Chapter 5. Raster Data Management, Queries, and Applications

Table of Contents

5.1. Loading and Creating Rasters

For most use cases, you will create PostGIS rasters by loading existing raster files using the packaged `raster2pgsql` raster loader.

5.1.1. Using raster2pgsql to load rasters

The `raster2pgsql` is a raster loader executable that loads GDAL supported raster formats into sql suitable for loading into a PostGIS raster table. It is capable of loading folders of raster files as well as creating overviews of rasters.

Since the raster2pgsql is compiled as part of PostGIS most often (unless you compile your own GDAL library), the raster types supported by the executable will be the same as those compiled in the GDAL dependency library. To get a list of raster types your particular raster2pgsql supports use the `-G` switch. These should be the same as those provided by your PostGIS install documented here ST_GDALDrivers if you are using the same gdal library for both.

The older version of this tool was a python script. The executable has replaced the python script. If you still find the need for the Python script Examples of the python one can be found at GDAL PostGIS Raster Driver Usage. Please note that the raster2pgsql python script may not work with future versions of PostGIS raster and is no longer supported.

When creating overviews of a specific factor from a set of rasters that are aligned, it is possible for the overviews to not align. Visit http://trac.osgeo.org/postgis/ticket/1764 for an example where the overviews do not align.

EXAMPLE USAGE:

```
raster2pgsql raster_options_go_here raster_file someschema.sometable > out.sql
```

-?

Display help screen. Help will also display if you don't pass in any arguments.

-G

Print the supported raster formats.

(c|a|d|p) These are mutually exclusive options:

-c

Create new table and populate it with raster(s), *this is the default mode*

-a

Append raster(s) to an existing table.

-d

Drop table, create new one and populate it with raster(s)

-p

Prepare mode, only create the table.

Raster processing: Applying constraints for proper registering in raster catalogs

-C

Apply raster constraints -- srid, pixelsize etc. to ensure raster is properly registered in `raster_columns` view.

-x

Disable setting the max extent constraint. Only applied if -C flag is also used.

-r

Set the constraints (spatially unique and coverage tile) for regular blocking. Only applied if -C flag is also used.

Raster processing: Optional parameters used to manipulate input raster dataset

-s <SRID>

Assign output raster with specified SRID. If not provided or is zero, raster's metadata will be checked to determine an appropriate SRID.

-b BAND

Index (1-based) of band to extract from raster. For more than one band index, separate with comma (,). If unspecified, all bands of raster will be extracted.

-t TILE_SIZE

Cut raster into tiles to be inserted one per table row. `TILE_SIZE` is expressed as WIDTHxHEIGHT or set to the value "auto" to allow the loader to compute an appropriate tile size using the first raster and applied to all rasters.

-R, --register

Register the raster as a filesystem (out-db) raster.

Only the metadata of the raster and path location to the raster is stored in the database (not the pixels).

-l `OVERVIEW_FACTOR`

Create overview of the raster. For more than one factor, separate with comma(,). Overview table name follows the pattern o_`overview factor_table`, where `overview factor` is a placeholder for numerical overview factor and `table` is replaced with the base table name. Created overview is stored in the database and is not affected by -R. Note that your generated sql file will contain both the main table and overview tables.

-N `NODATA`

NODATA value to use on bands without a NODATA value.

Optional parameters used to manipulate database objects

-q

Wrap PostgreSQL identifiers in quotes

-f COLUMN

Specify name of destination raster column, default is 'rast'

-F

Add a column with the name of the file

-I

Create a GiST index on the raster column.

-M

Vacuum analyze the raster table.

-T `tablespace`

Specify the tablespace for the new table. Note that indices (including the primary key) will still use the default tablespace unless the -X flag is also used.

-X `tablespace`

Specify the tablespace for the table's new index. This applies to the primary key and the spatial index if the -I flag is used.

-Y

Use copy statements instead of insert statements.

-e

Execute each statement individually, do not use a transaction.

-E ENDIAN

Control endianness of generated binary output of raster; specify 0 for XDR and 1 for NDR (default); only NDR output is supported now

-V `version`

Specify version of output format. Default is 0. Only 0 is supported at this time.

