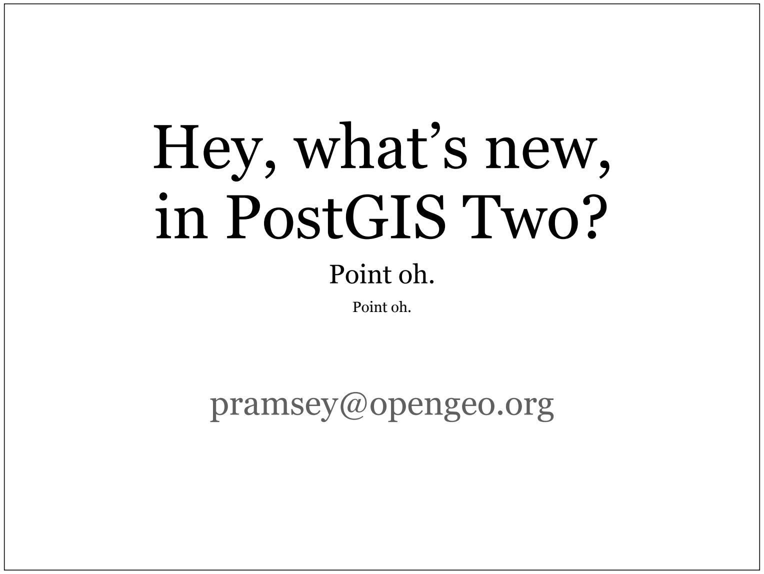
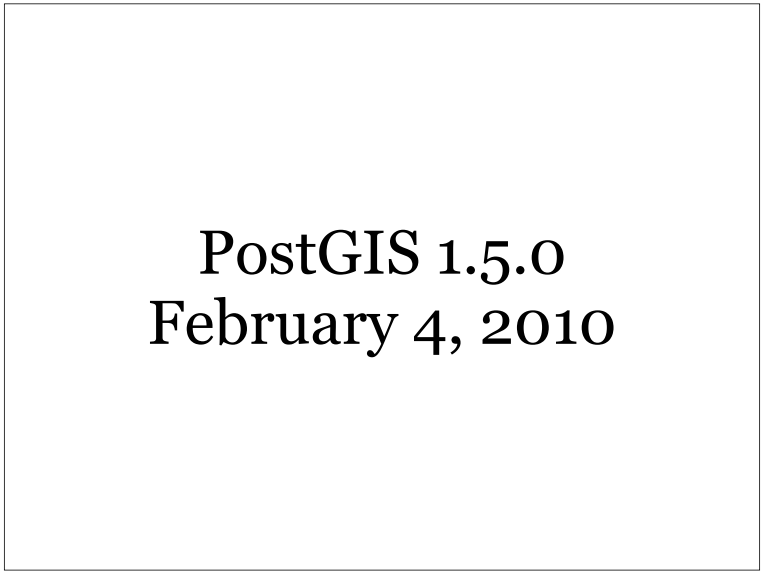


Hey, what's new, in PostGIS Two?

Point oh.

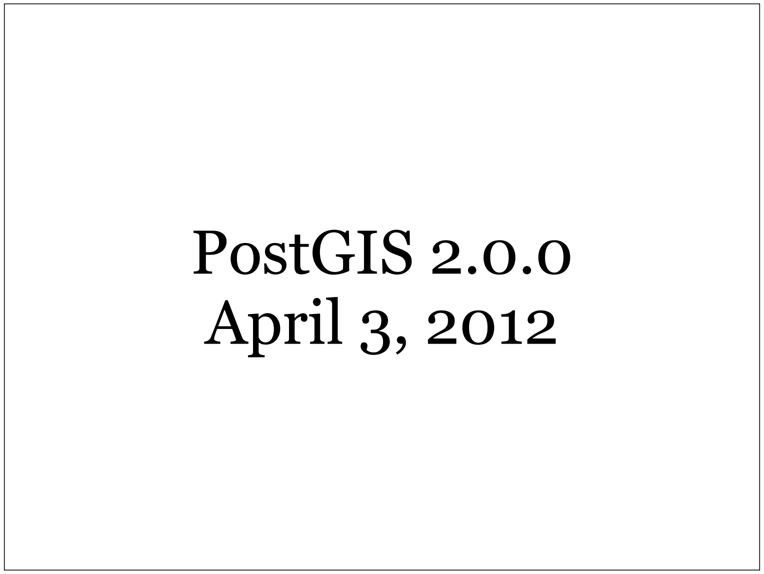
Point oh.

pramsey@opengeo.org



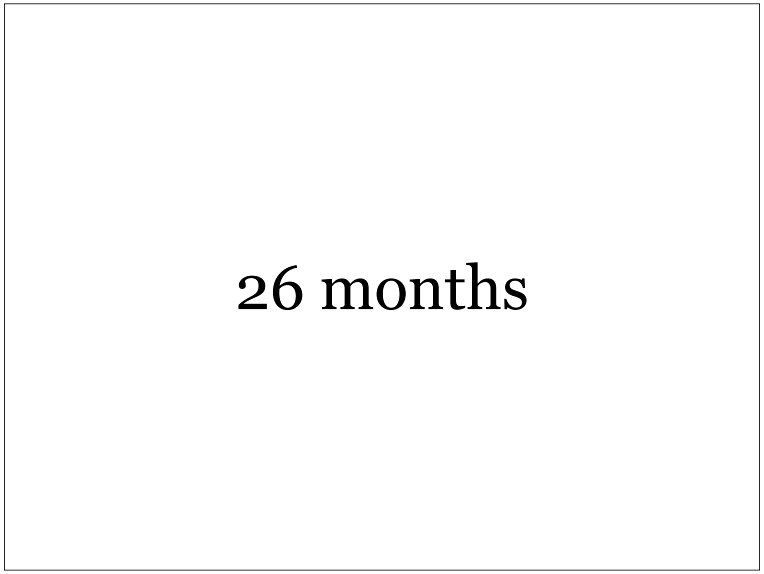
PostGIS 1.5.0 February 4, 2010

So...
The last major release of PostGIS was in early 2010



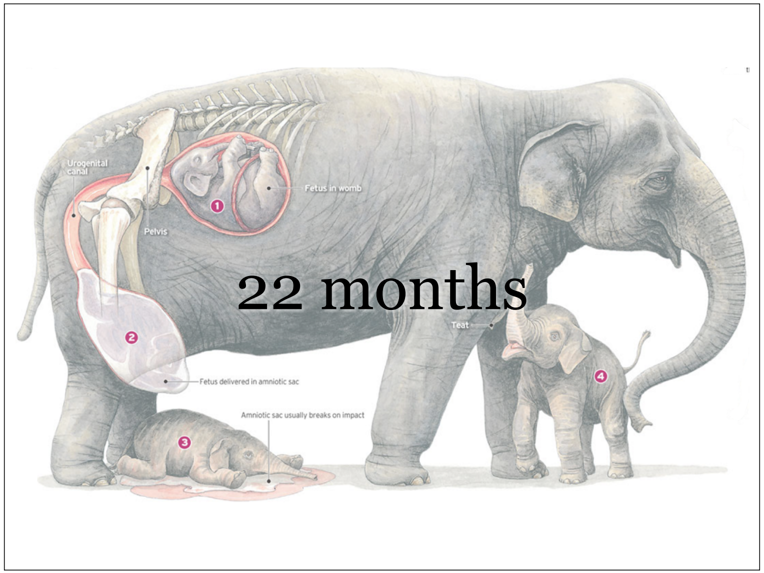
PostGIS 2.0.0
April 3, 2012

And PostGIS 2.0 was released just last week...
(right?)



26 months

So... it took us 26 months to birth our latest major version.

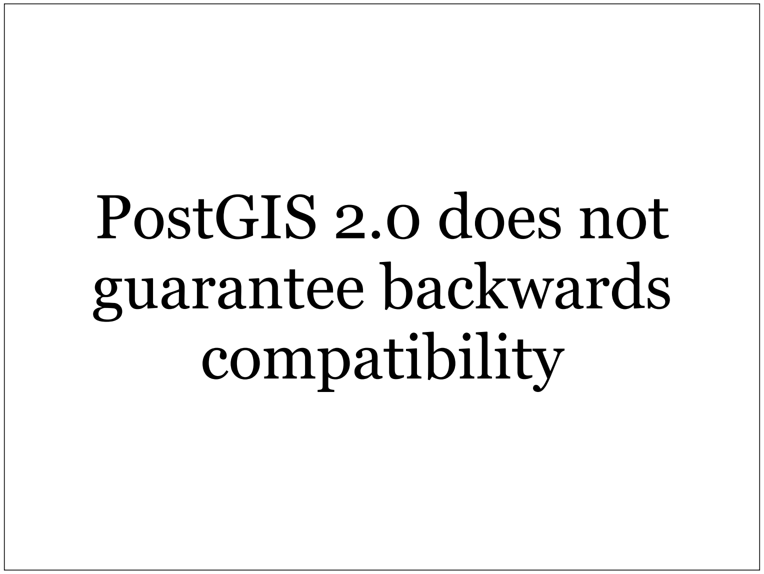

Urogenital canal
Fetus in womb
1
Pelvis
Teat
2
Fetus delivered in amniotic sac
Amniotic sac usually breaks on impact
3
4

22 months

And it takes an elephant 22 months to gestate a new elephant.

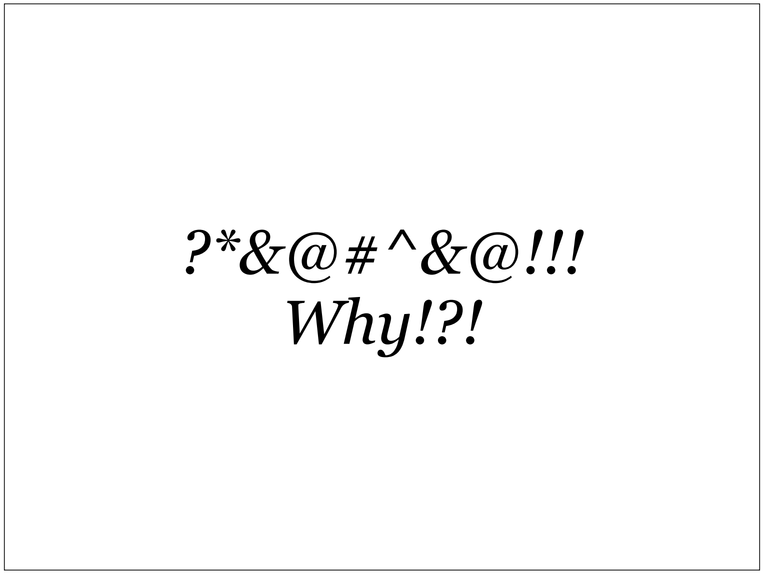
Why so long?
Why 2.0?
Why not 1.6?

And you might have some valid questions about the process...
Why so long, why 2.0, why not 1.6?
Not just because we like big round numbers.



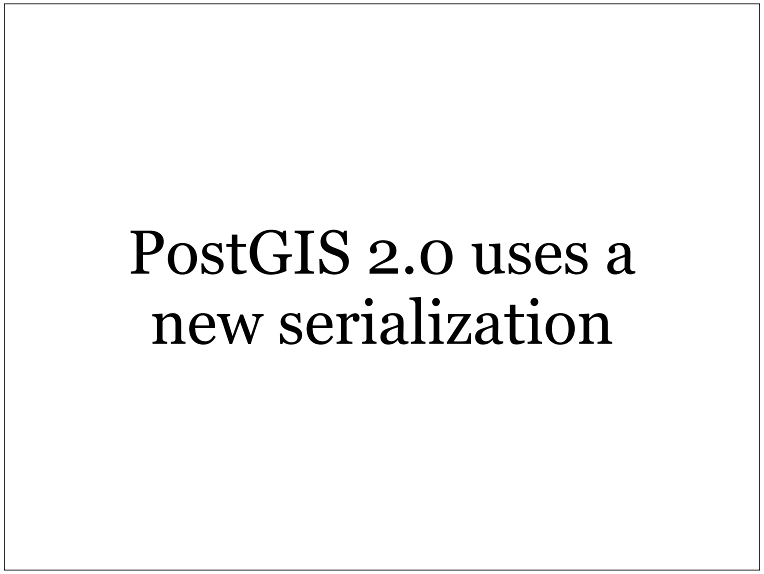
PostGIS 2.0 does not guarantee backwards compatibility

The big round number means that this release is not backwards compatible. That’s a big deal. We don’t do that very often. 1.X lasted 7 years.



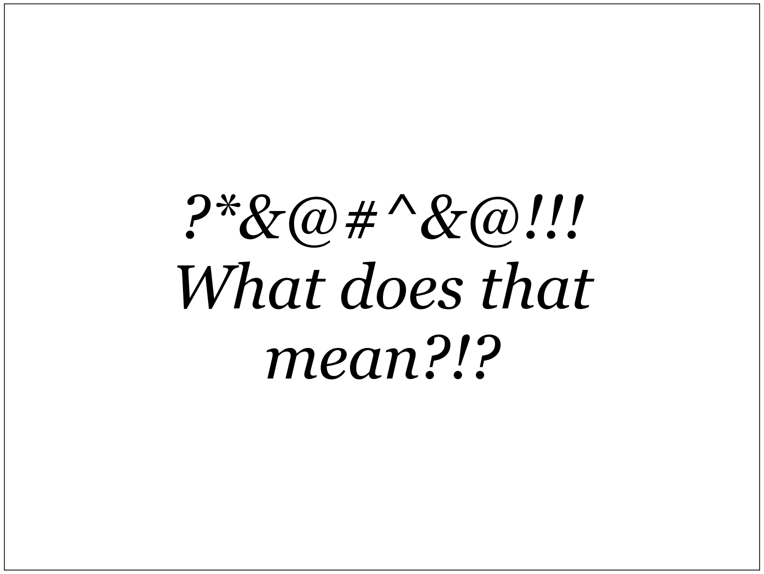
?&@#^&@!!!*
Why!?!

And I understand this might not be the most popular thing to do...



PostGIS 2.0 uses a new serialization

But the main reason we lost backward compatibility is that we use a new serialization...

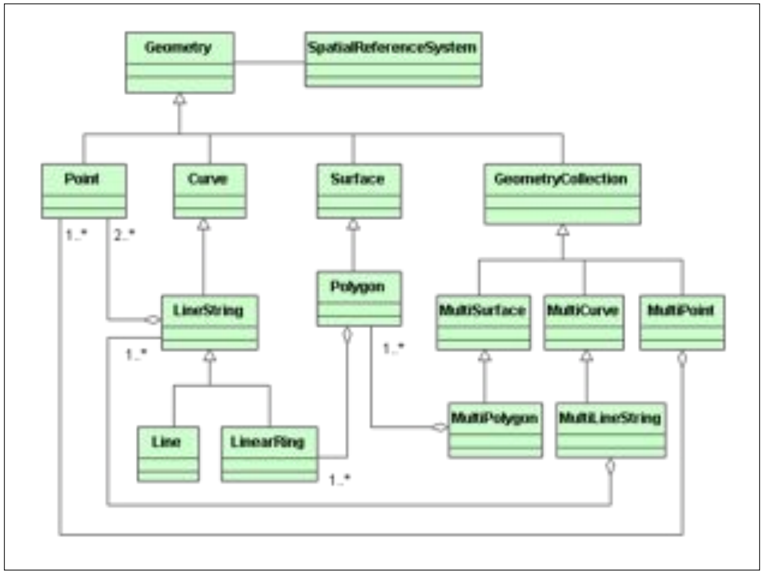


?&@#^&@!!! What does that mean?!?*

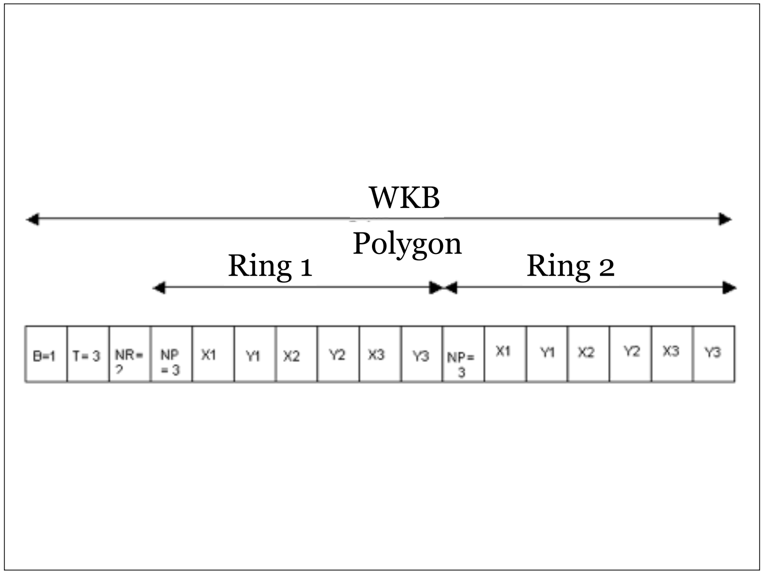
OK, yes, this is getting down into the weeds...

