) as well as Oracle Spatial GeoRaster.

QGIS also has MySQL support out of the box. To load a MySQL layer choose Layer > Add Vector Layer > Database > MySQL. The Add Vector Layer dialog box is shown in figure 12.11.

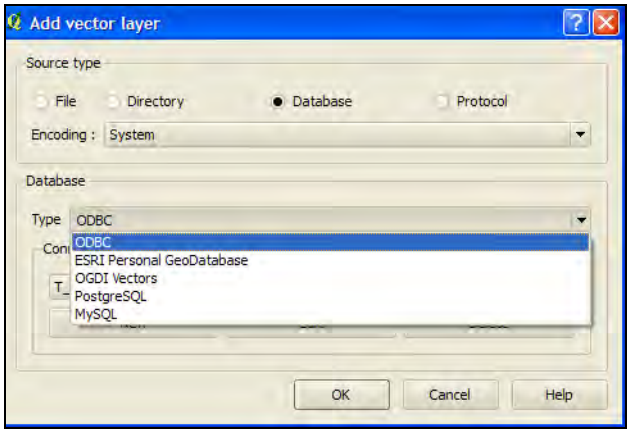


Figure 12.11 QGIS dd database vector layer

12.3.5 Loading other vector and raster layers

Loading layers is probably QGIS's strongest point; the number of types available is mind boggling. To load a vector layer choose Layer > Add Vector Layer > File or Directory; you'll be amazed at the number of options. Most are enabled using the OGR interface. Figure 12.12 shows what the list of options looks like.

QGIS also has a rich set of raster formats; about 15 different types are supported out of the box. Even more can be accessed with plug-ins.

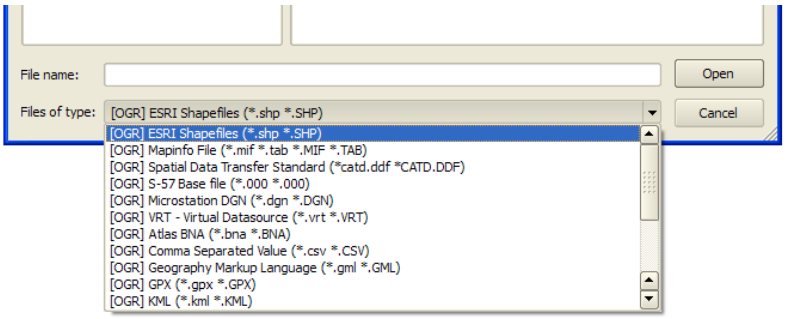


Figure 12.12 QGIS vector file sampling

12.3.6 Exporting data

QGIS comes packaged with a tool called SPIT, which allows you to batch load ESRI shapefiles into a PostgreSQL database. We already covered this in chapter 7. As mentioned there, the most annoying thing about this tool, even in the 1.4 incarnation, is that it converts all field names to uppercase. Uppercase fieldnames have to be quoted in queries, so you definitely need some sort of hack to convert them to lowercase. In terms of exporting, we found exporting to ESRI shapefile easy and available on the File > Save As menu. The other export options were tucked away and difficult to use. We couldn't detect 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-odd-page user manual packaged with it to explore its other features, as well as the numerous plug-ins made available by contributors. There's even an autoupdate plug-in module that informs you of updates to plug-ins. We didn't touch at all on its raster features or its GPS integration features, but these are two of its strongest points. In addition, QGIS offers a few other unique options. The one we found most alluring is the MapServer Export, which allows you to export your workspace as a MapServer mapfile with MapServer templates. If you're a big MapServer developer, then this could save you some time. Similar to OpenJUMP, it has plug-ins for doing geoprocessing and analysis such as union, buffer, and the like. These we didn't find all that interesting, because they can be done in general more efficiently with the raw power of PostGIS.

In the next section, we'll cover uDig, another popular GIS desktop tool.

12.4 uDig

User-friendly Desktop Internet GIS, more commonly known as uDig, has a rich feature set and is based on the Eclipse framework. It can be run as a standalone or packaged within any Eclipse environment. Its main focuses appear to be cartography and software development (SDK), and it seems to cater less to the casual GIS users of QGIS or to the hard-core geospatialists and database programmers of OpenJUMP. Although it

does have some database query functionality, it seems much more effort is directed to making good-looking presentations, enhancing map rendering speed, and extensibility. Therefore it probably caters more to the high-end GIS user/cartographer.

One thing that makes uDig stand out from the others is that it's licensed under LGPL rather than GPL. This license is a bit friendlier for those who want to build proprietary applications on top of it.

12.4.1 Feature summary

uDig, like OpenJUMP, QGIS, and gvSIG, grew up with PostGIS, and so its PostGIS support is probably stronger and better tested than that for any of the other spatial databases. It too started life as a Refractions Research project, the company that brought us PostGIS. Its most outstanding properties are its strong focus on cartography, its geometry-editing capability, and its fixing routines. It supports more commercial spatial databases out of the box and can integrate with GRASS via the JGrass interface. Note that JGrass itself was built using the uDig framework.

INSTALLATION

Installation was easy, although there were a few things to grumble about. 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 kill the 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, but 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 tables with multiple geometry columns can generally not be processed. In some cases, it refused to load a table or gave a yellow triangle denoting a problem. Neither did it allow us to filter the layers with Where conditions or to write ad hoc queries before loading the layer. This we found extremely annoying, especially with large tables. The speed in rendering large numbers of records seems a bit better than with OpenJUMP and QGIS, or at least it felt faster to us.

On the plus side, we liked the fact that it supports all the common spatial databases out of the box and that they were all right next to each other in a logical location. This wasn't the case with OpenJUMP and QGIS, which were both PostGIS centric and required acquiring separate plug-ins or looking in nonintuitive locations after the plug-in was 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, you can tell that uDig is quite flexible and easy to morph if you're a Java programmer. Therefore, it's probably well suited for a Java programmer who wants to build a complete GIS desktop suite. Like OpenJUMP, it's both a desktop tool and a platform for building desktop tools. One interesting example is

JGrass, an interface to GRASS built using the uDig framework. Various others are showcased on the uDig gallery page: http://udig.refractions.net/gallery/.

POSTGIS SUPPORT METRICS

uDig offers the following support for working with PostGIS data:

- *geometry-columns 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 with the rows listed in the table view, we were never able to get them displayed on the map.
- *SQL queries*—No. uDig supports a web query standard called Common Query Language (CQL), which in versions of CQL of 1.2 and above is now called Contextual Query Language. CQL filter conditions can be applied to layers. It appears to have no mechanism to write raw SQL. More details about the standard CQL spec can be found on Wikipedia at http://en.wikipedia.org/wiki/Contextual_Query_Language.
- *Curved geometries*—Although the 1.1.1 version doesn't support it, work is going on in version 1.3 with close collaboration with the PostGIS curve geometry developers. We expect it will be the first of these desktop tools to support PostGIS curved geometries.

12.4.2 Connecting to PostGIS and other spatial databases

uDig has the easiest interface for making the connection to PostGIS and other spatial databases. Choose Layer > Add, and you'll see the screen of available options shown in figure 12.13.

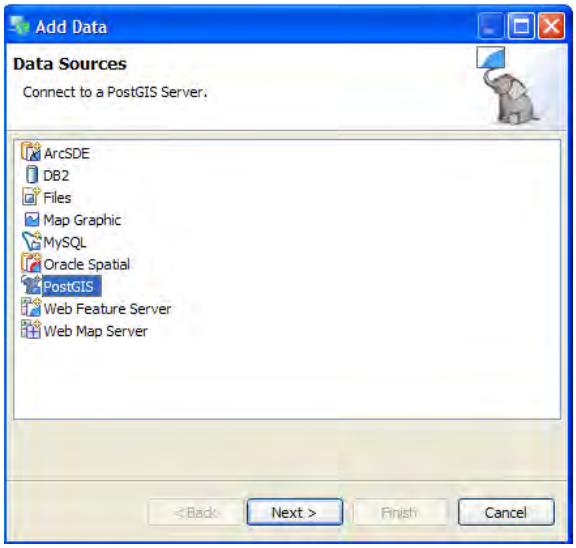


Figure 12.13 uDig Layer > Add connection

12.4.3 Viewing and filtering PostGIS data

In uDig filters can be applied by using the Contextual Query Language or by picking fields from the drop-down list and applying a single-column filter. We found both approaches cumbersome, because they require the full dataset to be loaded. You can't choose this option when first loading the layer, as can be done with OpenJUMP and QGIS. When you finally filter, though, it highlights the records that match the condition on the screen and on the map.

CQL FILTER

To use CQL do the following:

1. Add your PostGIS layer.
2. Right-click the layer and zoom to the layer.
3. Choose the Table tab. Pick CQL from the drop-down list and type in your CQL statement.

CQL uses pretty much the same conventions as SQL Where clauses, but because it's called from the uDig client side, you can't use PostgreSQL SQL-specific constructs like ILIKE because the queries aren't processed by PostgreSQL.

Figure 12.14 is a snapshot of a CQL screen.

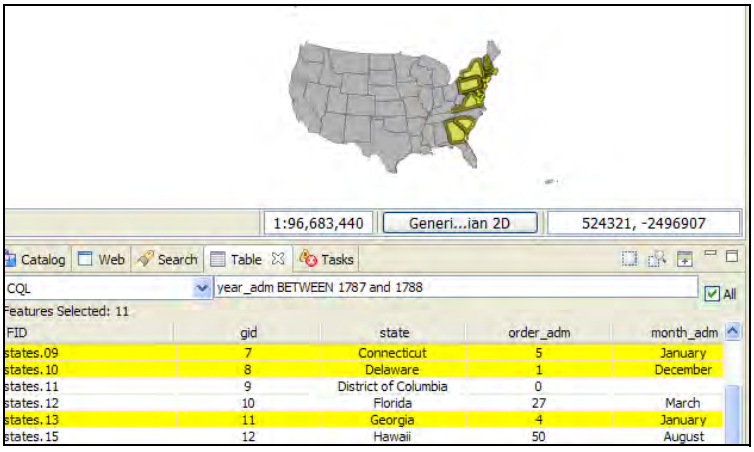


Figure 12.14 uDig CQL interface for filtering data

From within the table view of uDig, you can directly edit attribute fields, and from the map view you can edit geometry fields. uDig also supports WFS-T, so it can be used against a web-mapping server for pushing edits to various spatial databases using 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 supports only export to ESRI shapefile format, PDF, and image. All these you can access by right-clicking a layer and choosing Export or choosing 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 didn't cover it here, it does have some pretty good labeling and other cartography features you'd expect of a professional GIS desktop tool.

We didn't stress test it, but it seems on the surface to have the richest support for web mapping services of the bunch. In addition, its generous licensing model makes it enticing for building commercial products.

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 the ArcGIS desktop, which has a large install base within the Spanish government. As a result, you may find that some of the idioms used in ArcGIS are similar in gvSIG, and in many cases it tries to accomplish the basic functionality of the ArcGIS desktop in a somewhat similar fashion. It also has support not only for PostGIS but also for Oracle Spatial and ESRI ArcSDE. gvSIG started later as a project than JUMP and uDig, and the gvSIG developers strove to make it more responsive in speed than the other desktop applications. One of the major strengths of gvSIG is that the GUI responsiveness is better than that of uDig, OpenJUMP, or even QGIS.

The other unique feature of gvSIG is that it has a mobile version. This is still in beta and we didn't test it, so we can't speak for its merits. It's geared toward users in the field such as surveyors. We assume it's similar in concept to ArcGIS ArcPad.

In addition, it's the only tool that supports ArcIMS web services. It even has a link to connecting directly to the ESRI geography network.

12.5.1 Feature summary

Now let's explore the features gvSIG has to offer.

INSTALLATION

We found gvSIG the most annoying of all to install, mostly because it's a Spanish-born product with some Spanish screens popping up, and we're predominantly English speaking.

The first problem was that although it should work with an existing Java installation, we couldn't get it to work with our existing 1.6 install. We ended up using the defaults, by allowing it to install its own JRE 1.5 and JAI.

The second problem was that the installation defaulted to Spanish though most of the dialog boxes were in English, and it even correctly detected we were running an English OS. We changed the default to English during the install. When we launched the

application, all the menus were in Spanish. Thankfully, the FAQ tells you how to switch to some other language. On the General menu tab select Windows > Preferences > General > Language (or if you're in the default Spanish mode, select Ventana > Preferencias > General > Idioma). Once you've finished, click Accept (Aceptar) and then reopen the application. Hopefully this will be fixed in later versions, or perhaps it was 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. With the other tools, we were at least able to get a list of PostGIS tables and select one without opening the manual. Once you get past that hurdle and read the manual for about 10 minutes, everything becomes clear.

The manual for gvSIG is packaged as a PDF, is extensive, and is 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 doesn't use Eclipse but has its own Eclipse-like framework for extending it. It uses the concepts of projects that have three 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 an extendable mapping platform you can use to build 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, 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

gvSIG has the following features or limitations when working with PostGIS data:

- *geometry_columns required*—Yes. gvSIG lists all tables when browsing your PostGIS database, and you can select any table. However, for views and tables not registered in geometry_columns, the geometry field drop-down list 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 although the table view listed the rows, we were never able to get these to display on the map.
- *SQL queries*—No. gvSIG has a query builder tool similar to QGIS. It allows you to build the Where clause filter, but that's pretty much it. It looks like it does have an SQL filter in which you can use advanced SQL Where constructs, but it was broken. Scanning the newsgroups suggests this is a known issue, and it will be considerably improved in the 2.0 version.

- *Curved geometries*—No. When we tried to load curved geometries, it said "unsupported" and kept on popping up the error message, so we had to exit the application.
- *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 start, follow these steps:

1. Create a new view from the Project Manager window and rename it to PostGIS Test, as shown in figure 12.15.

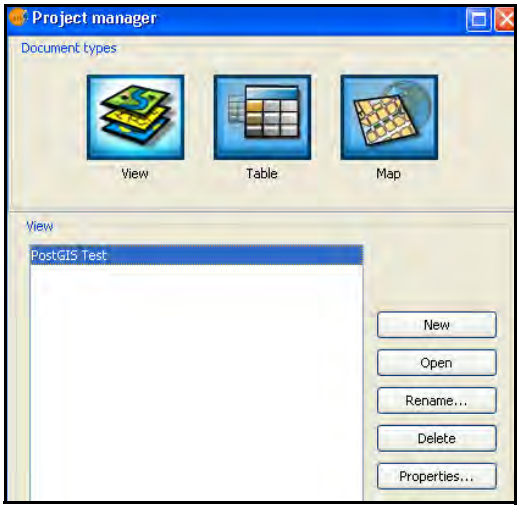


Figure 12.15 gvSIG Project Manager window

2. Click the Open button, and under the View menu choose Add Layer. Switch to the GeoDB tab and click the Connect button to create a new PostGIS connection, as shown in figure 12.16.

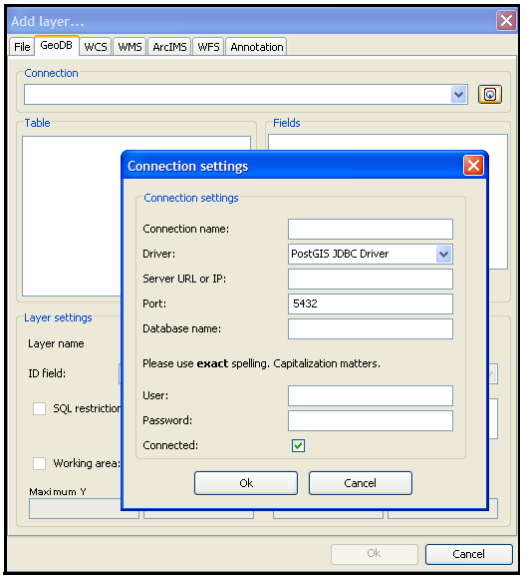


Figure 12.16 gvSIG adding a new PostGIS connection

3 Fill in all the information to connect to your PostgreSQL/PostGIS database. Then select the connection and the tables you want to add to the screen, filling in the parameters for each. The approach taken here, as you can see in figure 12.17, is similar to that of QGIS.

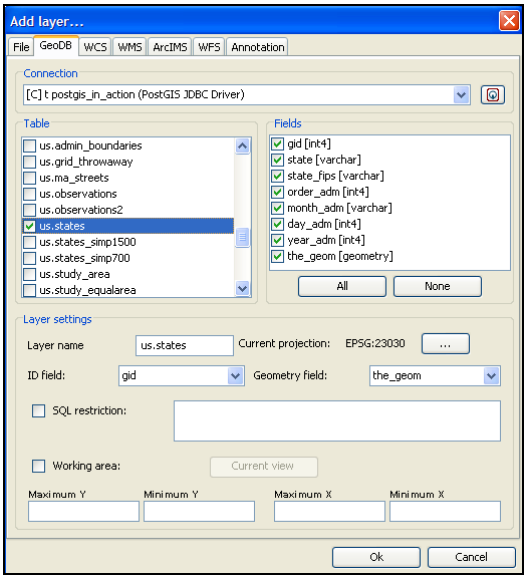


Figure 12.17 gvSIG pick PostGIS layers

gvSIG wasn't smart enough to read the spatial reference system for our table, but because we weren't 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 the layer once the layer is 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, follow these steps:

1 Select the layer.
2 On the Layer menu pick Export To.
3 Choose the format to export to, as shown in figure 12.18.

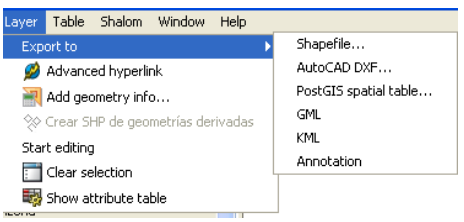


Figure 12.18 gvSIG basic export options

12.5.4 Connecting to other spatial databases

Although we didn't try it, gvSIG supports MySQL and Oracle Spatial as well as ArcSDE.

CONNECT TO MYSQL

Connecting to a MySQL database is done same way as with a PostGIS connection: Pick the JDBC driver from the drop-down menu.

CONNECT TO ORACLE SPATIAL

The drivers needed for Oracle Spatial aren't packaged with gvSIG. 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 at http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/htdocs/jdbc_10201.html and copy them to the bin/gvSIG/extensions/com.iver.cit.gvSIG/lib folder of your install. This is all explained in the manual.

We found gvSIG a bit tricky to get started with, mostly because its conventions were somewhat different from those of the other tools we've used. Once we got 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, and spatial databases, it also has many geometric processing options. These we didn't touch on because they're fairly easy to do in PostGIS but not so trivial with shape files. In addition, it has nice printing options, theming, and editing and measuring capabilities out of the box. We found its export features much easier to use than those of QGIS, and the fact it makes AutoCAD export a simple click away should make AutoCAD users or spatial analysts who work with AutoCAD users feel at home with this program.

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. We also provided a feature matrix that compared them based on functionality, installation, and ease of use. Although all these tools have some 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 tip of the iceberg of what each of these tools provides.

All these tools have end-user features you can use straight out of the box, as well as developer features that allow you to enhance the functionality via scripting or plugins. Sadly we didn't have time to go deeply into what these tools offer. We encourage you to explore them further to see what gems they have hidden. Each of these tools has a fairly extensive user manual.

We hope at the least that you were able to get a sense of what each of these tools offers, what audience of user it tries to target, and which ones will serve your needs best.

13 PostGIS raster

This chapter covers

- Differences between raster and vector data
- Using raster data in PostGIS
- The future of raster storage in PostGIS

Up to this point, we've focused on spatial vector data, because most of the functionality built into PostGIS is for storing and analyzing vector data. But there's another kind of spatial data that people commonly use: raster data. Vector data is used to represent shapes 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. Raster data, on the other hand, is a mosaic of pixels. Each pixel stores one or more different values. Areas of work where raster data is preferred over vector data include the following:

- Fine or detailed categorical coverages like land cover or land use
- Temperature, elevation, and all their derivatives
- True color coverages—aerial and satellite photos

Raster data almost always originates from instrumental data-collection processes and often serves as the raw material for vector data. As such, there's much more

free raster data in the world than vector data. Although much 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 often used independently of vector data.

What's a coverage?

A geospatial coverage is any raster or vector data (including Triangulated Irregular Networks (TINs) and point clouds) representing a common theme (or property) and covering a geographical area in 2, 2.5, or 3 dimensions. A raster or grid coverage generally implies many raster tiles or a mosaic of rasters representing the same theme and covering any geographical area. An OGC standard called Web Coverage Services (WCS) defines protocols for querying coverages.

In this chapter, we'll first look at raster data as a form of information and show how it differs from vector data. We'll look at how the new PostGIS raster data type makes analyzing raster data in a PostGIS-enabled database possible and easy. We'll also look at the various kinds of cross-querying you can accomplish by having both vector and raster support in the same database. We'll conclude by summarizing the enhancements planned and being advanced in PostGIS raster support.

13.1 What is PostGIS raster?

PostGIS raster started out as a subproject of PostGIS, created by Pierre Racine and others. Prior to its integration in the PostGIS 2.0 code base, it was known as PostGIS WKT Raster. It was created to support spatial analysis and processing of raster data as well as to make it possible to do queries that involve both raster and geometry data with a single tool. PostGIS 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 raster provides analysis and processing functions to work with raster data and also allows raster data to commingle with vector data. The main reason for this is that many operations involve both raster and vector data. Some processes involve both types of data as input or one type of data as input and the other type as output. For example, you can use a vector to clip a specific region of raster space you're interested in. In versions of PostGIS prior to 2.0, PostGIS raster (aka WKT Raster) is a separate package, with separate installation on top of an existing PostGIS (1.3–1.5) enabled database. In PostGIS 2.0, raster support is part of the core PostGIS installation. You can still experiment with raster if you're using older versions by installing standalone WKT Raster codebase. In this chapter we'll focus on the PostGIS 2.0 raster functionality.

Raster in PostGIS 2.0

In PostGIS 2.0, the raster support is no longer managed as a separate project from PostGIS and has shed its WKT primordial naming to be just *raster*, much like the geometry and geography types. Now that the project is merged, all new work on the raster front is happening in the PostGIS 2 series, and the pre-2.0 wktraster beta package is no longer being maintained.

Before we go into specifics about what you can do with raster using the raster type and functions, we'll cover some basic concepts about raster and its common use.

13.1.1 What is raster data and how is it different from vector data?

Raster data is stored as a rectangular grid of pixels (sometimes called cells). When this data is pinned down to a particular region of the world, it's referred to as georeferenced raster data, and we can georeference raster data much the same way we georeference vector data.

Georeferenced raster data

Georeferenced raster data is data where the rows and columns, usually based on the upper-left corner, are 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/feet/degrees or Y meters/feet/degrees of geographic space in the spatial reference system.

Rasters are composed of bands, sometimes also referred to as channels or dimensions. When you look at a picture such as a JPEG, PNG, or TIFF, it's generally composed of one to four bands, expressed as the Red Green Blue Alpha (RGBA) channels you see on typical computer screens. When you look at the picture in a microscope, you'll realize that it stores a lot of information. It stores a matrix with one or more numeric values per cell, where each cell has exactly the same number of values.

Vector data is 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. 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 will show the individual pixels that make up the image. The smaller the pixels, the more storage space needed to hold the raster data, but the crisper the image and the more sampling options available to

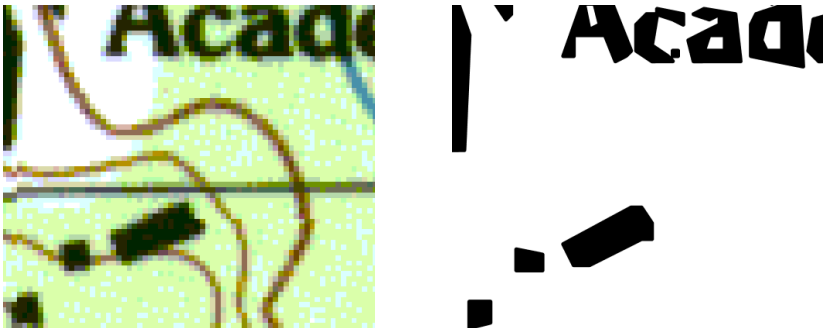


Figure 13.1 Example of a raster topo map and select pixel ranges vectorized using ST_DumpAsPolygons and further smoothed. In the raster version you can make out the pixels, whereas in the vector version you can't.

make lighter weight rasters (bigger pixels) or less-accurate vector geometries. Figure 13.1 shows 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 selected pixel value ranges have been vectorized using the PostGIS raster function 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 the vectorized version continues to show straight lines.

Although rasters are often limited to one to four bands, with each band storing whole integers, they need not be. The raster universe is much bigger than that. Rasters can have many bands and can store floating-point numbers or large integers in each band. The PostGIS raster data type supports many of these various band types and can even store rasters with different kinds of bands in the same raster. These kinds of rasters don't always have a directly viewable format. Each band can encode 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.

> **DEFINITION: REMOTE SENSING** Remote sensing is a technique that involves acquisition of information about objects and natural phenomena using real-time passive (receiving only) and active (emitting and receiving) sensing devices. Much of the automated geographic collection of information is done using remote sensing technologies. Techniques 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. Detection devices can be installed on satellites or airplanes to produce raw images from which we derive vector data such as building footprints or road networks.

These raster band pixel values can be reclassified via various algebraic calculations that mix band values from one set of bands with another or that converge pixel ranges based on values or proximity to neighboring pixels. From these processes we get

viewable rasters, easier to analyze rasters, or crisper vector geometries. When you hear the word *raster* or *picture*, we encourage you to not just admire the pretty image before you but also to 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 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 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 image recognition among other things, and with an engineer's or doctor's attuned sense even surmise subtle changes in the color of thermal camera or other imagery to diagnose things like problem regions, root causes of explosions, cancer, and so forth. Our minds automatically reduce the information overload of rasters into simplified vector like patterns or objects. Imagine what feats of analysis can be automated by imbuing more of this sophistication in computer image analysis.

Raster data is in some ways more versatile than vector data because it makes fewer assumptions about patterns. Vector already assumes a specific pattern formation. Raster offers many options for arriving at patterns that aren't available 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 a certain 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 and analyzing; for example, you can more easily take areas and measurements with vector data.

A pixel in raster data is generally modeled as a rectangle with a value for band1, band2, ... band*n* that can be related to vector data via the X and Y start coordinates of the pixel. Each rectangle in a raster has a width and height, and in most cases, the pixels in a raster are equal size, though they need not be. Keep in mind that this is simply

Regular vs. irregularly blocked rasters and how they are stored in PostGIS

For better access performance, rasters are often divided in blocks of pixels also called *tiles*. In a PostGIS tiled raster coverage, each tile is stored as one table row with at least one raster column data type field that holds the binary data. We say that a raster coverage is regularly blocked when all the tiles have the same width and height and are correctly aligned on a grid fitting the size of a tile, with no overlap and no gap. In an irregularly blocked raster coverage, tiles might be dispersed anywhere; they can overlap and they don't necessarily have the same width and height. This happens when you load many rasters forming a mosaic of overlapping rasters or when you rasterize a vector layer using one raster per geometry.

PostGIS raster is flexible in that it supports both kinds of blocking. As of this writing the GDAL driver supports only regularly blocked coverages, but support for irregular coverages is being implemented.

a model of a pixel; a pixel is really a measurement of a section of space, and as such it could just as well be a square, rectangle, triangle, or any other space-filling shape, but the math to deal with other shapes would be harder than that of a rectangle.

In the sections that follow we'll cover reasons for analyzing raster and how to analyze raster data with PostGIS.

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 volume is increasing as tools such as LIDAR imagery, thermal and infrared camera imagery, electron microscopes, and the like become cheaper and easier to use. Even the old map you scanned from an eighteenth-century drawing or nautical navigation charts downloaded from the National Oceanic and Atmospheric Administration (NOAA) are raster maps. By analyzing such a drawing, you can convert it to a smaller, crisper, and easier-to-manipulate vector format. Much of the data we think of as vector data is extruded from raster data, whether that be scanned paper survey maps or other kinds of imagery.

Vector data is generally smaller than raster data for the same region because it's 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 (recorded in DEM, DTM, or TIF). Much of this data is best expressed in raster format. In addition, raster data such as aerials is used to overlay on top of maps, 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, such as medical imaging analysis or image recognition of building equipment, many of which haven't been fully explored. Any analysis where matrices are useful or where machine-generated data is expressed in a pixel/cell format is suitable for the raster format and can be processed into other forms.

In the rest of this chapter, you'll learn how to load raster data, and we'll do some common exercises such as reading bands from raster data, polygonizing raster data, intersecting raster with vector, getting various attributes about raster data, and creating new rasters. We'll then summarize what advancements are happening on the PostGIS raster front.

Keep in mind that at this point PostGIS raster is a moving target accelerating in motion. By the time you read this, there'll be more possibilities than we've summarized here.