An example session using the loader to create an input file and uploading it chunked in 100x100 tiles might look like this:

You can leave the schema name out e.g `demelevation` instead of `public.demelevation` and the raster table will be created in the default schema of the database or user

```
raster2pgsql -s 4236 -I -C -M *.tif -F -t 100x100 public.demelevation > elev.sql
psql -d gisdb -f elev.sql
```

A conversion and upload can be done all in one step using UNIX pipes:

```
raster2pgsql -s 4236 -I -C -M *.tif -F -t 100x100 public.demelevation | psql -d gisdb
```

Load rasters Massachusetts state plane meters aerial tiles into a schema called `aerial` and create a full view, 2 and 4 level overview tables, use copy mode for inserting (no intermediary file just straight to db), and -e don't force everything in a transaction (good if you want to see data in tables right away without waiting). Break up the rasters into 128x128 pixel tiles and apply raster constraints. Use copy mode instead of table insert. (-F) Include a field called filename to hold the name of the file the tiles were cut from.

```
raster2pgsql -I -C -e -Y -F -s 26986 -t 128x128  -l 2,4 bostonaerials2008/*.jpg aerial
```

```
--get a list of raster types supported:
raster2pgsql -G
```

The -G commands outputs a list something like

```
Available GDAL raster formats:
  Virtual Raster
  GeoTIFF
  National Imagery Transmission Format
  Raster Product Format TOC format
  ECRG TOC format
  Erdas Imagine Images (.img)
  CEOS SAR Image
  CEOS Image
```

JAXA PALSAR Product Reader (Level 1.1/1.5)
Ground-based SAR Applications Testbed File Format (.gff)
ELAS
Arc/Info Binary Grid
Arc/Info ASCII Grid
GRASS ASCII Grid
SDTS Raster
DTED Elevation Raster
Portable Network Graphics
JPEG JFIF
In Memory Raster
Japanese DEM (.mem)
Graphics Interchange Format (.gif)
Graphics Interchange Format (.gif)
Envisat Image Format
Maptech BSB Nautical Charts
X11 PixMap Format
MS Windows Device Independent Bitmap
SPOT DIMAP
AirSAR Polarimetric Image
RadarSat 2 XML Product
PCIDSK Database File
PCRaster Raster File
ILWIS Raster Map
SGI Image File Format 1.0
SRTMHGT File Format
Leveller heightfield
Terragen heightfield
USGS Astrogeology ISIS cube (Version 3)
USGS Astrogeology ISIS cube (Version 2)
NASA Planetary Data System
EarthWatch .TIL
ERMapper .ers Labelled
NOAA Polar Orbiter Level 1b Data Set
FIT Image
GRIdded Binary (.grb)
Raster Matrix Format
EUMETSAT Archive native (.nat)
Idrisi Raster A.1
Intergraph Raster
Golden Software ASCII Grid (.grd)
Golden Software Binary Grid (.grd)
Golden Software 7 Binary Grid (.grd)
COSAR Annotated Binary Matrix (TerraSAR-X)
TerraSAR-X Product
DRDC COASP SAR Processor Raster
R Object Data Store
Portable Pixmap Format (netpbm)
USGS DOQ (Old Style)
USGS DOQ (New Style)
ENVI .hdr Labelled
ESRI .hdr Labelled
Generic Binary (.hdr Labelled)
PCI .aux Labelled
Vexcel MFF Raster
Vexcel MFF2 (HKV) Raster
Fuji BAS Scanner Image
GSC Geogrid
EOSAT FAST Format
VTP .bt (Binary Terrain) 1.3 Format
Erdas .LAN/.GIS
Convair PolGASP

Image Data and Analysis
NLAPS Data Format
Erdas Imagine Raw
DIPEx
FARSITE v.4 Landscape File (.lcp)
NOAA Vertical Datum .GTX
NADCON .los/.las Datum Grid Shift
NTv2 Datum Grid Shift
ACE2
Snow Data Assimilation System
Swedish Grid RIK (.rik)
USGS Optional ASCII DEM (and CDED)
GeoSoft Grid Exchange Format
Northwood Numeric Grid Format .grd/.tab
Northwood Classified Grid Format .grc/.tab
ARC Digitized Raster Graphics
Standard Raster Product (ASRP/USRP)
Magellan topo (.blx)
SAGA GIS Binary Grid (.sdat)
Kml Super Overlay
ASCII Gridded XYZ
HF2/HFZ heightfield raster
OziExplorer Image File
USGS LULC Composite Theme Grid
Arc/Info Export E00 GRID
ZMap Plus Grid
NOAA NGS Geoid Height Grids

5.1.2. Creating rasters using PostGIS raster functions

On many occasions, you'll want to create rasters and raster tables right in the database. There are a plethora of functions to do that. The general steps to follow.