Serialization:
A recipe to convert a memory structure into an array of bytes on disk

A serialization is the format used for on-disk storage. It’s a recipe for converting in-memory objects into bytes on-disk.



An in-memory model is something like the simple features object model, with discrete parts that might be stored in different parts of memory, and pointers tying them all together.



WKB

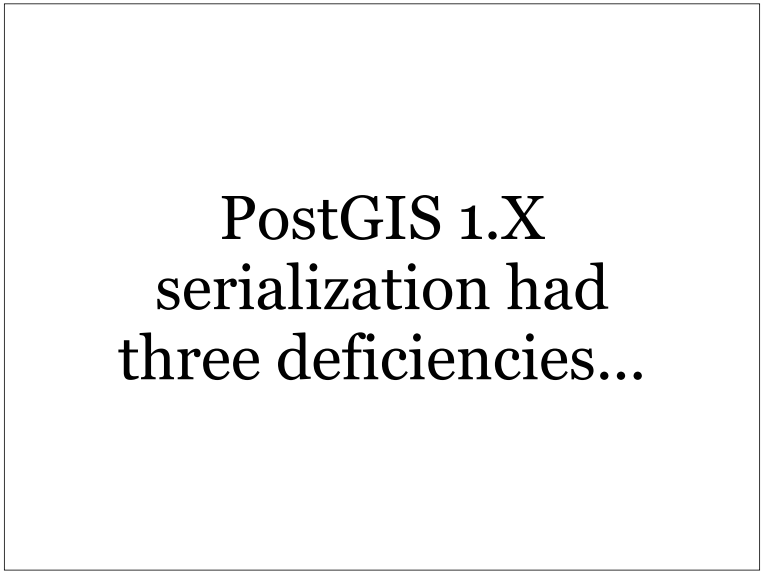
Polygon

Ring 1

Ring 2

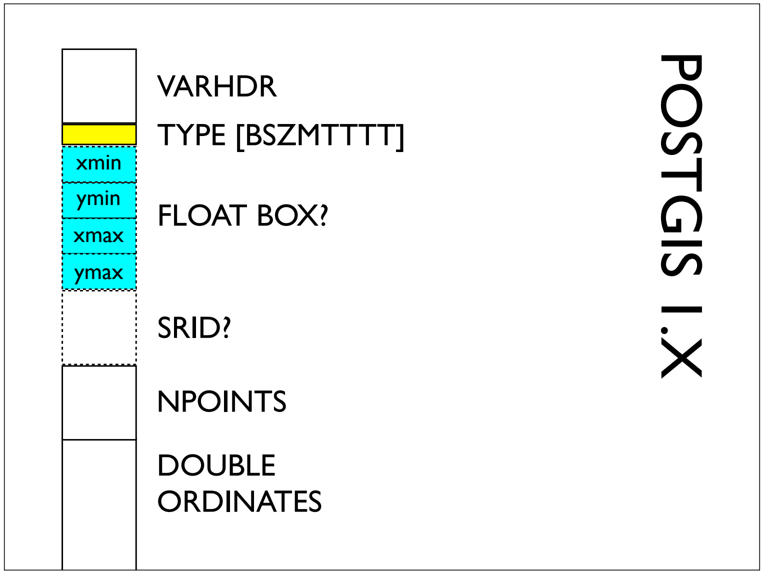
B=1	T= 3	NR= 2	NP = 3	X1	Y1	X2	Y2	X3	Y3	NP= 3	X1	Y1	X2	Y2	X3	Y3

A serialization is a contiguous sequence of bytes. The OGC well-known binary format is an example of a serialization.



PostGIS 1.X serialization had three deficiencies...

But why change? The old serialization had a few drawbacks...



POSTGIS 1.X

VARHDR

TYPE [BSZMTTTT]

xmin
ymin
xmax
ymax

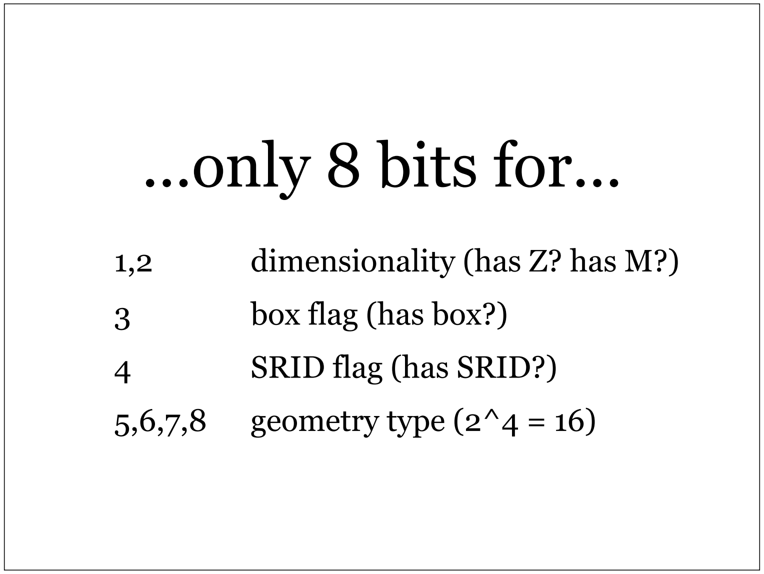
FLOAT BOX?

SRID?

NPOINTS

DOUBLE
ORDINATES

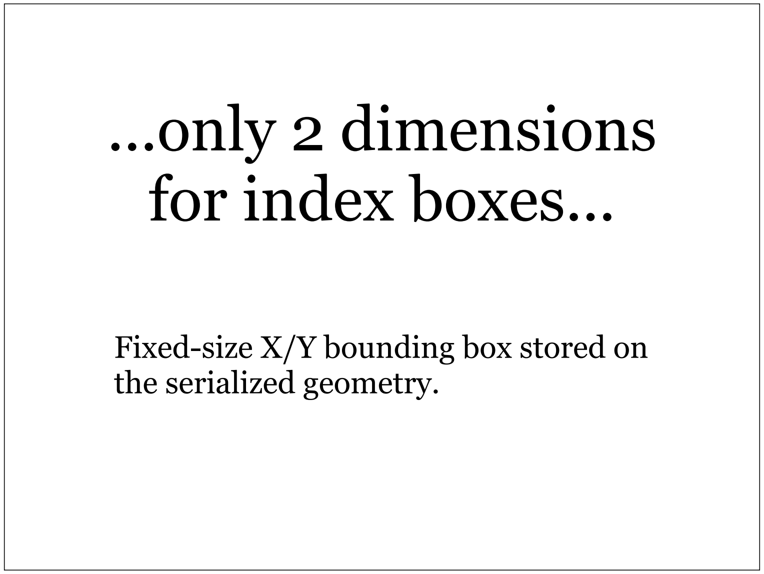
The old serialization started up with a “type byte” (in yellow) that included both the dimensionality information and the geometry type number. That’s a lot of information to pack into 8 bits.



...only 8 bits for...

1,2	dimensionality (has Z? has M?)
3	box flag (has box?)
4	SRID flag (has SRID?)
5,6,7,8	geometry type (2^4 = 16)

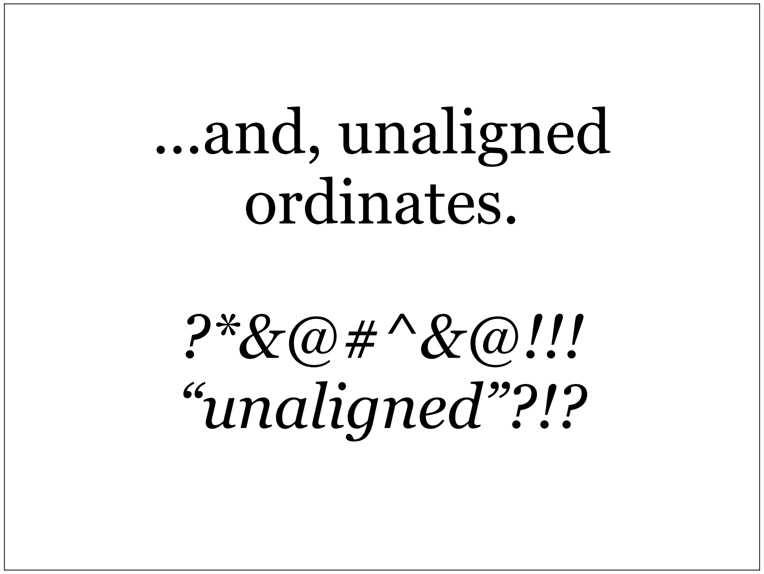
Two bits for dimensions, two bits for box/srid flags, and four bits for types.
Four bits can hold the numbers zero to 15.
type 1 is point, then there is linestring, polygon, multipoint, multilinestring, multipolygon, geometrycollection
circularstring, compoundcurve, curvepolygon, multicurve, multisurface
triangle, tin, polyhedral surface (15)
That’s it, we’re out of space for new types!



...only 2 dimensions for index boxes...

Fixed-size X/Y bounding box stored on the serialized geometry.

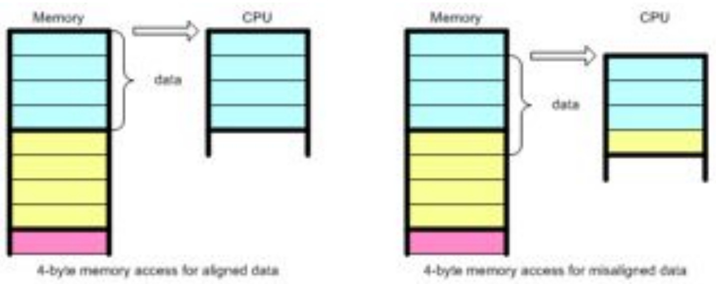
And the bounding box was fixed at four floats, so only room to index the X/Y plane.



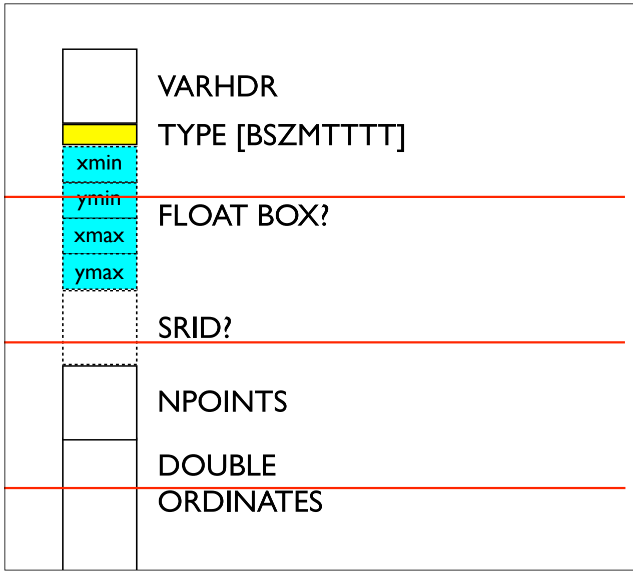
...and, unaligned ordinates.

?&@#^&@!!!*
“unaligned”?!?

And finally, because of that type byte, the coordinates are not double aligned.
OK, that “aligned” could use some explanation....


Memory
CPU
data
4-byte memory access for aligned data
Memory
CPU
data
4-byte memory access for misaligned data

Memory is accessed at addresses, counted in bytes. Data types have sizes expressible in bytes. If the data is stored in memory at an address that is evenly divisible by the size of the data type it is said to be “aligned”. Aligned data can be accessed faster and more directly than unaligned data. On some architectures (RISC) it cannot be directly accessed at all, it has to be copied into an aligned location first.



POSTGIS 1.X

VARHDR

TYPE [BSZMTTTT]

FLOAT BOX?

SRID?

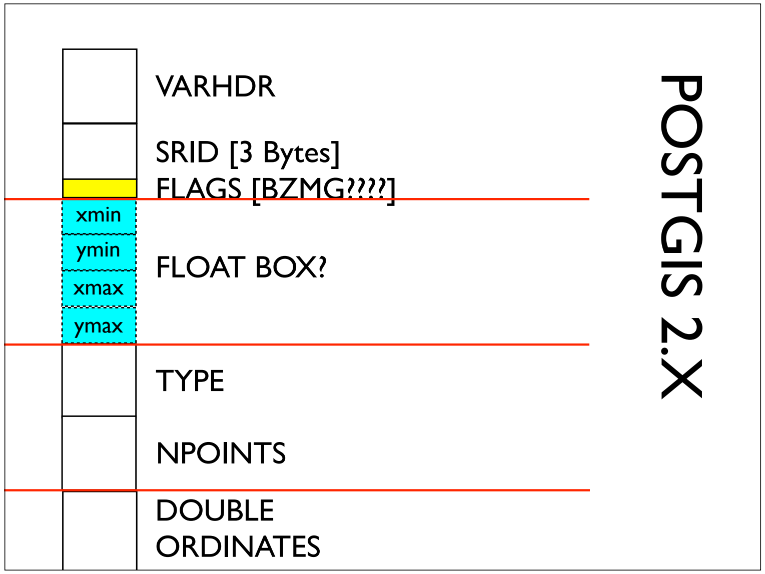
NPOINTS

DOUBLE ORDINATES

If you overlay the double precision alignment boundaries over the old serialization, you can see pretty quickly that the coordinates don’t fall on the alignment boundaries.

PostGIS 2.X serialization has room to grow!

The new serialization addresses all these drawbacks.



POSTGIS 2.X

VARHDR

SRID [3 Bytes]

FLAGS [BZMG????]

xmin

ymin

xmax

ymax

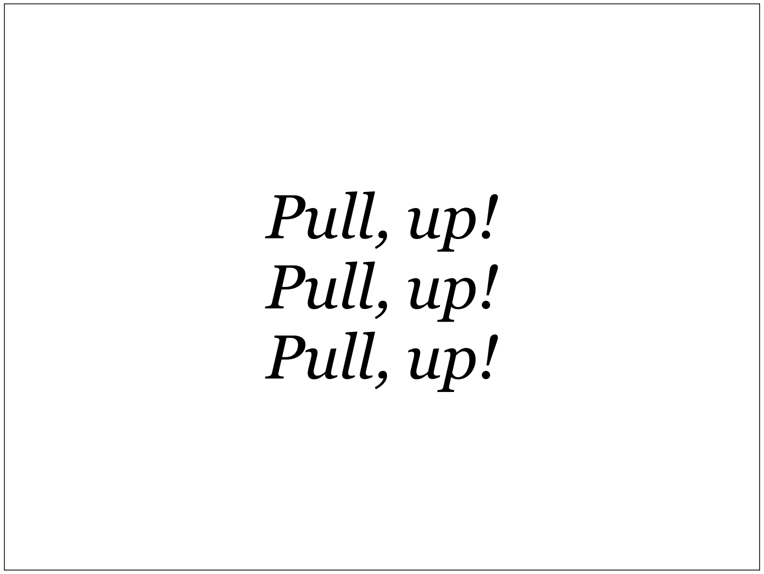
FLOAT BOX?

TYPE

NPOINTS

DOUBLE ORDINATES

By re-ordering the contents of the serialization and expanding a few components,
we've gotten space for more type numbers (whole integer!),
we have achieved double alignment for the coordinates,
and we have space for a version number (four spare bits in the "flags") so we can avoid future dump/reload situations.

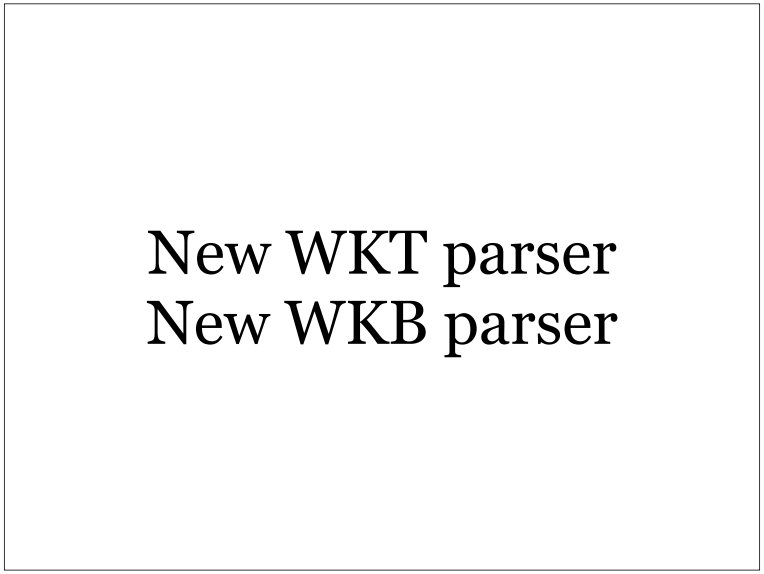


Pull, up!
Pull, up!
Pull, up!