To find out more about PostGIS raster, please refer to http://trac.osgeo.org/postgis/wiki/WKTRaster.

13.1.3 Getting started with raster support in PostGIS

In order to use PostGIS raster type and functions, you need the following items:

1. A working PostGIS database preferably 1.4 or above, though 1.3.5+ should work. (Note: If you're using PostGIS below 2.0, you have to use the standalone WKT Raster older version. If you're using PostGIS 2+, raster support is included.)

2. The rtpostgis* so/dll compiled against your version of Postgres and a version of PostGIS. If you're on Linux, you can download the source from http://www.postgis.org/download/. If you're on Windows, there are precompiled binaries of the older WKT Raster project for 8.3, 8.4, and 9.0 that should work fine for PostGIS 1.4–1.5 at http://www.postgis.org/download/windows/experimental.php.

   For versions of PostGIS equal to or above 2.0, the raster type is packaged in the Windows experimental builds and will be packaged in the final release. See http://www.postgis.org/download/windows/experimental.php (available for PostGIS 8.4, 9.0, and 9.1).

   For Linux OS, as of this writing, there are no available binaries, so you need to compile them yourself following instructions in the PostGIS official manual, available at http://www.postgis.org/documentation/manual-svn/ch02.html.

   For Mac OS X, there are compiled binaries of the older WKT Raster project for Leopard and Snow Leopard at http://www.kyngchaos.com/software:postgres.
3. Raster functions rely on the GDAL library for the more advanced processing features such as ST_DumpAsPolygons and ST_Polygon; see http://www.gdal.org/ for more information. If you're compiling it yourself, you'll need the source for GDAL 1.6+ or above and may need to reference the gdal-config in your postgis configure statement.
4. Copy the rtpostgis-2.0.so or .dll file into your PostgreSQL ..lib folder. This step is done by the install process for PostGIS 2+.
5. Run the rtpostgis.sql in your PostGIS-enabled database.

Now that we've covered how to install raster support, we'll go on to loading raster data into your PostgreSQL database, and we'll discuss storage considerations.

13.2 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 your database.

In the sections that follow, we'll go over the ways you can store your raster data and what options you need to consider about how you store the data.

13.2.1 Options for storage

PostGIS raster supports both in-database storage and out-of-database storage. Let's look at the pros and cons of each storage type.

IN-DB STORAGE

When raster data is stored in PostGIS, the pixels are in a data column of type raster, similar to how geometries are stored in a column of type geometry or geography. You can choose to store a full raster file in a single record in a single column or cut up your raster files into tiles and store each tile as a separate record. The data is not of the original raster binary format from which it came but converted to a native PostGIS raster form suitable for manipulation by the PostGIS raster functions.

If you choose to store your rasters and all the corresponding pixel data in the database, you'll get the following benefits:

- Raster data gets backed up with your database.
- It's more tested than out-of-database storage.
- It ensures transactional integrity of the database when editing rasters.
- You enjoy faster reading of data, aggregation, and vectorization.

Storing rasters in the database isn't without its issues, however:

- Rasters are big and generally bigger than vectors that cover the same space. This will make your database backups and restores take longer. If you have rarely changing rasters, you should probably store your raster tables in a separate schema from your more commonly changed tables. This will allow you to easily exclude that schema from your daily backup and also back it up separately (for example, monthly instead of daily).
- You can't easily share the rasters with tools designed to read only from flat files.

OUT-OF-DB STORAGE

You can also choose to only store the geographic extent associated with your rasters or tiles, keeping all the corresponding pixel data outside the database as raster files in the filesystem (in TIFF, JPEG, or any other format supported by GDAL). Paths to filesystem rasters are stored in the database along with the georeferencing information, so you can query them and access pixel values indirectly. This provides you with an easy way to catalog and index your raster files. The pros are as follows:

- You can share the rasters with other applications that need them and don't know how to read raster from the database.
- If your raster files are read-only, you back them up once. Your database with just metadata is smaller and easier to back up.

And here are the cons:

- It's not well tested at this point.
- You don't get transactional benefits of the database.
- The rasters can get deleted or moved apart from the database, which will make the database records useless.
- General path annoyance. You need to make sure the postgres server process can access the files and that the path setting is in a form that's relative to the server.
- Analysis of rasters, such as forming polygons and reading pixel values, isn't yet provided, and when it is provided will most likely be much slower.

13.2.2 Using a loader to load data

In order to use the packaged loader called raster2pgsql.py, you need Python 2.5–2.7 installed with Python bindings GDAL 1.6+ and NumPy.

raster2pgsql.py current and future

raster2pgsql.py (in pre-2.0 this was called 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, PNG, GIF, ArcGIS ASCII grid files, and MrSID, to name a few. Some aren't compiled in by default and may depend on your particular installation. Refer to http://www.gdal.org/formats_list.html for details.

Currently, the GDAL PostGIS raster driver can only read the PostGIS raster data type and export to other raster formats. This may change in the future so that the GDAL PostGIS raster driver will also be able to import rasters directly without need of Python.

Instructions on how to get started with GDAL are available at http://trac.osgeo.org/gdal/wiki/GdalOgrInPython.

If you're on Windows and don't want to compile the code yourself, you can use the binaries from http://pypi.python.org/pypi/GDAL/1.6.1 or use various other binary packages. The ReadMe.txt file packaged with the standalone WKT Raster and PostGIS 2.0 Windows builds covers configuring your Python environment to support raster loading and where to find various precompiled GDAL binaries.

The loader supports the ability to do the following:

- Load single rasters.
- Chunk a single raster into various raster records (storing each as evenly blocked tiles).
- Import raster coverages (several rasters all at once into same raster table) and store each file as a separate record or as several tiled records. In addition, if you choose to import several files, you can use an additional `-F` argument that will create a text column to store the filename that a raster tile originated from. This would be useful for reconstituting the original file from the set of raster tile records.
- Load select bands from a raster.
- Reference files as out-of-DB rasters and load in the metadata and path info for that.
- Create overviews, which is useful for displaying rasters at various zoom levels.

In the examples that follow, we'll demonstrate some of these features.

Python with raster2pgsql.py

If you're on Linux/Unix or have Python associated with .py files, you can usually leave out the direct Python call. If you have multiple Python installs, you may want to give the full path to the Python version that has GDAL bindings installed to use, for example, C:\Python27\python.exe. The raster2pgsql.py file is installed in your PostgreSQL bin

(continued)

folder. If PostgreSQL bin isn't part of your path, you'll need to provide the full path to the file.

To get more help about what is supported by raster2pgsql use this command:

```
python raster2pgsql.py --help
```

LOADING A 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.

Pele will guide us through various raster exercises. Although Pele lives in her own undefined coordinate system unencumbered by the earthly weight of the world, she's a useful specimen to start with. She's 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 install the .sql file. We can do this in one step by using the | command we demonstrated with shp2pgsql.

Change to the data directory of ch13:

```
python raster2pgsql.py -r pele.png -I -t ch13.pele -o pele.sql
psql -h localhost -U someuser -d postgis_in_action -f pele.sql
```

Both of these steps are run from the command line.

LOADING LARGER AND GEOREFERENCED RASTERS

Many rasters are big, in the megabyte or gigabyte range. The raster type uses GIST indexes, the same way as geometry and geography data types do. The bounding box is the bounding box of the raster in each record. Because index seeks are much faster than the final non-box intersects checks, you generally want to split raster files into smaller rasters (tiles) and store one raster fragment per record. Smaller rasters are also easier to manipulate if you need to work on only one small part at a time.

In the last example, we brought Pele into a table called ch13.pele, as a single record. Ideally, you probably want your chunks to be no more than 100x100 pixels if you expect to do a lot of processing and analysis with them.

Your choice of tile sizes is dependent on what kind of applications you'll be using the rasters for: bigger tiles 200x200 to 400x400 for web applications and 50x50 to 200x200 for raster/vector analysis applications. For this next example, we're going to load in an ArcView image file (BIL) representing elevations of the Hawaiian island of Kauai. We grabbed this particular elevation model from http://gis.ess.washington.edu/data/raster/index.html. We've also packaged these files as part of the chapter 13 download.

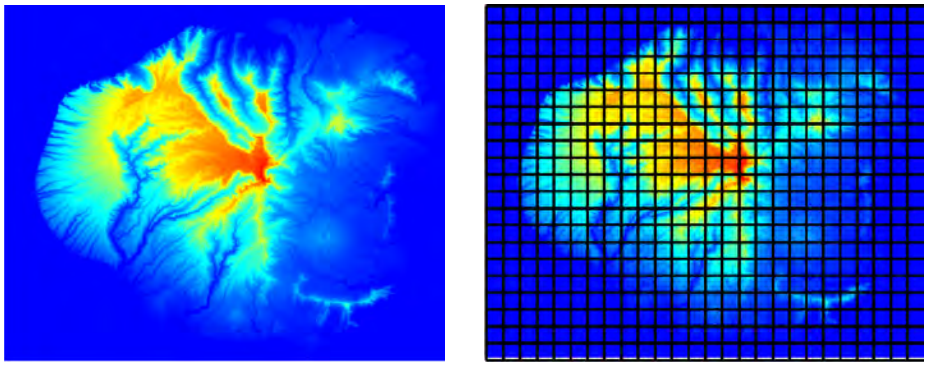


Figure 13.2 The Kauai BIL elevation model file in pseudocolor as single file and after chunking

The BIL we're using is georeferenced using UTM zone 4, NAD 83. The following query of our spatial_ref_sys table informs us that the SRID of this raster is 26904.

```
SELECT srid, proj4text
FROM spatial_ref_sys
   WHERE proj4text ILIKE '%utm%' AND proj4text ILIKE '%zone=4 %'
     AND proj4text ILIKE '%datum=NAD83%'
```

With this information in hand, we'll load the 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. Figure 13.2 is a snapshot of the image. The first is the original file using the 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.

Because this file is quite large at 14 MB, we're 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 the `-M` option to force an analyze of the table. Here's our command to generate and load the SQL file:

```
python raster2pgsql.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've loaded, we run this query:

Listing 13.1 Getting general summary info about rasters in the 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 metadata you need 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 previous query tells us that we have 546 raster tiles each of 200x200 in size and one band of 16-byte unsigned integer. We didn't include any other georeferencing information such as the upper-left corners because each tile would have a different value for upper left.

The code gives us an output of

num_rasters	height	width	srid	num_bands	btype
546	200	200	26904	1	16BUI

We'll go over some of these functions and various other useful functions in the coming sections. We'll demonstrate simple examples using Pele because she's easy to manipulate and spot check. We'll also demonstrate how we can bring Pele to Kauai.

LOADING A COVERAGE OF FILES

For this next exercise, we'll demonstrate loading many raster files at once. We're going to use *the* Vietnam elevation data we downloaded from geocommons.com:

```
python raster2pgsql.py -r vietnam/dted/*/*.dt0 -t ch13.vietelev
➥ -s 4326 -k 50x50 -F -I -o vietelev.sql
```

This command will scan our folder of dt0 files in our vietnam/dted folder included with the chapter 13 download and its subfolders and generate an SQL file that will load all the data into a table called vietelev in our ch13 schema. It will break the files into records, each with a column rast that contain 50x50 pixel dimensions rasters. This will make them easier for analytical use. The `-I` argument will index the table. The `-F` switch will add a text column to the table called *filename* that has the file path of the file the raster tile came from. Note that because we are both cutting the files with the `-k` 50x50 option as well as recursing with the vietnam/dted/*..., each file will be broken into several records.

To load the file we do as we did before:

```
psql -h localhost -U someuser -d postgis_in_action -f vietelev.sql
```

GEOREFERENCING A RASTER BEFORE LOAD

We clipped a very zoomed out National Elevation Dataset (NED) relief from a PDF we generated using the USGS National Map Seamless Server at http://seamless.usgs.gov/. You can read the details of how we massaged it so we could align the coordinates in the code USGSSeamless\ReadMe.txt of this chapter's data download. What we created to make our non-georeferenced image georeferenced was a text file with a .tfw extension. The contents of the usdem.tfw file are as follows:

```
     0.05
     0.00000000000000
0.00000000000000
-0.05
-124.85
49.42
```

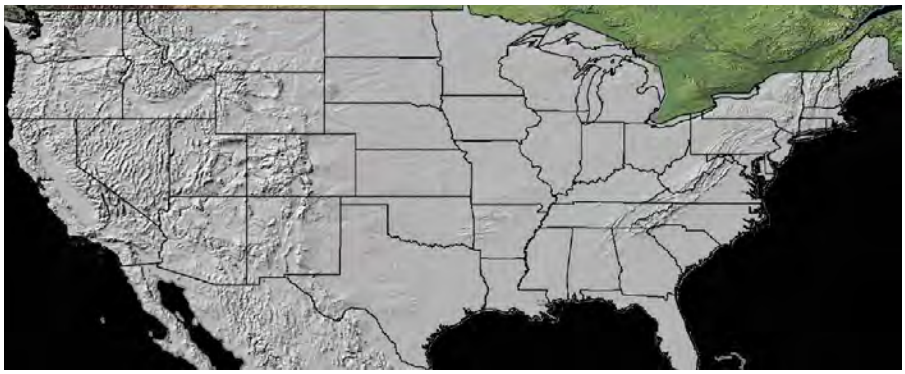


Figure 13.3 The shaded NED relief with terrain elevations

The .tfw file is a TIFF world file that GDAL uses to define the geographic extent of the raster file. The top upper-left corner is -124.85, 49.42 (lon lat), and our pixel size divided is about 0.05, -0.05 degrees per pixel. The zeros represent the skew. We'll go over these georeferencing factors in more detail in a later section of this chapter.

Our NED relief looks like figure 13.3 and it's in WGS 84 lon lat (SRID:4326). We have the option of keeping it in this projection or reprojecting it using gdalwarp. For this exercise, we'll keep it in its native projection and then later output it to a planar coordinate system using gdalwarp.

```
python raster2pgsql.py -s 4326 -r USGSSeamless\US.tif
➥ -t ch13.usdem -k 130x79 -I -o usdem.sql
```

LOADING SELECT BANDS OF A RASTER

In the case of USDem, the colors of each band for the areas we care about are pretty much the same, so we can probably do with just loading one band and save some space and processing. To load a single band as a raster, we'd use the following statement, which would generate the script to load just the first band:

```
python raster2pgsql.py -s 4326 -b 1 -r USGSSemaless\US.tif
➥ -t ch13.usdemb1 -k 130x79 -I -o usdemb.sql
```

In the next section, we'll go over the raster maintenance functions that AddRasterColumn is a member of. The SQL output of raster2pgsql.py includes a call to AddRasterColumn to add the raster column to the generated PostgreSQL table.

13.3 Raster maintenance tables and functions

Many of the raster maintenance functions are similar to the functions for geometry support. If you're 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 registered raster tables. This table is called raster_columns. The functions for adding constrained raster columns are also similar in name to the geometry add/drop column functions.

13.3.1 raster_columns metadata table

When you import rasters with the raster2pgsql utility, it also registers the raster table in the raster_columns metatable using the AddRasterColumn function. You can query this table much like any other:

```
SELECT r_table_name As tname, r_column As col_name,
   nodata_values As noval, srid, pixel_types,
   scale_x As sx, scale_y As sy
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're showing just a subset here. These are similar to the geometry_columns of the same name except they're prefixed with *r_* instead of *f_*. pixel_types is most similar to geometry_type except that because each band has its own type, it's 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 doesn't have a single value representing nodata.

The previous example returns a table with basic information about the rasters. This isn't the only information available in the raster_columns table:

tname	col_name	noval	srid	pixel_types	sx	sy
pele	rast		-1	{8BUI,8BUI,8BUI,8BUI}	1	1
kauai	rast		26904	{16BUI}	10	-10
usdem	rast		4326	{8BUI,8BUI,8BUI,8BUI}	0.05	-0.05

As you can see from the raster_columns query table, Pele is composed of four bands with pixels of 8-bit unsigned integers. Because we didn't specify an SRID, pele's SRID is unknown. Kauai on the other hand, is composed of one band with values of 16-bit unsigned integers, representing elevation levels on the island. Its SRID is 26904, as we specified, and raster2pgsql has read from the file that each pixel size is 10x10 meters square. Also note that Kauai has -10 for the height of the pixel, which we'll explain shortly. The USGS Seamless DEM is in WGS 84 lon lat, and pixel sizes represent 0.05 degrees.

Note that although our table is homogeneous across *scale and band types*, you can define a table with mixed rasters just as we defined tables with mixed geometries. This is because each raster object has its own metadata specifying its scale, band types, srid, and other key attributes even if the table is not registered in raster_columns. By default, raster2pgsql calls the raster column *rast* and the id column *rid*, though you're free to name them whatever you want or even store multiple raster columns as we did with geometry columns. The downside of using a mixed rasters is that the GDAL driver doesn't know how to export such a monster to other raster formats unless you export each record separately as a single file.

13.3.2 AddRasterColumn function

The function used to create a uniform raster column and to register it in the raster_columns table is called AddRasterColumn. Similar to its AddGeometryColumn sibling, it also adds constraints to the created raster column to ensure that only rasters of the specified SRID can be inserted in that column. It currently doesn't constrain block size and other properties designated when registering the column, though this may change in the future. It also won't 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 or not having your raster column registered in the raster_columns table. Here's an example that adds a secondary raster column to the pele table:

```
ALTER TABLE ch13.pele ADD COLUMN rast_singband raster;
```

The more formal way would be to create the raster column using AddRasterColumn. This will register the column in the raster_columns table in addition to adding the raster column to the designated table:

```
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 nodata_values:'{255,255,255,255}'
scale_x:'10' scale_y:'-10' blocksize_x:'299' blocksize_y:'439' extent:NULL
```

We'll use the pele_in_kauai table and rast column later to hold georeferenced versions of Pele in Kauai.

13.3.3 Other management functions

In addition to the AddRasterColumn function, there are DropRasterColumn and DropRasterTable functions that are parallels to the geometry DropGeometryColumn and DropGeometryTable maintenance functions.

13.4 Commonly used functions

Once we have raster data in our database, we can access some metadata information about each raster table.

13.4.1 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 number of pixels making

up the width and height, respectively. For this example we have only one record in our table holding the full raster image we imported:

```
SELECT ST_Height(rast) As nrows, ST_Width(rast) As ncols,
    ST_NumBands(rast) As numbands, ST_SRID(rast)
FROM ch13.pele;
```

This outputs the following:

nrows	ncols	numbands	st_srid
439	299	4	-1

From this we learn that Pele's image is 439 pixels tall by 299 pixels wide and has four channels of information. Two other common functions are ST_SRID and ST_NumBands, which tell us the spatial reference system and the number of bands per pixel. Because we didn't specify a spatial reference system for her, her spatial reference system came in as unknown.

ST_VALUE , ST_BANDNODATAVALUE

With the 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 in figure 13.4, we can tell that the upper quadrant is whitespace and that whitespace is most likely nodata pixel value. So to confirm the pixel value of that whitespace 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 b1nodata
FROM ch13.pele;
```

This yields

b1val	b2val	b3val	b4val	b1nodata
255	255	255	255	

The ST_Value function returns the value of a pixel for a given band and row/col in a raster. This query tells us that the *nodata* band value in band 1 is currently NULL, but the nodata band value of all bands would be better represented by the value of 255 (white).

Another variant of the ST_Value function takes as input a PostGIS point geometry and a band, instead of an X/Y position and a band, and returns the band value intersecting with that point. Both variants assume the band number to be 1 if not specified. We'll show a use of the geometric ST_Value variant later in this chapter.

ST_SETBANDNODATAVALUE

The ST_SetBandNoDataValue is a complement to the ST_BandNoDataValue that allows you to redefine the pixel value in each band to represent nodata.

The nodata value is used by some functions to determine pixels to skip over during analysis.

To convert our nodata value to 255, we use the ST_SetBandNoDataValue function:

```
UPDATE ch13.pele
     SET rast =
          ST_SetBandNoDataValue(ST_SetBandNoDataValue(
               ST_SetBandNoDataValue(
                    ST_SetBandNoDataValue(
                         rast,1, 255) ,2,255),
                    3,255),4,255) ;
```

This update query has the effect of changing the nodata value, as demonstrated in the output of the following query:

```
SELECT  ST_BandNoDataValue(rast,1) As b1ndval
FROM ch13.pele;
```

This gives us an output of

```
 b1ndval
---------
 255
```

In pre-2.0 versions of raster, the ST_BandNoDataValue would return 0 if there was no nodata value set. In 2.0, this has been changed to NULL, and you can also explicitly set the value to NULL if you want all pixel values to be considered in operations that normally ignore the nodata pixel values.

ST_ENVELOPE

Before we continue, we need to get the envelope of the raster. We'll use this in OpenJUMP to overlay it with the original imported raster to make sure everything lines up:

```
SELECT ST_Envelope(rast) FROM ch13.pele;
```

POLYGONIZE RASTER WITH ST_POLYGON

The most basic polygonizing function is the ST_Polygon function. The ST_Polygon function unions all the pixels in a raster that aren't equal to the band nodata value. In figure 13.4, we show three queries we rendered in OpenJUMP.

The first is the ST_Envelope function, which you saw earlier. The next is the ST_Polygon function, which takes as argument a raster and an optional band number and returns a PostGIS geometry. The

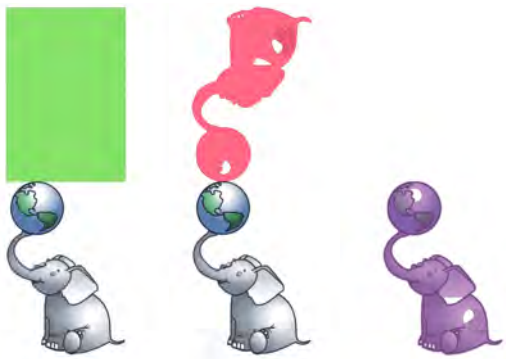


Figure 13.4 Original Pele file overlaid with her database envelope, her database band 1 polygon self, and her database polygon self flipped back to line up with her original file self.

final is ST_Polygon output after correcting Y pixel orientation. If no band number is specified, then band number 1 is assumed.

```
SELECT ST_Polygon(rast) FROM ch13.pele;
```

In the last shot, we corrected Pele by setting her Y pixel scale to -1, which we'll demonstrate shortly. We can do this with the query

```
SELECT ST_Polygon(ST_SetScale(rast,1, -1)) FROM ch13.pele;
```

Later on in this chapter, we'll describe in a bit more detail the georeferencing functions of which ST_SetScale is a member and why Pele came in upside down.

ST_CONVEXHULL

The ST_ConvexHull returns more or less the same answer for regularly blocked and nonskewed rasters. The only difference is that the envelope returns a slightly larger box than the convex hull with the coordinates rounded.

ST_ConvexHull doesn't exclude nodata values from the mix. If you have rotated rasters, then the convex hull will be quite different from the envelope.

In the next example, shown in figure 13.5, we rotate Pele and overlay her new envelope and convex hull:

```
SELECT ST_Envelope(rast_skew) As envelope,
       ST_ConvexHull(rast_skew) As convexhull
FROM (SELECT ST_SetSkew(rast,0.5) As rast_skew
      FROM ch13.pele) As foo;
```

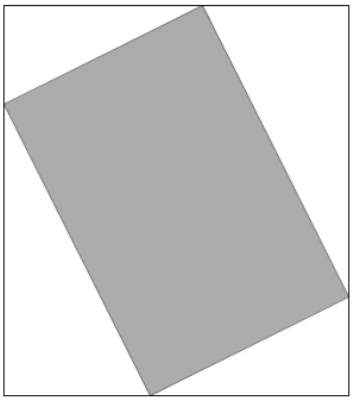


Figure 13.5 The envelope of a rotated Pele overlaid with the convex hull. The shaded area is the convex hull of the new rotated image.

CREATING A SPATIAL INDEX ON RASTER DATA

Currently no GIST operator is defined for raster, so to use a GIST index we use a functional one. If you included the `-I` option in raster2pgsql 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 this command:

```
CREATE INDEX idx_gist_ch13_pele_rast
  ON ch13.pele USING gist (ST_ConvexHull(rast));
```

In the original versions of WKT Raster before Raster got rolled into PostGIS 2.0, the spatial index was built using ST_Envelope instead of ST_ConvexHull. This has been changed in 2.0, and the cast operator that converts a raster to a geometry now uses the ST_ConvexHull function.

Now that we've demonstrated some simple things you can do with raster data, we'll move on to more exciting topics: combining the power of raster processing with vector processing.

13.4.2 Georeferencing functions

Raster data usually has an origin that starts at the upper left. Spatial coordinates, on the other hand, usually have an origin starting at the lower left. Because common files such as PNGs, GIFs, and TIFFs usually have an upper-left origin, the Y pixel size is negative to denote that it's in the opposite direction of the spatial coordinate system.

World file and upside-down Pele

A world file is a metadata sister file that lists the six numbers necessary to locate a rotated (or unrotated) raster in its reference system: four numbers for the size and shape of the pixel and two numbers for the upper-left corner of the raster. We generated one of these in the section "Georeferencing a raster before load."

For some kinds of raster formats, this metadata is embedded directly in the file rather than as a separate text file. If no information is provided, raster2pgsql guesses at the X and Y pixel scale sizes and direction. If your Python GDAL is below GDAL version 1.8, then most likely the Y scale will default to 1 (meaning the direction of the Y pixel space is the same as the spatial coordinate space), which is generally wrong.

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*.

Table 13.1 lists the georeferencing edit functions currently supported. These are all described in the "Raster Editor Functions" section of the PostGIS official reference manual.

Table 13.1 Georeferencing raster edit functions

Function	Purpose
ST_SetGeoReference	Sets the basic six georeferencing numbers in one statement. This includes ScaleX, SkewY, SkewX, ScaleY, and upper-left corner X and Y coordinates in that order. For example, `ST_SetGeoReference(rast, '10 0 0 -10 446139 2440440')` would set the rast PostGIS raster object to have 1 pixel width to represent 10 spatial units, 1 pixel height to represent -10 spatial units, and no skew and would set the upper-left corner to be X: 446139 and Y 2440440. If our spatial units are meters, then 1 pixel is 10 meters wide.
ST_SetSRID	Sets the spatial coordinate system the raster uses. The upper-left corner coordinates and pixel size should be expressed in coordinates and units of this system.
ST_SetUpperLeft	Sets the X, Y coordinates of the upper-left corner of the raster to spatial ref units.
ST_SetScale	Sets the x and y size of pixels in units of the coordinate reference system, number of units/pixel width/height. There are two versions of this function. One takes X and Y to set the ratios differently, and one takes a single XY to set the ratios the same.

THE SCALE FAMILY OF FUNCTIONS

ScaleX and ScaleY represent the ratio of pixels to a spatial coordinate system. ScaleX is almost always positive because raster coordinates and spatial coordinates go in the same direction on the X axis. ScaleY is most often negative because raster pixel origins almost always go in the opposite direction of the spatial coordinate system. By setting the Y pixel size of Pele to -1, we're saying that Pele is vertically oriented in the opposite direction of what we consider our spatial coordinate system.

First, to see how we screwed, we run an informational query:

```
SELECT ST_ScaleX(rast) As pixx, ST_ScaleY(rast) As pixy
FROM ch13.pele;
```

This query informs us that the X and Y pixel sizes are both 1, meaning that our raster coordinate direction is assumed to be the same as our spatial coordinate system, and that each X width of pixel and Y height of pixel represent one unit of X and Y of our spatial coordinate system.

Next we correct our mistake by setting the Y pixel size to -1 to denote that Y pixel coordinates are in the opposite direction of our geometry spatial coordinates:

```
UPDATE ch13.pele SET rast = ST_SetScale(rast, 1,-1);
```

Then we vectorize the first band of Pele using a PostGIS raster function called ST_Polygon:

```
SELECT ST_Polygon(rast) FROM ch13.pele;
```

This is equivalent to writing

```
SELECT ST_Polygon(rast,1) FROM ch13.pele;
```

Now we repeat the same exercise but create a fatter and taller version of Pele by setting the Y Scale to -1.5 and the X Scale to 2, as shown in figure 13.6.

This means the width of a pixel represents 2 units in our spatial coordinate system, and the height of a pixel represent 1.5 units in our spatial coordinate system but oriented in the opposite direction:



Figure 13.6 A fatter and taller shadow of Pele overlaid on her original self

```
SELECT ST_Polygon(ST_SetScale(rast,2,-1.5)) FROM ch13.pele;
```

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're going to create clones of her that are transported to Kauai. These new Peles will have a pixel size that is 10 meters by 10 meters per pixel in UTM zone 4, NAD 83 spatial reference system (SRID:26904).

In the following listing we use the ST_SetGeoreference and ST_SetSRID functions using these new coordinates as the upper-left corner and also use generate_series to create three copies of Pele separated by their body widths.

Listing 13.2 Creating three clones of Pele in Kauai