1. Create a table with a raster column to hold the new raster records which can be accomplished with:

   ```
   CREATE TABLE myrasters(rid serial primary key, rast raster);
   ```

2. There are many functions to help with that goal. If you are creating rasters not as a derivative of other rasters, you will want to start with: ST_MakeEmptyRaster, followed by ST_AddBand

   You can also create rasters from geometries. To achieve that you'll want to use ST_AsRaster perhaps accompanied with other functions such as ST_Union or ST_MapAlgebraFct or any of the family of other map algebra functions.

   There are even many more options for creating new raster tables from existing tables. For example you can create a raster table in a different projection from an existing one using ST_Transform

3. Once you are done populating your table initially, you'll want to create a spatial index on the raster column with something like:

   ```
   CREATE INDEX myrasters_rast_st_convexhull_idx ON myrasters USING gist( ST_Convex
   ```

   Note the use of ST_ConvexHull since most raster operators are based on the convex

hull of the rasters.

Pre-2.0 versions of PostGIS raster were based on the envelope rather than the convex hull. For the spatial indexes to work properly you'll need to drop those and replace with convex hull based index.

4. Apply raster constraints using AddRasterConstraints

5.2. Raster Catalogs

There are two raster catalog views that come packaged with PostGIS. Both views utilize information embedded in the constraints of the raster tables. As a result the catalog views are always consistent with the raster data in the tables since the constraints are enforced.

1. `raster_columns` this view catalogs all the raster table columns in your database.

2. `raster_overviews` this view catalogs all the raster table columns in your database that serve as overviews for a finer grained table. Tables of this type are generated when you use the `-l` switch during load.

5.2.1. Raster Columns Catalog

The `raster_columns` is a catalog of all raster table columns in your database that are of type raster. It is a view utilizing the constraints on the tables so the information is always consistent even if you restore one raster table from a backup of another database. The following columns exist in the `raster_columns` catalog.

If you created your tables not with the loader or forgot to specify the `-C` flag during load, you can enforce the constraints after the fact using AddRasterConstraints so that the `raster_columns` catalog registers the common information about your raster tiles.

- `r_table_catalog` The database the table is in. This will always read the current database.

- `r_table_schema` The database schema the raster table belongs to.

- `r_table_name` raster table

- `r_raster_column` the column in the `r_table_name` table that is of type raster. There is nothing in PostGIS preventing you from having multiple raster columns per table so its possible to have a raster table listed multiple times with a different raster column for each.

- `srid` The spatial reference identifier of the raster. Should be an entry in the Section 4.3.1, “The SPATIAL_REF_SYS Table and Spatial Reference Systems”.

- `scale_x` The scaling between geometric spatial coordinates and pixel. This is only available if all tiles in the raster column have the same `scale_x` and this constraint is applied. Refer to ST_ScaleX for more details.

- `scale_y` The scaling between geometric spatial coordinates and pixel. This is only

available if all tiles in the raster column have the same `scale_y` and the `scale_y` constraint is applied. Refer to ST_ScaleY for more details.