Uh, oh, I think this talk is getting a little too technical!!!!....

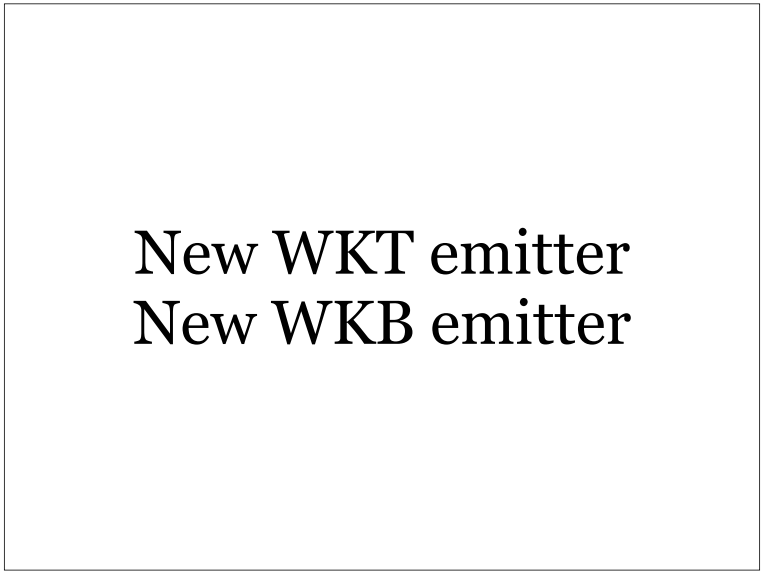
New serialization meant...

So, *because* we changed the serialization, we had to do some other work...



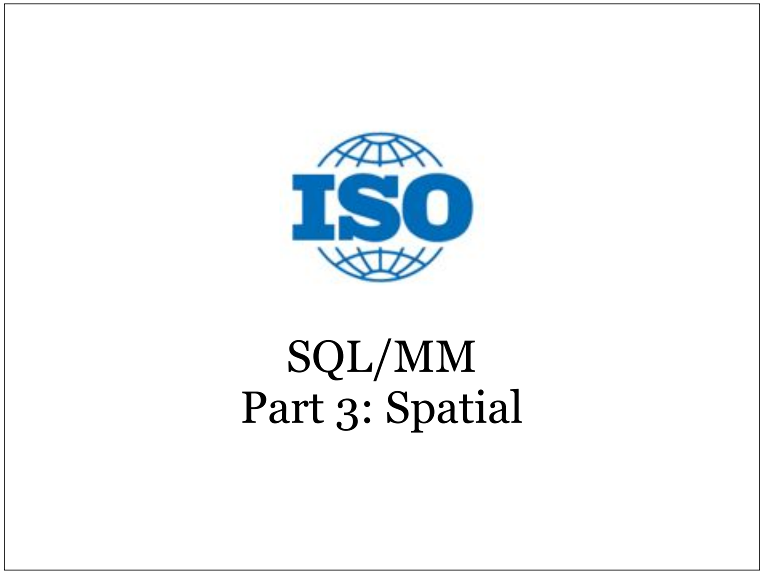
New WKT parser
New WKB parser

The old well-known-text and well-known-binary parsers were both tightly bound to the old serialization. They were also both pretty hard to read and maintain. So, they have been completely re-written. They are now more generic and easier to support.



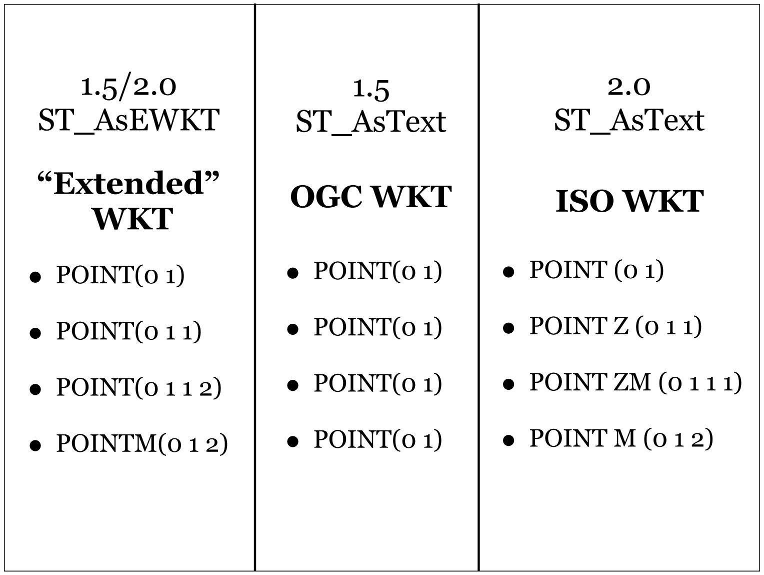
New WKT emitter
New WKB emitter

Similarly the well-known-text and well-known-binary emitters were bound to the old serialization and have been completely re-written.



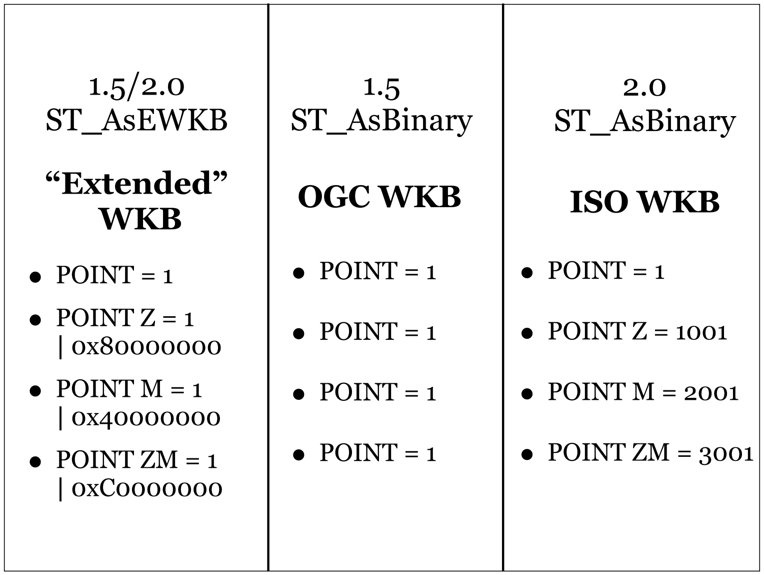
SQL/MM
Part 3: Spatial

Since we were re-writing them anyways, this provided an opportunity to add in full support for both consuming and producing ISO SQL/MM versions of well-known-text and well-known-binary.



1.5/2.0 ST_AsEWKT **"Extended" WKT**	1.5 ST_AsText **OGC WKT**	2.0 ST_AsText **ISO WKT**
POINT(0 1)	POINT(0 1)	POINT (0 1)
POINT(0 1 1)	POINT(0 1)	POINT Z (0 1 1)
POINT(0 1 1 2)	POINT(0 1)	POINT ZM (0 1 1 1)
POINTM(0 1 2)	POINT(0 1)	POINT M (0 1 2)

The main thing to note is that the ISO forms support 3d and 4d geometry, and that the ST_AsText() function now emits those extra dimensions. The ST_GeomFromText and other text consumers will accept any of the forms (OGC WKT, EWKT, or ISO WKT).



1.5/2.0
ST_AsEWKB

“Extended” WKB

- POINT = 1
- POINT Z = 1 | 0x80000000
- POINT M = 1 | 0x40000000
- POINT ZM = 1 | 0xC0000000

1.5
ST_AsBinary

OGC WKB

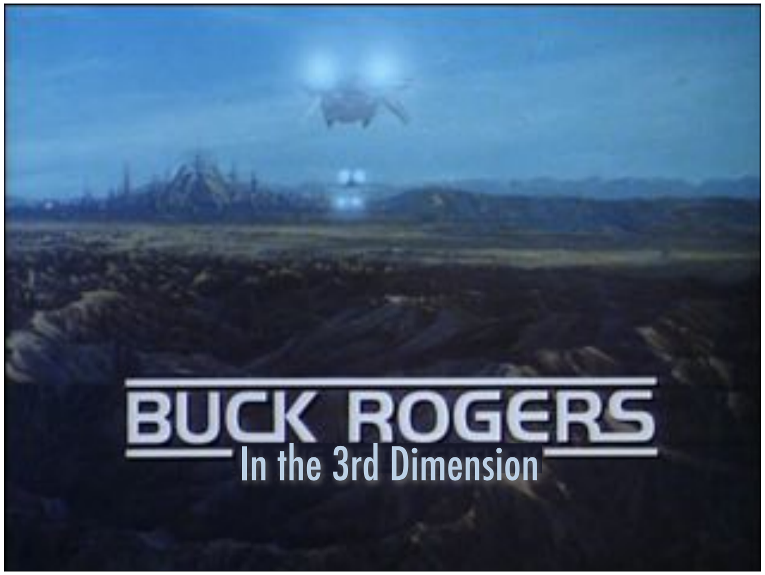
- POINT = 1
- POINT = 1
- POINT = 1
- POINT = 1

2.0
ST_AsBinary

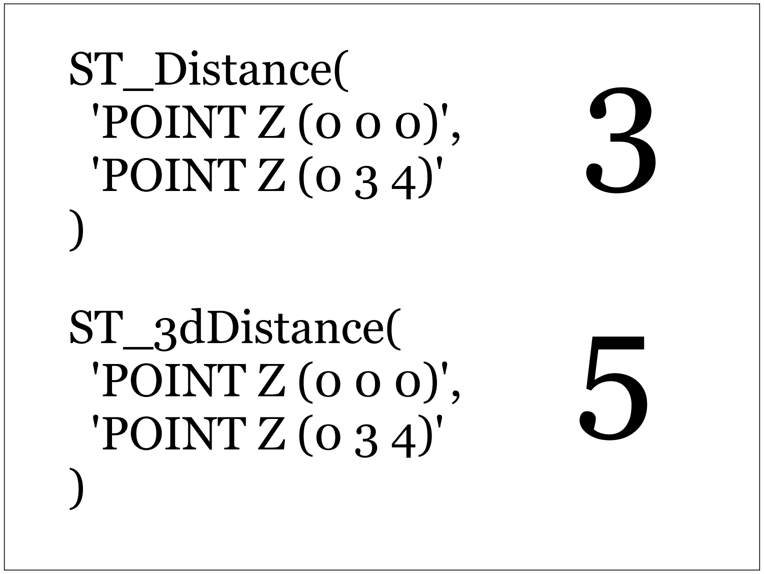
ISO WKB

- POINT = 1
- POINT Z = 1001
- POINT M = 2001
- POINT ZM = 3001

ISO SQL/MM also defined new type numbers and support for 3d and 4d geometry. Again, the standard ST_AsBinary function now emits ISO well-known-binary.

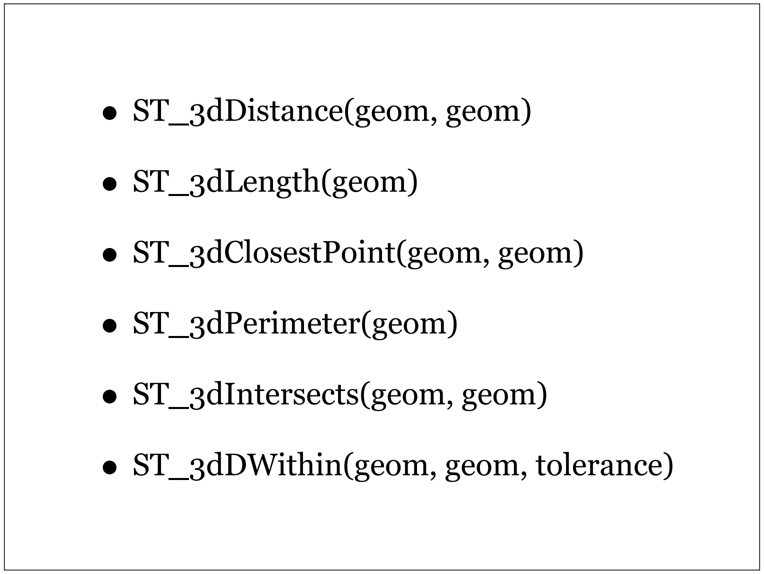


And hey, all this new core support for 3D is good, because we have a lot of new support for the third dimension in other functions.



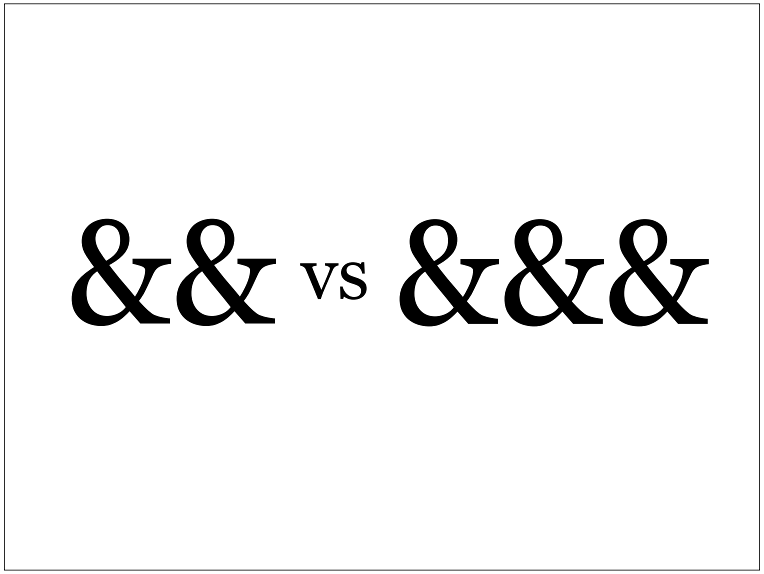
```
ST_Distance(
  'POINT Z (0 0 0)',
  'POINT Z (0 3 4)'
)                              3

ST_3dDistance(
  'POINT Z (0 0 0)',
  'POINT Z (0 3 4)'
)                              5
```

For example, you can now do 3D distance calculations on geometries.



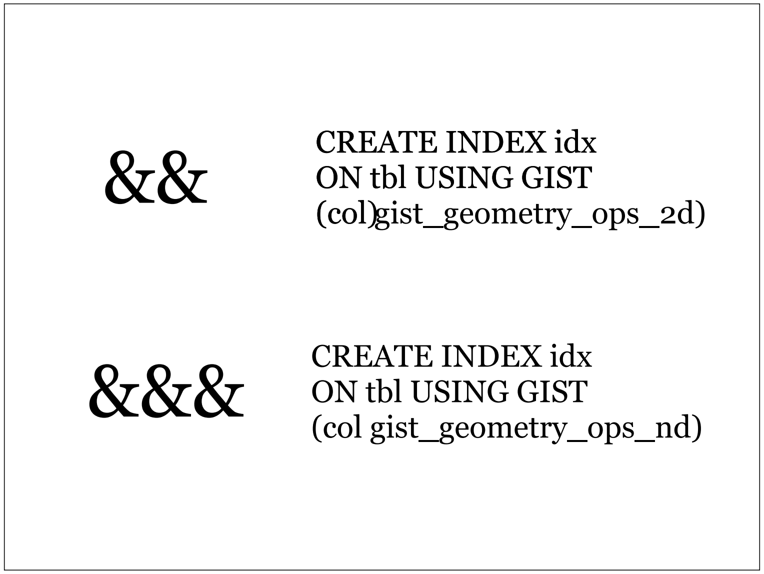
- ST_3dDistance(geom, geom)
- ST_3dLength(geom)
- ST_3dClosestPoint(geom, geom)
- ST_3dPerimeter(geom)
- ST_3dIntersects(geom, geom)
- ST_3dDWithin(geom, geom, tolerance)

The collection of 3D enabled functions has grown a great deal. Distance, length, nearest points, even intersects and within.



&& vs &&&

But all those functions won’t be good for much on large data sets, without support for 3D and 4D indexes, and good news, the new serialization means we can and do support high dimensional indexes.