```
INSERT INTO ch13.pele_in_kauai(twin, rast)
SELECT 'pele ' || i,
          ST_SetSRID(
            ST_SetGeoReference(rast, '10 0 0 -10 '
               || 433149.653768 + i  * ST_Width(rast) * 30
          || ' ' ||  2440440.99542, 'GDAL'),
                     26904)
     FROM ch13.pele
     CROSS JOIN generate_series(1,3) As i;
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';
CREATE INDEX idx_gist_ch13_pele_in_kauai_rast
   ON ch13.pele_in_kauai USING gist (ST_ConvexHull(rast));
```

1 Create three Peles across

2 Correct raster_columns

Index for better performance

We 1 insert three copies of Pele using the PostgreSQL built-in generate_series and use 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.

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 in figure 13.7 shows the relative positioning of the clones to 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 to vector data.

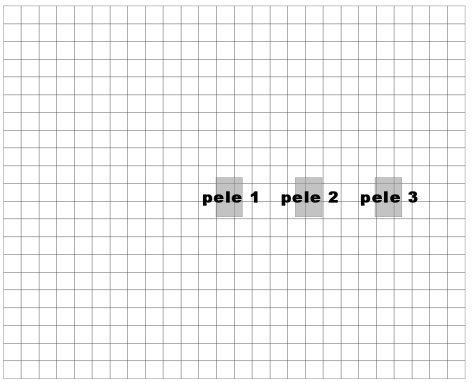


Figure 13.7 The clone envelopes of Pele overlaid on the envelopes of the Kauai raster tiles

13.5 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.

13.5.1 Pixel value getters and setters

There are two key functions for getting pixel values and setting pixel values. These functions are called ST_Value for getting pixel value information and ST_SetValue for setting pixel value information. We'll demonstrate examples of these in this section.

GETTING A PIXEL VALUE AT A GEOMETRIC POINT ST_VALUE

There are two variants of the ST_Value function. One variant takes the raster, optional band number, column, and row. The other variant takes a raster, an optional band number, and a point geometry that has the same spatial reference system as the raster. Both return the pixel value in that location of the raster for the given band. Band 1 is always assumed if no band is specified.

Here's an example of the ST_Value point geometry variant function in action:

```
SELECT p.twin, ST_Value(k.rast,1,p.geom) As elev_twin
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(k.rast, p.geom)
ORDER BY p.twin;
```

SETTING A PIXEL VALUE AT A GEOMETRIC POINT

A companion to the ST_Value function is the ST_SetValue function, which returns a new raster with the value and the specified location set to the specified value. This function also has two basic variants. One version takes a raster, band number, row, column, and value and sets the pixel at that location to that value. The second variant takes a raster, optional band number, and geometric point and sets the value of the row column that intersects that point.

There is currently no variant that takes an areal geometry such as a polygon and sets all pixels intersecting the geometry, although this is a planned feature. For this next example, we'll initialize the elevation where the sister centroids are in Kauai to the same elevation:

```
UPDATE ch13.kauai
  SET rast = ST_SetValue(k.rast,1,p.geom,400) As elev_twin
FROM (
 SELECT twin, ST_Centroid(ST_Envelope(rast)) As geom
 FROM ch13.pele_in_kauai) As p
    WHERE ST_Intersects(k.rast, p.geom);
```

13.5.2 Intersects and Intersections

In the PostGIS 2.0 raster function set, a function called ST_Intersection takes as argument a raster band and a geometry and returns a set of geomval objects that contain a pixel value and a polygon. A companion ST_Intersects function takes a raster band and a geometry and returns true or false if the raster intersects the geometry. Both the

ST_Intersects and the ST_Intersection function exclude pixels that are the nodata value. You can get an even faster but less accurate result by passing in an optional *hasnodata* boolean = false that will use only the convex hull of the raster for intersection checks rather than a more intensive and time-consuming check for nodata values. By default, the hasnodata is read from the metadata of the raster if it's not specified.

The hasnodata option

If a raster has a null nodata value, then the hasnodata argument is set to false. In WKT Raster versions, nodata could never be NULL, and there was a separate hasnodata metadata boolean property. This was changed in PostGIS 2.0.

We'll demonstrate these features by returning a portion of our Kauai raster intersected with a buffer. We used the following query to display an intersection in OpenJUMP.

Listing 13.3 Intersection of raster with geometry

```
SELECT CAST((gval).val As integer) AS val,
     ST_AsBinary((gval).geom) As geom
FROM (
SELECT ST_Intersection(rast,1,buf.geom) As gval
FROM ch13.kauai
     INNER JOIN (
     SELECT ST_Buffer(
      ST_GeomFromText('POINT(444205 2438785)',26904),100)
     As geom) As buf ON
    ST_Intersects(rast,buf.geom))      As foo
ORDER BY (gval).val
```

❶ pixelval and geometry

❷ Intersected output

In ❷ we create a subquery that returns the intersection of all Kauai raster rows that intersect our 100-meter buffer. The intersection returns a set of composite objects called a geomval that contains the properties geom (a geometry) and val (the pixel value of all points in that geometry). ❶ We then format these fields so we can display them in OpenJUMP and use a gradient theming. The reason we CAST to integer is that val returns a double precision object, which the current version of OpenJUMP treats as text and doesn't allow for gradient theming. We set the theming in OpenJUMP to a Quantile/Equal Number classification so that the color gets darker as the values increase.

The output of this query is shown in figure 13.8.

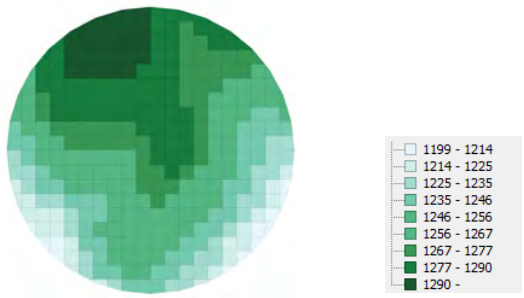


Figure 13.8 Kauai raster intersected with a 100-meter radius buffer as detailed in listing 13.3. The darker patches represent higher elevations.

PIXEL STATS

One common use of raster analysis is to calculate statistics across raster coverages that intersect our region of space. For this next exercise, we'll calculate the average elevation for our buffer region:

```
SELECT SUM((gval).val* ST_Area((gval).geom))
     / ST_Area(ST_Collect((gval).geom)) As avg_elesqm
FROM (
SELECT ST_Intersection(rast,1,buf.geom) As gval
FROM ch13.kauai
     INNER JOIN
(SELECT ST_Buffer(
      ST_GeomFromText('POINT(444205 2438785)',26904),
        100) As geom
) As buf ON
    ST_Intersects(rast,buf.geom)) As foo;
```

This yields an answer of 1258.409. Observe that this agrees with our visual spot check.

ADDING A Z COORDINATE TO A 2D LINESTRING USING PIXEL VALUES

A 2D linestring that represents a trail in Kauai can be converted to a 3D linestring with the elevation stored in the Z coordinate. We can do this by dumping all the points that make up the linestring, getting the elevation pixel values at each of these points, and then reconstituting the linestring by adding in the Z coordinate. From this we get a 3D linestring, for which we can calculate length distances relative to other trails. This requires the ST_DumpPoints function introduced in PostGIS 1.5. The result of the query in the following listing will yield a 3D linestring from our 2D linestring:

```
SRID=26904;LINESTRING(444210 2438785 1278,434125 2448785 1267,466666 2449780
     84,47000 2459000 0)
```

Listing 13.4 Adding a Z coordinate to a 2D linestring

```
SELECT ST_AsEWKT(
    ST_SetSRID(
     ST_MakeLine(
          ST_MakePoint(
               ST_X((gd).geom), ST_Y((gd).geom),
               COALESCE(ST_Value(rast, (gd).geom),0 )
          )
        ), 26904
     )
) As line_3dwkt
FROM (
  SELECT
    ST_DumpPoints(
     ST_GeomFromText('LINESTRING(444210 2438785,
          434125 2448785,  466666 2449780,
          47000 2459000)',
          26904 )
        ) As gd
   ) As trail
     LEFT JOIN ch13.kauai
      ON ST_Intersects(rast, (gd).geom);
```

1 Aggregate points into a line

2 Make pixel value Z

3 Get points from linestring

4 Rasters that intersect

In ❸ we have a linestring, and we dump the constituent vertices of the linestring to form our trail virtual table. ❹ For each point in the trail we determine which raster tiles intersect with the point and consider only those tiles. The left join ensures that we'll still get all points back even if a point doesn't intersect a tile. ❷ For each point returned we construct a new point that has the same X and Y as the original one and uses the intersecting pixel value from our Kauai raster. From these new 3D points we use the ❶ ST_MakeLine aggregate function to form a 3D linestring, set the spatial reference to the same as Kauai, and output the PostGIS extended well-known text representation.

For the 3D linestring example, we can get an even more accurate 3D linestring by using ST_Segmentize on the linestring to yield more points to dump. This would allow us to store more Z coordinates along the trail.

13.5.3 Adding bands

Because these next exercises are a bit more analytical, we want a chopped version of Pele. We use the following commands to generate a chunked version:

```
python raster2pgsql.py -r pele.png -I
➥ -k 50x50 -t ch13.pele_chunked -o pele_chunked.sql
```

Then we load in

```
psql -h localhost -U someuser
➥ -d postgis_in_action -f pele_chunked.sql
```

Again our chunked version came in upside down, but in addition to correcting the ScaleY we need to correct the upper-left Y corner by negating it:

```
UPDATE ch13.pele_chunked
     SET rast = ST_SetUpperLeft(ST_SetScale(rast, 1, -1),
     ST_UpperLeftX(rast), -ST_UpperLeftY(rast));
```

After we've finished this operation and overlay the original image with the envelopes in OpenJUMP generated by the following query

```
SELECT rid, ST_AsBinary(ST_Envelope(rast)) FROM ch13.pele_chunked;
```

we get the image shown in figure 13.9.

Even the chunked Pele has four bands. Ideally we want only one band for easier analysis, and we don't need so many colors. To accomplish this, we're going to create a new raster column consisting of one band and with two possible values to distinguish Pele's ball outline from her body outline.



Figure 13.9 Chunked Pele's envelope overlaid with the original Pele image

We first create the function in the following listing:

Listing 13.5 Reclassify Pele's pixels

```
CREATE FUNCTION ch13.reclass_pele(rast raster, rast2 raster)
    RETURNS raster AS
    $$
    DECLARE
        newrast raster := rast2;
        cx int;
        cy int;
        newwidth int := ST_Width(rast);
        newheight int := ST_Height(rast);
    BEGIN
      FOR i IN 1 .. newwidth  LOOP
         FOR j IN 1 .. newheight LOOP
               IF ST_Value(rast,1,i,j) BETWEEN 1 and 45
                  AND ST_Value(rast,2,i,j) BETWEEN 1 and 50 THEN
                  newrast := ST_SetValue(newrast,1,i,j,1);
               ELSIF ST_Value(rast,1,i,j) BETWEEN 50 AND 70 AND
                  ST_Value(rast,2,i,j) BETWEEN 51 and 70
                        AND NOT EXISTS(SELECT 1
                     FROM generate_series(-2,2) As x
                        CROSS JOIN generate_series(-2,2) AS y
                        WHERE ST_Value(rast,1,
                           greatest(1,least(newwidth,i - x)
                              ),
                           greatest(1,
                              least(newheight,j - y)
                              )
                           ) BETWEEN 1 and 45
                     AND ST_Value(rast,2,greatest(1,
                        least(newwidth,i - x)
                           ),
                           greatest(1,
                              least(newheight,j - y)
                              )
                        )
                        BETWEEN 1 and 50) THEN
                            newrast := ST_SetValue(newrast,1,i,j,2);
               END IF;
         END LOOP;
      END LOOP;
      RETURN newrast;
    END;
    $$
LANGUAGE 'plpgsql';
```

1 Isolate globe outline

2 Isolate non-globe outline

In 1 we pick out the blackish pixels that make up the ball. In 2 we pick out the remaining blackish pixels that are within 4 pixels of the ball pixels. We need to do this to get rid of false positives. Note that in 2 we use generate_series to generate a matrix of all the neighboring pixels and consider the current pixel only if the neighboring pixels aren't in the ball.

To test our reclassification function, we first create a new raster column:

```
ALTER TABLE ch13.pele_chunked ADD COLUMN rast_simp raster;
```

Then we initialize each tile to a blank one-band raster (2-byte unsigned integer) where all the pixels are 0 and the nodata value is 0 with the same dimensions and georeference information as our original raster:

```
UPDATE ch13.pele_chunked
   SET rast_simp =  ST_AddBand(ST_MakeEmptyRaster(rast),'2BUI',0,0) ;
```

We then update each tile using our reclassification function:

```
UPDATE ch13.pele_chunked
    SET rast_simp = ch13.reclass_pele(rast,rast_simp) ;
```

We use the following query to output each pixel value into separate geometries in OpenJUMP and then theme them so they show in different colors. The output of the new raster dumped using ST_DumpAsPolygons and coloring each pixel value differently is shown in figure 13.10.

```
SELECT (foo.g).val,
  ST_AsBinary(ST_Union((foo.g).geom)) As geomwkb
FROM (SELECT ST_DumpAsPolygons(rast_simp) As g
  FROM ch13.pele_chunked) As foo
  GROUP BY (foo.g).val;
```

We've reduced Pele to two possible values and one band.



Figure 13.10 Pele reclassified output from OpenJUMP rast_simp created using listing 13.5

13.5.4 Adding additional attributes to raster records

One of the nice features about having raster data in the database is that you can attach various additional attributes to each record. You may want to specify where you got the data or even what rows can be safely ignored for most processing purposes.

In the Kauai data file a good chunk of the tiles are just water around Kauai and not terribly useful. 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 kinds of operations is to add another attribute. For this example we'll add another column we'll call is_filler.

```
ALTER TABLE ch13.kauai ADD COLUMN is_filler boolean;
```

Now we're 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 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 one 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
   (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;
```

The filled-in section in figure 13.11 represents the tiles with information.

Now that we've demonstrated some raster operations, we'll demonstrate how to output our raster data into other raster formats.

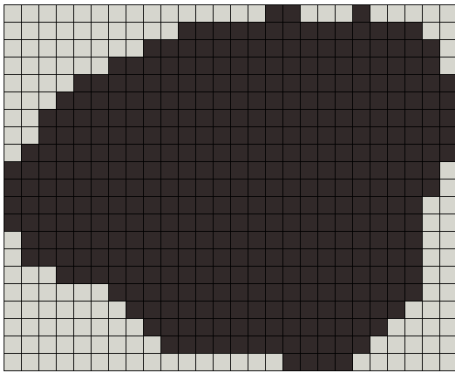


Figure 13.11 Dark tiles represent tiles that have pixels with information.

13.6 Exporting raster data into other raster formats

The GDAL library version 1.7.1+ has a driver called PostGIS WKT Raster; in later versions this will be renamed to PostGIS Raster. You can enable this driver during compile by compiling with `--with-pg=path/to/pg_config`. If you're on Windows, you can get prebuilt binaries. The FW Tools package we discussed in chapter 7 includes this driver from version 2.4.6 on.

Also, a nightly build of GDAL that has the newer trunk version (currently 1.8) with all the improvements being added to the driver is maintained by Tamas Szekeres at http://vbkto.dyndns.org/sdk/. This contains the latest and greatest of GDAL, MapServer, and Python bindings for Windows, and so it's probably the best one to use for the newest changes in PostGIS raster driver.

To verify that you have the driver compiled in your version of the binaries enter

```
gdalinfo --formats
```

The driver supports only regularly blocked PostGIS rasters, though that will be enhanced with time. To get basic information about our Kauai raster type we do the following:

```
gdalinfo  "PG:host=localhost port=5432 dbname='postgis_in_action'
     user='postgres' password='whatever' schema='ch13' table=kauai"
```

This gives us the following output:

```
Driver: WKTRaster/PostGIS WKT Raster driver
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 = (418205.000000000000000,2459785.000000000000000)
```

```
Pixel Size = (10.000000000000000,-10.000000000000000)
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 output raster data into a flat file format, 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. Gdalwarp is a tool also packaged with GDAL that both converts from one raster format to another and does a spatial coordinate transformation. We'll demonstrate them in the next section.

13.6.1 Gdal_translate basics to convert to other formats

To export our data into some other raster format, we use gdal_translate, as shown in the following listing:

Listing 13.6 Exporting raster data from PostGIS raster type

```
gdal_translate -of PNG -outsize 10% 10%                    <-- 1 Export at 10% of original size
➥ PG:"host=localhost dbname='postgis_in_action' user='postgres'
➥ password='whatever' schema='ch13' table='kauai' mode='2'" kauai_small.png

gdal_translate -of JPEG PG:"host=localhost                 <-- 2 Export specific raster column and band
➥ dbname='postgis_in_action' port='5432' user='postgres'
➥ password='whatever' schema='ch13'
➥ table='pele_chunked' column='rast' mode='2'" -b 1 pele_grey.png

gdal_translate -of GTiff                                   <-- 3 Export select rows
➥ PG:"host='localhost' port='5432' dbname='postgis_in_action'
➥  user='postgres' password='whatever' schema='ch13' table='kauai'
➥where='rid BETWEEN 1 and 200' mode='2'" subset.tif

gdal_translate -of GTiff                                   <-- 4 Export parts with Pele clones
➥ 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_kaui))' mode='2'" pelespots.tif
```

In 1 we export the whole Kauai table into a single file but shrunk to 10% of the original size and export it as a PNG. 2 We export the rast column of the pele_chunked raster table, but we export only the first band, which makes it look gray scaled instead of colored. Note for this example we needed to specify the raster column to export because pele_chunked has two raster columns. 3 We export only raster rows 1–200 of

Kauai into a single GeoTiff image. ❹ We do intersects with pele_in_kauai to get only those tiles that contain Pele clones.

> **Mode setting**
>
> In the earlier versions of the GDAL PostGIS raster driver, there was no mode setting. In the newer versions, there is a mode setting. In order to export tiled rasters as a single file, you have to set mode = 2.

The GDAL PostGIS raster driver makes it possible to export PostGIS raster data to 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 the 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. Because 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 raster layer.

13.6.2 Using gdalwarp to transform from one spatial ref to another

You can use the GDAL toolkit to transform from one spatial reference system to another using the gdalwarp executable. Its parallel for geometry would be reprojection using the ST_Transform function in PostGIS. As of this writing, the PostGIS raster GDAL driver supports only reading, not writing. As a result, you can't as of this time directly transform PostGIS raster data in the database without going through an intermediary step of exporting using gdalwarp and reimporting using raster2pgsql.

REPROJECTING DATA BEFORE LOAD

If we did this initially, we would have warped our US.tif raster before loading. The following snippet of code warps 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
```

This transforms our original U.S. map to the one shown in figure 13.12.

EXPORTING RASTER DATA IN A DIFFERENT PROJECTION

Although you can't use gdalwarp to reproject PostGIS raster data within the database, you can use it to export the data in a different projection. For this next example, we're 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 raster support. GDAL 1.7.1+ will run, but it's a bit buggy and cuts off a good chunk of the image.

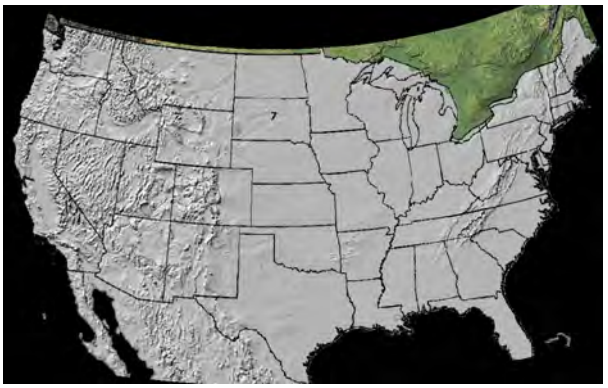


Figure 13.12 Our original WGS 84 map after being warped to NA LAEA (SRID:2163)