- `blocksize_x` The width (number of pixels across) of each raster tile . Refer to ST_Width for more details.
- `blocksize_y` The width (number of pixels down) of each raster tile . Refer to ST_Height for more details.
- `same_alignment` A boolean that is true if all the raster tiles have the same alignment . Refer to ST_SameAlignment for more details.
- `regular_blocking` If the raster column has the spatially unique and coverage tile constraints, the value with be TRUE. Otherwise, it will be FALSE.
- `num_bands` The number of bands in each tile of your raster set. This is the same information as what is provided by ST_NumBands
- `pixel_types` An array defining the pixel type for each band. You will have the same number of elements in this array as you have number of bands. The pixel_types are one of the following defined in ST_BandPixelType.
- `nodata_values` An array of double precision numbers denoting the `nodata_value` for each band. You will have the same number of elements in this array as you have number of bands. These numbers define the pixel value for each band that should be ignored for most operations. This is similar information provided by ST_BandNoDataValue.
- `extent` This is the extent of all the raster rows in your raster set. If you plan to load more data that will change the extent of the set, you'll want to run the DropRasterConstraints function before load and then reapply constraints with AddRasterConstraints after load.

5.2.2. Raster Overviews

`raster_overviews` catalogs information about raster table columns used for overviews and additional information about them that is useful to know when utilizing overviews. Overview tables are cataloged in both `raster_columns` and `raster_overviews` because they are rasters in their own right but also serve an additional special purpose of being a lower resolution caricature of a higher resolution table. These are generated along-side the main raster table when you use the `-l` switch in raster loading.

Overview tables contain the same constraints as other raster tables as well as additional informational only constraints specific to overviews.

The information in `raster_overviews` does not duplicate the information in `raster_columns`. If you need the information about an overview table present in `raster_columns` you can join the `raster_overviews` and `raster_columns` together to get the full set of information you need.

Two main reasons for overviews are:

1. Low resolution representation of the core tables commonly used for fast mapping zoom-out.
2. Computations are generally faster to do on them than their higher resolution parents because there are fewer records and each pixel covers more territory. Though the computations are not as accurate as the high-res tables they support, they can be

sufficient in many rule-of-thumb computations.

The raster_overviews catalog contains the following columns of information.

- o_table_catalog The database the overview table is in. This will always read the current database.
- o_table_schema The database schema the overview raster table belongs to.
- o_table_name raster overview table name
- o_raster_column the raster column in the overview table.
- r_table_catalog The database the raster table that this overview services is in. This will always read the current database.
- r_table_schema The database schema the raster table that this overview services belongs to.
- r_table_name raster table that this overview services.
- r_raster_column the raster column that this overview column services.
- overview_factor - this is the pyramid level of the overview table. The higher the number the lower the resolution of the table. raster2pgsql if given a folder of images, will compute overview of each image file and load separately. Level 1 is assumed and always the original file. Level 2 is will have each tile represent 4 of the original. So for example if you have a folder of 5000x5000 pixel image files that you chose to chunk 125x125, for each image file your base table will have (5000*5000)/(125*125) records = 1600, your (l=2) o_2 table will have ceiling(1600/Power(2,2)) = 400 rows, your (l=3) o_3 will have ceiling(1600/Power(2,3)) = 200 rows. If your pixels aren't divisible by the size of your tiles, you'll get some scrap tiles (tiles not completely filled). Note that each overview tile generated by raster2pgsql has the same number of pixels as its parent, but is of a lower resolution where each pixel of it represents (Power(2,overview_factor) pixels of the original).

5.3. Building Custom Applications with PostGIS Raster

The fact that PostGIS raster provides you with SQL functions to render rasters in known image formats gives you a lot of optoins for rendering them. For example you can use OpenOffice / LibreOffice for rendering as demonstrated in Rendering PostGIS Raster graphics with LibreOffice Base Reports. In addition you can use a wide variety of languages as demonstrated in this section.

5.3.1. PHP Example Outputting using ST_AsPNG in concert with other raster functions

In this section, we'll demonstrate how to use the PHP PostgreSQL driver and the ST_AsGDALRaster family of functions to output band 1,2,3 of a raster to a PHP request stream that can then be embedded in an img src html tag.

The sample query demonstrates how to combine a whole bunch of raster functions together to grab all tiles that intersect a particular wgs 84 bounding box and then unions with ST_Union the intersecting tiles together returning all bands, transforms to user specified projection using ST_Transform, and then outputs the results as a png using ST_AsPNG.