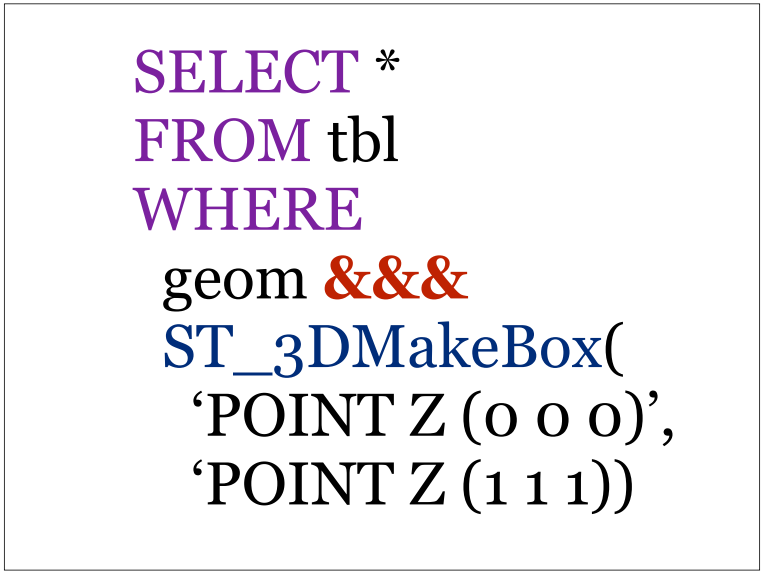
&&

```
CREATE INDEX idx
ON tbl USING GIST
(col)gist_geometry_ops_2d)
```

&&&

```
CREATE INDEX idx
ON tbl USING GIST
(col gist_geometry_ops_nd)
```

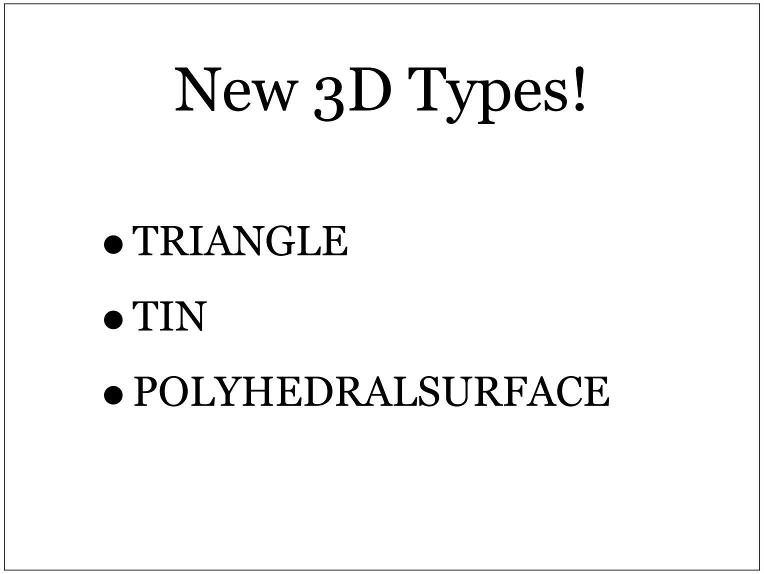
Creating a higher-dimension index looks almost exactly like creating a standard 2D one, the only difference is you have to specify your “opclass” as “gist_geometry_ops_nd”. You don’t have to specify opclass for 2D indexes, since the 2D opclass is the default, but it’s there under the covers.



```
SELECT *
FROM tbl
WHERE
  geom &&&
  ST_3DMakeBox(
    ‘POINT Z (0 0 0)’,
    ‘POINT Z (1 1 1))
```

So, an index-enabled 3D query

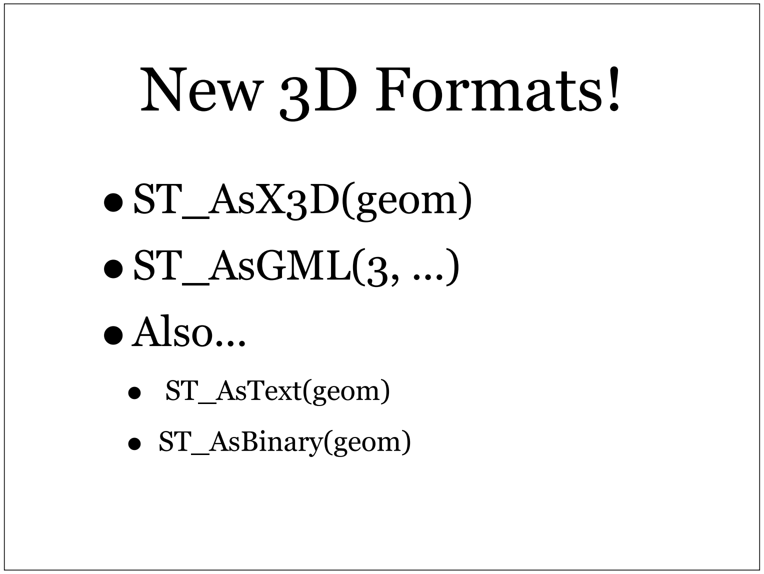
You’d think that with all these amazing changes, that would be it, but wait there’s more!....



New 3D Types!

- TRIANGLE
- TIN
- POLYHEDRALSURFACE

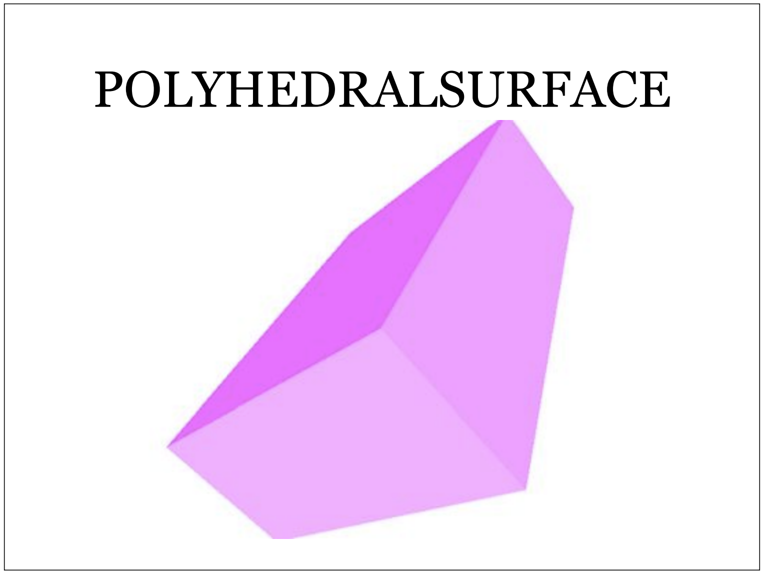
We also have 3D types to go with those new functions and indexes.



New 3D Formats!

- ST_AsX3D(geom)
- ST_AsGML(3, ...)
- Also...
  - ST_AsText(geom)
  - ST_AsBinary(geom)

And new 3D formats to write those 3D objects out to the wire.



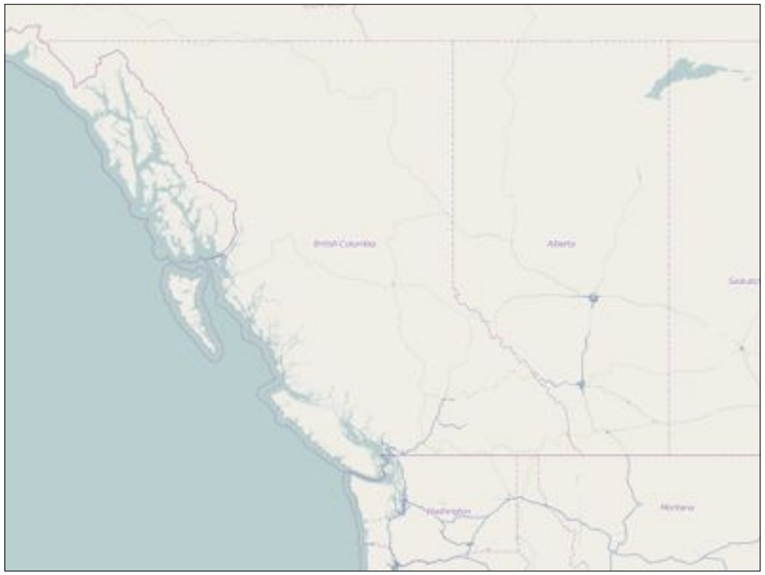
POLYHEDRALSURFACE

When I first heard about PolyhedralSurface, I asked “What the heck good is that?”

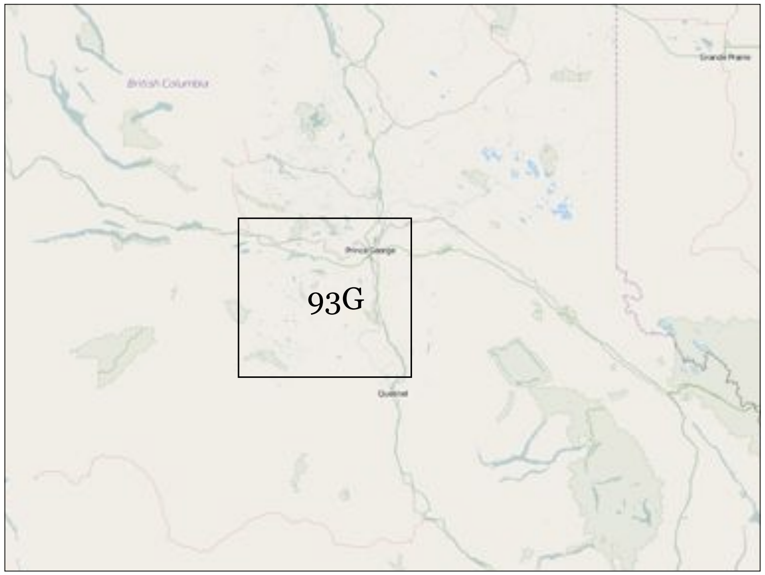
And I was told hey, what about 3D buildings. Yep!

That must be all, right? Heck no!
I wanted to show you some real examples of new PostGIS 2.0 features in action,

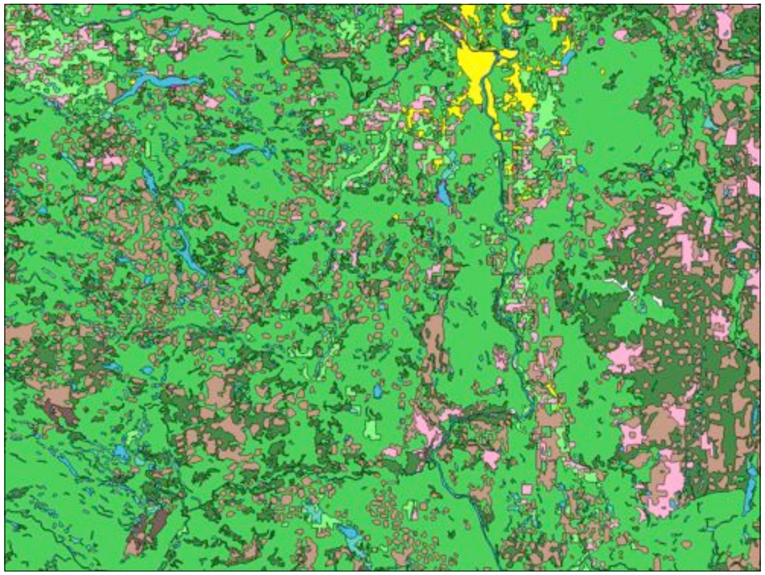
so I went to my favorite country, Canada



and favorite province, British Columbia

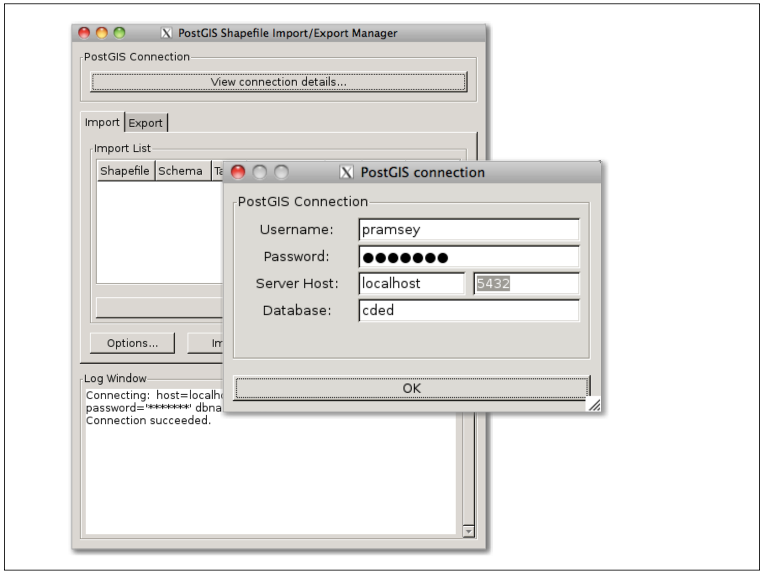


and favorite part of British Columbia, Prince George, and I downloaded some landuse data for one mapblock



It looks like this, the yellow is urban area, and the redish stuff is new logging

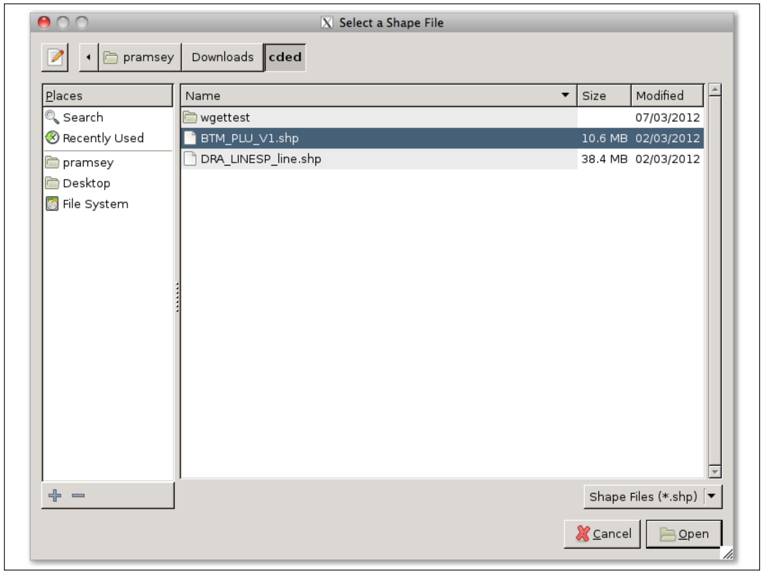
and I went to load it, and wow! there was even more new functionality... a new design for the shape GUI!