EPSG vs. PROJ4

The `gdalwarp –t_srs` command is the same command we used in the previous example, except for using the EPSG code instead of the proj4text string. Gdalwarp will accept both forms, but the EPSG version requires that your GDAL_DATA path environment variable be set and that you have the EPSG file to look up the proj4 settings. Therefore, writing out the proj4 string is more verbose but it's more likely to work and is guaranteed to match with your 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(rast,
➥ ST_MakeEnvelope(-115.60,32.54, -112.96, 26.03,4326))' mode='2'"
    usdem_sub.tif
```

Although it's great to be able to analyze raster data programmatically, it's also nice to be able to view your data to spot check it. Right now, few tools can view PostGIS raster data directly. MapServer, Quantum GIS, and GvSig currently can, and in the next section we'll go over how to do this. Quantum GIS can view by using a PostGIS raster plug-in, which, as of this writing, isn't currently available in the Quantum GIS compiled downloads. We briefly mentioned this plug-in in chapter 12.

13.7 Viewing raster data with MapServer

If you're using GDAL 1.7+ with the PostGIS raster driver enabled, you should be able to view PostGIS raster layers via MapServer. The following listing shows an example layer.

Listing 13.7 Example MapServer PostGIS 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 is currently the only web-mapping tool that we've tested that can directly read the PostGIS raster data type, but we expect other tools, particularly ones that are built with GDAL, to follow shortly.

13.8 The future of PostGIS raster support

Although the raster in PostGIS is still fairly new functionality, we hope we've presented enough information here so you can get a feel for what it can do for you now as well as its potential. We also hope you're as impressed as we are at the amount of functionality and speed it has already achieved.

The following section lists planned features, some of which you can expect to see with the PostGIS 2.0 release.

13.8.1 Input/output functionality

There are currently two utilities you can use for inputting and outputting raster data. These are the GDAL driver we just discussed and the raster2pgsql.py script we discussed earlier. These are expected to change in the future.

GDAL DRIVER

Currently the PostGIS raster GDAL driver can only export to other raster formats. This driver is being continually enhanced and hopefully will be able to import rasters as well without dependency on Python. It currently has a couple of rough spots when dealing with exporting irregularly blocked rasters and large single rasters. These are the main enhancements being worked on.

TOOLKITS SUPPORTING POSTGIS 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 raster support. Tools in this family include PyGDAL for interfacing with Python, Rgdal for interfacing with R, which we covered in an earlier chapter, and SharpMap.Net for interfacing with the .NET Framework. Although we haven't tested these except for PyGDAL, we expect them to be just a compile away against the latest GDAL 1.8+ source.

13.8.2 Open source viewing tools

Currently MapServer, Quantum GIS, and gvSIG are the only open source tools that can render PostGIS raster. Both MapServer and Quantum GIS piggyback on GDAL support, whereas gvSIG uses Java. You should see support for most of the other tools that leverage GDAL in the future if they're compiled with GDAL 1.7 or above.

MAPSERVER

We briefly tested MapServer with Kauai rasters. More testing with larger rasters and improvement in performance is on the list.

GEOSERVER

GeoServer has a new raster API that can be leveraged to build support for PostGIS raster in GeoServer. This new API is currently being used to build Oracle GeoRaster support, and there's talk of doing the same for PostGIS raster.

QUANTUM GIS

Quantum GIS has a plug-in that displays a listing of PostGIS raster tables and can render PostGIS raster layers. We haven't tested this.

GVSIG

gvSIG recently came out with a plug-in to render PostGIS raster. This plug-in can be downloaded from http://www.osor.eu/projects/gvsig-postgisra, the gvSIG PostGIS raster site. The plug-in is also being incorporated in gvSIG 1.11 and above.

13.8.3 Database raster functions

The raster functionality currently available is mostly for converting back and forth between geometry and raster data types. Future plans are to expand raster-only functionality that you'd commonly find in raster desktop tools.

AGGREGATE FUNCTIONS

You'll begin to see aggregate functions that work natively with rasters. Aggregates on the list are ST_Union and ST_Accum.

GEOMETRY-LIKE OFFERINGS

These include ST_Area, ST_Centroid, ST_Count, and ST_Transform.

RASTER-SPECIFIC OFFERINGS

These include the following:

- *ST_Reclass*—Currently a prototype of this is in the works and can be found in the PostGIS 2.0 raster/scripts/plpgsql codebase. This work will hopefully be integrated in the core of PostGIS 2.0 and may also be implemented as a C function. It takes as argument a raster, a band number, and a reclassification expression and replaces the passed-in band with the reclassified band.
- *ST_Resample*—This allows you to recompute pixel size and origin based on 'NEAREST NEIGHBOR', 'LINEAR', and 'BICUBIC' algorithms.
- *ST_SelectByValue(raster|geometry, 'expression')*—This allows for selecting pixels based on band value or intersection with a geometry.
- *ST_MapAlgebra*—The single band version is available and undergoing performance enhancements.

RASTER OUTPUT FUNCTIONS

Currently you can output PostGIS raster data using the GDAL PostGIS raster driver. Functions are planned that will allow you to output some basic formats using just SQL. ST_AsGDALRaster is a planned output function that will return binary data (bytea) of the specified raster output format.

ST_AsGDALRaster will return raw images of the specified type using the built-in GDAL library support in PostGIS. These will be useful for displaying raster portions without any need for additional tools. These functions will also eventually allow you to rasterize geometries because they'll support both geometry and raster types.

13.9 Summary

In this chapter we demonstrated how to work with the new raster data type. These examples built on what you've 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. In addition to raster support, PostGIS 2+ series will have 3D spatial indexes, support for 3D surfaces, 3D-aware distance and relationship functions, improved topology support, and more efficient nearest-neighbor queries using KNN GIST functionality introduced in PostgreSQL 9.1. 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 includes links to resources useful for PostGIS users of all walks of life. Some of these links we may have already covered in previous chapters, but they're also listed here so you can have all of these resources in one place.

PostGIS-focused tutorials and sites

These lists present good tutorials as well as sites focused on PostGIS content.

Getting-started tutorials

Below is a compendium of tutorials accumulated over the years. We tried 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 for viewing in uDig. http://revenant.ca/www/postgis/workshop/

BostonGIS—Part 1: Getting Started With PostGIS: An Almost Idiot's Guide. Mostly geared to the Windows user, this covers how to install PostGIS and load data and offers quick-use examples. It's still useful to Mac and Linux users because 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

OpenGeo's Open Source Geostack tutorial—Includes setting up PostGIS, MapServer, GeoServer, and QuantumGIS and creating an application. Also includes a link to download the stack and tutorial data. http://workshops.opengeo.org/stack-intro/. For a PostGIS-specific tutorial, see http://workshops.opengeo.org/postgis-spatialdbtips/

FOSS4G 2007—Introduction to PostGIS. These are PowerPoint and data documents for the intro workshop given by Paul Ramsey at the 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 through 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 gives a quickie tour of QGIS, uDig, and gvSig. It's worth a read even if you aren't using Ubuntu. http://www.paolocorti.net/public/wordpress/index.php/2008/01/20/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/

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)—This is the body that defines standards for interoperability between GIS products that are both open source and commercial. It defines data portability, web service standards, and spatial SQL standards. http://www.opengeospatial.org/

PostGIS main site—http://www.postgis.org/

PostGIS User Wiki and Bug Tracker—http://trac.osgeo.org/postgis

PostgreSQL main site—http://www.postgresql.org

Free GIS—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. http://www.freegis.org

Spatial Reference Org—This is an invaluable site for looking up spatial reference systems. It's so important we're listing it twice. http://spatialreference.org

Noteworthy PostGIS blogs and sites

The blogs and sites in this list 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—We set up this site where you can download data and code discussed in this book. We'll also be posting our presentations, chapter summaries, other chapter-related information, links, and demos. http://www.postgis.us

Paul Ramsey—One of the original co-developers of PostGIS; many think of him 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's a member of the PostGIS steering committee and the 2008 recipient of the Sol Katz Award for Geospatial Free and Open Source leadership. He's also a contributor to MapServer and the founder of the uDig desktop kit. http://blog.cleverelephant.ca

Martin Davis aka Dr. JTS—Martin is the lead architect behind the Java Topology Suite (JTS), which GEOS (Geometry Engine Open Source) is a C++ port of. A lot of the great geometry-manipulation algorithms you'll find in PostGIS and other commercial and open source packages that rely on GEOS and JTS are due in 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-th-inking.blogspot.com/

Simon Greener, Spatial DB Advisor—If you want to see how it's done in other spatial databases such as Oracle and SQL Server and other tools such as Manifold as well as find some extra tips for getting the most out of PostGIS, then Simon's your man.

His blog is full of freely available functions he's 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

Dylan Beaudette, California Soil Resource Lab—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, GeoMusings—Bill's blog is one of the best in terms of its breath 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's 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/

Paolo Corti, Thinking in GIS—I don't think there is any web GIS, particularly of an open source nature, that Paolo hasn't tried and oftentimes blogged about with helpful tutorials. His site contains everything from ArcGIS, using PostGIS in ArcGIS, to open source topics such as using GeoDJango, KML Overlays, UMN MapServer, Ruby on Rails, TileCache, OpenLayers, and even NoSQL databases. It also has 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 online, full month in HTML, and full month in PDF format). It covers general PostgreSQL tips and tricks such as doing automated backups, the TSearch integrated full-text search engine, as well as PostgreSQL integration with other tools such as MS Access, Open Office, and web programming (PHP, ASP.NET, Flex), and comparisons of different databases and administration tools. http://www.postgresonline.com/

BostonGIS—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 also tutorials on SpatiaLite, SQL Server 2008, SharpMap.NET, and PL/R and various cheat sheets we've developed over the years. The theme of the site centers on using Boston data to demonstrate spatial concepts. http://www.bostongis.com

Nicklas Avén—Nicklas is a member of the core PostGIS developer team. He made major contributions to the distance functions in PostGIS 1.5. He improved the efficiency of the existing functions on large geometries and also introduced some new ones such as ST_MaxDistance, ST_ClosestPoint, ST_LongestLine, ST_ShortestLine, and various others. In PostGIS 2.0 he's working on 3D measurement functions. In his blog, he chronicles some of his thought processes in adding to the PostGIS code base. http://blog.jordogskog.no/

Mateusz Loskot—Mateusz is a core GEOS developer and PostGIS Raster developer. He blogs a lot about various geo processing kits, packaging, and also PostGIS and PostGIS Raster. http://mateusz.loskot.net/

Sandro Santilli aka strk—Sandro is a long-time PostGIS and GEOS core developer. He's responsible for integrating much of the GEOS functionality you find in PostGIS and has done work on PostGIS Raster. 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 them 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's probably the most-read GIS blogger around. He's also the maintainer for Planet Geospatial. http://www.spatiallyadjusted.com/

Planet OSGeo—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. http://planet.osgeo.org/

Planet Geospatial—This is an aggregator of the more popular and up and coming GIS focused blogs and news sites. http://www.planetgs.com/

JASPA (Java Spatial)—This is an open source spatial extender patterned after PostGIS but written in Java instead of C. It currently has two implementations: PostgreSQL and HSQL. The PostgreSQL implementation has more or less the same functions as PostGIS 1.5 plus some additional ones (minus the geography support). The other database is HSQLDB—a Java-built relational database. The core of its logic is built using JTS, GeoTools, and PL/Java. http://www.osor.eu/projects/jaspa

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—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. http://www.joeconway.com/web/guest/pl/r

R—This is where you can download the R software package and basic tutorials on R. http://www.r-project.org/

California Soil Resource Lab—These pages are chock 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. http://casoilresource.lawr.ucdavis.edu/drupal/blog

Quick-R—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*. http://www.statmethods.net/

Boston GIS: Getting started with PL/R—These are three quick tutorials we've 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_tut02
http://www.bostongis.com/PrinterFriendly.aspx?content_name=postgresql_plr_tut03

Connecting PostGIS with R—This 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

R-sig-Geo—This is a newsgroup focused on using R in geoinfomatics and geographical mapping. It's a good group to join if you want to learn tricks of the trade and what R packages are available for doing geospatial analysis with R. https://stat.ethz.ch/mailman/listinfo/r-sig-geo

Analysis of Spatial Data—This is a 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—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 the 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. http://www.asdar-book.org/

A Practical Guide to Geostatistical Mapping—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. http://spatial-analyst.net/book/order

pgRouting installation and examples

These are sites specific to pgRouting where you can download source and binaries or learn how to compile and use it:

pgRouting official site—Here you'll find links to download the source and various binaries available for different OSes. http://pgrouting.postlbs.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 by Daniel Kastl. http://www.osgeo.jp/wordpress/wp-content/uploads/2009/11/workshop_manual.pdf

Complete documents and sample data for workshop that details using pgRouting with OpenLayers and MapFish—http://pgrouting.postlbs.org/wiki/WorkshopFOSS4G2008

PL/Python installation and examples

Our Postgres OnLine Journal—We have a cheatsheet and collection of intro articles on PL/Python detailing installing, samples, and basic flow. http://www.postgresonline.com/journal/archives/106-PL-Python.html

Official docs for PostgreSQL 8.4 on PL/Python—Similar docs exist for 8.2 and 8.3. http://www.postgresql.org/docs/8.4/interactive/plpython.html

GDAL—Geographic Data Abstraction Library has both a C and a Python interface. It's probably the most common Python library used by the 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's similar in functionality to things like MatLab. It's used for dealing with complex matrices. It's 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—This is a Python library for interfacing with Gnuplot that allows generating attractive graphical plots in Python. http://gnuplot-py.sourceforge.net/

Windows 32-bit and 64-bit nightly build binaries by Tamas Szekeres—These contain the latest Python PyGDAL. http://vbkto.dyndns.org/sdk/

Raster-related information

PostGIS Raster Home Page—This page will give you links to other PostGIS Raster resources, provide you with status of the project, show 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— these have a PostGIS Raster driver that will allow you to export data out of PostGIS raster format. This page will also give you more extensive details about using the gdal2raster.py script we described. http://trac.osgeo.org/gdal/wiki/frmts_wtkraster.html

GDAL Raster Formats—This page lists 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 gdal2raster.py. http://www.gdal.org/formats_list.html

RasterLite—This is a raster extender for SQLite similar to how SpatiaLite is a vector extender for SQLite. Its focus is more on storing raster in a database for rendering

rather than PostGIS Raster for analysis. GDAL 1.7+ has drivers that support it. http://www.gaia-gis.it/spatialite/rasterlite-man.pdf.

Precompiled binaries of GDAL and Python needed for PostGIS Raster loading—For Windows the aforementioned Tamas Szekeres binaries are the latest and greatest and are built nightly. http://trac.osgeo.org/gdal/wiki/DownloadingGdalBinaries.

rasdaman—This is a raster server implemented using PostgreSQL with its own matrix-like query language called rasql. Its focus is high performance and raster support for both rendering and matrix-like analysis, but it currently lacks support for georeferencing, integration with the core SQL base of PostgreSQL and PostGIS functions. The Rasdaman Group is currently working on providing integration with GDAL and the ability to use it directly in PostgreSQL SQL queries and georeferencing capabilities. http://www.rasdaman.com/

Open source tools and offerings

The following listings offer 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.

Installers and self-contained suites that include/work with PostGIS

The following GIS suites contain PostGIS as part of an integrated GIS desktop and/or web mapping tool:

For Mac users—There are binaries for Mac OS X graciously supplied by KyngChaos. The offerings include 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. http://www.pgrpms.org/

OpenGeo Stack—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 OS X, 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 Editions. The main differences are that Enterprise includes support, training, and Service Level Assurances (SLA), as well as hand-holding help with upgrades for those new to GIS or who need more predictable professional support. The stack also comes prepackaged with sample data to get you started. The Community Edition is a 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. http://opengeo.org/community/suite/download/ Comparison between the community and enterprise editions of OpenGeo Stack can be found at http://opengeo.org/products/suite/compare/

Portable GIS—If you're a Windows user, check out Portable GIS managed by Jo Cook. It's really cool and comes packaged with PostgreSQL, PostGIS, MySQL, Quantum, GRASS, FWTools, MapServer, GeoServer, FeatureServer, and OpenLayers, all of which can be run from a thumb drive. http://www.archaeogeek.com/blog/portable-gis/

GISVM—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 and that contains best-of-the-breed open source GIS tools, check out GISVM. This is an Ubuntu VM that comes in three flavors: GIS VM Basic English, GIS VM Geostatistics English, and a Portuguese version. http://www.gisvm.com/

The basic International English version comes packaged with PostgreSQL/PostGIS, GeoServer, Mapserver, FWTools, QGIS/Grass, gvSIG, uDig, OpenJump, and 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—This 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're on a Windows system or a desktop Linux or Mac OS X, 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

Free open source desktop GIS

The following desktop tools have integration features with PostGIS to allow viewing and editing PostGIS data:

OpenJUMP—This is one of our favorite desktop GIS tools and what we used 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 plug-in architecture and has many user-contributed plug-ins. It runs on Linux/Windows/Mac OS X. http://www.openjump.org/

QuantumGIS (QGIS)—This is perhaps the most popular of the free open source desktop tools. It also has a plug-in 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. QGIS is GNU GPL licensed. http://www.qgis.org/

gvSIG—This Java-based desktop platform offers lots of integration features to ArcIMS and other ESRI services. http://www.gvsig.gva.es/

uDig—This is an Eclipse-based Java desktop application and SDK. It has lots of integration features with OGC-compliant web services and more advanced cartography. http://udig.refractions.net

OSGeo4W Installer—This is an online 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 it yourself, this is currently the easiest package to use. http://trac.osgeo.org/osgeo4w/

Geographic Resources Analysis Support System (GRASS)—GRASS is probably the oldest and one of the most advanced free and open source tools for analyzing vector, raster, and other GIS data. It's designed more for the advanced GIS analyst rather than a new GIS or pure spatial database user. Although it's not set up specifically for PostGIS, there are many avenues of integration such as the PostGRASS driver, JGRASS, and QGIS GRASS integration tools. http://grass.osgeo.org/

Extract Transform Load (ETL)

GDAL/OGR—This is the most popular of all-purpose open source free ETL tools. It's licensed under the MIT license, which is similar to BSD. http://gdal.org/ogr2ogr.html. 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. These are downloadable as part of the source tar ball. http://www.postgis.org/download/ These are also available in binary form for Windows from http://www.postgis.org/download/windows/.

osm2pgsql—This is a command-line tool specifically designed for converting OpenStreetMap XML (OSM) format to PostgreSQL/PostGIS. Binaries for most OSes including Windows and Mac OS X can be downloaded from http://wiki.openstreetmap.org/wiki/Osm2pgsql.

Spatial Data Integrator—This is a GPL v2 licensed open source ETL tool with geospatial capabilities spearheaded by CamptoCamp and Talend. It's based on Talend Open Studio, Talend's generic ETL solution, and extends it with geospatial components. http://www.spatialdataintegrator.com/

Some Talend Geospatial use case examples can be found at http://www.talendforge.org/wiki/doku.php?id=sdi:examples and http://www.talendforge.org/wiki/doku.php?id=sdi:geocomponentslist.

As of this writing current formats supported are ESRI shapefile, MapInfo, WKT, WFS, GPX, OSM, and PostGIS.

GeoKettle—This LGPL-released open source ETL loader is based on Pentaho Kettle ETL. It currently has built-in support for PostGIS, Oracle Spatial, MySQL, and ESRI shapefiles. http://sourceforge.net/projects/geokettle/

Proprietary tools that support PostGIS

Cadcorp SIS—This suite of products includes desktop GIS and web-mapping OGC-compliant WMS, WFS, great raster, and CAD support. http://www.cadcorp.com/

Safe FME (ETL)—This is the most recognized name in the industry for spatial ETL and automating spatial ETL workflows. http://www.safe.com/

ESRI ArcGIS 9.3—This requires an ArcSDE license to work with PostGIS. http://webhelp.esri.com/arcgisdesktop/9.3/index.cfm?TopicName=The%20PostGIS%20geometry%20type

zigGIS for ArcGIS—This plug-in for ArcGIS is 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—Pretty nice all in one package, it also has some neat SQL functions for dealing with Raster. It has its own dialect of Spatial SQL very similar in style to Microsoft Access Jet (for example, it supports cross tabs using Transform PIVOT and TOP and has lots of spatial functions). It works with all the popular spatial databases without additional cost: PostGIS, Oracle, DB2, and SQL Server 2008. http://www.manifold.net

Pitney Bowes MapInfo 10—It's a favorite among data analysts and casual GIS users because of the ease with which you can link to data sources, import data, and run basic SQL queries. http://www.pbinsight.com/products/location-intelligence/applications/mapping-analytical/mapinfo-professional/

MapDotNet—This web-mapping toolkit for ASP.NET similar is in style to UMN MapServer and its mapfile format follows a similar scheme. It includes wizards to build maps. http://www.mapdotnet.com

Places to get free vector data

Following 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.

All geographic regions

OpenStreetMap—This community-driven spatial database and map repository 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 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 waypoints in GPX format. You can use it as an overlay directly with your maps using something like OpenLayers. http://www.openstreetmap.org

In addition, 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—This offers 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 includes world administrative boundaries, city and town points with population, and various natural land and water geometries. http://www.naturalearthdata.com/

Centers for Disease Control administrative boundary files—The United States Centers for Disease Control maintains boundary files for all the continents and countries in

ESRI shapefile format. These are circa 2000 and are all in the WGS 84 long lat spatial reference system (SRID = 4326). http://www.cdc.gov/epiinfo/shape.htm

Some of notable interest:

By Country—This contains the multipolygon boundaries for each country as well as information such as name of country, currency, population, and iso-code. We'll be using this layer in some of our future exercises—Cntry00.zip.

By State—This is more granular than the country boundaries and breaks the various countries into states and provinces. It lists the administrative name, country, population, and type of boundary—Admin00.zip.

Some of the files that used to be located at http://biogeo.berkeley.edu are now redirected here: http://www.gadm.org/.

Global Administration Areas is a fairly new site that tries to maintain an up-to-date version of administration boundaries at various resolutions. Data is licensed for free use for noncommercial and educational purposes. Data is stored in ESRI shapefiles, ESRI geodatabase, and Google KMZ, and some is in RData format for R statistical packages. You can download files by country at http://www.gadm.org/country or download the whole set at http://www.gadm.org/world.

GeoCommons—GeoCommons is a directory of both free and non-free GIS data sources in ESRI shapefiles, KML, and GeoRss. http://finder.geocommons.com/

It contains spatial data containing statistical information on a wide variety of topics including health, demographics, and boundaries. Many of these are user contributed.

Infochimps—This is a search engine specifically for finding datasets. Many of the data sets are of a geospatial nature. http://infochimps.org

GIS Data Depot—This is a source of both free and premium downloads of both vector and raster data. All require free registration. Premium downloads require paid subscription. Much of the data provided is public domain. http://data.geocomm.com/

North America

U.S. Census Bureau data and TIGER data—By far the most popular, complete, and free source of data for the United States is the U.S. Census Bureau's Topologically Integrated Geographic Encoding and Referencing (TIGER) system. The most recent distribution of this data is the 2010 version, which was released on a rolling basis starting November 3, 2010. This can be downloaded from http://www.census.gov/geo/www/tiger/.

The latest release of TIGER data is distributed in ESRI shapefile and dBase DBF format, so it 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—U.S. state boundaries

Census block groups, blocks, and tracts broken out by state-county—A census tract is the smallest demarcation for population. The U.S. census tries to maintain the same population across all census tracts or tracts of a particular type. A lot of statistics such

as employment and disease 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 the entire United States as single files. Keep in mind that U.S. 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, and water polygons, address ranges, points of interest, voting districts (aka wards and precincts), and congressional and senatorial districts.

New England cities and towns

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, and air emissions. We encourage you to explore it. http://www.data.gov/

U.S. National Atlas—The National Atlas offers numerous geographic layers for the United States such as railroads, airports, and political boundaries. Most are in ESRI shapefile format. http://nationalatlas.gov/atlasftp.html

U.S. National Weather Service—The National Weather Service has a catalog listing of ESRI shapefile boundaries and weather-related data mostly of the United States, but some of it covers the globe. http://www.nws.noaa.gov/geodata/

NatureServe—If you are into the study of animals and ecological effects, NatureServe has an extensive assortment of ecological data for the United States and Canada and some other regions of North America, all in ESRI shapefile format. These are mostly point and polygon regions where these species are naturally found. http://www.natureserve.org/getData/animalData.jsp

GeoBase.ca—GeoBase is a portal that provides free spatial data for Canada. http://www.geobase.ca

It provides fairly up-to-date data for following types of features:

- Administrative boundaries
- Digital elevation data
- Hydrology
- Satellite imagery
- Roads for each province
- *Statistics Canada, the National Statistical Agency*—Statistics Canada has data for free as well as for cost download, including Canadian boundary and road network files from 2005 through 2007. http://www.statcan.gc.ca/mgeo/boundary-limite-eng.htm

- *GeoGratis*—Natural Resources Canada maintains mostly boundary files for Canada as well as raster data in ESRI shapefiles, raster tiffs, and tabular data. Data is free for download for both commercial and noncommercial. http://geogratis.cgdi.gc.ca/

Probably the one for most general use is the framework data files: http://geogratis.cgdi.gc.ca/geogratis/en/download/framework.html

Other countries and continents

UK Ordnance Survey—Ordnance Survey recently launched its 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/

Regional

These locations cover a small region such as a state or city but have a lot of spatial data for that region:

MassGIS—We couldn't talk about data without talking about our favorite state and the state we live in. MassGIS 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, it has various web services you can use to consume its spatial data if you just need to use it 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 are just looking for data to play with, you'll want to check out DataSF.org. It's part of a pilot project called CivicDB, which hopes to provide a reference implementation for other government agencies. Details can be found here: http://it.toolbox.com/blogs/database-soup/datasforg-is-now-up-33563?rss=1.

At DataSF, you can find lots of spatial vector and aerial data for San Francisco, including data such as streets, shorelines, bridges, Zip Codes, city projects such as renewal and revitalization, city parcels, and zones. http://www.datasf.org

Sample data for training

OpenGeo Introduction to PostGIS—This includes both data and a workshop mostly licensed under Creative Commons. http://workshops.opengeo.org/postgis-intro/

North Carolina Educational Dataset—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/ncexternal/index.html.

This is from our PGCon 2009 presentation and is 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

Spatial reference systems resources

Next, we list some resources on spatial reference systems that we've found useful:

Enchanted Learning—A good primer on map projections. http://www.enchantedlearning.com/geography/glossary/projections.shtml

Spatial Reference—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 that don't come packaged with the default PostGIS spatial_ref_sys table) and 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 user-submitted ones. http://spatialreference.org

Summary by Morten Nielsen—http://www.sharpgis.net/post/2007/05/Spatial-references2c-coordinate-systems2c-projections2c-datums2c-ellipsoids-e28093-confusing.aspx

A good description of conical/cylindrical and oblique, equatorial, transverse—http://geology.isu.edu/geostac/Field_Exercise/topomaps/map_proj.htm

Gory details of the mathematical definition of an ellipsoid—http://en.wikipedia.org/wiki/Ellipsoid

Figure of the earth and various ellipsoids used over the years—http://en.wikipedia.org/wiki/Figure_of_the_Earth

SQL Server 2008 documentation by Isaac Kunen explaining spatial coordinate systems and the difference between flat and round earth models—It's a surprisingly good description complete with pictures. http://msdn.microsoft.com/en-us/library/cc749633.aspx

PROJ.4 Wiki—PROJ.4 is the Cartographic Projections 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

Projections Transform List—This article is a quick primer on common spatial reference systems and the PROJ.4 equivalents. Again, this is of use for those who want to look at an example of how to define a custom spatial reference system with PROJ.4 syntax. http://www.remotesensing.org/geotiff/proj_list/

appendix B Installing, compiling, and upgrading

There are several ways to install PostgreSQL/PostGIS. When we were starting out, the only way was to compile the code yourself. Life has become much simpler, and the general user doesn't need to experience the joys and frustrations of compiling their own source code. Compiling from source is still an adventurous journey and builds character, but most take the easy road.

Installing PostgreSQL and PostGIS

It goes without saying that you need a functioning PostgreSQL server to use PostGIS. The installation options we discuss describe the base PostgreSQL installation as well as the additional PostGIS installation.

Desktop Linux, Windows, Mac OS X using one-click installers

If you're on a Windows system or a desktop Linux or Mac OS X system, the easiest way to get started is to use one of the one-click installers provided by EnterpriseDB at http://www.enterprisedb.com/products/pgdownload.do.

EnterpriseDB one-click installers will work for any desktop Linux system (32 bit and 64 bit), Windows system (2000, XP, 2003, 2008), and Mac OS X. The installer comes with the following prepackaged goods:

- PostgreSQL Server
- pgAdmin III (GUI database administration tool)
- Application Stack Builder—Allows you to install PostgreSQL add-ons such as PostGIS, JDBC, and ODBC, plus application development environments such as Apache and Ruby on Rails, the PostgreSQL Tuning Wizard (to help you

quickly configure memory and other settings for your desired profile), and the MySQL Migration Wizard (requires Java to 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, you can download the PostgreSQL binaries directly from the PostgreSQL website at http://www.postgresql.org/download/ and the precompiled Stack Builder PostGIS pieces (Windows precompiled but 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 getting started, check out appendix A, "Additional resources."

WINDOWS VISTA GOTCHAS

When trying to install on Windows Vista you may get an error something of the form shown in figure B.1.

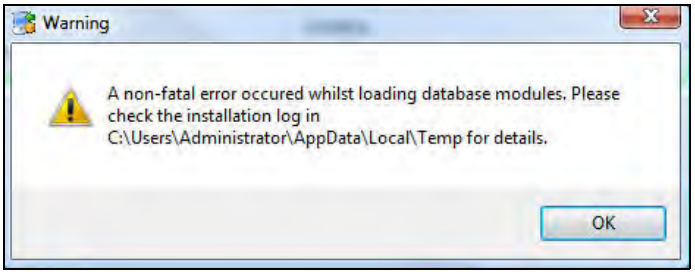


Figure B.1 Common error on Windows Vista

And then it proceeds to uninstall PostgreSQL. This is because of the security measures added to Vista. If you're installing on Windows Vista, follow the instructions outlined in the PostGIS Wiki at http://trac.osgeo.org/postgis/wiki/UsersWikiWinVista.

> **Turn off User Account Control**
>
> It may be sufficient to just turn off User Account Control (UAC) located in Control Panel > User Accounts. In most cases the newer PostgreSQL installers seem capable of creating a postgres account on their own.

Installing on Linux server (Red Hat EL, CentOS) using YUM

The PostgreSQL Yum repository has the latest and greatest for the main and beta versions of PostgreSQL, if you're running a variant of Red Hat Enterprise Linux, Red Hat Fedora, or CentOS. In addition to core PostgreSQL, the Yum repository has additional packages, such as PostGIS and other add-ons, available at http://yum.pgrpms.org/.

The Yum repository is most suitable for command-line server installs, but it can also be used for desktop installs via the Yum installer.

Details on how to install can be found at the following link we've written: Postgres OnLine Journal, An 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've written about on YUM check out http://www.postgresonline.com/journal/categories/53-yum.

OpenSUSE and SUSE aren't available yet through this repository, but these are a planned addition in the future. For OpenSUSE you can use the EnterpriseDB one-click installers or via distros of OpenSUSE/SUSE.

Mac OS X–specific installers

If you're a Mac user, you might want to check out KyngChaos: http://www.kyngchaos.com/software:postgres.

KyngChaos has packages for the latest stable releases of PostGIS/PostgreSQL and pgRouting. Packages are usually for the latest two versions of Mac OS X (currently Leopard and Snow Leopard) as well as a number of other interesting GIS open source packages. KyngChaos generally stays up to date with the latest and greatest PostGIS stable offerings and has them available right after the PostGIS source is officially released.

Other available binaries and distros

Most of the other distros such as Ubuntu and Debian make PostgreSQL and PostGIS available via their package manager. Ubuntu and Debian both use apt-get. These don't always have the latest version binaries, though things are improving nowadays, so they're more up to date. As of this writing, we'd suggest using the aforementioned PostgreSQL binaries if you can, because those are specifically maintained by PostgreSQL high-end users and so are most always up to date.

Compiling and installing from PostGIS source

If you want the most bleeding-edge version of PostGIS, compiling it yourself is still the only way to go. This is covered in the PostGIS manual and wiki, which are kept fairly current.

- Chapter 2 of the official PostGIS manual covers standard compilation on Linux systems; see http://www.postgis.org/documentation/manual-svn/postgis_installation.html#PGInstall.

  Although you can compile your own PostgreSQL, if you're on Linux you don't need to even if you want to compile PostGIS yourself. But you'll need the PostgreSQL development headers (usually packaged in something called postgresql-devel) in addition to installing postgresql and postgresql-server, which you can install using your Linux packager, such as YUM, YatZ, apt-get, or whatever your distribution uses for software install. The YUM, YatZ, and apt-gets of the world are similar in concept to Windows Update and provide already precompiled

binaries of PostgreSQL and PostGIS for your particular Linux distribution from well-defined repositories.

- There is a whole user-contributed section on the PostGIS wiki with details on compiling PostGIS on various operating systems; see http://trac.osgeo.org/postgis/wiki/UsersWikiMain.
- For Windows, as of this writing (headers aren't available for Windows because 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.

PostGIS under Visual C++/Visual Studio

PostgreSQL 9.0 introduced support for 64-bit on Windows. A few people have been able to compile PostGIS under Visual Studio (targeting 64-bit) and Visual C++ Express editions of 2005/2008, 32-bit only (note that Express doesn't support 64-bit compilation), by creating their own project and solution files. The PostGIS development team is currently working on direct support for Windows Visual C++, Visual Studio, and also Msys64 to make support of the newer PostgreSQL versions on Windows 64-bit easier.

If you don't want to experience 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.

Creating a PostGIS database

The Windows one-click installer already creates a template_postgis database for you, but many of the binary packages won't.

A template database is a database that serves as a model for new databases. If you create a template specifically for PostGIS work, you can use it as a quick way to create PostGIS-enabled databases. You can also add other stuff you commonly use; for example, if you're a Python programmer you may want to enable plpython in this template). For those with a SQL Server background, 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 to be used for various use cases. PostgreSQL comes packaged with two template databases: template0 and template1. Template0 is a super-plain-vanilla database with the absolute minimum required for a functioning PostgreSQL database. Template1 is the more commonly used one, with the common function libraries and languages already installed.

Before you can even create a spatial database, or any database for that matter, you need to be able to log in to your PostgreSQL server via pgAdmin III or psql. If you have problems doing that, then please refer to appendix D.

Creating template_postgis under PostGIS 1.3.x

Listing B.1 is a simple script to create a template_postgis database for PostGIS 1.3.x installations using psql. Please note that the paths may be different from installation to installation, so you may need to change the paths.

For Red Hat Enterprise Linux installations, the path locations are the ones listed. For Windows, the path is usually C:/Program Files/PostgreSQL/8.x/share or C:/Program Files (x86)/PostgreSQL/8.something/share (for 64-bit Windows running 32-bit PostgreSQL).

To launch psql, enter the following from the command line on the PostgreSQL server or in PgAdmin3:

```
psql -d postgres -U postgres
```

If psql isn't accessible without the full path name, you may need to include the full path to your postgresql/bin folder or add the postgresql bin folder to your system PATH variable.

Once you're in psql, enter the code in the following listing.

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';
GRANT ALL ON geometry_columns TO PUBLIC;
GRANT ALL ON spatial_ref_sys TO PUBLIC;
\q
```

1 Create postgis template database

2 Grant permissions

3 Quit psql

This code creates a 1 new template PostGIS 1.3 database with all the PostGIS functions and metatables and 2 grants all permissions to all for the geometry_columns and spatial_ref_sys. This means that when a database is created based on this template, new geometry columns can be registered by the user and new spatial_ref_sys records can be added as needed. 3 The `\q` is a psql-only command not available in pgAdmin that exits psql.

Creating template_postgis under PostGIS 1.4,1.5+

Listing B.2 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 Red Hat Enterprise Linux installs, the path locations are the ones listed. For Windows, the path is usually C:/Program Files/PostgreSQL/8.x/share/contrib/post-

gis-1.4 (postgis-1.5) or C:/Program Files (x86)/PostgreSQL/8.*x*/share/contrib/postgis-1.5 (for 64-bit Windows running 32-bit PostgreSQL).

To launch psql, enter the following from the command line on the PostgreSQL server:

```
psql -d postgres -U postgres
```

If psql isn't accessible without the full path, you may need to include the full path to your postgresql/bin folder.

Once you're in psql, enter the code in the following listing.