You would call the below using

```
http://mywebserver/test_raster.php?srid=2249
```

to get the raster image in Massachusetts state plane feet.

```
<?php
/** contents of test_raster.php **/
$conn_str ='dbname=mydb host=localhost port=5432 user=myuser password=mypwd';
$dbconn = pg_connect($conn_str);
header('Content-Type: image/png');
/**If a particular projection was requested use it otherwise use mass state plane met
if (!empty( $_REQUEST['srid'] ) && is_numeric( $_REQUEST['srid']) ){
                $input_srid = intval($_REQUEST['srid']);
}
else { $input_srid = 26986; }
/** The set bytea_output may be needed for PostgreSQL 9.0+, but not for 8.4 **/
$sql = "set bytea_output='escape';
SELECT ST_AsPNG(ST_Transform(
                        ST_AddBand(ST_Union(rast,1), ARRAY[ST_Union(rast,2),ST_Union(
                                ,$input_srid) ) As new_rast
 FROM aerials.boston
        WHERE
         ST_Intersects(rast, ST_Transform(ST_MakeEnvelope(-71.1217, 42.227, -71.1210,
$result = pg_query($sql);
$row = pg_fetch_row($result);
pg_free_result($result);
if ($row === false) return;
echo pg_unescape_bytea($row[0]);
?>
```

5.3.2. ASP.NET C# Example Outputting using ST_AsPNG in concert with other raster functions

In this section, we'll demonstrate how to use Npgsql PostgreSQL .NET driver and the ST_AsGDALRaster family of functions to output band 1,2,3 of a raster to a PHP request stream that can then be embedded in an img src html tag.

You will need the npgsql .NET PostgreSQL driver for this exercise which you can get the latest of from http://npgsql.projects.postgresql.org/. Just download the latest and drop into your ASP.NET bin folder and you'll be good to go.

The sample query demonstrates how to combine a whole bunch of raster functions together to grab all tiles that intersect a particular wgs 84 bounding box and then unions with ST_Union the intersecting tiles together returning all bands, transforms to user specified projection using ST_Transform, and then outputs the results as a png using ST_AsPNG.

This is same example as Section 5.3.1, “PHP Example Outputting using ST_AsPNG in concert with other raster functions” except implemented in C#.

You would call the below using

```
http://mywebserver/TestRaster.ashx?srid=2249
```

to get the raster image in Massachusetts state plane feet.

```
 -- web.config connection string section --
<connectionStrings>
    <add name="DSN"
        connectionString="server=localhost;database=mydb;Port=5432;User Id=myuser;pas
</connectionStrings>
```

```
// Code for TestRaster.ashx
<%@ WebHandler Language="C#" Class="TestRaster" %>
using System;
using System.Data;
using System.Web;
using Npgsql;

public class TestRaster : IHttpHandler
{
        public void ProcessRequest(HttpContext context)
        {

                context.Response.ContentType = "image/png";
                context.Response.BinaryWrite(GetResults(context));

        }

        public bool IsReusable {
                get { return false; }
        }

        public byte[] GetResults(HttpContext context)
        {
                byte[] result = null;
                NpgsqlCommand command;
                string sql = null;
                int input_srid = 26986;
        try {
                    using (NpgsqlConnection conn = new NpgsqlConnection(System.Config
                            conn.Open();

                if (context.Request["srid"] != null)
                {
                    input_srid = Convert.ToInt32(context.Request["srid"]);
                }
                sql = @"SELECT ST_AsPNG(
                            ST_Transform(
                                        ST_AddBand(
                                    ST_Union(rast,1), ARRAY[ST_Union(rast,2),ST_Union(ras
                                                        ,:input_srid) ) As new_rast
                            FROM aerials.boston
                                    WHERE
                                        ST_Intersects(rast,
                                        ST_Transform(ST_MakeEnvelope(-71.1217, 42.227, -7
                                command = new NpgsqlCommand(sql, conn);
                command.Parameters.Add(new NpgsqlParameter("input_srid", input_srid));
```

```
                        result = (byte[]) command.ExecuteScalar();
                conn.Close();
                        }

                }
          catch (Exception ex)
          {
              result = null;
              context.Response.Write(ex.Message.Trim());
          }
                return result;
          }
}
```

5.3.3. Java console app that outputs raster query as Image file

This is a simple java console app that takes a query that returns one image and outputs to specified file.