Username: pramsey
Password: ●●●●●●●
Server Host: localhost 5432
Database: cded

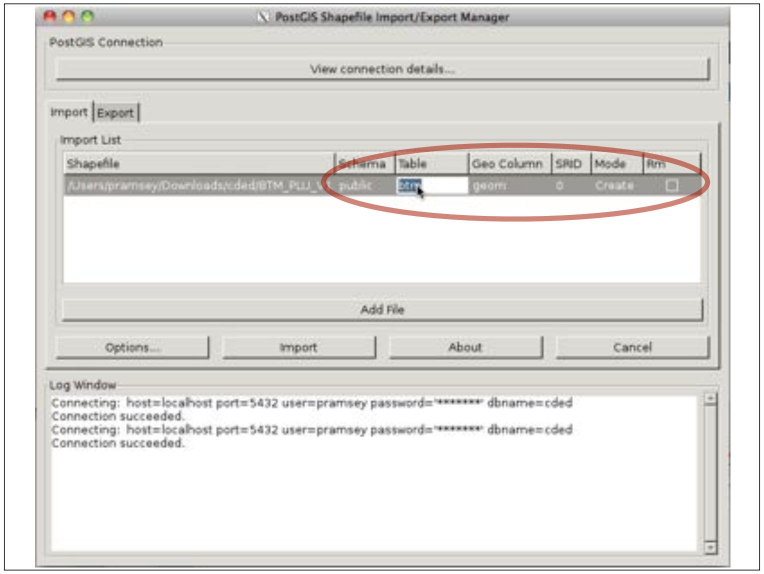
```
Connecting:  host=localh
password='********' dbna
Connection succeeded.
```

I connected to the database



Name	Size	Modified
wgettest		07/03/2012
BTM_PLU_V1.shp	10.6 MB	02/03/2012
DRA_LINESP_line.shp	38.4 MB	02/03/2012

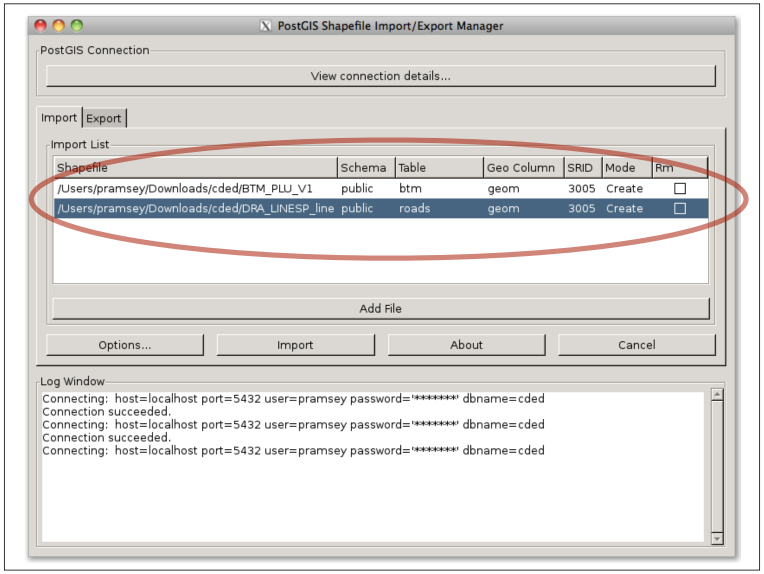
Chose my shape file



Shapefile	Schema	Table	Geo Column	SRID	Mode	Rm
/Users/pramsey/Downloads/cded/BTM_PLU_V	public	btm	geom	0	Create	□

```
Connecting:  host=localhost port=5432 user=pramsey password='*******' dbname=cded
Connection succeeded.
Connecting:  host=localhost port=5432 user=pramsey password='*******' dbname=cded
Connection succeeded.
```

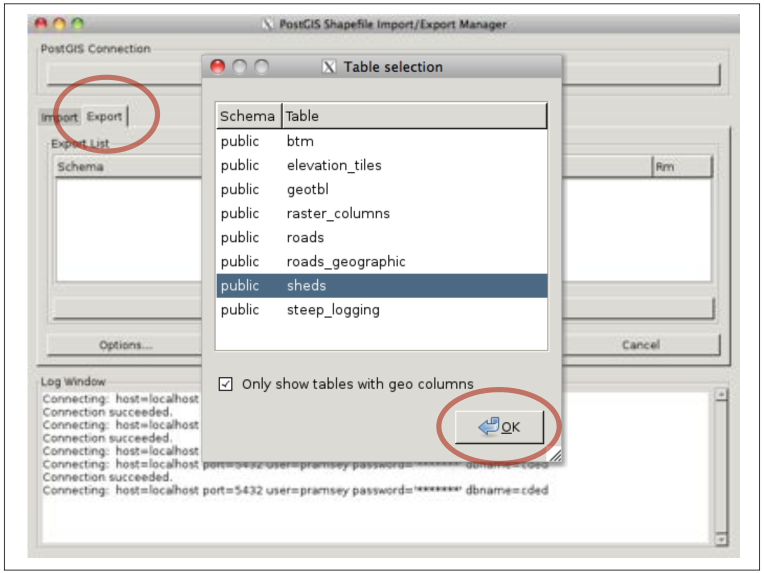
Set the target table name



Shapefile	Schema	Table	Geo Column	SRID	Mode	Rm
/Users/pramsey/Downloads/cded/BTM_PLU_V1	public	btm	geom	3005	Create	□
/Users/pramsey/Downloads/cded/DRA_LINESP_line	public	roads	geom	3005	Create	□

```
Connecting:  host=localhost port=5432 user=pramsey password='*******' dbname=cded
Connection succeeded.
Connecting:  host=localhost port=5432 user=pramsey password='*******' dbname=cded
Connection succeeded.
Connecting:  host=localhost port=5432 user=pramsey password='*******' dbname=cded
```

Added another shape file! Yes, the GUI now supports batch loading of multiple files. Then I clicked on “Import” and in it went.

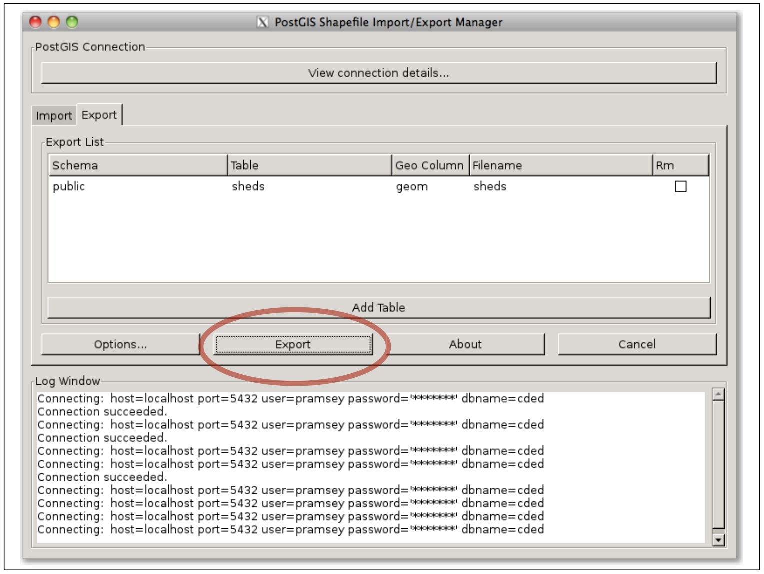


Schema	Table
public	btm
public	elevation_tiles
public	geotbl
public	raster_columns
public	roads
public	roads_geographic
public	sheds
public	steep_logging

☑ Only show tables with geo columns

```
Log Window
Connecting:  host=localhost
Connection succeeded.
Connecting:  host=localhost
Connection succeeded.
Connecting:  host=localhost
Connecting:  host=localhost port=5432 user=pramsey password='********' dbname=cded
Connection succeeded.
Connecting:  host=localhost port=5432 user=pramsey password='********' dbname=cded
```

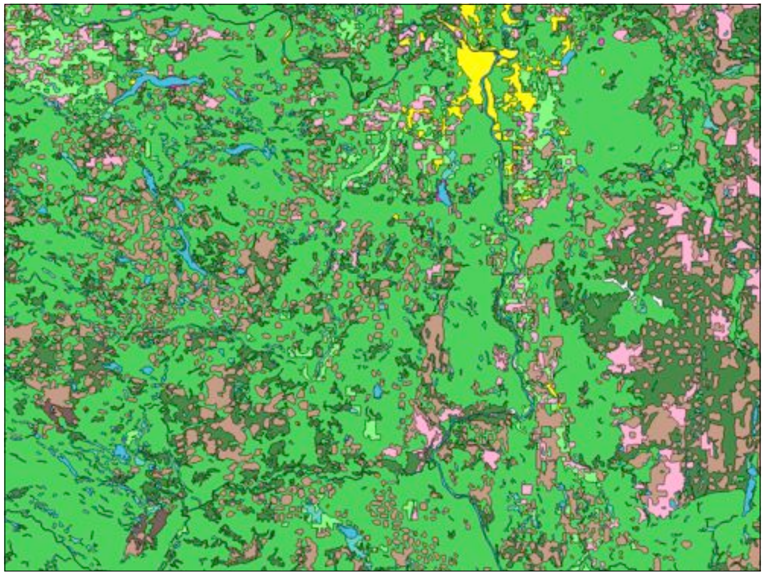
Close observers will have noticed an “Export” tab there too!
Click export, choose tables.



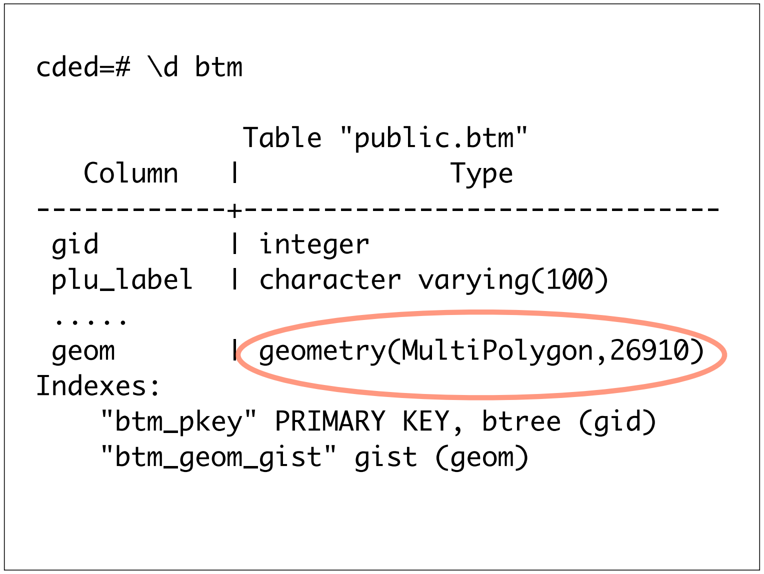
Schema	Table	Geo Column	Filename	Rm
public	sheds	geom	sheds	□

```
Log Window
Connecting:  host=localhost port=5432 user=pramsey password='********' dbname=cded
Connection succeeded.
Connecting:  host=localhost port=5432 user=pramsey password='********' dbname=cded
Connection succeeded.
Connecting:  host=localhost port=5432 user=pramsey password='********' dbname=cded
Connecting:  host=localhost port=5432 user=pramsey password='********' dbname=cded
Connection succeeded.
Connecting:  host=localhost port=5432 user=pramsey password='********' dbname=cded
Connecting:  host=localhost port=5432 user=pramsey password='********' dbname=cded
Connecting:  host=localhost port=5432 user=pramsey password='********' dbname=cded
Connecting:  host=localhost port=5432 user=pramsey password='********' dbname=cded
```

Hit the export button and out it goes!

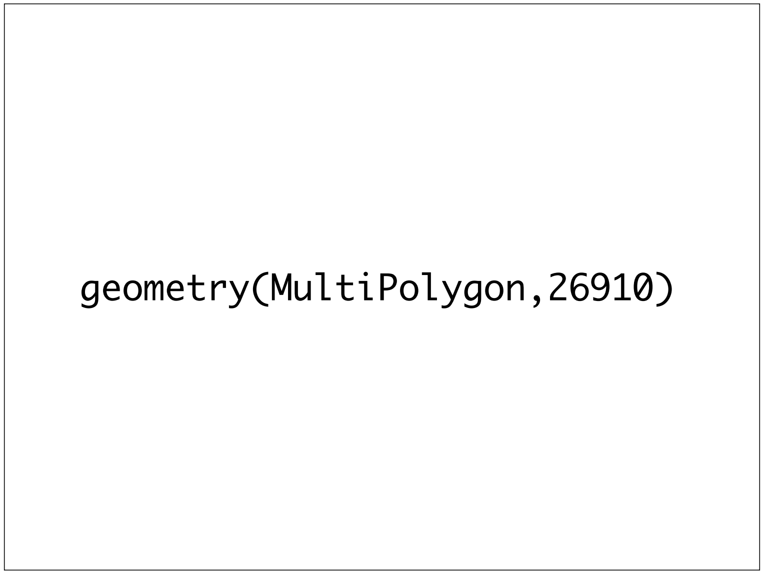


So, now my landuse data was loaded up.



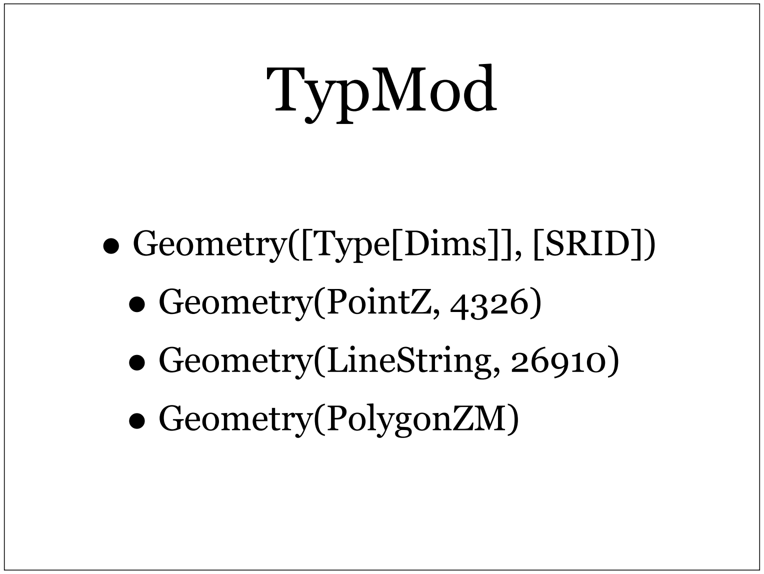
```
cded=# \d btm

             Table "public.btm"
   Column    |            Type
------------+-----------------------------
 gid         | integer
 plu_label   | character varying(100)
 .....
 geom        | geometry(MultiPolygon,26910)
Indexes:
    "btm_pkey" PRIMARY KEY, btree (gid)
    "btm_geom_gist" gist (geom)
```

Take a look at the table description...
The geom column is no longer just “geometry”



```
geometry(MultiPolygon,26910)
```

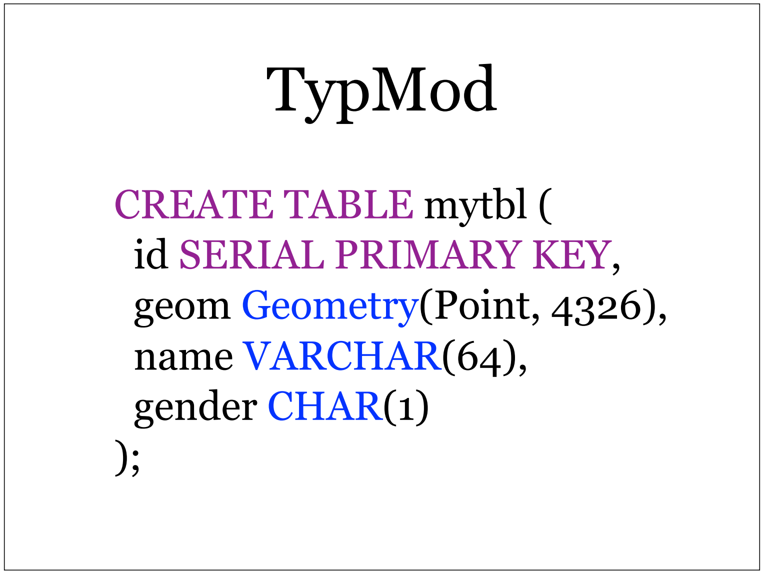
It’s “a multipolygon with srid 26910”!
This is enabled by the magical “typmod” improvement



TypMod

- Geometry([Type[Dims]], [SRID])
  - Geometry(PointZ, 4326)
  - Geometry(LineString, 26910)
  - Geometry(PolygonZM)

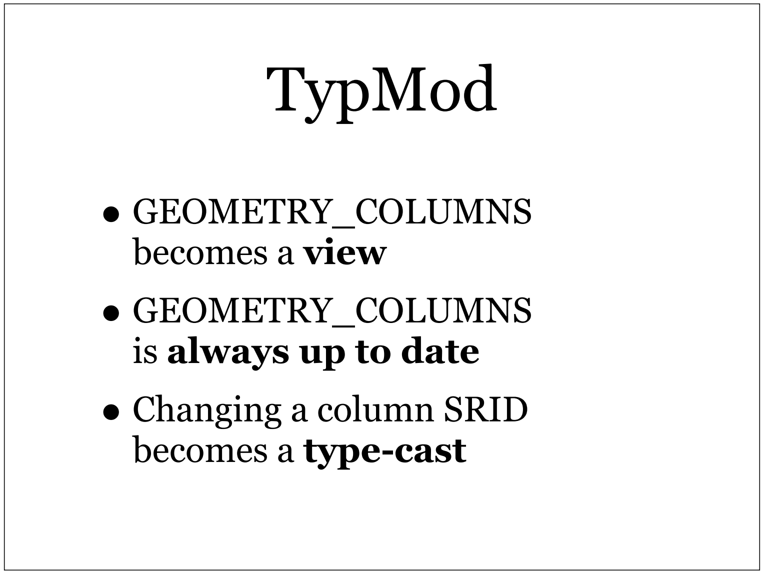
All geometry columns now have extra information in the PostgreSQL system tables, flagging the geometry type, the dimensionality and the spatial reference ID.