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-1.5/postgis.sql;
\i /usr/share/pgsql/contrib/postgis-1.5/spatial_ref_sys.sql;
\i /usr/share/pgsql/contrib/postgis-1.5/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
```

You use this listing at the PSQL command line or pgAdmin III to create a new template PostGIS 1.5 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.

Packaging change in PostGIS 1.4/1.5

In PostGIS 1.4 the library generated changed to include the version number, for example, postgis-1.4.so or postgis-1.4.dll or postgis-1.5.so. This allowed for the possibility of running multiple versions of PostGIS on the same server. In PostGIS 1.5 the install process was changed so that PostGIS scripts are also installed in their own versioned folders so they don't overwrite each other during install. All Windows installs via Stack Builder have this naming change from PostGIS 1.4 on, but Linux and Mac OS X installations didn't change to this standard until PostGIS 1.5.

Creating a new spatially enabled database

Once you've created a template_postgis database, you can use it to create new PostGIS-enabled databases. You can create a new spatially enabled database with pgAdmin III or the psql command-line/shell tool.

USING PGADMIN III

If you're using pgAdmin III, right-click the database tree icon and create a new database. Then choose the template_postgis database as your template, as shown in figure B.2.

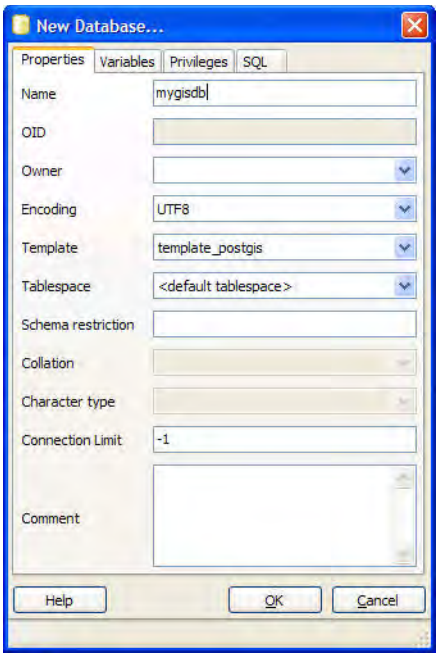


Figure B.2 New Database dialog box in pgAdmin III with the template_postgis database selected

USING PSQL OR SHELL VIEW

If you don't have pgAdmin III or prefer to live in the shell, you have two options (accessed in Linux by regular shell commands, in Windows by launching Start > Programs > PostgreSQL > SQL Shell). pgAdmin III versions 1.10 and above also include on the plug-ins menu an option to launch psql connected to your selected database.

Connect via psql to any database and run the following command:

```
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
```

Spatially enabling an existing PostgreSQL database

To spatially enable a PostgreSQL database that doesn't have PostGIS installed, follow these steps:

1. Install PostGIS binaries, which you can get from the one-click installer, from Stack Builder (should be on your PostgreSQL menu), or via Yum or your distro. Or you can compile and install the binaries following the directions on the PostGIS wiki.
2. Then run the same scripts we demonstrated to install template_postgis.
3. To verify your install, run the following query after connecting to the newly created database from either psql or pgAdmin III:

```
SELECT postgis_full_version();
```

Upgrading an existing install

Before you upgrade your PostGIS install, you should first verify which version of PostGIS you're running:

```
SELECT postgis_full_version();
```

After verifying your current install, compile or upgrade the PostGIS binaries as described earlier, and then follow the steps in the following sections to upgrade your database.

> **PostGIS 1.3, PostGIS 1.4, and PostGIS 1.5 can coexist**
>
> As of PostGIS 1.4, it's possible to have PostGIS 1.3, PostGIS 1.4, and PostGIS 1.5 all installed on the same PostgreSQL server but using different PostgreSQL databases, which would be useful for testing functionality. If you have multiple versions, you'll want to name your templates something like template_postgis13, template_postgis14, and template_postgis15.
>
> They all have to share the same GEOS and Proj libraries, however, so keep that in mind.

Upgrading database from 1.3.x to 1.3.x+

If you're 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 /path/to/pgsql/share/contrib/postgis/lwpostgis_upgrade.sql to upgrade your database.

Upgrading database from 1.3.x to 1.4.x or 1.3.x to 1.5.x

If you're 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 example, for 1.5 it would be `postgis_upgrade_13_15.sql` and so on.

Then once you've finished, to verify you're running 1.4, enter

```
SELECT postgis_full_version();
```

Hard upgrades

Hard upgrades are required when upgrading from a PostGIS major release to another major release (for example, 0.9 to 1.3). Although they aren't 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 that's a bit more time consuming than what we'll demonstrate and also requires that you have Perl installed, which may not be an option on a Windows Server.

Official hard upgrade instructions are provided on the following sites:

- *1.5*—http://www.postgis.org/documentation/manual-1.5/ch02.html#upgrading
- *Latest*—http://www.postgis.org/documentation/manual-svn/postgis_installation.html#hard_upgrade

HARD UPGRADE FROM 0.X, 1.X TO 1.3.X OR 1.4 OR 1.5

If you're running an older version between 1.1 and 1.2.2 (or lower) or are upgrading to a major release (such as 1.2 to 1.3), you should do a hard upgrade even though running the upgrade script may lead you to believe that you can do a soft upgrade. The reason is that certain things such as changes to CASTS, types, and operators can't be accomplished with a soft upgrade. Also, you can't drop items 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, however.

You'll want to dump your database with the following command (replace `localhost` with your server's name if you aren't local to it). You do this step from the command line, or if you're 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 figure B.3.

```
/path/to/pgsql/bin/pg_dump -i -h localhost -p 5432
➥ -U youruser -F c -b -v
➥  -f "/path/to/backup/yourdbhere.backup" yourdbhere
```

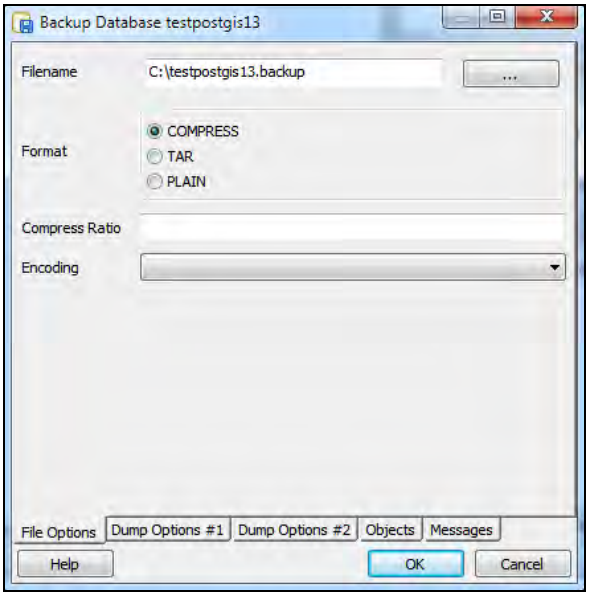


Figure B.3 If you're using pgAdmin III for backup, your screen should look like this.

Launch psql (or you can use pgAdmin III):

```
/path/to/pgsql/bin/psql -i -h localhost -p 5432 -U postgres
```

Then perform the following steps from the psql prompt.

Install a new PostGIS and create a new template_postgis based on the new PostGIS install. If you're on the same server as your old database, drop the old database (but make note of the database encoding because you'll want to create your new database with the same encoding).

The example shown in the following listing uses UTF8; you should replace it with whatever your old database was encoded with.

Listing B.3 Hard upgrade

```
DROP DATABASE yourdbhere;
CREATE DATABASE yourdbhere WITH ENCODING='UTF8'
➥ TEMPLATE=template_postgis;
\q
```

Drop your old

Create new version based on template_postgis

Quit out of psql

Alternatively, you can use pgAdmin III, as shown in figure B.4.

Once you have a fresh PostGIS database, you can restore your old data on top of it. The pg_restore won't put in the old functions (it will skip over functions and aggregates already present), so the new functions you install will take precedence. As for data, it won't re-create tables already present but will insert data into those tables if the structure is the same.

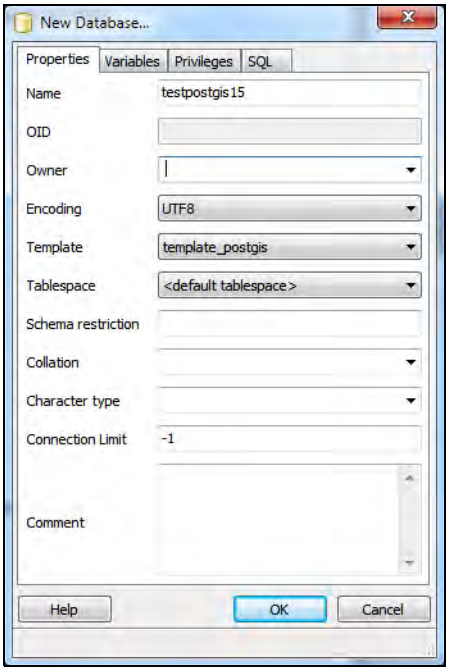


Figure B.4 Creating a new database with pgAdmin III using template_postgis

Old spatial reference records not preserved

Because spatial_ref_sys has a primary key and data, it will fail trying to add records, so any custom records you added will need to be added again. Alternatively, you can delete the spatial_ref_sys table before doing the restore so your old spatial_ref_sys will get restored, but then you'll lose the corrections made in the newer spatial_ref_sys.

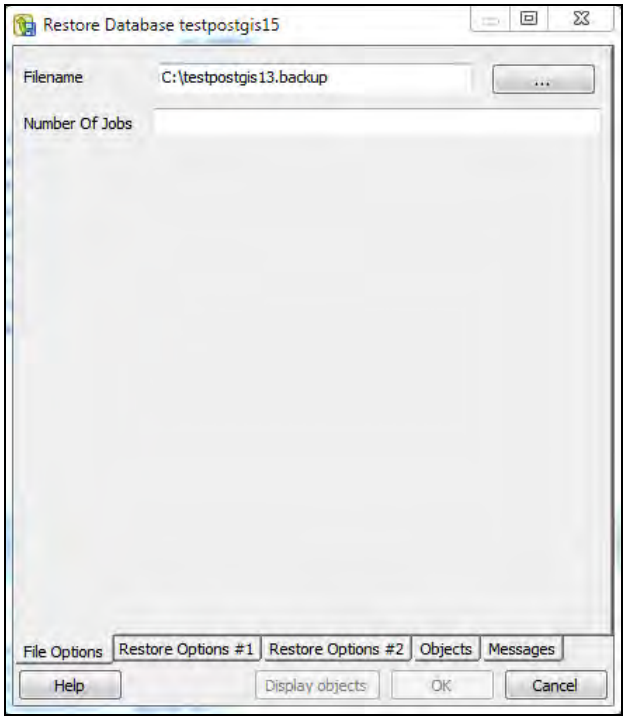


Figure B.5 Doing a restore with pgAdmin III

You can restore using the command-line pg_restore.exe packaged with PostgreSQL:

```
/path/to/bin/pg_restore --host=localhost --port=5432 --username=postgres --
dbname=yourdbgoeshere --verbose "/path/to/yourbackupfile"
```

Or you can use pgAdmin III, if network bandwidth between the PostgreSQL server and your workstation is good (or your database is small), as shown in figure B.5.

Versions of pgAdmin III from 1.12 on provide additional options, such as the number of jobs, shown in figure B.5, which allows you to specify the number of parallel jobs to use for restore. This feature works only with PostgreSQL 8.4+. It can reduce the restore process by half or more. In addition, the other Restore tabs provide options such as selective restore of tables and how to fail on errors.

appendix C
SQL primer

PostgreSQL supports almost the whole ANSI SQL-92, 1999 standard logic as well as many of the SQL:2003, SQL:2006, and some of the SQL:2008 constructs. In this appendix we'll cover these as well as some PostgreSQL-specific SQL language extensions. Because we're remaining fairly focused on standard functionality, the content in this appendix is applicable to other standards-compliant relational databases.

Information_schema

The information_schema is a catalog introduced in SQL-92 and enhanced in each subsequent version of the specs. Although it's a standard, sadly most commercial and open source databases don't completely support it. We know that the following common databases do: PostgreSQL (7.3+), MySQL 5+ (not sure about 4), and Microsoft SQL Server 2000+.

The most useful views in this schema are tables, columns, and views; they provide a catalog of all the tables, columns, and views in your database.

To get a list of all non-system tables in PostgreSQL, you can run the following query, which will work equally well in MySQL (except that in MySQL *schema* means "database" and there's only one information_schema shared across all MySQL databases in a MySQL cluster). MS SQL Server behaves more like PostgreSQL in that each information_schema is unique to each database, except that in SQL Server the system views and tables aren't queryable from the information_schema, whereas they are in PostgreSQL. The tables view in PostgreSQL will list only tables that you have access to:

```
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 following example we list all the geometry columns found in a schema called hello.

Listing C.1 List all columns in hello schema

```
SELECT c.table_name, c.column_name, c.data_type, c.udt_name,
    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.

Table C.1 Results of query in listing C.1

table_name	column_name	data_type	udt_name	ord_pos cmax	cdefault
coastline	coastline_id	integer	int4	1	nextval('hello....)
coastline	coastline_name	character varying	varcha	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. Because PostGIS geometry is an add-on module and not part of PostgreSQL, you'll see the standard ANSI data_type listed as USER-DEFINED and the udt_name storing the fact that it's a geometry.

This view provides numerous other fields, so we encourage you to explore it. We've listed here what we consider the most useful fields:

- *table_name and column_name*—These 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 that we created coastline_id as a serial data type, and PostgreSQL behind the scenes created an integer column and 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 in which the column appears in the table.
- *character_maximum_length*—With character fields, 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 gives you the name and the view_definition for each view 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, because the information_schema is excluded from this system view.

```
SELECT table_schema, table_name, view_definition,
is_updatable, is_insertable_into
FROM information_schema.views
WHERE table_schema = 'information_schema';
```

In these examples, we demonstrated the common metatables you'd find in the ANSI information_schema. We 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.

Querying data with Structured Query Language

The cornerstone of every relational database is the declarative language called Structured Query Language (SQL). Although each relational database has a slightly different syntax, the fundamentals are pretty much the same across all relational DBMSes.

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'll cover in the next section.

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 clause isn't needed in PostgreSQL, SQL Server, or MySQL, whereas in databases such as Oracle and IBM DB2, you need to select FROM dual or sys.dual or some other dummy table.

BASIC SELECT

A basic select looks something like this:

```
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'd do the following or use the non-portable ILIKE PostgreSQL predicate:

```
SELECT gid, item_name, the_geom
FROM feature_items
WHERE upper(item_name) LIKE 'QUEENS%';
```

There's no guaranteed order for results to be returned, but sometimes you care about order. The SQL ORDER BY clause satisfies this need for order.

Following 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 should uppercase your ORDER BY field, but PostgreSQL 8.4 provides a new per-database collation feature that makes this not as necessary depending on the collation order you've designated for your database.

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 want to select all fields from only 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 it 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's 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's useful to put an index on that field. For a few distinct groupings of records by a column, the index is more harmful than helpful, because the planner will ignore it and do a faster table scan, and updating will even incur a heavy performance penalty.

ALIASING

In the examples using the 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's defined in the database. It's indispensible when doing 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 to make your code easier to read and also reduce typing by shortening long table 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-in support for subselects. Much of the expressiveness and complexity of SQL consists of keeping subselects straight and knowing when and when not to use them. For PostgreSQL most valid SELECT ... clauses can be used as subselects, and when used in a FROM clause, they 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:

- *UNION, INTERSECT, EXCEPT*—You'll learn about these shortly.
- *In the FROM clause*—Where you'd normally put a table name and where it acts like a virtual table. The subselect needs to have an alias name to define how it will be called in other parts of the query, and it 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 2005+ CROSS APPLY.
- *In the definition of a calculated column*—When used in this context, the subselect can return only 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 multicolumn row as a column expression. Because this isn't a feature you'll commonly find in other databases and is of limited use, we won't cover it in this appendix. You can, however, return multiple rows as an array if they contain only one column using ARRAY in PostgreSQL. This will return the column as an array of that type. We demonstrate this in various parts of the book. Again this is a feature that's fairly unique to PostgreSQL but very handy for spatial queries.

- *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 available only in PostgreSQL 8.4+. You'll also find it in Oracle, SQL Server 2005+, IBM DB2, and Firebird. You won't 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 because they have to be calculated for each unique combination of fields and have a dependency on the outer query.

In the following listing are some examples of subselects in action. Don't worry if you don't completely comprehend them, because some require an understanding of topics that we'll cover shortly.

Listing C.2 Subselects 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
        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;
```

This statement uses a subselect to define the derived table we define as r. This is the common use case. We'll demonstrate the same statement in listing C.3 using the PostgreSQL 8.4 WITH clause. The WITH clause, sometimes referred to as a Common Table Expression (CTE), is an advanced ANSI-SQL feature that you'll find in SQL Server, IBM DB2, Oracle, and Firebird, to name a few.

Listing C.3 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 query using a correlated subquery.

Listing C.4 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 you can use any of these to get the same result, the strategies used by the planner are very different, and depending on what you're doing, one can be much faster than the other. With large numbers of returned rows, you should avoid the correlated subquery approach, but in certain cases it can be necessary to use a correlated subquery, for example, to prevent duplication of count.

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. In a spatial database you'll find that the JOIN is often based on a proximity condition rather than on keys. The clauses LEFT JOIN, INNER JOIN, CROSS JOIN, RIGHT JOIN, FULL JOIN, and NATURAL JOIN exist in the ANSI SQL specifications. PostgreSQL supports all of these. SQL Server supports them as well. MySQL lacks FULL JOIN support. Oracle does support these, but it also has its own proprietary syntax (WHERE *= etc.) that is non-standard and is still often used today by long-time 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 maximum number of records returned by a LEFT JOIN is m*n rows, where m is the number of rows in M and n is the number of rows in N. The number of columns is the number of columns selected from M plus the number of columns selected from N.

Generally speaking, if your M table has a primary key that's the joining field, you can expect the minimum number of rows returned to be m and the maximum to be m + mxn – n.

NULL placeholders are put in N table's columns where there's no match in the M table. You can see a diagram of a LEFT JOIN in figure C.1.

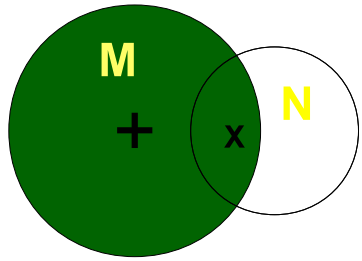


Figure C.1 Diagram of a LEFT JOIN. The darkened region represents the portion of records returned by a LEFT JOIN. The x stands for multiplication and the + is additive. The first circle is M and the second circle is N.

Following 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;
```

This query will list both cities that have airports and cities that don't have airports based on the city_code. We assume city_code to be the city primary key with a foreign key in the airports table. If the LEFT JOIN were changed to an INNER JOIN, only cities with airports would be listed. With a LEFT JOIN, cities that have no airports will get 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 placeholders where there's no match. When using this, make sure the field you're joining with is guaranteed to be filled in when there are matches; otherwise, you'll get spurious results. For example, a good candidate would be the primary key of a table. Here's an example of such a trick:

```
SELECT c.city_name
FROM city As c
LEFT JOIN airports As ON a.city_code=c.city_code
WHERE a.airport_code IS NULL;
```

In this example code we're returning all cities with no matching airports. We're making the assumption here that the airport_code is never NULL in the airports table. If it were ever NULL, this wouldn't work.

INNER JOIN

The INNER JOIN returns only records that are in both M and N tables, as shown in figure C.2. The maximum number of records you can expect from an inner join is (m x n). Generally speaking, if your M table has a primary key that's the joining field, you can expect the maximum number of rows to be n. A classic example is customers joined

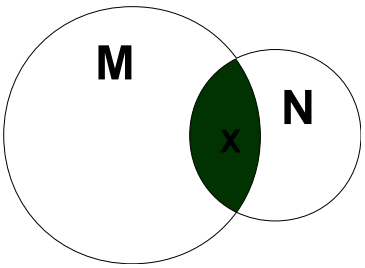


Figure C.2 Diagram of an INNER JOIN. The darkened region represents the portion of records returned by the INNER JOIN. The x denotes that it's multiplicative. The first circle is M and the second circle is N.

with orders. If a customer has only five orders, the number of rows you'll get back with that customer id and name is five.

Following is an example of an INNER JOIN:

```
SELECT c.city_name, a.airport_code, a.airport_name, a.runlength
FROM city AS c
    INNER JOIN airports a ON a.city_code = c.city_code;
```

In this example we list only cities that have airports and only the airports in them. If we had a spatial database, we could do a JOIN using a spatial function such as ST_Intersects or ST_DWithin and could also find airports in proximity to a city or in a city region.

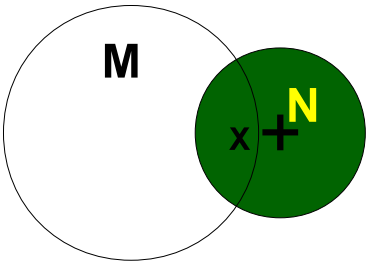


Figure C.3 Diagram of a RIGHT JOIN. The darkened region represents the portion of records returned by a RIGHT JOIN. The x stands for multiplication and the + is additive. The first circle is M and the second circle is N.

RIGHT JOIN

The RIGHT JOIN returns all records in the N table and only records in the M table that match records in N, as shown in figure C.3. 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, but flipping the table order in the clause.

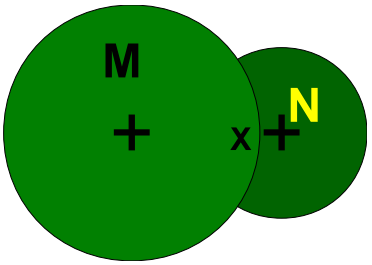


Figure C.4 Diagram of a FULL JOIN. The darkened region represents the portion of records returned by a FULL JOIN. The x stands for multiplication and the + is additive. The first circle is M and the second circle is N.

FULL JOIN

The FULL JOIN, shown in figure C.4, returns all records in M and N and puts in NULLs as placeholders in fields where there's no matching data. There's a lot of debate about the usefulness of this. In practice it's 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, or INNER join; the 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's possible to do a FULL JOIN using spatial functions like ST_DWithin or ST_Intersects, in practice this isn't currently supported, even as of PostgreSQL 9.0, PostGIS 1.5.

CROSS JOIN

The CROSS JOIN is the cross product of two tables, where every record in the M table is joined with every record in the N table, as illustrated in figure C.5. The result of a CROSS JOIN without a WHERE clause is m x n rows. It's sometimes referred to as a Cartesian product.

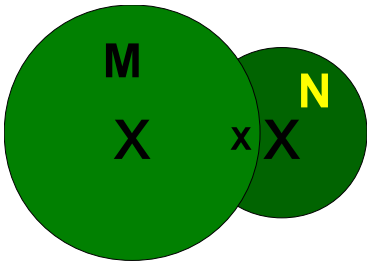


Figure C.5 Diagram of a CROSS JOIN. The darkened region represents the portion of records returned by the CROSS JOIN. The x stands for multiplication. The first circle is M and the second circle is N.

Here's an example of a good use for a CROSS JOIN. The following calculates the total price of a product including state tax for each state:

```
SELECT p.product_name, s.state, p.base_price * (1 + s.tax) As total_price
FROM products AS p
CROSS JOIN state AS s;
```

It can also be written as

```
SELECT p.product_name, s.state, p.base_price * (1 + 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 the WHERE part, but we prefer the more explicit INNER JOIN because it's 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's no absolute rule about it. The distinction becomes important when doing LEFT JOINs, as you saw with the LEFT JOIN orphan trick.

NATURAL JOIN

A NATURAL JOIN is like an INNER JOIN without an ON clause. It's supported by many ANSI-compliant databases. It automagically joins same named columns between tables; thus there's no need for an ON clause.

> **Just say no to the NATURAL JOIN**
>
> We highly suggest you stay away from using this. It's a lazy and dangerous way of doing joins that will come to bite you when you add new fields with the same names that are totally unrelated. We feel so strongly about not using this that we won't 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 their order. Here's an 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.rental_id);
```

This example is from the PostgreSQL pagila database. The pagila database is a favorite for demonstrating new features of PostgreSQL. You can download it from http://pgfoundry.org/projects/dbsamples/. In the previous example we find all the customers who have had rentals and list the rental fields as well (note that the INNER JOIN kicks out all customers who haven't made rentals). We then pull the payments they've made for each rental and have NULLs if no payment was made but still list the rentals.

Sets

A set looks like a JOIN and is 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're *not* talking about the SET clause you'll find in UPDATE statements.

SQL clauses we consider as sets are UNION [ALL], INTERSECT, and EXCEPT. PostgreSQL supports all three, though many databases support only 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 autocast 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, and EXCEPT, which talk 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.

UNION AND UNION ALL

The most common type of set includes the UNION and UNION ALL sets, illustrated in figure C.6. Most relational databases have at least one of these and most have both. A UNION takes two 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.

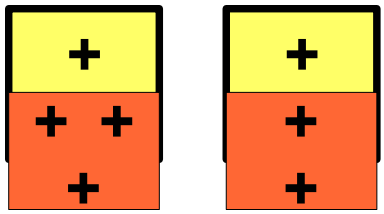


Figure C.6 UNION ALL versus UNION. The thick box is M and the thinner box is N. The first UNION ALL shared regions are duplicated; in UNION only one of the shared regions is kept, resulting in a distinct set.

A union can have multiple chains each separated by a UNION ALL or UNION. The ORDER BY can appear only 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. The following example will list all water features and land features greater than 500 units in area and all architecture monuments greater than 1000 dollars and will order results by item name.

Listing C.5 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;
```

This example will pull data from three 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 mistakenly used

The plain UNION statement is often mistakenly used because it's the default option when ALL isn't 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. We covered this in chapter 4. In short, be careful. In general, you want to use a UNION ALL except when deduping data where you want your datasets to be distinct.

INTERSECT

INTERSECT is used to join multiple queries, similar to UNION. It's defined in the ANSI-SQL standard, but not all databases support it; for example, MySQL doesn't support it, and neither does SQL Server 2000, although SQL Server 2005 and above do.

INTERSECT returns only the set of records that are common between the two result sets, as shown in figure C.7. It's different from INNER JOIN in that it isn't multiplicative and in that both queries must have the same number of columns. In the diagram in figure C.7, the green represents what's returned by an SQL INTERSECT. Later we'll look at a spatial intersection involving an intersection of geometries rather than an intersection of row spaces.

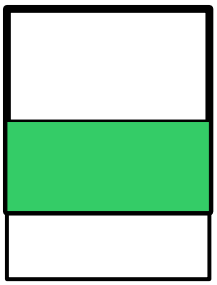


Figure C.7 INTERSECT—the darkened region is the intersection of two data sets returned by an INTERSECT clause.

INTERSECT is rarely used. There are a couple of reasons for that:

- Many relational databases don't support it.
- It tends to be slower than doing the same trick with an INNER JOIN. In PostgreSQL 8.4, the speed of INTERSECTs has been improved, though in prior versions it wasn't that great.
- In some cases, it looks convoluted when you're talking about the same table.
- In some cases it does make your code clearer, such as when you have two disparate tables or when you chain more than two queries. We demonstrate an example using INTERSECT and the equivalent query using INNER JOIN in the following listing.

Listing C.6 INTERSECT compared to INNER JOIN

```
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;

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;
```

❶ INTERSECT example

❷ Same done with INNER JOIN

❶ The query lists all water features greater than 500 square units that are also designated as protected areas inducted after the year 2000.

Note that if the feature_id field isn't unique, the INNER JOIN runs the chance of multiplying records. To overcome that, you may do a subselect, as shown in ❷.

The next example demonstrates 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 and EXCEPT as well. The order of precedence is from top query down unless you have subselect parenthetical expressions.

EXCEPT

An EXCEPT chains queries together such that the final result contains only records in A that aren't 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 section in figure C.8 represents the result of the final query.

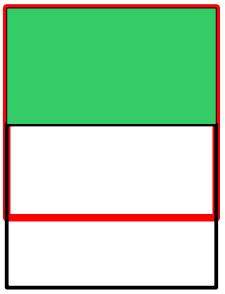


Figure C.8 A demonstration of EXCEPT

EXCEPT is rarely used, but it does come in handy when chaining multiple clauses:

```
SELECT r
    FROM generate_series(1,3) AS r
EXCEPT
SELECT n
    FROM generate_series(3,8) AS n
INTERSECT
SELECT s
    FROM generate_series(2,3) AS s;
```

Using SQL aggregates

Aggregate functions roll a group of records into one record. In PostgreSQL the standard SUM, MAX, MIN, AVG, COUNT, and various statistical aggregates are available out of the box. PostGIS adds approximately nine to the list, of which ST_Collect, ST_Union, and ST_Extent are the most commonly used. We demonstrate an example of spatial aggregates in listing C.7 and several examples in this book. In this section we'll focus on using aggregates. How you use aggregates is pretty much the same regardless whether they're spatial or not.

Aggregates in SQL have generally the following parts:

- *SELECT and FROM*—This is where you select the fields and where you pull data from. You also include the aggregated functions in the select field list.
- *SOMEAGRREGATE(DISTINCT somefield)*—On rare occasions, you'll use the DISTINCT clause within an aggregate function to denote that you use only a distinct

set of values to aggregate. This is commonly done with the COUNT aggregate to count a unique name only once.

NOTE With geometries, what is DISTINCTed is the bounding box, so different geometries with the same bounding box will get thrown out.

- *WHERE*—Non-aggregate filter; this gets applied before the HAVING part.
- *HAVING*—Similar to WHERE, except used when applying filtering on the already aggregated data.
- *GROUP BY*—All fields in the SELECT that are non-aggregated and function calls must appear here (pre PostgreSQL 9.1).

PostgreSQL 9.1 GROUP BY functional dependency enhancement

PostgreSQL 9.1 introduced the functional dependency feature, which means that if you're already grouping by a primary key of a table, you can skip grouping by other fields in that table. This feature is defined in the ANSI SQL-99 Standard. It saves some typing as well as makes it easier to port some MySQL apps.

FAST FACTS ABOUT AGGREGATE FUNCTIONS

There are some important things you should keep in mind when working with aggregate functions. Some are standard across all relational databases, some are specific to PostgreSQL, and some are a consequence of the way PostGIS implements = for geometries.

- For most aggregate functions, NULLs are 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, for example, 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's the bounding box of the geometry that's 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's distinct for each geometry, like the primary key of the table the geometry is coming from.

The following listing is an example that mixes aggregate SQL functions with spatial aggregates.

Listing C.7 Combining standard SQL and spatial aggregates