You can download the latest PostgreSQL JDBC drivers from http://jdbc.postgresql.org/download.html

You can compile the following code using a command something like:

```
set env CLASSPATH .:..\postgresql-9.0-801.jdbc4.jar
javac SaveQueryImage.java
jar cfm SaveQueryImage.jar Manifest.txt *.class
```

And call it from the command-line with something like

```
java -jar SaveQueryImage.jar "SELECT ST_AsPNG(ST_AsRaster(ST_Buffer(ST_Point(1,5),10,
```

```
 -- Manifest.txt --
Class-Path: postgresql-9.0-801.jdbc4.jar
Main-Class: SaveQueryImage
```

```
// Code for SaveQueryImage.java
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.io.*;

public class SaveQueryImage {
  public static void main(String[] argv) {
      System.out.println("Checking if Driver is registered with DriverManager.");

      try {
        //java.sql.DriverManager.registerDriver (new org.postgresql.Driver());
        Class.forName("org.postgresql.Driver");
      }
```

```
        catch (ClassNotFoundException cnfe) {
          System.out.println("Couldn't find the driver!");
          cnfe.printStackTrace();
          System.exit(1);
        }

        Connection conn = null;

        try {
          conn = DriverManager.getConnection("jdbc:postgresql://localhost:5432/mydb","my
          conn.setAutoCommit(false);

          PreparedStatement sGetImg = conn.prepareStatement(argv[0]);

          ResultSet rs = sGetImg.executeQuery();

                  FileOutputStream fout;
                  try
                  {
                          rs.next();
                          /** Output to file name requested by user **/
                          fout = new FileOutputStream(new File(argv[1]) );
                          fout.write(rs.getBytes(1));
                          fout.close();
                  }
                  catch(Exception e)
                  {
                          System.out.println("Can't create file");
                          e.printStackTrace();
                  }

          rs.close();
                  sGetImg.close();
          conn.close();
        }
        catch (SQLException se) {
          System.out.println("Couldn't connect: print out a stack trace and exit.");
          se.printStackTrace();
          System.exit(1);
        }
    }
}
```

5.3.4. Use PLPython to dump out images via SQL

This is a plpython stored function that creates a file in the server directory for each record.

```
//plpython postgresql stored proc.  Requires you have plpython installed
CREATE OR REPLACE FUNCTION write_file (param_bytes bytea, param_filepath text)
RETURNS text
AS $$
f = open(param_filepath, 'wb+')
f.write(param_bytes)
return param_filepath
$$ LANGUAGE plpythonu;
```

```
--write out 5 images to the PostgreSQL server in varying sizes
-- note the postgresql daemon account needs to have write access to folder
-- this echos back the file names created;
 SELECT write_file(ST_AsPNG(
        ST_AsRaster(ST_Buffer(ST_Point(1,5),j*5, 'quad_segs=2'),150*j, 150*j, '8BUI',
         'C:/temp/slices'|| j || '.png')
         FROM generate_series(1,5) As j;

     write_file
--------------------
 C:/temp/slices1.png
 C:/temp/slices2.png
 C:/temp/slices3.png
 C:/temp/slices4.png
 C:/temp/slices5.png
```

5.3.5. Outputting Rasters with PSQL

Sadly PSQL doesn't have easy to use built-in functionality for outputting binaries. This is a bit of a hack and based on one of the suggestions outlined in Clever Trick Challenge -- Outputting bytea with psql that piggy backs on PostgreSQL somewhat legacy large object support. To use first launch your psql commandline connected to your database.

Unlike the python approach, this approach creates the file on your local computer.

```
SELECT oid, lowrite(lo_open(oid, 131072), png) As num_bytes
 FROM
 ( VALUES (lo_create(0),
   ST_AsPNG( (SELECT rast FROM aerials.boston WHERE rid=1) )
  ) ) As v(oid,png);
-- you'll get an output something like --
   oid   | num_bytes
---------+-----------
 2630819 |     74860

-- next note the oid and do this replacing the c:/test.png to file path location
-- on your local computer
 \lo_export 2630819 'C:/temp/aerial_samp.png'

-- this deletes the file from large object storage on db
SELECT lo_unlink(2630819);
```