TypMod

```
CREATE TABLE mytbl (
  id SERIAL PRIMARY KEY,
  geom Geometry(Point, 4326),
  name VARCHAR(64),
  gender CHAR(1)
);
```

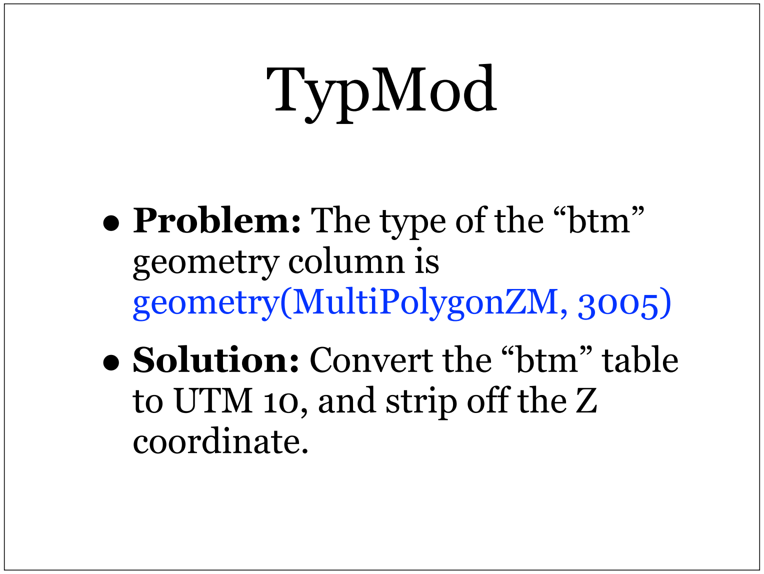
That means that it's possible to fully define a geometry column during the CREATE TABLE statement.



TypMod

- GEOMETRY_COLUMNS becomes a **view**
- GEOMETRY_COLUMNS is **always up to date**
- Changing a column SRID becomes a **type-cast**

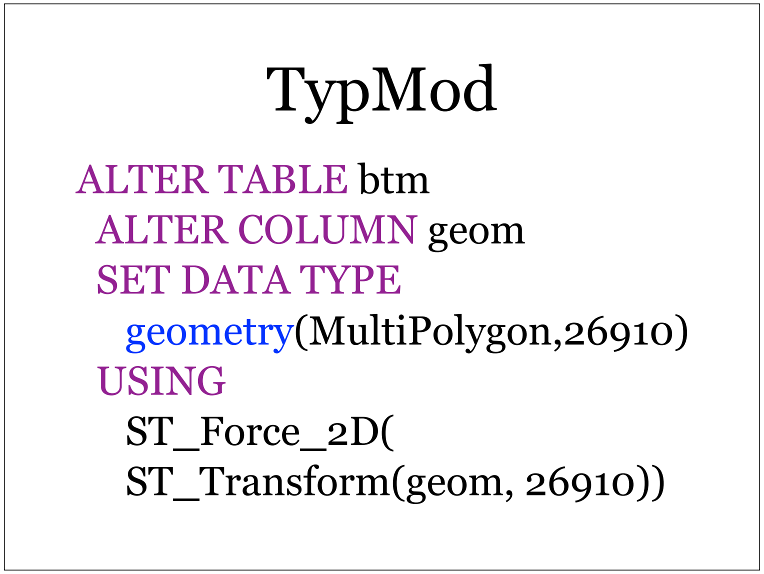
Which in turn means that the geometry_columns metadata table can be turned into a VIEW on the system tables. So it is always up to date! And even crazier, you can change geometry types and SRIDs for a table using typecasting in one step.



TypMod

- **Problem:** The type of the “btm” geometry column is geometry(MultiPolygonZM, 3005)
- **Solution:** Convert the “btm” table to UTM 10, and strip off the Z coordinate.

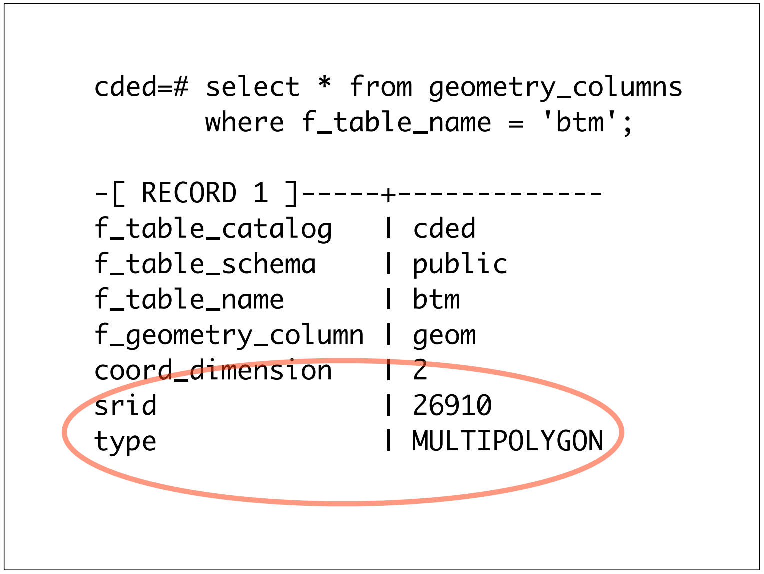
The classic problem is you import your data in one SRID and want to transform it to another SRID and geometry type. In PostGIS 1.5, solving the problem was a multi-step affair: constraints had to be dropped, table updates run, geometry_columns updated, and constraints re-added.



TypMod

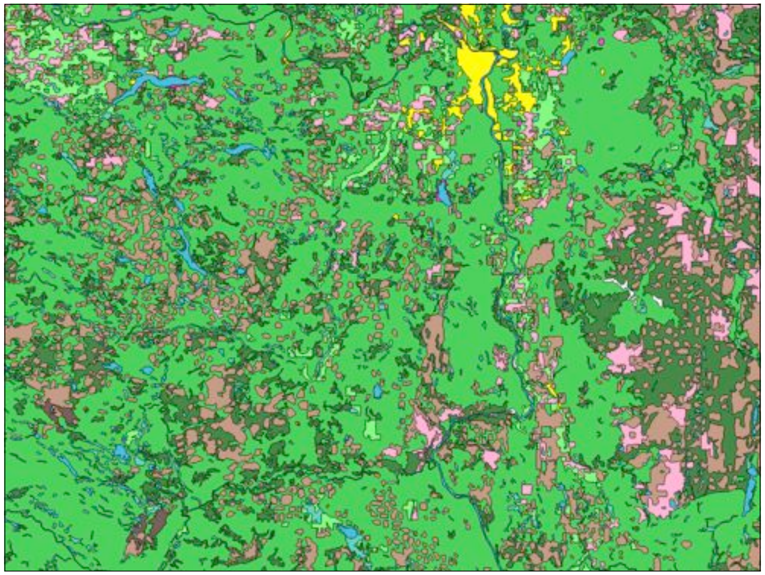
```
ALTER TABLE btm
 ALTER COLUMN geom
 SET DATA TYPE
   geometry(MultiPolygon,26910)
 USING
   ST_Force_2D(
   ST_Transform(geom, 26910))
```

Now it’s a one-step process.
Just alter the geometry column type, and supply the functions necessary to alter the data to match the new column type.



```
cded=# select * from geometry_columns
       where f_table_name = 'btm';

-[ RECORD 1 ]-----+-------------
f_table_catalog   | cded
f_table_schema    | public
f_table_name      | btm
f_geometry_column | geom
coord_dimension   | 2
srid              | 26910
type              | MULTIPOLYGON
```

After running the update, I check the geometry_columns view, and lo and behold the metadata matches the new SRID and geometry type automatically!

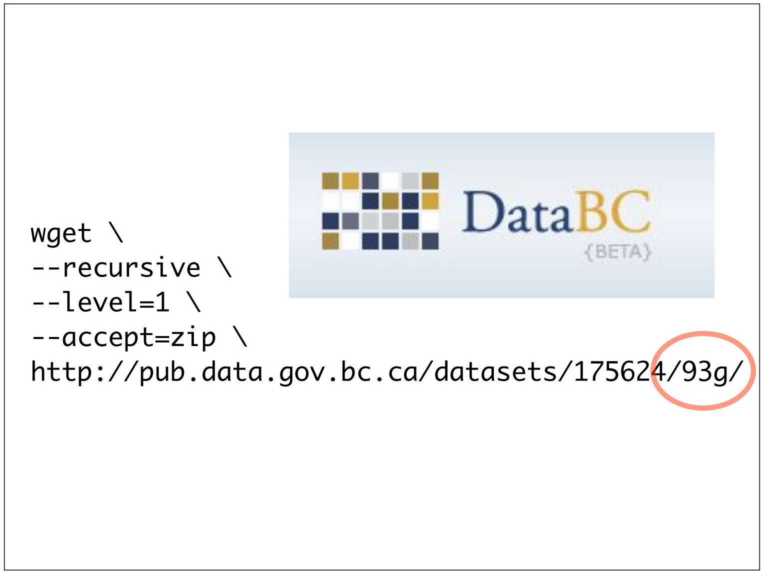


So, my data is in, I want to do some analysis with it...

I find your lack of raster slides...

...disturbing.

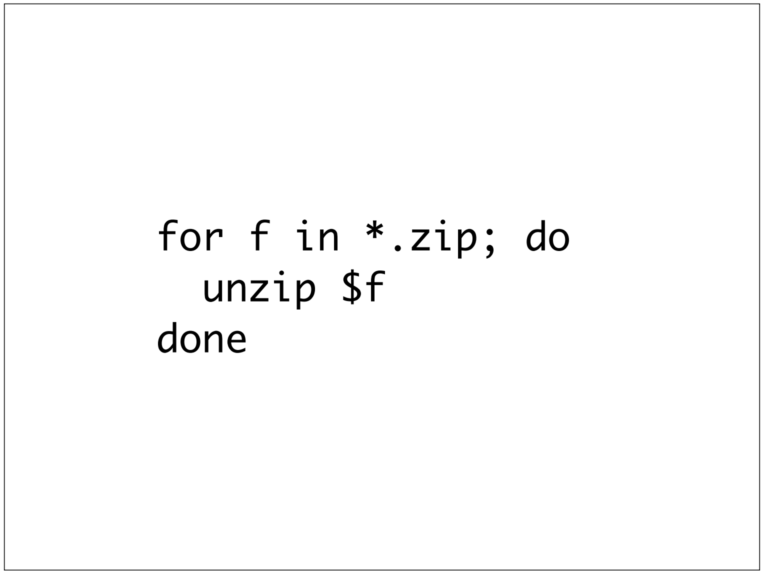
And this is a PostGIS 2.0 talk, and I know, I haven’t yet talked about the headline new features in PostGIS 2.0.



DataBC {BETA}

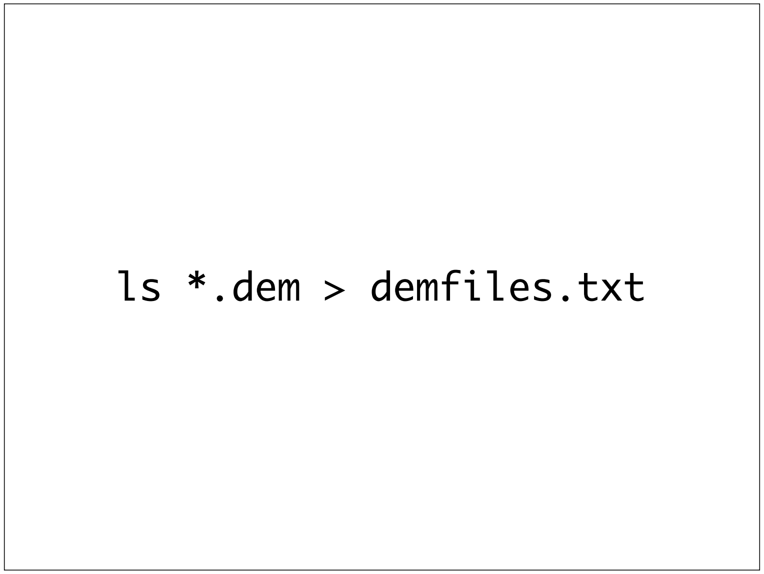
```
wget \
--recursive \
--level=1 \
--accept=zip \
http://pub.data.gov.bc.ca/datasets/175624/93g/
```

So here we go... I went to the BC open data site and downloaded all the elevation grids for my test map block.



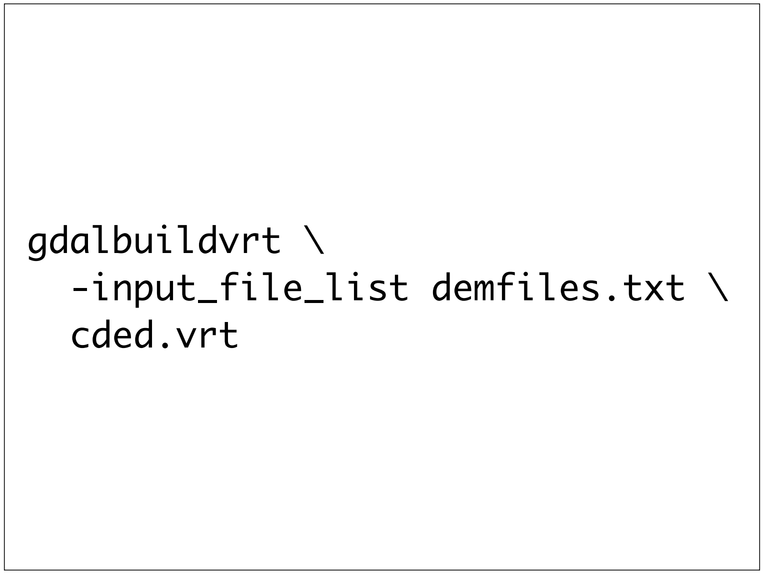
```
for f in *.zip; do
  unzip $f
done
```

Unzipped them all



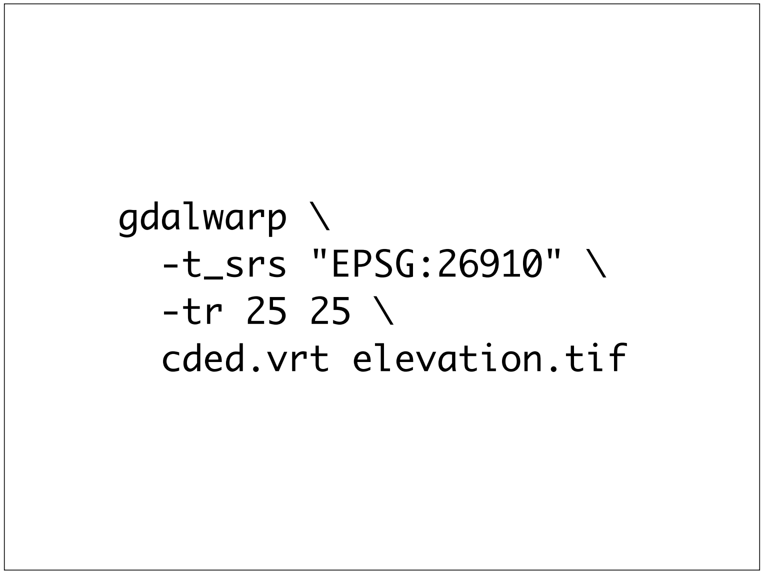
```
ls *.dem > demfiles.txt
```

Created a list of files.



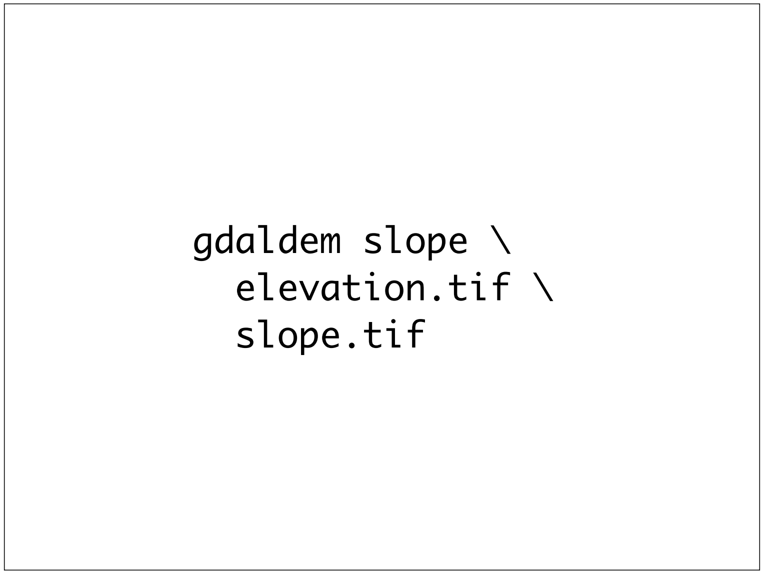
```
gdalbuildvrt \
  -input_file_list demfiles.txt \
  cded.vrt
```

Used that list of files to create a GDAL “Virtual Raster Table”



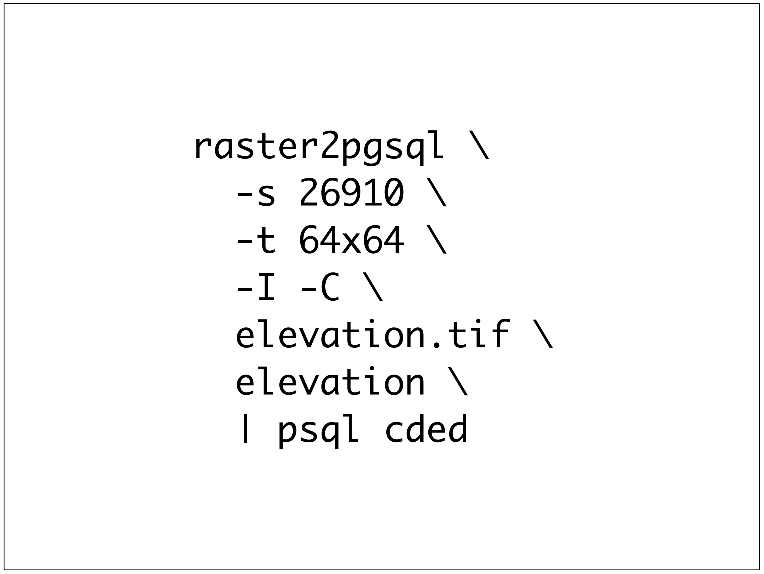
```
gdalwarp \
  -t_srs "EPSG:26910" \
  -tr 25 25 \
  cded.vrt elevation.tif
```

Converted that GDAL virtual raster into a unified elevation file.