```
SELECT n.nei_name,
    SUM(ST_Length(roads.the_geom)) as total_road_length,
    ST_Extent(roads.the_geom) As total_extent,
```

```
    COUNT(DISTINCT roads.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(roads.the_geom)) > 1000;
```

The query for each neighborhood specifies the total length of road and the extent of roadway. It also includes a count of unique road names and counts only neighborhoods where the total area of the extent covered is greater than 1000 square units.

Window functions and window aggregates

PostgreSQL 8.4 introduced the ANSI-standard Window functions and aggregates, and PostgreSQL 9.0 improved on this feature by expanding the functionality of BETWEEN ROWS AND RANGE.

Window functionality allows 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 ST_Union and ST_MakeLine calls, which are perhaps solutions 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 that of IBM DB2 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 also supports named window frames that can be reused by name.

The following example uses the ROW_NUMBER() Window function to number streets sequentially that are within one kilometer of a police station, ordered by their proximity to the police station.

Listing C.8 Find roads within 1 km from each police station and number sequentially

```
SELECT  ROW_NUMBER() OVER (
    PARTITION BY loc.pid
        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;
```

1 Number rows

2 Restart numbering for each pid

3 Order numbers by distance

In this listing we're using ❶ the Window function called ROW_NUMBER() to number the results. The ❷ partition by clause forces numbering to restart for each unique parcel id (identified by pid) that uniquely identifies a police station. The ❸ ORDER BY defines the ordering. In this case we're incrementing based on proximity to the 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 table C.2 we show a subset of our resulting table just for two police stations.

Table C.2 Results of window query in listing C.8

row_num	pid	road_name	dist_km
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

In the next section you'll learn about another key component of SQL. SQL is good for querying data but also useful for updating and adding data as well.

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 you 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'll demonstrate simple constructs as well as ones that are more complex.

Updates

We use the SQL UPDATE statement to update existing data. You can update individual records or a batch of records based on some WHERE condition.

SIMPLE UPDATE

A simple UPDATE will update data to a static value based on a where condition. Following is a simple example of this:

```
UPDATE things
SET status = 'active'
WHERE last_update_date > (CURRENT_TIMESTAMP - '30 day'::interval);
```

UPDATE FROM OTHER TABLES

A simple UPDATE is one of the more common UPDATE statements used. In certain cases, however, you'll need to read data from a separate table based on some sort of related criteria. In this case you'll need to utilize joins within your UPDATE statement.

Here's a simple example that updates the region code of a point data set if the point falls within 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 SUBSELECTS

A subselect, as you 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't 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. Following is such an example that tallies the number of objects in a 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're updating all rows in a table, it's 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 it with the new row in the active heap. In the next section you'll learn how to perform INSERTs.

INSERTs

Just like the UPDATE statement, you 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 three basic forms.

The single-value constructor approach has been in existence in PostgreSQL since the 6.0 days and is pretty well supported across all relational databases. Here we insert a single point:

```
INSERT INTO points_of_interest(fe_name, the_geom)
    VALUES ('Highland Golf Club',
            ST_SetSRID(ST_Point(-70.063656, 42.037715), 4269));
```

The next most popular is the multirow value constructor syntax introduced in SQL-92, which we demonstrated often in this book. This syntax was introduced in PostgreSQL 8.2 and IBM DB2, has been supported for a long time in MySQL (we think since 3+) and was introduced in SQL Server 2008. As of this writing, Oracle has yet to support this useful construct. The multirow constructor is useful for adding more than a single row or as a faster way of creating a derived table with just constants. Following is such

an example excerpted from earlier chapters. The multirow insert is similar to the single. It starts with the word *VALUES*, and then each row is enclosed in parentheses and separated with a comma.

Listing C.9 Multivalue row INSERT: two 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, as shown in listing C.10. In the simplest example it doesn't have a FROM. Some people prefer this syntax because it allows you to alias what the value is right next to the constant. It's also a necessary syntax for the more complex kind of INSERT we'll demonstrate in the next section. Note that this syntax is supported by PostgreSQL (all versions), MySQL, and SQL Server. To use it in something like Oracle or IBM DB2, you need to include a FROM clause, like FROM dual or sys.dual.

Listing C.10 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;
```

This is the standard way of inserting multiple values into a table. It was the only way to do a multirow in pre PostgreSQL 8.2. This is also the only way to do it in SQL Server 2005 and below.

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're copying a filtered set of data from another table. It uses the SELECT syntax usually with a FROM and sometimes accompanying joins. Here we 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. The following listing is a somewhat complex case: Given a table of polygon chain link edges, it constructs polygons and stuffs them into a new table of polygons.

Listing C.11 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 poly.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're also creating the table to hold the data 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 it's more cross platform (will work on SQL Server as well as MySQL, for example).
- The other is a CREATE TABLE .. AS SELECT .., which isn't as well supported by other relational databases.

In both cases any valid SELECT statement or WITH statement can be used. The following listing shows examples of the same statement written using SELECT INTO and CREATE TABLE AS.

Listing C.12 Example SELECT INTO and CREATE TABLE

```
SELECT t.region_code, COUNT(t.gid) As cnt
    INTO thingy_summary
    FROM things AS t
GROUP BY t.region_code;

CREATE TABLE thingy_summary AS
    SELECT t.region_code, COUNT(t.gid) As cnt
        FROM things AS t
    GROUP BY t.region_code;
```

1 cross-platform bulk insert

2 less cross-platform way

❶ This is the standard more cross-database-platform way of creating a table and inserting the data in one go. ❷ This is more of a PostgreSQL-specific way that's a bit clearer in style but not as cross platform. If you need your code to support multiple vendor databases, you're better off with ❶.

DELETEs

DELETEs are the most limiting as far as joins go. When doing a DELETE you can't 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 subselects but usually has a WHERE clause. All the data in a table is deleted and logged if you're missing a WHERE clause. Following 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 because it does much less transaction logging than a standard DELETE FROM, but it can be used only in tables that aren't involved in foreign key relationships. Here's an example of it at work:

```
TRUNCATE TABLE streets;
```

ADVANCED DELETE

An advanced DELETE involves subselects in the WHERE clause. These are useful for cases where you need to delete all data in your current table that's in the table you're adding from or you need to delete duplicate records. The following example deletes duplicate records:

```
DELETE
FROM     sometable
WHERE    someuniquekey NOT IN
    (SELECT     MAX(dup.someuniquekey)
        FROM           sometable As dup
        GROUP BY     dup.dupcolumn1, dup.dupcolumn2, dup.dupcolum3);
```

Now that we've covered the basics of SQL in PostgreSQL, this 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

In this appendix, we cover features and behaviors that are fairly distinctive to PostgreSQL, which make it a little different from working with other relational databases.

Useful PostgreSQL resources

Below you'll find a list of key PostgreSQL resources that cover general PostgreSQL usage in addition to resources for add-on tools and performance.

General

- *PostgreSQL wiki*—User contributed articles about various PostgreSQL topics ranging from administration, performance tuning, and writing queries to using PostgreSQL in various application and programming environments. http://wiki.postgresql.org/wiki/Main_Page

  Check out the code snippets repository for lots of useful PostgreSQL functions you can copy and paste into your database. http://wiki.postgresql.org/wiki/Category:Snippets
- *Planet PostgreSQL*—Blog roll of PostgreSQL-specific blogs. Learn from hardcore long-time PostgreSQL users how to get the most out of PostgreSQL. Also learn what's new and hot in PostgreSQL. http://planet.postgresql.org/
- *Our blog/journal*—We try to cater to new PostgreSQL users, programmers, and database users coming from other database systems such as MySQL, SQL Server, or Oracle. http://www.postgresonline.com
- *PostgreSQL main site*—You can download the source from here as well as get flash news. You can also download the manual in PDF form or leaf through the HTML version online. The manual is huge and consists of five volumes. http://www.postgresql.org

- *PostgreSQL 9.0 High Performance and PostgreSQL 9 Administration Cookbook*—A fairly recent couple of books on PostgreSQL are written by 2ndQuadrant consultants who are major contributors to PostgreSQL. These books cover PostgreSQL 8.1–9.0. These and other PostgreSQL books are listed on http://www.postgresql.org/docs/books/.
- *PostgreSQL Yum repository*—If you're a Centos, Fedora, or Red Hat Enterprise Linux user, this is a painless way of installing PostgreSQL service and keeping it up to date. There are Yum updates for even the latest beta versions of PostgreSQL. http://yum.pgsqlrpms.org/reporpms/repoview/letter_p.group.html

Performance

- *Explaining EXPLAIN*—Covers planner basics up through PostgreSQL 8.3 and some of PostgreSQL 8.4. http://wiki.postgresql.org/images/4/45/Explaining_EXPLAIN.pdf

PostgreSQL-specific tools

- *Packaged with PostgreSQL are psql, pg_dump, pg_dump_all, and pg_restore*—These are command-line utilities for querying, backing up, and restoring PostgreSQL databases. You can get them from your Linux or Mac OS X distribution or from EnterpriseDB one-click installers, or you can download the source from the PostgreSQL core site and compile them yourself.
- *pgAdmin III*—Comes packaged with PostgreSQL, but binaries and source can be downloaded separately if you need to install it on a workstation without a PostgreSQL server. http://www.pgadmin.org/ It's also available via the common operating system distributions.
- *phpPgAdmin*—A PHP web-based database administration tool for PostgreSQL, patterned after phpMyAdmin. http://phppgadmin.sourceforge.net/

Connecting to a PostgreSQL server

Before you can even create a spatial database (or any database for that matter), you need to be able to log in to your PostgreSQL server via pgAdmin III or psql. In this section, we'll cover the basics, the most common problems, and how to work around them.

Core configuration files

If you're just starting out and have just installed your PostgreSQL server, you'll want to pay attention to the following key files, which are all located in the data cluster of your installation.

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, for 32-bit systems is located in C:\Program Files\PostgreSQL\8.4\data 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.

In PostgreSQL 9.0, native 64-bit for Windows OS was introduced. But as of this writing, no PostGIS 64-bit installers are available and 64-bit is not well tested for PostGIS on Windows, though some have claimed success compiling and running it. We hope to have native 64-bit PostGIS for Windows soon. For the time being, even if you're on a 64-bit Windows platform, you should use the 32-bit PostgreSQL installers if you want to use PostGIS.

POSTGRESQL.CONF

The postgresql.conf file is the most important file. It contains all the memory configurations and defaults as well as the listening addresses and ports. If you're 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. But you can have multiple instances sharing the same binaries as long as they have a different data cluster. In practice that's rarely done.

The two settings of most importance for getting started are 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 as follows:

```
listen_addresses = '*'          # what IP address(es) to listen on;
                                # comma-separated list of addresses;
                                # defaults to 'localhost', '*' = all
                                # (change requires restart)
```

The port setting defaults to 5432, but if you want to run multiple PostgreSQL services, you'll need to have each one set differently. For example, we run PostgreSQL 8.2, 8.3, and 8.4 on our servers for testing. We have our port for 8.4 set to something like this:

```
port = 5434
```

We discuss some of the other important settings in the postgresql.conf file in chapter 9, "Peformance tuning."

PG_HBA.CONF

The pg_hba.conf file controls which users on which IP address ranges can connect to the PostgreSQL service/daemon as well as which authentication scheme is allowed for them.

Launching psql

If you're using a Linux server with just a command-line console (standard for most production web/app servers), you'll need to use psql at least once on the server to get everything rolling.

From the server do the following

```
psql -h localhost -U postgres
```

to verify that you can connect. If you can't, refer to the "Connection difficulties" section, which follows shortly.

Launching pgAdmin III

For new users, we highly recommend the pgAdmin III GUI. If you installed using 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 distributions the path varies.

You can also install pgAdmin on a regular desktop PC that doesn't have a PostgreSQL server installed. Download one of the available binaries from http://www.pgadmin.org/download/.

For versions of pgAdmin III 1.10 and above, you can also launch psql for that specific database within pgAdmin III.

This is useful for taking advantage of special features of psql, like redirecting output to files or importing data from files. To access psql from pgAdmin III, follow these steps:

1. Select the database.
2. Under the Plug-ins icon, choose psql. If the wrong version of psql launches, you can change it in the plugins.ini file in the pgAdmin III install folder or by changing the bin location in the Options tab. This is discussed briefly at http://www.postgresonline.com/journal/archives/145-PgAdmin-III-Plug-in-Registration-PostGIS-Shapefile-and-DBF-Loader.html.

In pgAdmin III 1.13 and above, the plug-in architecture has changed a bit to allow easier adding of more plug-ins without affecting prior or distributed ones. Instead of a single plugins.ini file, there's a plugins.d folder where you would put all the INIs for your plug-ins. These INIs can be given descriptive names. We discuss this change at http://www.postgresonline.com/journal/archives/180-PgAdmin-III-1.13-change-in-plugin-architecture-and-PostGIS-Plugins.html, which also covers registering more custom plug-ins.

Connection difficulties

If you're connecting to the server from a separate desktop PC, then the server needs to listen on one or more IP addresses and should allow remote connections. After making the required changes to configuration files, you must restart the PostgreSQL service. On Windows, you go into Services Manager and restart.

On Linux, if you did install it as a service, which is usually the case when you installed from YUM or a one-click installer, you can usually enter the following from a shell prompt:

```
service postgresql restart
```

CONNECTION REFUSED

If you get an error in pgAdmin III like

```
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 you have one of the following problems:

- Your PostgreSQL server is not started.
- Your PostgreSQL server service is only listening on localhost or a non-accessible IP.
- Your PostgreSQL server is not listening on the port you think it's listening on.
- Your firewall is getting in the way. Generally for firewall issues you can set up an SSH tunnel, which we describe briefly here: http://www.postgresonline.com/journal/index.php?/archives/38-PuTTY-for-SSH-Tunneling-to-PostgreSQL-Server.html. Some third-party PostgreSQL tools 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 file isn't configured right for remote connections or contains errors. We generally set our configuration to something like the example pg_hba.conf in listing D.1 and allow all local connections to be trusted. This means that if you connect from the local machine, you don't need to provide a password, only a valid PostgreSQL username. If you're very security conscious, you could leave this line out and require MD5 or some other security scheme. To make connecting a bit easier, you can set up a .pgpass file, which we'll 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 credentials 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'll have to provide a password even when connecting locally.

Enabling advanced administration for pgAdmin III

Once you're able to connect, you'll be able to administer the postgresql.conf and pg_hba.conf configuration files and also to check server status and do other administrative tasks remotely from another computer using pgAdmin III. For that, you need to run adminpack.sql first, which is located in the contrib folder of your PostgreSQL install.

Do the following from the command line on the PostgreSQL server. The example uses the default Windows install path for PostgreSQL 8.4; if you're on Linux or running a lower version of PostgreSQL, the installation path will be different and you may

not even need to include the psql binary in your path, because it's usually in the default bin folder.

```
"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 PostgreSQL installed on a 64-bit version of Windows, it gets installed by default in the Program Files (x86) directory because PostgreSQL as of this writing doesn't 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 because 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, do the following:

1. Register the server in pgAdmin III.
2. Choose Tools > Server Configuration (you should see both config files there).

Controlling access to data

There are two parts of access control in PostgreSQL. First is control of who can log in and how they can log in, which you 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.

Connection rules

The connection rules are controlled by three files:

- *postgresql.conf*—Controls on what ports and IP addresses the server listens and whether a Secure Socket Layer (SSL) connection is required. For LDAP-like connectivity, it also holds information such as the Kerberos settings to use 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 required for each connection.

  Common authentication schemes include the following:
  - *md5*—MD5 encryption; what most people use, particularly for web apps.
  - *trust*—Ignores the password. Never use this except in a tightly secured local network or on a local PC that has good firewall protection against IP spoofers or that listens only on a local port.
  - *ident*—Trusts a user based on their local identity determined by the OS. Again, it's generally used only for local authentication.
  - *reject*—This kind of authentication doesn't allow you to authenticate. You use this if you want to ban certain IP ranges or everyone who isn't on your network

but who would otherwise be allowed by a broader IP range rule. In such cases, you'd put this rule above the broader rule so that it's resolved first.

The following are less-common schemes, but they're particularly useful if you're in an enterprise network using LDAP or Active Directory (there are even more such as PAM and some others).

- *krb4/5*—Kerberos connection. This is deprecated; don't use it.
- *sspi*—Supported only on Windows and requires the PostgreSQL server to be running on Windows. It's designed for connecting via Windows authentication or NT authentication. It sits on top of Kerberos.
- *gss*—Industry defined protocol similar to SSPI but doesn't require a Windows server; it sits on top of Kerberos.
- *ldap*—authentication via an 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 log in as postgres.
- *pgpass.conf, .pgpass*—This is a local configuration file that stores the usernames, server, and passwords for the database that you connect with. If you're using pgAdmin, then pgAdmin creates this for you automatically and you can export its contents using the File > Open pgpass.conf. Under Windows, this file 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, and pg_restore to automatically log you into a PostgreSQL server without prompting for a username or password. It's useful to have if you don't have trust enabled and you need to schedule backup jobs and such. Keep in mind that the file must exist under the account doing the work, so if you're doing backup under a service account such as postgres or Administrator, then you need to copy the file into the respective home directory of these accounts.

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. Prior versions of PostgreSQL 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 doesn't make a distinction between a user and a group. You can easily morph a user into a group 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 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 these are 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 it also deviates, 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.

Table D.1 lists general user database concepts and their equivalent in PostgreSQL.

Table D.1 PostgreSQL role concepts and parallels to other databases

General concept	PostgreSQL equivalent
User (login)	A role with login rights and generally contains no member roles.
Group	A role with member roles (usually no login rights).
Database user	A 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've worked on arbitrarily implements the ones they prefer in this regard, to the point of relying on any of the role/privilege-based tables in information_schema is not very portable between database management systems.

Rights management

PostgreSQL roles can contain and be contained by many other roles. In other words, each role/user/group can have many parents or belong to many groups. Roles in PostgreSQL don't necessarily inherit rights from their parent roles, which is a cause of confusion for many people. We'll go over this shortly.

CORE SERVER RIGHTS

A role can have a couple of core rights that 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 won't by default be able to do these things even if you mark them as inheriting from their roles. To relinquish these, prefix them with NO, such as NOSUPERUSER, NOINHERIT.

- *SUPERUSER*—Has super powers (to relinquish superuser rights, use NOSUPERUSER).
- *INHERIT*—When marked as INHERIT, the role inherits the rights of its parent roles. In later versions of PostgreSQL, this is the default behavior.
- *CREATEDB*—This gives a role the ability to create a database.
- *CREATEROLE*—This gives a role the ability to create other roles that are not SUPERUSER roles and that it's not a member of. Only a superuser can create other superusers.

- *LOGIN*—This gives the role rights to log in. Generally speaking, people create group roles by not giving the group role rights to log in, though in theory you can have a group role that has rights to log in. In practice, having group roles that can log in is confusing, so pgAdmin 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 superpowers always has superpowers. This is never the case, as mentioned previously, because SUPERUSER rights and the like are never inheritable. It's 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 SUPERUSER role but not cause damage casually. How? When the boss demands, "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. We'll demonstrate this superuser without superuser powers with this simple exercise:

```
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;
```

Here we have a simple script that creates a group called office_of_president and two users, Leo and Regina. Leo has SUPERUSER rights, and Regina is a member of a group that has SUPERUSER rights. Leo is always omnipotent. Regina is only omnipotent when she summons her powers of omnipotence. We'll 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's frustrated. She's a member of the mighty role of office_of_president and she's 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;
```

Now 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 Regina is always omnipotent without the need to summon superpowers.

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 the fact, as we mentioned earlier, 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 idiosyncrasy dumbfounds people because they 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 would anyone ever create a user that doesn't inherit rights of its membership groups?

One reason is for testing. Let's imagine that you create a user that's a member of every single group role under the sun, but that user (login role) doesn't inherit rights from any role it's a member of. What can this user do? It can for a specific session, promote itself to have rights of any role it's a member of, much like Regina promoted herself to the rights of her powerful group. This can be useful for testing different membership rights or for giving a user only certain rights within an application. As we also demonstrated earlier, this prevents you from doing superuser damage without trying to deliberately do superuser damage.

SESSION AUTHORIZATION

Session authorization is similar to SET ROLE but the distinction is that in SET ROLE you summon your powers as a member of a role, while with SET SESSION AUTHORIZATION you become that role. Basically you're impersonating another user. Only a superuser can impersonate another user, but any member of a role can do a SET ROLE to the roles of which they're a member. Impersonation is useful when you're creating a bunch of objects that you want to be owned by a specific user without having to change owner to that person for each creation. Another example would be when you want to run commands but limit yourself 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 or the owner of an object can grant rights to others, and in addition to granting rights to objects, they can give others the right to grant rights to objects using WITH GRANT OPTION. GRANT, as you 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 the user can add or remove users to or from that role.

PostgreSQL 8.4 column-level permissions

PostgreSQL 8.4 introduced column-level permissions, allowing granting read/write/update permissions to individual columns of a table, largely thanks to the work of Stephen Frost, who is also a longtime contributor of the PostGIS project (TIGER geocoder).

In the following exercise, we list the common GRANT usages.

Listing D.2 Common GRANT options

```
GRANT ALL PRIVILEGES
  ON DATABASE postgis_in_action to leo
  WITH GRANT OPTION;

GRANT ALL PRIVILEGES ON SCHEMA world to leo;
GRANT SELECT, INSERT ON geometry_columns TO leo;
GRANT SELECT ON spatial_ref_sys TO public;
GRANT UPDATE(proj4text,srtext)
   ON spatial_ref_sys TO leo;
```

1 Right to connect and create objects

2 Right to create objects in world

3 Right to view and insert

4 Right for all to view

5 Right to update specific columns

1 We grant all rights to Leo for our database and also allow him to give grant rights to whomever he chooses, but this only means that Leo can connect to the database and create new schemas. It doesn't give him the right to view existing tables, for example, or to create objects in schemas for which he doesn't have rights. 2 We give Leo all rights to the world schema. This doesn't allow him to view or edit existing data, but it does allow him to create new objects in world. 3 We give Leo the right to view and add data in geometry_columns but not to update or delete. 4 We give everyone the right to view data in spatial_ref_sys and 5 Leo the right to update the proj4text and srtext columns in spatial_ref_sys (works only for PostgreSQL 8.4+).

As you can see, the process of granting rights in PostgreSQL 8.4 or below is somewhat annoying. It's annoying in the sense that you often want to grant rights to a whole database or schema, and there's no one-liner in PostgreSQL for doing such a thing. To get around this annoyance, you 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-andDCL-Making-structural-and-Permission-changes-to-multiple-tables.html.
- Use the pgAdmin III Grant Wizard, which allows you to select a list of objects and grant rights to specified roles. To use the Grant Wizard, follow these steps:
  - Select a schema.
  - Select Tools > Grant Wizard.
  - Select the objects you want, privileges, and roles.

PostgreSQL 9.0 enhancements to GRANT and REVOKE

PostgreSQL 9.0 introduced enhancements to the GRANT and REVOKE feature that allow you to GRANT ALL to all tables or functions and the like 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/.

If you're using PostgreSQL 9.0 or above, life is much simpler as far as rights management is concerned.

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's granted with the GRANT command. We'll demonstrate our favorite REVOKE command, revoking connection rights from the Public group. As we mentioned, the Public group is the group that 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. This isn't always desirable. To prevent this, you can run the following command:

```
REVOKE CONNECT ON DATABASE postgis_in_action FROM public;
```

Now that we've covered the basics of security, we'll cover something perhaps even more important, backup and restore.

Backup and restore

PostgreSQL has perhaps the richest backup and restore tools of any open source database, and they rival and often surpass those offered by the commercial relational database systems. Backup is accomplished with the following commands:

- *pg_dump*—This 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*—This 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 it doesn't allow the selective restore that's possible 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*—This is 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-tune what you would like to restore from the backup.

- *psql*—This is used for restoring or running an SQL file such as those generated by pg_dumpall or pg_dump when SQL mode is chosen or by the 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 back up a database with PostgreSQL 8.4+ pg_dump, then you can specify jobs=2 or more. This does a parallel restore. Depending on your disk IO and CPU, setting this can halve or reduce even more the time of a restore. For example, a restore of a PostGIS 800 GB database would take about 12 hours in prior versions and only 6 hours or less in 8.4.

Backup

The pg_dump command-line tool packaged with PostgreSQL is our preferred tool for database backups. The main reason we prefer it is that it creates a nice compressed backup and allows for selective restore of objects. Most of the time a restore is needed 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, as shown in the following listing.

Listing D.3 Common backup statements

```
pg_dump -i -h localhost -p 5432 -U someuser
➥ -F c -b -v -f  "/pgbak/somedb.backup" somedb
pg_dump -i -h someserver -p 5432 -U someuser -E latin1
➥ -F c -b -v -f "/pgbak/somedb.backup" somedb
pg_dump -i -h someserver -p 5432 -U postgres
➥ -F p -o -v -n pgagent  -f "C:/pgagent.sql" postgres
pg_dumpall -i -h someserver -p 5432 -U someuser -c -o
➥ -f "/pgbak/alldbs.sql"
pg_dumpall -h localhost -p 5432 -U postgres
➥ --globals-only > /pgbak/globals.sql
pg_dump -h localhost -p 5432 -U postgres
➥ -F c -b -v  -f "/pgbak/work_poi.backup" -t "work.poi" somedb
```

1 Compressed database backup
2 Compressed Latin encoded backup
3 Plain-text schema backup
4 Plain-text all databases
5 Plain-text roles and tablespaces
6 Single table compressed backup

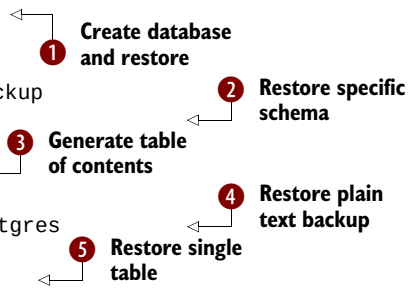
1 Dump the database in compressed format; include blob and show verbose progress (-v). 2 Dump the database in Latin1 encoding, which is useful if you want to restore a database but want to use a different encoding in the new database. 3 Back up pgagent schema or any schema of postgres DB in plain-text copy format, and maintain oids. 4 Dump all databases—note that pg_dumpall can only output to plain text. 5 Back up users/roles and tablespaces. 6 Back up a single table in compressed format.

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. In this section, we'll demonstrate some common examples, as shown in the following listing.

Listing D.4 Common restore statements