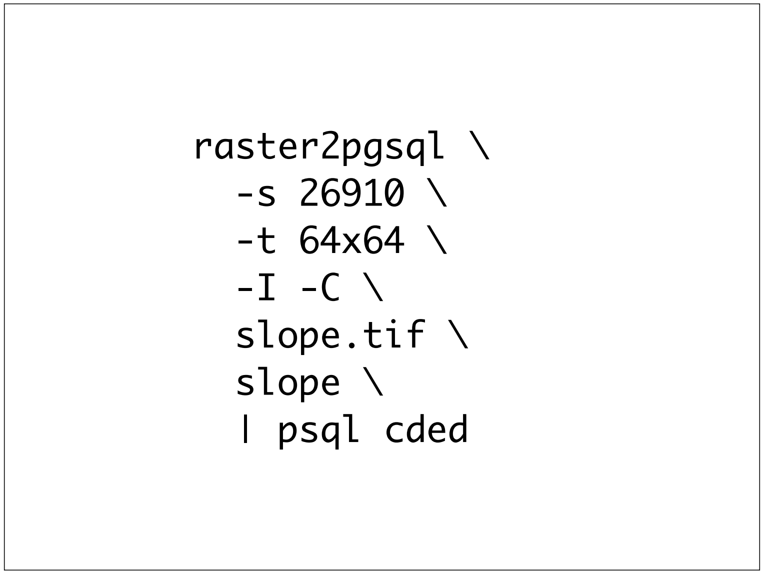
```
gdaldem slope \
  elevation.tif \
  slope.tif
```

Calculated the a slope grid from that raster file



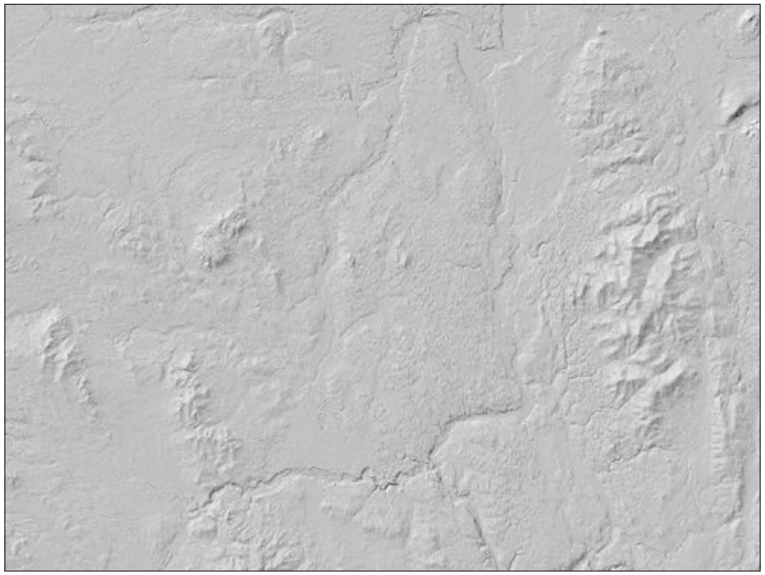
```
raster2pgsql \
  -s 26910 \
  -t 64x64 \
  -I -C \
  elevation.tif \
  elevation \
  | psql cded
```

Then loaded the elevation file into PostGIS raster using the new “raster2pgsql” data loading utility.

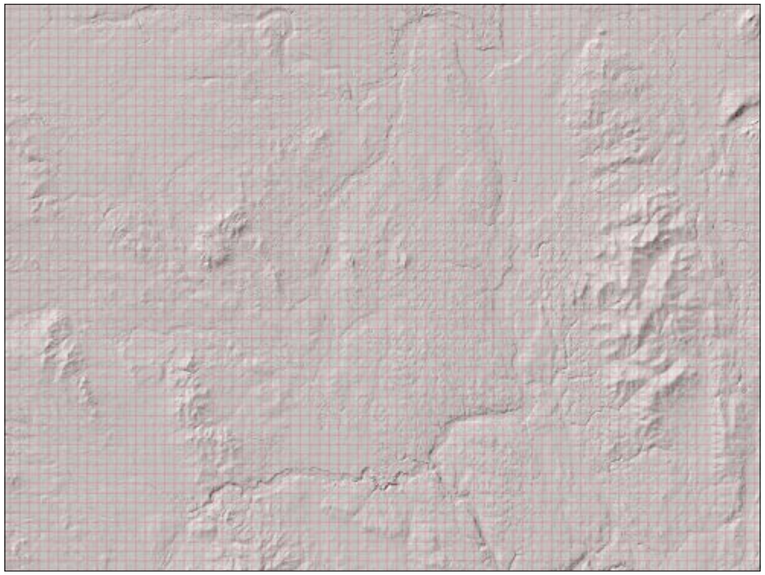


```
raster2pgsql \
  -s 26910 \
  -t 64x64 \
  -I -C \
  slope.tif \
  slope \
  | psql cded
```

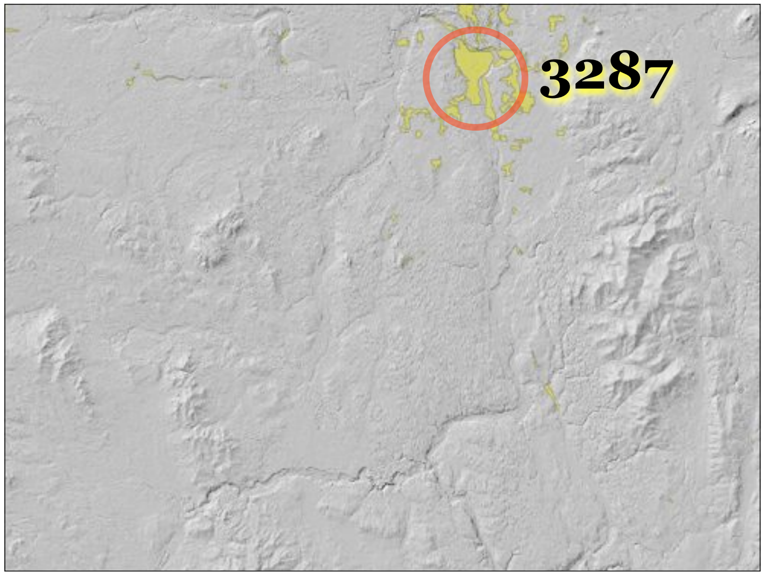
And also loaded the slope the same way.



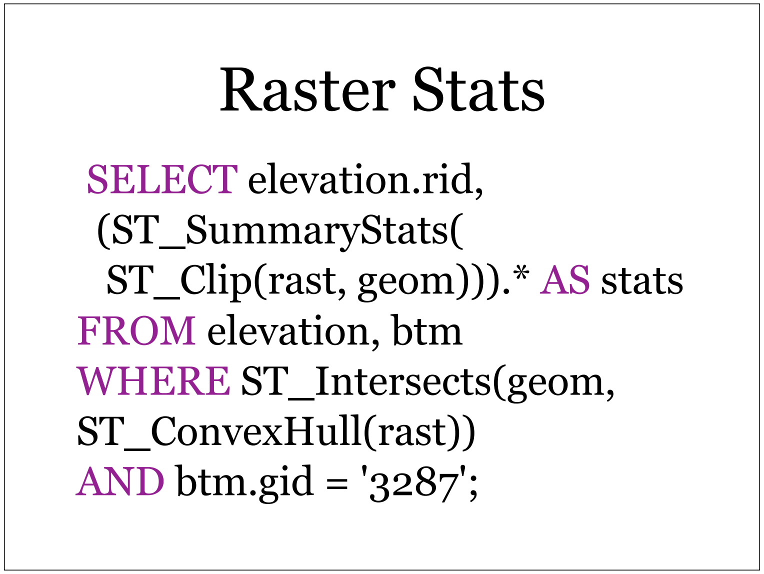
Boom, I had elevation and slope.



Actually I had this, thousands of wee slope and elevation tiles in slope and elevation tables.



I wanted to find out the elevation of my old home town, Prince George, so I identified the polygon that made up the urban area.

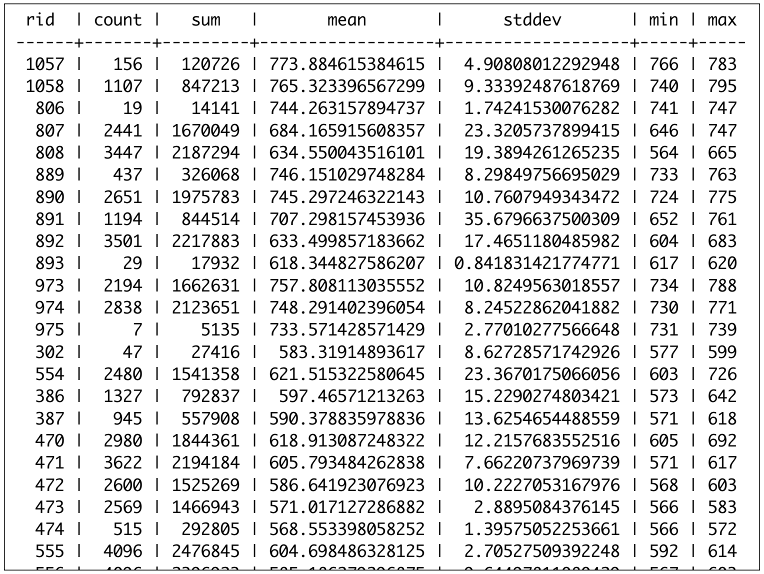


Raster Stats

```
SELECT elevation.rid,
 (ST_SummaryStats(
  ST_Clip(rast, geom))).* AS stats
FROM elevation, btm
WHERE ST_Intersects(geom,
ST_ConvexHull(rast))
AND btm.gid = '3287';
```

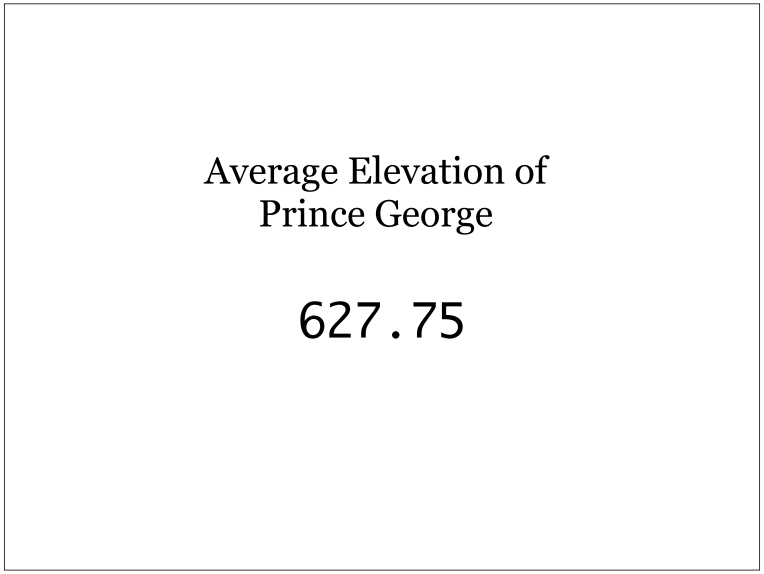
And ran this query to pull the elevation summary from the elevation table. Note that this is a join, between the elevation raster table and the land use vector table.

Also note the use of the ST_Clip() function that clips raster data to a vector geometry. From the clipped rasters, we pull summary statistics for each tile we found that intersected the urban area polygon.



```
 rid  | count |   sum   |       mean       |       stddev       | min | max
------+-------+---------+------------------+--------------------+-----+-----
 1057 |   156 |  120726 | 773.884615384615 |  4.90808012292948 | 766 | 783
 1058 |  1107 |  847213 | 765.323396567299 |  9.33392487618769 | 740 | 795
  806 |    19 |   14141 | 744.263157894737 |  1.74241530076282 | 741 | 747
  807 |  2441 | 1670049 | 684.165915608357 |  23.3205737899415 | 646 | 747
  808 |  3447 | 2187294 | 634.550043516101 |  19.3894261265235 | 564 | 665
  889 |   437 |  326068 | 746.151029748284 |  8.29849756695029 | 733 | 763
  890 |  2651 | 1975783 | 745.297246322143 |  10.7607949343472 | 724 | 775
  891 |  1194 |  844514 | 707.298157453936 |  35.6796637500309 | 652 | 761
  892 |  3501 | 2217883 | 633.499857183662 |  17.4651180485982 | 604 | 683
  893 |    29 |   17932 | 618.344827586207 | 0.841831421774771 | 617 | 620
  973 |  2194 | 1662631 | 757.808113035552 |  10.8249563018557 | 734 | 788
  974 |  2838 | 2123651 | 748.291402396054 |  8.24522862041882 | 730 | 771
  975 |     7 |    5135 | 733.571428571429 |  2.77010277566648 | 731 | 739
  302 |    47 |   27416 |  583.31914893617 |  8.62728571742926 | 577 | 599
  554 |  2480 | 1541358 | 621.515322580645 |  23.3670175066056 | 603 | 726
  386 |  1327 |  792837 |  597.46571213263 |  15.2290274803421 | 573 | 642
  387 |   945 |  557908 | 590.378835978836 |  13.6254654488559 | 571 | 618
  470 |  2980 | 1844361 | 618.913087248322 |  12.2157683552516 | 605 | 692
  471 |  3622 | 2194184 | 605.793484262838 |  7.66220737969739 | 571 | 617
  472 |  2600 | 1525269 | 586.641923076923 |  10.2227053167976 | 568 | 603
  473 |  2569 | 1466943 | 571.017127286882 |   2.8895084376145 | 566 | 583
  474 |   515 |  292805 | 568.553398058252 |  1.39575052253661 | 566 | 572
  555 |  4096 | 2476845 | 604.698486328125 |  2.70527509392248 | 592 | 614
```

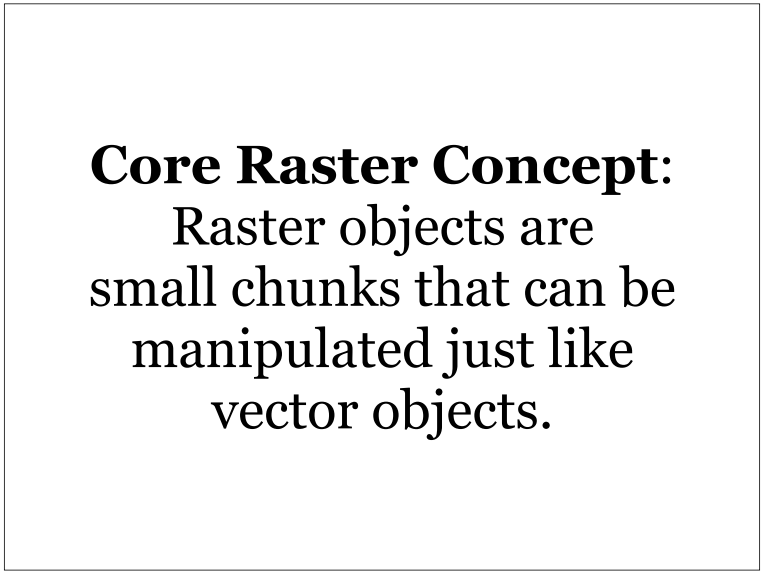
Boom! Why are there so many rows? Because the urban polygon intersected lots and lots of the little raster chips in the elevation table. To get the final answer the rows must be summarized... (more complex SQL) so



Average Elevation of
Prince George

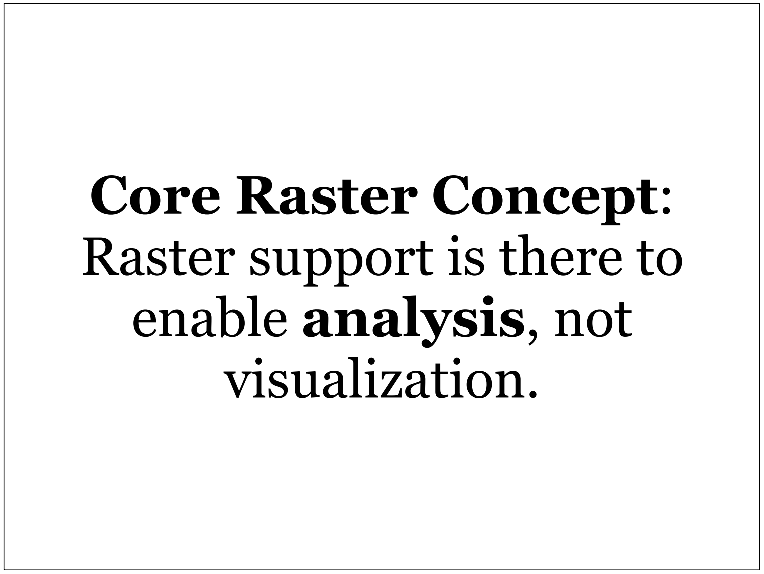
```
627.75
```

you’ll have to take my word that the answer is 627.75 meters. But look what we’ve done! A raster/vector analysis problem run entirely inside the database.