```
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
pg_restore --schema=us --dbname=somedb
➥ -U postgres /pgbak/somedb.backup
pg_restore --list /pgbak/somedb.backup
➥ --file=/pgbak/somedb_list.txt
psql -h localhost -p 5432 -U postgres -d postgres
➥ -f /pgbak/globals.sql
pg_restore -h localhost -p 5432 -U postgres
➥ -t "work.poi" /pgbak/somedb.backup
```



❶ Create a new database and restore the backup file to this new database using two threads for restore. (Remember, jobs works only for PostgreSQL 8.4+.) ❷ Restore only a specific schema, in this case the us schema. ❸ Generate a table of contents for a backup file and store it in the file somedb_list.txt. ❹ Restore user accounts and custom table spaces. You can specify any SQL file here, such as the one we created to back up all databases. ❺ Restore a single table from backup. In this case, we're restoring the table poi in the schema work.

In the next section we'll provide some tips for automating the backup process.

Setting up automated jobs for backup

There are two common ways for automating backups for PostgreSQL, and they vary slightly depending on your OS:

- Use an OS-specific scheduling agent such as cronjob in Unix/Linux or Windows Scheduler in Windows.
- Use pgAgent, a free scheduling agent for PostgreSQL manageable from pgAdmin III.

We prefer the pgAgent way because it's cross platform allows us to manage the same way we manage and view other parts of PostgreSQL (via the pgAdmin III tool), and is also designed for running SQL jobs. On the downside, it's sometimes more finicky to set up. We describe the details of the setup and also how to define a backup script for Windows and Linux at http://www.postgresonline.com/journal/index.php?/archives/19-Setting-up-PgAgent-and-Doing-Scheduled-Backups.html.

Note that because the backup scripts use only shell commands of the respective Unix/Linux/Windows environment, you can also run them with your scheduling agent of choice.

Data structures and objects

PostgreSQL, like many sophisticated relational databases, has a rich collection of objects to accomplish different tasks. In addition to database objects, it has built-in data types, many of which you'll find in other relational databases, as well as some data types that are unique to it. If that isn't enough, PostgreSQL allows you to extend the system and 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.

PostgreSQL objects

When we speak of *objects*, we're 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 part of the core makeup of PostgreSQL. They are tables, views, schemas, and so on. Some of these you've already been exposed to. In the following list, we give a brief synopsis of the core PostgreSQL objects and their function:

- *Server service/daemon*—This is PostgreSQL itself that houses everything.
- *Tablespaces*—These are physical locations of data that map to a named location in the server. When you run out of disk space, objects can be moved to different locations on disks by moving them to a different tablespace. You can define the default for these by setting the Global User Control (GUC) variables default_tablespace and temp_tablespace. These 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's also fast and easy 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 filesystem, and database data in any tablespace is always stored in a folder, such as <tablespace path>/<databaseoid>/objectoid.
- *Schemas*—These are the logical location of tables, views, functions. They have 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 the default at the function level via the search_path configuration. This allows you to maintain a logical separation without having to schema qualify commonly used schemas. Think of a schema 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 will be chosen.
- *Roles*—These include users and groups. They sit at the database level and are granted rights to objects in a database.

- *Rules*—Rules rewrite SELECT, INSERT, and 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 windows to the real data and allow you to see summaries or a subset of data by selecting from an abstracted virtual table. A view generally consists of only an INSTEAD OF SELECT rule, which is a rule that defines the SELECT statement of the view. An updateable view will also have rules on the UPDATE, INSERT, and DELETE actions of a view.
- *Triggers*—Triggers are actions that are performed when data changes. They are often used to update additional data. A common example in PostGIS would be if you have an application that updates a lon lat field and uses 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 comprise other data types. Table columns are composed of things with the same data type. The rows of a table itself are implemented in PostgreSQL as a composite data type. You'll 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 distinctive among relational databases 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 that expects a different data type, the data will be automatically cast for you if there's an unambiguous autocast type. Take care when doing this. Because of the overloading features of PostgreSQL, it's possible to have two functions with the same name but different data types. In this case, if an object that has an autocast for both is used without an explicit casting, you'll 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 shorthand `mybox::geometry`.
- *Operators*—These are things like =, >, <, and again PostgreSQL allows you to define custom operator behavior for your custom types. Some operators have special meaning, such as = > <, that are used by internal SQL querying to define ORDER BY, GROUP BY DISTINCT ordering. These are useful to override if you're building custom data types and want them to sort in a certain way. As you saw in earlier chapters, PostGIS overrides = to order by the bounding box of geometry/geographies.
- *Functions*—These are functions you can use within an SQL statement. PostgreSQL comes with a lot of built-in and contributed ones such as the soundex you saw earlier and those provided by PostGIS. You can also build your own. Functions can return simple data types, sets, or arrays. There are three 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've worked with Oracle, then a sequence object will be very familiar to you. It's 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, which is tied to a specific table. SQL Server 2011 introduced support for sequences as well, which follow the same ANSI SQL standard as Oracle and PostgreSQL. If you wanted a sequence to be tied to a specific table in PostgreSQL, then you'd create the column as serial or serial8, which behind the scenes will create a sequence object and set the default of the column to the next value of the sequence.

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:

- *int4, int8*—These go by more familiar names such as int, integer, and bigint.
- *float, double precision*—These are 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's not a true type. It's shorthand for "give me an integer with a sequence object to increment it." It's still an integer. The parallel in MySQL would be marking the column as an AUTO_INCREMENT or in SQL Server setting the Identity property to Yes.
- *numeric*—This has a scale and precision and is named the same in other relational databases. It's also often referred to as decimal in other databases.
- *varchar, text*—This means character varying. Unlike most other relational databases, PostgreSQL doesn't put a limit on the maximum length of a varchar or a text variable. Varchar and text behave much the same, except that text has no maximum limit and varchar may or many not have a specified maximum limit. Some other databases decide on storage handling based on the specified size of a field. For example, in SQL Server if text is noted, then a pointer to the text field is stored and data is stored elsewhere outside the table. The closest parallel in SQL Server is the varchar(MAX) option. PostgreSQL doesn't care about this and bases storage considerations on the size of data actually stored in the field. Only if a field goes beyond its allotted storage size is storage relegated to toast tables. This means, as many PostgreSQL people will argue, that there's no penalty for using text over varchar with a limit. But if you care about interoperability, we argue that there's a big penalty. For exporting purposes such as tab delimited and so forth, it's important to have a limit on the size of a field, and if you export to another system often, you want your size limits to mirror those of the other side. For those using autogenerated screens with screen painters,

such applications refer to the system tables to determine the width for fields on the screen, and if everything is text, you end up with big text boxes everywhere. If you use an ODBC driver, for example in MS Access, a varchar is treated very differently from a text. It will allow you to sort by varchar but not by text.

- *char*—These are padded characters. If you say it's a char(8), then the field will always be of length 8. This is the same in almost all relational databases. This is more a presentation feature than a storage consideration in PostgreSQL because PostgreSQL presents padded eight characters but doesn't actually store eight characters if the text is shorter. Most other relational databases store eight characters.
- *date*—This is a date without time. MySQL has this, Oracle has this, and SQL Server 2008+ has this (in prior versions of SQL Server you couldn't have a date without time).
- *timestamp, timestamp with timezone*—Again, these are very similar in other relational databases although they may be called datetime or some 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. As far as we know, only Oracle and IBM DB2 have them. 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 it in functions as well. Arrays play an important role in building aggregate functions, because 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 an array. A typed row is a row in a table or a specific type. You can cast compatible rows to compatible types, as we'll demonstrate shortly.

Anatomy of a database function

Stored functions and procedures are useful for compartmentalizing reusable nuggets of functionality and embedding them in SQL statements. Unlike most relational databases, PostgreSQL (even as of PostgreSQL 9.0) doesn't make a distinction between a stored procedure and a 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 that of any relational database system you'll find, both commercial and open source. Common favorites are sql, plpgsql, and 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) and pl/proxy (designed by Skype Corporation and freely provided and that's designed to replicate commands between PostgreSQL servers).

The only languages preinstalled in all PostgreSQL databases are SQL and C. PostgreSQL allows you to bind a C function in a C library to a stored function wrapper so that it can be used in an SQL statement. Most PostGIS functions are C functions. PL/PgSQL isn't always installed by default in versions of PostgreSQL prior to 8.4 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*—This dictates the return of the function; for functions that don't return anything, it's void.
- *The body*—This 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 the form *$somename$*. Oftentimes people leave out the *somename* so it reduces down to $$ *body goes here* $$. This works for all languages. Prior versions required quoting with a single quote mark ('), 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*—This is always LANGUAGE '*somelanguage*'.
- *For PostgreSQL 8.3+ the ROWS expected and COST as a function of CPU cycles*
- *Cachability*—Designated as IMMUTABLE, STABLE, or VOLATILE, this allows PostgreSQL to know under what conditions the results can be cached. IMMUTABLE means with the same inputs you can always expect the same output. STABLE within the same query means that you can expect the same inputs to result in the 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, 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 superuser 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 contain only a body and no name and can be run straight from the command line. It currently supports only plpgsql, plpython, and plperl.

Next we'll demonstrate how to use these PostgreSQL objects.

Defining custom data types

Defining custom data types is fairly simple in PostgreSQL. As we mentioned earlier, when you create a new table, you create a new data type as well. The next listing is a simple example of a data type we'll call vertex that contains x and y attributes. We then create instances of it and then pull out just one of its attributes.

Listing D.5 Create a simple type and use it

```
CREATE TYPE vertex AS
   (x double precision,
    y double precision);
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;

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;
```

1 Create type

2 Convert row to type

3 Get element of typed object

In 1 we define a new type called vertex that has an x attribute 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'll recognize that 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.

Creating tables and views

Creating tables and views is done just like in any other relational database. The following listing shows some simple examples that create a table and view in the assets schema.

Listing D.6 Creating a table and a view

```
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);

CREATE VIEW assets.vwpoi_active AS
    SELECT poi_gid, the_geog, poi_name,is_active
      FROM assets.poi
      WHERE is_active = true;
DROP TABLE assets.poi CASCADE;
```

1 Create table with geography field

2 Create view against table

3 Drop table

In 1 we create a table with a geography field (requires PostGIS 1.5+) that is of type POINT and WGS 84 lon lat, with an autoincrement primary key call poi_gid and an

active flag that defaults to true for new entries. In ❷ we create a view against this new table that will list only active records. In ❸ we drop the table and include the CASCADE command, which will drop all dependent objects such as the view we created in ❷. When using CASCADE, proceed with caution because you could be dropping a lot of dependent objects. Without the CASCADE we'd be informed that assets.vwpoi_active depends on assets.poi and thus can't be dropped. We'd then have to drop the view first and then the table.

Now that we've covered the basic features of PostgreSQL, we'll get into greater detail about functions and rules.

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 the PL/SQL supported by IBM DB2 and MySQL in that the PostgreSQL SQL function language has no support for procedural control structures. What other databases call PL/SQL is closer in family to PostgreSQL's PL/PgSQL.

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 this is that an SQL function can be treated like any other SQL statement and optimized by the SQL planner. In many cases very useful pieces of reusable code can be compartmentalized in such a simple structure.

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 embed its definition in the query. Essentially, it treats it like a macro similar to the way C macros are expanded where they're 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 subexpression within a query, then the planner can collapse the expression. A common example is the && operator, which is used in many PostGIS functions. If you use two functions with &&, the planner will see && and &&, and it will collapse the two into a single &&.

As a general rule of thumb, here's when to use an SQL function:

- When you use constructs that could benefit from an index.
- When logic is fairly simple and short.
- In a rule; you can only write rules with SQL. Rules aren't really functions, but they serve a similar purpose.

There's one situation where you absolutely can't use SQL to write a function even if you wanted to, and that's for a trigger function. This may change in later versions of PostgreSQL, but as of PostgreSQL 9.0, you can't write triggers in the SQL language.

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's true that 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've demonstrated throughout this book.

This makes them fairly limited but easy to fold into a larger SQL statement. The other disadvantage 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.

The next listing is a trivial function that returns a square of numbers starting with the first and ending with the last.

Listing D.7 Example SQL function returns square

```
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;
SELECT i, fnsquare(i,i + 3) As squared_range
 FROM generate_series(1,3) As i;

SELECT *
   FROM fnquare(1,10) As foo;
```

1 Define set returning function

2 Use function in SELECT

3 Use function in FROM

In 1 we define our function that takes a range and 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 written in SQL or C. For PostgreSQL 8.4+, you can do this with PL/PgSQL and other functions as well. 3 This shows the standard way for calling SET-returning functions.

Creating rules

Rules are objects that are bound to tables or views. They are often used in place of triggers, and for views in PostgreSQL 9.0 and below, you can only use rules. Rules don't perform any action but help in rewriting SQL statements to do something in addition to 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 behind the scenes changes it to something that has a SELECT rule. If you were ever nosy enough to inspect your view, you'd 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;
```

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 you make a view updateable? You create an UPDATE rule that rewrites the update to update the raw tables, as shown here. Note that PostgreSQL 9.1+ supports binding insert/update/delete triggers to views, and using triggers for update/insert/delete is generally the preferred way for making views updateable in PostgreSQL 9.1+.

Listing D.8 Making a view updateable

```
CREATE RULE updvwpoi_active AS                          <-- Update rule
    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;
      );
CREATE RULE insvwpoi_active AS                          <-- Insert rule
  ON INSERT TO assets.vwpoi_active
    DO INSTEAD (
      INSERT INTO poi(poi_name, the_geog)
         VALUES(NEW.poi_name, NEW.the_geog)
    );
CREATE RULE delvwpoi_active AS                          <-- Delete rule
  ON DELETE TO assets.vwpoi_active
    DO INSTEAD (
      DELETE FROM  poi WHERE poi.poi_gid = OLD.poi_gid);
```

In a rule or trigger are two records called NEW and OLD. NEW exists when there's an insert or update to an object. OLD exists when there's an UPDATE or DELETE to an

object. In the examples in listing D.8 we make our view updateable by pushing updates to the base tables the view is based on.

In PostgreSQL 9.1 triggers can be bound to views to update, insert, or delete data and can be used instead of rules for these events.

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, even with a language as simple as SQL. This is one of the coolest features of PostgreSQL. Part of the power of doing this is because of the malleability of the 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, listing D.9 shows an example that simulates (but we think better), the MS Access First and Last aggregate functions. It's excerpted from one of our articles titled "Who's on first and who's on last," available at http://www.postgresonline.com/journal/index.php?/archives/68-More-Aggregate-Fun-Whos-on-First-and-Whos-on-Last.html.

Listing D.9 Creating first and last aggregate functions

```
CREATE OR REPLACE FUNCTION first_element_state(
      anyarray, anyelement)
  RETURNS anyarray AS
$$
    SELECT CASE WHEN array_upper($1,1) IS NULL
     THEN array_append($1,$2) ELSE $1 END;
$$
  LANGUAGE 'sql' IMMUTABLE;

CREATE OR REPLACE FUNCTION first_element(anyarray)
  RETURNS anyelement AS
$$  SELECT ($1)[1] ;$$  LANGUAGE 'sql' IMMUTABLE;

CREATE OR REPLACE FUNCTION last_element(
     anyelement, anyelement)
  RETURNS anyelement AS
$$  SELECT $2;  $$ LANGUAGE 'sql' IMMUTABLE;

CREATE AGGREGATE first(anyelement) (
  SFUNC=first_element_state,STYPE=anyarray,
  FINALFUNC=first_element);

CREATE AGGREGATE last(anyelement) (
  SFUNC=last_element,STYPE=anyelement);
```

1 State function

2 FINALFUNC function

3 State function

4 Aggregate for first

5 Aggregate for last

As you can see in listing D.9, an aggregate function is composed of at least one state function (SFUNC) 1 3 and one state type (SType). The FINALFUNC 2 is sometimes present and is needed if the result of each subsequent state is not enough or the data type 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 in the next listing we take it for a test drive.

Listing D.10 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;
```

We put our functions to work with a simple query. This example and the creation of aggregates work in most versions of PostgreSQL, even back to 8.1.

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, employ other control flow such as FOR and WHILE loops, cursors, RAISE errors, and so on and also write SQL. Unlike the pure SQL language, it's 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 whether 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.

When to use PL/PgSQL functions

PL/PgSQL is desirable for functions where using an outer index gives no benefit, for example, when the values that go into the function are already filtered by a where condition, or when very fine-grained step-by-step control is needed.

As a general rule of thumb, here's when to use a PL/PgSQL function:

- No construct could benefit from an outer index check.
- Logic is complex and needs several breaks, or you need variables or the ability to raise errors.
- In a trigger. You can't use SQL in a trigger. Although you can use other languages such as PL/Python, PL/R, or PL/Perl for writing triggers, PL/PgSQL tends to be more stable and also has more integration with PostgreSQL. Therefore, PL/PgSQL is generally a better language for writing triggers unless you need to leverage specific functionality only offered in the other languages.

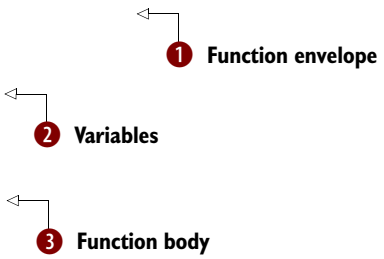
As mentioned earlier, you can't write rules with PL/PgSQL, and in earlier versions of PostgreSQL (pre 8.4), you can't use a set-returning PL/PgSQL function in the SELECT clause of a statement, whereas you can with an SQL function.

Creating a PL/PgSQL function

The following listing is a simple PL/PgSQL function. This is the utmzone function we've used often in the book, and it offers a good example of when to use a PL/PgSQL function. Let's study its parts.

Listing D.11 utmzone

```
CREATE OR REPLACE FUNCTION utmzone(geometry)
   RETURNS integer AS
$$
DECLARE
    geomgeog geometry;
    zone int;
    pref int;
BEGIN
    geomgeog:= ST_Transform($1,4326);
    IF (ST_Y(geomgeog))>0 THEN
       pref:=32600;
    ELSE
       pref:=32700;
    END IF;
    zone:=floor((ST_X(geomgeog)+180)/6)+1;
    RETURN zone+pref;
END;
$$ LANGUAGE 'plpgsql' IMMUTABLE
  COST 100;
```



❶ The first part of a PL/PgSQL function, like any function, is the envelope, which defines the parameters that go into the function and the return type. ❷ Then there is the DECLARE, which is part of the body, the place where we declare the variables we'll use through the rest of the function. SQL functions don't have this, though other languages may but specify it differently. ❸ Then comes the meat of the function, which is encapsulated between 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 section uses FOR loops, while the RETURN statement may contain a RETURN NEXT loop when returning a set. For 8.3+ there's also RETURN QUERY, which allows returning results of precompiled SQL, and 8.4 introduced RETURN QUERY EXECUTE, which allows returning results of dynamic SQL. An 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. These newer constructs are more efficient and shorter to write than the older RETURN NEXT. But if you need finer grained control of which records you return, RETURN NEXT will still be needed. The pros and cons are discussed in Andrew Dunstan's "Experiments in Efficiency" at http://people.planetpostgresql.org/andrew/index.php?/archives/131-Experiments-in-efficiency.html.

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're reusing the same trigger across multiple tables, TG_TABLE_NAME and TG_TABLE_SCHEMA are useful 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't be written using SQL; they must be written in PL/PgSQL 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. The following list shows kinds of triggers and what data is available to each. A trigger is either a row-level or statement-level trigger and is triggered on the UPDATE/INSERT/DELETE event or a 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. It's not 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 affect the insert. A common mistake is trying to set values of NEW.. in the AFTER INSERT trigger.
- *UPDATE row-level trigger*—You can think of an update as a delete followed by an insert. Therefore the UPDATE trigger has both OLD and NEW variables 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 for AFTER, your NEW changes go into a black hole when the trigger is completed.
- *DELETE*—Just the OLD object is available.

Listing D.12 shows an example trigger that will update a geography column whenever a longitude and latitude are updated or a new record is added. It will also log changes to a log table. Keep in mind that you can do other useful things such as geocode records when address information is updated. 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. The downside is that you can't just write the trigger function as part of the table definition as you can in some other databases.

Listing D.12 Trigger function applied to geography table inPL/PgSQL

```
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);

CREATE OR REPLACE FUNCTION trig_set_thegeog_pt()
  RETURNS trigger AS $$
DECLARE
changed boolean := false;
oldgeog geography := NULL;
BEGIN
   IF tg_op = 'INSERT' AND NEW.longitude IS NOT NULL
      AND NEW.latitude IS NOT NULL THEN
     changed = true;
   ELSIF COALESCE(NEW.longitude, -1000)
         != COALESCE(old.longitude, -1000)
  OR COALESCE(NEW.latitude, -1000)
      != COALESCE(old.latitude, -1000) THEN
        changed = true;
   END IF;
   IF changed THEN
      IF NEW.longitude IS NOT NULL AND NEW.latitude IS NOT NULL THEN
        NEW.the_geog :=
            ST_GeographyFromText('SRID=4326;POINT('
           || NEW.longitude || ' ' || NEW.latitude || ')');
  ELSE
    NEW.the_geog = NULL;
  END IF;
  INSERT INTO poi_log(logtype, geogtable, geog_gid,
        old_geog, new_geog)
    VALUES(TG_OP, TG_TABLE_NAME, NEW.gid, oldgeog, NEW.the_geog);
   END IF;
   RETURN NEW;
END;
$$
LANGUAGE 'plpgsql' VOLATILE;

CREATE TRIGGER step01_trigupdpt
  BEFORE INSERT OR UPDATE
  ON poi
  FOR EACH ROW
  EXECUTE PROCEDURE trig_set_thegeog_pt();

INSERT INTO poi(poi_name, longitude, latitude)
  VALUES('My back yard', -72.1234, 41.3456);
```

1 Table

2 Trigger function

3 Insert conditional code

4 Update conditional code

5 Log change

6 Bind trigger to table events

7 Test trigger

```
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;
```

❶ We create a test table that we'll later apply our trigger to. ❷ 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. ❸ We check to see what kind of event caused the trigger to fire; if it's an insert we know we need to update if the longitude and latitude values are not NULL. ❹ If it's an update, then we need to either update the geography column or wipe out the contents. We use COALESCE here to set NULLs to -1000 so as to never compare NULLs. NULLs are tricky to compare because even when two NULLs are compared, the comparison returns false. To shorten our code, we write the COALESCE hack. ❺ 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, because we have only one trigger, it's simpler to combine into our before event with the assumption that represents the final data. ❻ 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, it's advantageous to categorize your triggers by functionality rather than writing 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_..., and so on. Each subsequent BEFORE trigger will see the change of the previous if the previous makes sure to return NEW; otherwise trigger execution stops. ❼ Then we test our trigger to make sure it's 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.

This example just scratches the surface of what you can do with triggers in PostgreSQL in general and PL/PgSQL 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. We hope that we've demonstrated enough here for you to envision the potential it holds.

PostgreSQL 9.0 introduced ANSI SQL column-level triggers, which allow specifying a WHEN condition that can contain column names and will be executed only when the WHEN condition evaluates to true. This feature saves a bit of processing time because the body of the trigger doesn't always need to be checked.

We'll next cover some key elements for achieving optimal performance for your PostgreSQL database.

Performance

In chapter 9 we talked a bit about performance. Now, we'll cover some of the loose ends we left out of that chapter.

Index

Just like in other databases, PostgreSQL uses indexes to improve performance. There are various flavors of indexes to 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 listed here:

- *Partial indexes*—These are indexes with a where clause where the WHERE constrains the data that's actually indexed.
- *Functional indexes*—You can choose to index a function calculation such as UPPER, LOWER, SOUNDEX, or ST_Transform, 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 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*—These include B-tree, Gist, GIN, and Hash. The most common is the B-tree index. The kind of index controls how the index leaves are structured.

B-TREE INDEX GOTCHAS

A B-tree index is the most commonly used of all index types in PostgreSQL. Under certain conditions where you'd expect it to be used, it's not. These conditions are probably very unintuitive to people coming from databases that are not necessarily case sensitive.

- *Trying to do a functional compare for example, UPPER/LOWER*—Some people think that when you do a query upper(item_name) = 'DETROIT', then that condition should be able to use a B-tree 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) doesn't match what is indexed. You need to change the index to

  ```
  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) LIKE 'D%'
  ```

  then you can't use an unqualified B-tree. You need one with varchar_pattern_ops. Prior to PostgreSQL 8.4 a varchar_pattern_ops was not able to service an equality like the previous one. To handle both, you needed 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 (sometimes called expression indexes) are indexes that are built from 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 don't change given the same arguments. The main reason for that is that once an index is calculated, the index value is changed only 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 PostgreSQL, even as of 8.4, will not try to validate whether it demands 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 the function is immutable.

The other gotcha with functional indexes is that if you redefine a function to do something other than what it was doing before, namely changing the output value, PostgreSQL will not go back and reindex the affected tables. So if you change a function used in an index and you know that change will affect output, you need to go back and reindex the affected tables.

It's fairly easy to determine which tables are affected, particularly in pgAdmin, using the dependents tab of a function.

Summary

In this appendix, we've given you a foretaste of the uniqueness and versatility of PostgreSQL that makes it stand apart from the other databases you may be familiar with. We've also demonstrated its less nice and cumbersome features such as the rights management in pre-PostgreSQL 9.0, which has been much alleviated in PostgreSQL 9.0. 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.

index

Symbols

&& operator 140
&< operator 141
&<| 141
&> operator 141
~= operator 141
$HWNPARENT/bin folder 319

Numerics

3D geometry 33, 51–52, 146, 238
3D model 155
7-Zip 181, 281
64-bit windows 424

A

abline 295
abstract class 57
ad-hoc SQL 176
AddGeometryColumn function
  creating new geometry point column with 40
  in adding linestrings example 40
  in compound curve example 50
  in curved polygon example 51
  in data breakdown example 65
  in data conversion example 184
  in data storage examples 23, 27
AddRasterColumn function 385
addy 283
aerial imagery 376
AFTER DELETE event 73
aggregate functions 55, 443
aggregation 227
aliasing 433–434
ALTER DATABASE statement 265
ALTER FUNCTION statement 267
ANSI SQL 11, 29, 137, 430
Apache 323
Application Stack Builder 419
Applied Spatial Data Analysis with R (Bivand, Pebesma, Gómez-Rubio) 292
apt-get command 421
Arc de Triomphe 63
ARC Digitized Raster Graphics driver 299
Arc/Info ASCII Grid driver 299
Arc/Info Binary Grid driver 299
ArcGIS 13, 162, 366
ArcGIS ArcPad 366
ArcGIS desktop 318
ArcGIS IMS server 327
ArcGIS SDE 175
ArcIMS web services 351, 366
arcs 48
ArcSDE (Spatial Database Engine) 353, 357
ArcView image file 380
areal polygons 124
array_to_string 244
arrondissements 60–62
ASCII data 20
ascii grid 379
ASP.NET 313, 341
associate arrays 62
AutoCAD 187, 370
Autocasting 83, 146, 237
available.packages() command 297
Azimuthal Projection 160

B

bands 373
base layer 61, 332
Beanshell 346
BeanTools 353
BEFORE INSERT trigger 74
bezier curves 49
BIL (ArcView image file) 380
Bing 212, 309, 327
bisections 132–133, 231
bitmap scans 241
boundaries 102, 123–124, 152
bounding box 139–141, 150
bowtie 149
box2D 100, 141
box3D 100
Boxes. *See* bounding box
breaking linestrings 223
Btree indexes 250
buckets 232
buffer zones 6
buffering 206

buffers 30–31, 166
bulk loads 67, 71
bunching 273
bytea_output setting 83

C

C language 70
c() function 299
C# language 323
C# MapScript 315
Cadcorp SIS 318
cartesian 17, 35, 39–40, 209
Cartesian coordinate system 96
cartography 153
CASE statement 262
CAST 146
cells 236
Census, U.S. 280
center of gravity 103
CentOS 420
centroids 103, 238
CGI (Common Gateway Interface) 315
ChangeProperly 128
channel. *See* bands
character_maximum_length field 431
Cheeseshop package repository 307
child table 59, 67
chooseCRANmirror() 297
circularstring 48
Clarke 1866 ellipsoid 157, 159
clipping 121
closed linestrings 40
closed rings 43
closest objects. *See* nearest neighbor search
closest point 41, 205
CLUSTER 274
CLUSTER ON 273
clustered indexes 273
clustering 273
colinearity 270
collection geometries 52
column_default field 432
column_name field 431
columns 430
comma separated data 21–22
Common Gateway Interface (CGI) 315
Common Table Expression (CTE) 263
commutative relationships 119
companion relationships 125
composition 108
compound indexes 254
compoundcurves 50
Comprehensive R Archive Network (CRAN) 297
Conic Projection 160
constraint_exclusion 58, 60, 78, 266
constraints 37, 60–61
constructors 81–83, 115
Contains relationship 125
Contextual Query Language (CQL) 364
coord_dimension 34
coordinate dimensions 34–35, 91
coordinate reference system (CRS) 158
coordinate rounding. *See* ST_SnapToGrid
coordinates 101
COPY command 22
correlated subqueries 258, 435
COST setting 268
COUNT function 63
COUNT(DISTINCT) construct 29
coverage, defined 372
COVEREDBY construct 127
covering indexes 254
Covers 127–128
CRAN (Comprehensive R Archive Network) 297
CREATE DATABASE statement 423
CREATE LANGUAGE statement 290
CREATE OR REPLACE FUNCTION statement 306
CREATE PROCEDURAL LANGUAGE statement 306
CREATE TABLE AS statement 449
CREATE TABLE statement 55
CREATE TRIGGER statement 76
Creative Commons Attribution-ShareAlike 2.0 179
CROSS APPLY clause 434
CROSS JOIN clause 207, 234, 436
crosses 130
CRS (coordinate reference system) 158
CSV format 350
CTE (Common Table Expression) 263
curved geometry 47, 119, 146, 364
curvepolygons 50–51
Custom Query Language (CQL) 365
cuts 231
cylindrical projections 160

D

daemons 265
data integrity 54
Data Manipulation Language (DML) 432, 446
data_type field 431
database abstraction layer 343
database design 53
database planner 242
DataSF.org 204
datum 158
DB2 348
DBF (dBase) 174
DBM 316
DBMS (Database Management System) 432
DE-9IM (Dimensionally Extended 9-Intersection Matrix) 147–152
Debian distro 421
declarative language 241
decomposition 99
Deegree 134
DELETE statement 450
DEM (Digital Elevation Model) 376
demo() command 297
design process 53
desktop Linux 419
dev.off() function 295
Dimensionally Extended 9-Intersection Matrix (DE-9IM) 147–152
dimensions 35, 124
direction 160
disinheritance 58
disjoint relationship 131, 148
Distance_Spheroid 211
distance. *See also* ST_Distance 206
DISTINCT 67, 136, 144, 443
DISTINCT ON clause 136–137
distros 421

DML (Data Manipulation Language) 432, 446
DO INSTEAD rule 71, 73
domain language 291
Douglas-Peuker algorithm 114
driving directions 309
DropGeometryColumn function 27, 385
DropGeometryTable function 37, 385
DropRasterColumn function 385
DropRasterTable function 385
DTM 376
dump 427
DWG format 350
DWithin filter operator 134
DXF format 350

E

Easy Install tool 307
easy_install xlrd 307
Eclipse 347
ECW format 350
electron microscopes 376
elipsoids 156–157
ellipses 156
eminent domains 120
empty geometry 120
enable_bitmapscan setting 267
enable_hashagg setting 267
enable_hashjoin setting 267
enable_indexscan setting 267
enable_mergejoin setting 267
enable_nestloop setting 267
enable_seqscan setting. *See also* planner strategies 267
enable_sort setting. *See also* planner strategies 267
EnterpriseDb 419
envelopes 99–101
Environmental Systems Research Institute. *See* ESRI
EPSG (European Petroleum Survey Group) codes 17, 35, 154, 162, 168
EPSG:3785 161
EPSG:4326 35, 162–164, 198
equality 141–146
  geometric 142
  spatial 142, 148
equator 154
equatorial projection 160
equi-gravitational surface 155
equipotential surface 155
ESRI ArcGIS 349
ESRI ArcIMS 347
ESRI ArcSDE 366
ESRI Personal Geodatabase 191–192, 349
ESRI Shape format 20, 56, 169, 175, 280, 349
  exporting 196–197
ESRI tools 89
Euclidean geometry 159
EUMETSAT Archive native (.nat) driver 299
European Petroleum Survey Group. *See* EPSG
Excel 308, 350, 353
EXCEPT 434, 440
EXISTS 435
EXPLAIN 247
EXPLAIN ANALYZE 247, 249
EXPLAIN ANALYZE VERBOSE 247
EXPLAIN PLAN 242, 245
exporting data 195, 357, 362, 365, 369
expression index. *See* functional index
exterior ring 41–42
exteriors 123, 151
ExtJS 313, 333

F

FAA data 215
factor 302
FeatureServer 314–315
FILLFACTOR setting 274
filter_rings 273
finite points 124
Firebird 435
flat file data 22
FME 175
foreign key constraint 57
foreign keys 256
free data 173
free geographic data 173
FROM clause 434
FULL JOIN 436, 438
Full Text Search 250
functional indexes 167–168, 254
fuzzystrmatch.sql file 256
FWTools 188

G

Gauss, Friedrich 155
GDAL (Geospatial Data Abstraction Library) 175, 297–298, 403
gdal2raster 384
gdalDrivers 299
gdalwarp executable 383, 400
Generalized Inverted Tree (GIN) index 250
generate_series 107, 235, 356
geocoding 219
geocoding webservice 310
GeoDb tab, Add Layer dialog box, gvSIG Project Manager 368
geodesy 156
geodetic measurement 94
geodetics 14, 24, 96, 153, 156, 159, 161
GeoDjango web suite 304
GeoExt 333–342
GeoExt extension to OpenLayers 313, 318
Geographic Information Systems (GIS) 118
geographic modeling 156
Geographic Resources Analysis Support (GRASS) 177, 357
geography data type
  distance/area calculations 210
  for EPSG:4326 163
  measurement with 98–99
  of SQL Server 2008 vs. PostGIS OGC 23
  proximity queries using 342
  ST_DWithin function for 207
  storing WGS 84 lon lat (4326) in 161
  using to store data 24, 36
  vs. geometry data types 20, 54, 96–97, 159
Geography JavaScript Object Notation (GeoJSON) 316, 326, 341
Geography Markup Language (GML) 86, 175, 316, 349
geography_columns 384
Geohash geocoding system 87
geoid 154–156
GeoJSON (Geography JavaScript Object Notation) 316, 326, 341

geometric dimensions 123
Geometric Engine Open Source (GEOS) 13
geometric processing 97, 152
geometry
  boundary of 124
  defined 38–39
  measurement functions for 98
geometry columns 34, 61, 79
geometry comparators 139
geometry data type
  general discussion 6
  of SQL Server 2008 vs. PostGIS OGC 23
  vs. geography data types 20, 54, 96–97, 159
Geometry JavaScript Object Notation (GeoJSON) 86
geometry processing 269
geometry types 57, 90
geometry_columns table
  COORD-DIMENSION column 34–35
  interacting with 37–38
  overview 34
  SRID column 35–36
  TYPE column 36–37
geometry_dump 105, 107
geometrycollection 8, 45, 52
geomval objects 392–393
georeferenced raster data 373
georeferencing 280, 389
GeoRSS 175, 316
GEOS (Geometric Engine Open Source) 13, 51, 93, 204, 269
GeoServer 319
  accessing PostGIS layers via WMS/WFS 326–327
  and GeoExt 333
  installing 324–325
  setting up with PostGIS workspaces 325–326
  vs. other server products and 314–315
Geospatial Data Abstraction Library (GDAL) 175, 297–298, 403
geostatisticians 291
GeoStatistics Canada 192
GetCapabilities 324
GetFeatureInfo 324
GIN (Generalized Inverted Tree) index 250
GIS (Geographic Information Systems) 118, 154
GIST indexes 136, 250
GML (Geography Markup Language) 86, 175, 316, 349
GML See geography markup language
Google Maps 3, 212, 309, 343
Google Mercator 161, 164, 331
GPS 221
GPS Exchange Format (GPX) 175, 190–191, 221, 349
GPS track points 221
GPX 175, 190–191, 349, 353
grandchild tables 78
graphical explain 250
graphical explain plan 247, 261
GRASS (Geographica Resources Analysis Support System) 177, 357
graticule 236
gravitational measurement 155
gravity meter 155
grid 229
GROUP BY 63, 146, 227, 444
GroupAggregate 267
GRS 80 spatial reference system 96, 156, 193
gvSig tool 318, 347, 366–370

H

Haiti Crisis Map 333
hash indexes 250
hash joins 241
HashAggregate 262
hasnodata option 393
HAVING clause 227, 444
heat maps 236
help command 297
heterogeneous geometry columns 54–55
hexagon 239
hexagonal grid 236
HEXEWB 146
hole. *See* interior ring
homogeneous geometry columns 54, 56–57
hstore data type 62, 78, 194
HTML 313, 316
HTTP 314

I

IBM DB2 database 12, 432, 435, 445
ILIKE predicate 170
Illustra 9
IMMUTABLE function 208, 268
immutable function 254
IMS server 327
IN clause 435
index clustering 273
index scans 241
indexes 242, 250
information_schema catalog 430
Informix 9
infra red camera 376
INHERIT 68
inheritance hierarchies 69
inheritance. *See* table inheritance
INNER JOIN clause 436–437
INSERT construct 447
install.packages 297
installing from PostGIS source 421
interior 123, 149
interior ring 42–43
INTERSECT clause 434, 440, 442
intersection 119
intersection matrix model 142, 148
intersects 124, 148
intersects with tolerance 135
invalid geometry 42, 269
irregularly blocked raster 375

J

JAI 366–367
Java 290
Java Topology Suite (JTS) 13, 351
Java Web Archive (WAR) 315, 324
JDBC driver 317
Jetty web server 315, 323
JGrass 357
JOIN clause 436
join operations 118
JPG files 350, 373
JSON format 247, 314
JTS (Java Topology Suite) 13
Jython framework 346

K

k nearest neighbor (kNN) 136
KML (Keyhold Markup Language) 175, 316
  exporting 198
  overview 85–86
  template to format in 340
  tools supporting 349
kNN (k nearest neighbor) 136
KNN GIST 243
KyngChaos 421

L

LAEA (Lambert Azimuthal Equal Area) 160–161
land cover 371
land use 371, 376
LATIN1 encoding 182
least function 220
LEFT JOINs 242, 436, 439
length 302
Length_Spheroid functions 96
levels 302
library() command 297
LIDAR tool 376
limit theorems 124
line fitting 376
lineal 124
linear referencing 215
Lines 302
linestring 7, 19, 54, 63, 132, 394
LINESTRINGM 40
Linux 177, 282, 419
list() function 302
load command 294
loader_generate_script function 281
loader_lookuptables 281
loader_platform 281
loader_variables 281
localhost command 427
locate 305
lon lat 114, 193, 270, 283
ls( ) command 294
LTS (Long Term Support) 358

M

M coordinate 39, 102, 110
Mac OS X 177, 282, 419, 421
Macromedia Flash/Flex 86
maintenance_work_mem 266
Manifold tool 13, 318
map file 321
map reduce 291
MapFish 318
MapInfo 358
  and OGR2OGR 175–176
  exporting to tab format 198–199
  importing to tab format 192–193
MapInfo WFS 134
mapping server 314–324
MapQuest 3, 309, 327
MapScript 315
MapServer 314
  calling mapping service using reverse proxy 322–324
  exporting as mapfile using templates 362
  installing 319–320
  OGC WMS and WFS functionality 320–322
Mapserver 134, 362
MassGIS layers 332
match address 283
materialization 250, 263–264
MATLAB 291
MAX 450
measurement 94, 160
Mercator 161, 164, 331
meta programming 291
MetaCarta 327
metatables 432
metro stations 54
Microsoft Access 191
Microsoft Bing maps 212, 309, 327
Microsoft Excel 308, 350, 353
Microsoft SQL Server. *See* SQL Server
MID format 350
MIF format 350, 353
miles 209
minimum distance 205
Mobile feature 346
model database 422
modeling 66
models 63, 79
MrSID format 350, 353, 379
multi geometries 113
multicurve 88
multilinestrings 8, 44, 223
multipointm 44
multipoints 8, 44, 274
multipolygons
  and ST_Intersects function 204
  defined 45
  in city model 61
  states as 43
MySQL 175, 245, 314, 348, 361, 370, 432, 435–436, 448

N

NAD (North American Datum) 158
NAD 27 27, 89, 159, 169
NAD 83 27, 169–170, 193
National Grid System 161
National Oceanic and Atmospheric Administration (NOAA) 376
NATURAL JOIN 436, 439
nearest neighbor 134, 204
nested loops 241
.NET MapScript 319
NEW record variable 72
NEW.*, used in trigger functions 76
NOAA (National Oceanic and Atmospheric Administration) 376
nodata 384
Nominatim 281
non-commutative relationships 119
non-dimensional intersection 124
norm_addy object 283
normalize_address function 285
North American Datum. *See* NAD
NOT IN clause 435
NULL 75

O

object relational database 9
oblique 160
ODBC driver 175, 317
OFFSET 263
OGC (Open Geospatial Consortium) standards 13, 38, 313, 316
OGC web services 313, 344
OGR 316

OGR2OGR 175
  environment variables 189
  export 197–199
  GEOM_TYPE option 189
  GEOMETRY_NAME option 189
  LAUNDER option 189
  layer creation 188
  PG_USE_COPY variable 189
  PGCLIENTENCODING variable 189
  PGSQL_OGR_FID variable 189
  PRECISION option 189
  use 187–193
ogrDrivers() command 298
OLD record variable 72
one-click installer 425
opar 295
Open Database License 179
Open Geospatial Consortium. *See* OGC
OpenGeo Suite 25
OpenGIS SQL/MM 90
OpenJUMP 13, 17, 30, 63, 304, 318, 346, 351–357, 393
OpenJump 30
OpenLayers 318, 327–333
OpenStreetMap 179, 193–195, 199, 281, 316
Oracle database 12, 137, 353, 432, 445
Oracle SDO (spatial data option) 14, 127, 175, 348, 366, 370
ordinal_position field 431
OSGEO (Open Source Geospatial Foundation) 13
OSM. *See* OpenStreetMap
osm2pgrouting utility 193
osm2pgsql utility 62, 193–195
output functions 84
OVER 235
overlaps layer 8
overlay 129, 332

P

package 298
Pago Pago 155
parent table 59–60
paris_polygons 65
partial index 167, 253
partitions 58, 60, 266
Pele 380
Perl 290
Personal GeoDatabase 191–192, 349
pg_catalog tool 167
pg_dump tool 174, 427
pg_dumpall tool 174
pg_read_file tool 304
pg_restore tool 174, 429
pg.spi.exec 294
pgAdmin III 16–17, 22, 25, 78, 174, 247, 419, 422, 425, 429
PGeo. *See* Personal Geodatabase
pgRouting tool 280, 286–290, 311
PgSphere 250
pgsql2shp tool 175, 195–197, 199
PHP 313, 337, 341–343
PHP ADOdb 337
PHP MapScript 315
PHP PEAR 338
PHP Smarty. *See* Smarty
pixel 371
pixel_types 384
PL handler 290
PL languages 290
PL/Java 10, 290
PL/Perl 10, 290, 304
PL/PgSQL 10, 70, 74, 290
PL/Proxy 290
PL/Python 10, 70, 280, 304–311
PL/R 10, 70, 280, 292–304, 311
PL/Sh 10, 290
PL/TCL 10, 70
planar measurement 94, 160
planar model 94, 156
planner strategies 242, 267
plot 302
plpython.so 305
plpythonu 306, 310
plr 303
PNG format 350, 373
png() command 295
Point MZ data type 39
point on surface 103
points 7, 54, 295
polar axes 156
polygon 6, 19, 41, 54
polygonizing 376
polyhedral surface 8, 35
Populate_Geometry_Columns function 37, 64, 67, 228
PostGIS
  history 13–14
  proprietary tools 15
  version 1.4 37, 228
  version 1.5 36, 94, 106, 204
  version 2.0 8, 93, 226, 269
PostGIS raster 372–376
PostGIS WKT raster 398
postgis_full_version() command 426
postgresql-plpython 305
PostgreSQL. *See also* PostGIS
  and CONSTRAINT EXECUTION variable 266
  and ORDER BY field 433
  and statement AS 434
  common table expressions in 263
  cost and row settings 268
  features of 10–13
  functional dependency 444
  GIS, adding to 13–14
  history of 9
  PL/Python caveats 305
  using tables in functions 208
  window aggregates in 445–446
  Window functions in 137, 445–446
premature optimization 186
primary keys 57, 256
print() command 295
prj file 172, 197
Probe_Geometry_Columns function 37
proj4text 381
projections 159–161
proprietary software 345
proximity analysis 204
psql tool 174, 195, 199, 422–424, 428
pushpin 4
PyGDAL tool 403
Pythagorean theorem 36, 159
Python 280, 290, 305–306, 311, 313, 341, 346, 353, 357
Python MapScript 315

Q

q() command 298
QGIS 13, 24, 177, 318, 347, 357
QL:2008 430
Qt 346
quadrants 54
Quantum GIS 13, 177, 318, 347, 357–362
query builders 257

query plan 257
query planner 242

R

R environment 292
R_HOME environment variables 293
radii 156
random_page_cost 57
range 160, 306–307
RANK window function 265
raster data 8, 357, 362, 371
raster type 176, 373
raster_columns table 383
raster2pgsql.py 379–383
RData (R's custom binary format) 293
readOGR method 301
rectangular grid 236
Red Hat Enterprise Linux 420
Red Hat Fedora 420
reduce 306
reflection 291
Refractions Research 13
region tagging 215
relational data type 79
relational database 204, 242
remote sensing 374
REST Web feature server 315, 327
reverse geocoder 309
reverse proxy server 322–323
rewriting 73
RGBA (Red Green Blue Alpha) channels 373
rgdal. *See* GDAL
RGTK2 R package 298, 304
RIGHT JOIN clause 436, 438
RotateAtPoint function 240
rotation 239
round 283
ROW_NUMBER() window function. *See also* window functions 234–235, 445
rows 268, 445
rules 53, 69, 71–73, 79

S

Safe FME commercial GIS systems 304
sandboxed PL 291
SAS 291
saveOGR functions 303
Scalar Vector Graphics (SVG) 86, 350, 353
scaling family 238–239, 390
schema-less models 79, 194
SDO_GEOM.WITHIN_DISTANCE function 134
SDO_RELATE 127
sea level 155
SECURITY DEFINER 291
segmenting 120
select 299, 432
SELECT ... INTO statement 449
SELECT * 259
SELECT clause 227
self intersection 149
self joins 264, 433
seq_page_cost setting 57
sequential scans 241
sets 436, 440
setters 115
shape feature 160
Shapefile 25
Shapefile to PostGIS Import Tool (SPIT) 177, 362
shared web host 314
shared_buffers 266
SharpMap.NET open source server product 314–315, 403
short-circuiting 252
show plans 242
SHOWPLAN_ALL 245
shp2pgsql command-line loader 26, 174–175, 182, 199
shp2pgsql-gui 25–27, 175, 186–187, 199
Silverlight 86
simplicity 44, 269
simplification functions 112
slicing table geometries 229
Smarty PHP helper library 337
snapping points 217
SOAP standard messaging stream 314
soundex function 256
sp 303
spatial aggregates functions 227
spatial analysis 8
spatial clustering 273
spatial database 4–5, 54, 118, 152, 347
spatial design pattern 273
spatial equality 142, 148
spatial functions 152, 203
spatial index. *See* GIST index
spatial intersections 62
spatial orientation 100
spatial predicates 269
spatial processing 8
spatial query 7–8
spatial reference system (SRS) 17, 19, 54, 88, 119, 154–161, 172, 205
spatial references 152
spatial relationship function 118
spatial relationships 152
spatial SQL 118
spatial_ref_sys metatable 35–36, 89, 172, 381, 428
SpatiaLite spatial extender 175, 314, 349
SpatialLines 302
SpatialLinesDataFrame 302
Sphere 96
spherical coordinate system 96
sphericalMercator setting 331
spheroid 94, 96
spheroid function 210
SPIT (Shapefile to PostGIS Import Tool) 177, 362
splines 49
split 302
spplot plot function 303
SQL (Structured Query Language) 5–6, 8–12, 14–15, 20–23, 25, 30, 32, 203, 241
SQL COPY command 22
SQL joins 204
SQL patterns 257
SQL primer 203
SQL Server 432, 434
  version 2005 137, 435
  version 2008 23, 137, 159, 314
  version 2008 R2 12
SQL/MM Spatial standard 14
SQL/MM standard function 88
SQLite data source 175, 316, 349
squashing. *See* projection
srid 35
SRID 4326 85
SRID spatial reference system 162, 289, 357
  See also spatial reference systems

SRS (spatial reference system) 162
See also spatial reference systems
SRS ID (spatial reference system identifier) 35
ST_3DClosestPoint 3D measurement function 95
ST_3DDistance 3D measurement function 95
ST_3DIntersects 3D measurement function 95
ST_Area function 95, 98, 255
ST_AsBinary function 31, 143, 164, 356
ST_AsEWKB function 85, 143
ST_AsEWKT function 85
ST_AsGeoJSON 338
ST_AsGeoJSON function 87
ST_AsGML function 86–87, 212
ST_AsKML function 85, 87, 212
ST_AsSVG function 86–87
ST_AsText function 44, 46–47, 84–85, 212
ST_Boundary function 98, 102, 124
ST_Box2D function 100
ST_Buffer function 30–31, 99, 166, 214, 356
ST_BuildArea function 111
ST_Centroid function 103, 105
ST_ClosestPoint function 219
ST_Collect function 227, 272, 302, 443
ST_Contains function 119, 125, 127, 150
ST_ContainsProperly function 128
ST_ConvexHull function 388
ST_CoordDim function 91
ST_CoveredBy function 99, 127
ST_Covers function 99, 127–128
ST_Crosses function 130
ST_CurveToLine function 48–49, 119
ST_DFullyWithin function 205
ST_Difference function 119, 131
ST_Dimension function 91
ST_Disjoint function 131, 148
ST_Distance function 36, 98, 205, 209, 242, 271
ST_Distance_Sphere function 96, 209, 212
ST_Distance_Spheroid function 96, 163, 209
ST_Dump function 105, 134, 204, 226, 272–273
ST_DumpAsPolygons function 374
ST_DumpPoints function 106, 301, 394
ST_DumpRings function 107
ST_DWithin function 29–31, 96, 98, 135–136, 163, 206, 208, 218, 242, 255, 271, 343
ST_Envelope function 100, 387
ST_Equals function 142, 148
ST_Extent function 443
ST_ExteriorRing function 272
ST_GeoHash function 87
ST_Geohash function 87
ST_GeometryN function 105–107, 272
ST_GeomFromEWKB function 85
ST_GeomFromEWKT function 46, 82
ST_GeomFromText 18–19
ST_GeomFromText function 18–19, 23–24, 41, 46–47, 83, 89, 109, 150, 242, 448
ST_GeomFromWKB function 83
ST_Height function 386
ST_InteriorRingN function 107
ST_Intersect function 147
ST_Intersection function 99, 120–121, 123, 234, 392
_ST_Intersects function 248, 268
ST_Intersects function 119, 204, 268, 393
ST_IsSimple function 41
ST_IsValidDetail function 93
ST_IsValidReason function 92
ST_Length function 95, 98, 255
ST_Length_Spheroid function 96
ST_Length3D function 95
ST_Line_Interpolate_Point function 218
ST_Line_Locate_Point function 218, 224
ST_Line_Substring function 224
ST_LineMerge function 220, 302
ST_LineToCurve function 48
ST_MakeLine function 221, 227, 394
ST_MakePoint function 83, 108, 394
ST_MakePointM function 109
ST_MakePolygon function 110, 112, 272
ST_MakeValid function 93
ST_MapAlgebra function 404
ST_Multi function 112
ST_NPoints function 49, 93
ST_NumBands function 386
ST_NumGeometries function 107
ST_NumInteriorRings 273
ST_NumInteriorRings function 273
ST_NumPoints function 93
ST_OrderingEquals function 143–144
ST_Perimeter function 95, 98
ST_Point function 17–18, 83, 108
ST_PointFromText function 83
ST_PointOnSurface function 103, 105
ST_Polygon function 387, 390
ST_Polygonize function 111, 112, 227
ST_Reclass function 404
ST_Relate function 149
ST_Resample function 404
ST_Rotate function 239
ST_RotateX function 239
ST_RotateY function 239
ST_RotateZ function 239
ST_Scale function 238
ST_ScaleX function 390
ST_ScaleY function 390
ST_Segmentize function 395
ST_SetGeoReference function 389
ST_SetPoint function 224
ST_SetScale function 389
ST_SetSRID function 18, 89, 389
ST_SetUpperLeftX function 389
ST_SetValue function 392
ST_Simplify function 114
ST_SimplifyPreserveTopology function 112–114, 115, 271
ST_SnapToGrid function 113, 274
ST_Split function 134, 226

ST_SRID function 89, 386
ST_SymDifference function 119, 131
ST_Touches function 119, 129, 150
ST_Transform function 23, 27, 89, 113, 167, 210, 255, 400
ST_Translate function 234, 236, 356
ST_Union function 227–229, 443
ST_Value function 386, 392
ST_Width function 386
ST_Within function 125, 127, 152
ST_X function 101, 283, 302
ST_XMin function 84
ST_Y function 101, 283, 302
stable function 208, 268
Stack Builder 25
standard_conforming_strings 83
State Plane class 161
statistical analysis 215
statistical functions 291
statistical packages 292
Stonebraker, Michael 9
str() R base function 298
street centerlines 219
Structured Query Language. *See* SQL
subselects 258, 447
subset 299
SVG (Scalar Vector Graphics) 350, 353
Symmetric Difference function 131
system variables 265
System-R 9

T

TAB format 350
table inheritance 57, 59–60, 66–69, 79
table layouts 53
table partitioning 58
table scan 242, 257
table_name field 431
tablespace 56
tagging data 215
tar 181
TCL language 290
template databases 422
template_postgis database 422–423, 425
tessellate 228
textual explain plans 246
theming feature 356
thermal imagery 376
TIFF world file 350, 373
TIGER data 43, 279
TIGER_geocoder_2009 folder 282
TIN (Triangulated Irregular Network) 35
title 295
topology 269
transform error 89, 117
transformations 236
translations 239
Transverse flavors of projections 160
Traveling salesperson (TSP) 288
Triangulated Irregular Network (TIN) 35
trigger functions 53, 69–70, 73–74, 76, 78
TRUNCATE TABLE statement 76, 78, 450
trusted language function 291
trusted PL sandboxed PL 291
TSP (Traveling salesperson) 288–289
Twitter 316

U

UAC (User Account Control) 420
Ubuntu distros 421
uDig (User Friendly Desktop GIS) 48, 318, 347, 362–366
UMN MapServer. *See* MapServer
unary function 116
UNION ALL set 65, 440
UNION set 65, 146, 434, 440
unique key 256
units conversion table 207
Universal Trans Mercator (UTM) 61, 161, 213, 255
Unix systems 177, 282
unknown spatial reference system 118–119
unknown SRID 36
unzip processes 181, 281
UPDATE statement 168
update.packages() command 297
UpdateGeometrySRID function 37, 171
upgrading 426, 428
US Highways 7
US National Atlas Equal Area 136
US TIGER census data 280
use_spheroid argument 98
User Account Control (UAC) 420
USER-DEFINED data type 431
User-Friendly Desktop GIS (uDig) 48, 318, 347, 362, 366
UTF8 182, 428
UTM (Universal Trans Mercator) 61, 161, 213, 255
UTM zone 213

V

vacuum analyze processes 27–28, 256
validity 42, 123
VB.NET 323
vector data 371
vector space 88
vectorize 374, 378
VERBOSE 249
vertex 100
views 430
vignette() command 297
VirtualEarth 327
VOLATILE function 268, 309

W

WAR (Java Web Archive) 315, 324
Washington DC 54
WCS (Web Coverage Service) 316, 351, 372
Web Feature Service (WFS) 134, 316, 318, 321, 324, 326, 350–351
Web Feature Service Transactional (WFS-T) 316, 318, 328, 351
Web Mapping Service (WMS) 316, 318–319, 321, 324, 350–351
Web Mercator 211

web servers 314
web services 314
well known binary 85
well-known text (WKT) 42, 44–46, 48, 51–52, 81–82, 85, 349
WFS (Web Feature Service) 134, 316, 318, 321, 324, 326, 350–351
WFS-T (Web Feature Service Transactional) 316, 318, 328, 351, 365
wget command-line tool 180, 281
WGS 84 (World Geodetic System) 96, 156, 217
WHEN trigger clause 70
WHERE clause 433, 444
window frames 264–265, 445
WINDOW function 265
window functions 138, 264, 445
Windows 177, 188
Windows Server 426
Windows Vista 420
Within relationship 125
WKT (well-known text) 42, 44–46, 48, 51–52, 81–82, 85, 349
WKT Raster 176, 372–376
WKT SRS notation 169
WMS (Web Mapping Service) 316, 318–319, 321, 324, 350–351
WMS capabilities 321
work_mem memory 266
World Wide Web 344
WPS format 351

X

XAML 86
xlrd package 307
XML 193, 314

Y

Yahoo 327
Yahoo Maps 3, 309
YatZ Linux packager 421
yield 308
YUM Linux packager 421
Yum repository 305, 420

Z

Z coordinate 102

DATABASES

PostGIS IN ACTION

Obe • Hsu

PostGIS is an open source spatial database extender for PostgreSQL. It equals or surpasses proprietary alternatives, allowing you to create location-aware queries with just a few lines of SQL code, and provides a back-end for mapping applications with minimal effort.

PostGIS in Action teaches readers of all levels to write spatial queries that solve real-world problems. It first gives you a background in vector-based GIS and then quickly moves into analyzing, viewing, and mapping data. You'll learn how to optimize queries for maximum speed, simplify geometries for greater efficiency, and create custom functions for your own applications. The book covers PostgreSQL 8.4, 9.0, and 9.1 features and shows you how to integrate with other GIS tools.

What's Inside

- An introduction to spatial databases
- Geometry types, functions, and queries
- Applying PostGIS to real-world problems
- Extending PostGIS to web and desktop applications

Familiarity with relational database concepts is helpful but not required.

Regina Obe and **Leo Hsu** are database consultants. Regina is a member of the PostGIS core development team and the Project Steering Committee. They are hosts of BostonGIS.com and PostgresOnLine.com.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/PostGISinAction

Free eBook
SEE INSERT

"A concise guide that's truly one of a kind."
—From the foreword by Paul Ramsey, Chair PostGIS Steering Committee

"An elegant introduction to a difficult domain."
—Mark Leslie, LISAsoft Pty Ltd

"The ultimate PostGIS tour guide."
—Brent Wood, NIWA

"Will give you the *Aha!* moment you've been waiting for."
—Jeff Addison Southgate Software Ltd

"Want to get the most out of PostGIS? This is required reading for you."
—James Fee, WeoGeo.com

ISBN 13: 978-1-935182-26-9
ISBN 10: 1-935182-26-9
54999
9 781935 182269

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]