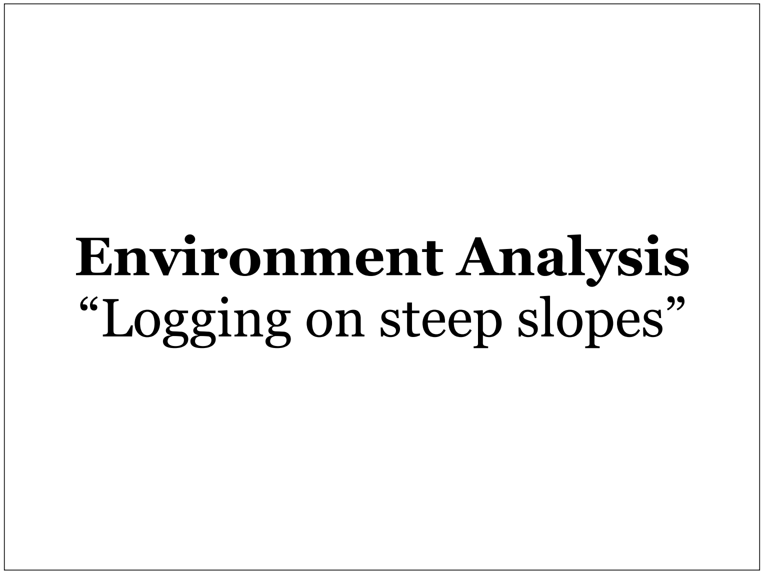
Core Raster Concept:
Raster objects are
small chunks that can be
manipulated just like
vector objects.

Two core concepts to remember working with raster. First, rasters are modelled as large collections of tiny chunks of raster.



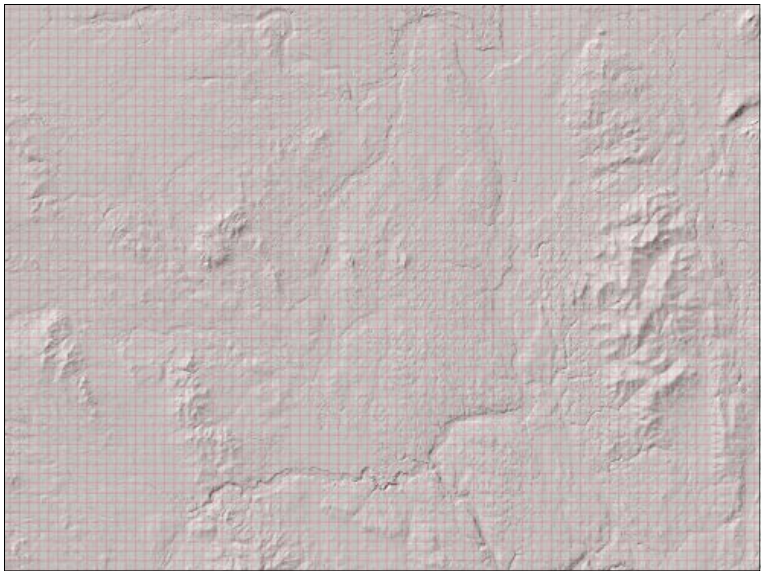
Core Raster Concept: Raster support is there to enable **analysis**, not visualization.

Second, the point of rasters in the database is to enable analysis, bringing together your raster and vector data to get an answer.

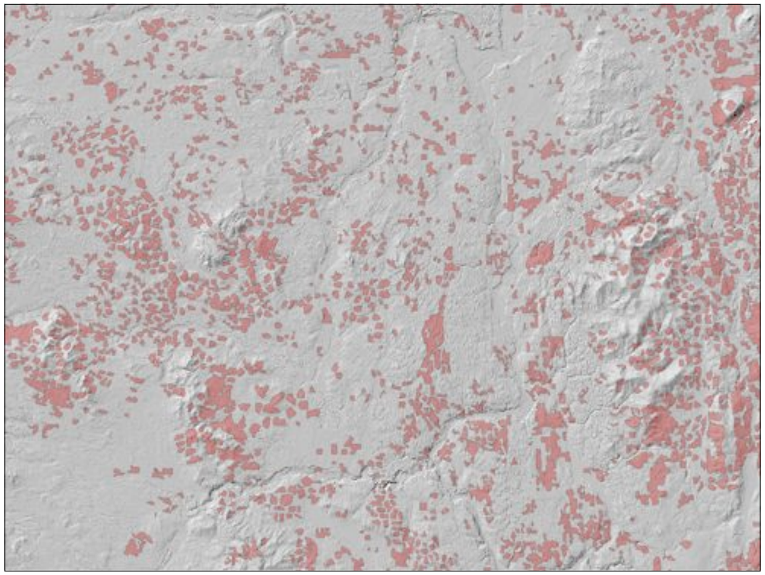


Environment Analysis
“Logging on steep slopes”

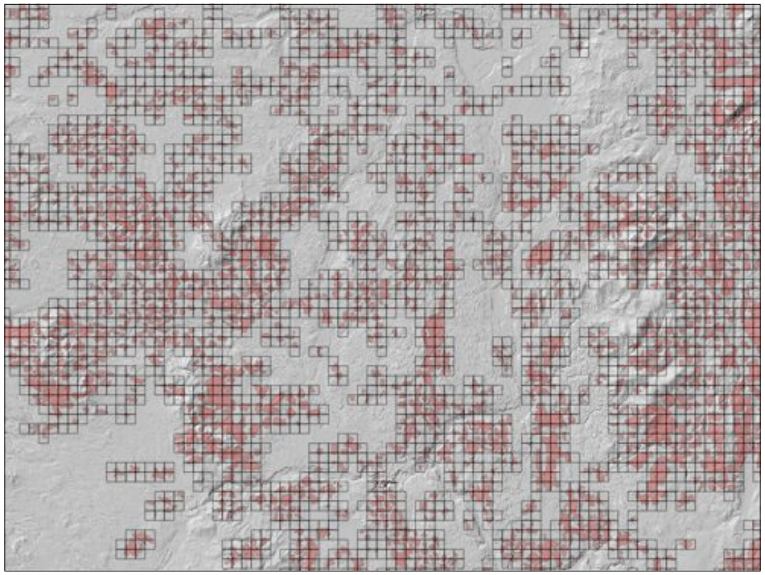
We can do real analyses with this data. We have a data set that shows where logging is. We have a data set of slopes. Logging on steep slopes is bad, because it allows greater run-off of top-soil and degrades future forest growth. Where is this happening in our area?



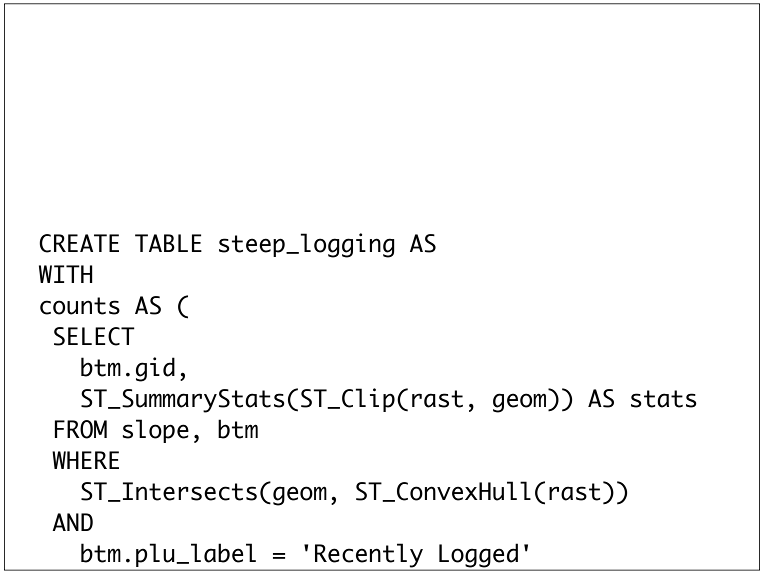
We have slope.



We have logging areas.

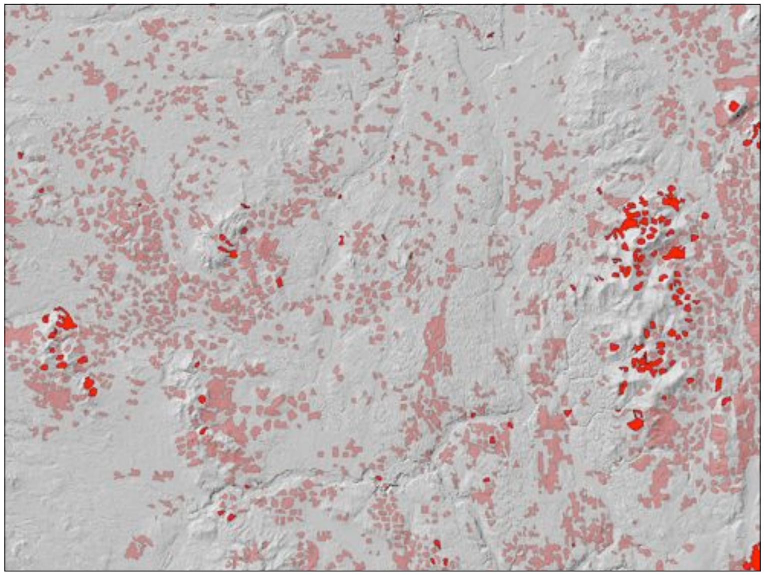


We can join the two tables, finding the slope grid chips that intersect logging areas. And then summarize to find the actual steep slope logging.

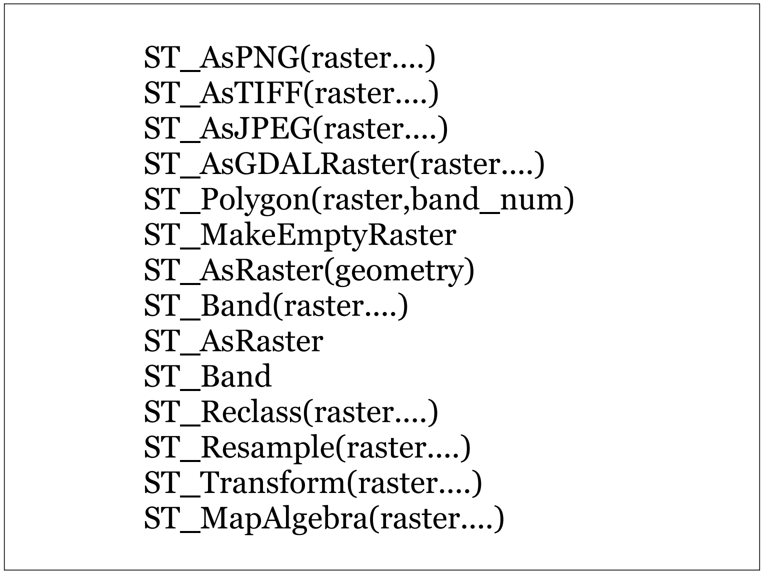


```
CREATE TABLE steep_logging AS
WITH
counts AS (
 SELECT
   btm.gid,
   ST_SummaryStats(ST_Clip(rast, geom)) AS stats
 FROM slope, btm
 WHERE
   ST_Intersects(geom, ST_ConvexHull(rast))
 AND
   btm.plu_label = 'Recently Logged'
```

The SQL is... a bit complex.

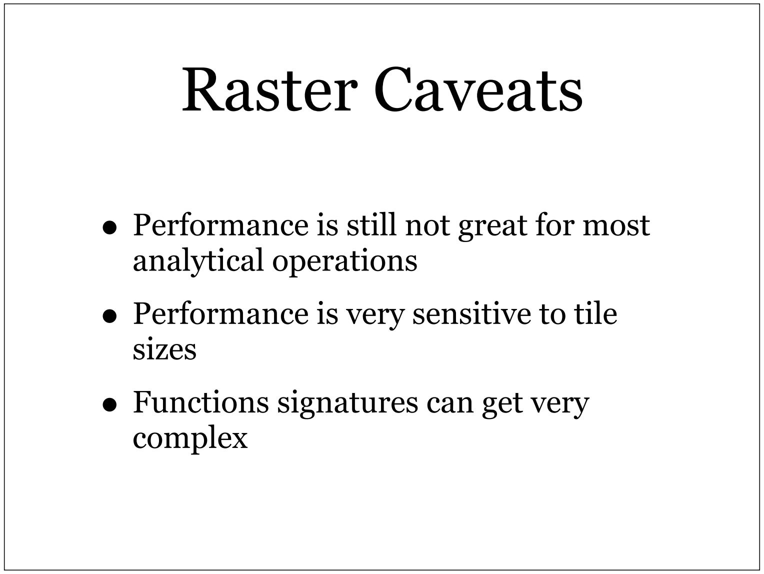


But it works, and we get a result, inside the database!



```
ST_AsPNG(raster....)
ST_AsTIFF(raster....)
ST_AsJPEG(raster....)
ST_AsGDALRaster(raster....)
ST_Polygon(raster,band_num)
ST_MakeEmptyRaster
ST_AsRaster(geometry)
ST_Band(raster....)
ST_AsRaster
ST_Band
ST_Reclass(raster....)
ST_Resample(raster....)
ST_Transform(raster....)
ST_MapAlgebra(raster....)
```

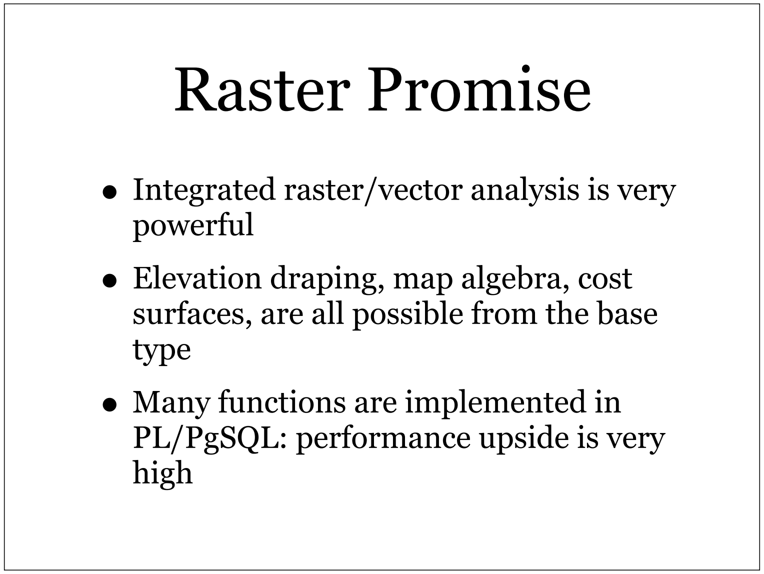
There are lots of new functions in PostGIS 2.0 for handling rasters, including fancy things like output to image formats, polygonization, reprojection, and even map algebra.



Raster Caveats

- Performance is still not great for most analytical operations
- Performance is very sensitive to tile sizes
- Functions signatures can get very complex

Even though this is PostGIS 2.0, it’s important to remember that raster is a brand new feature. If it was not being released inside PostGIS, it would probably have a number like 0.5. There is much work still to do.

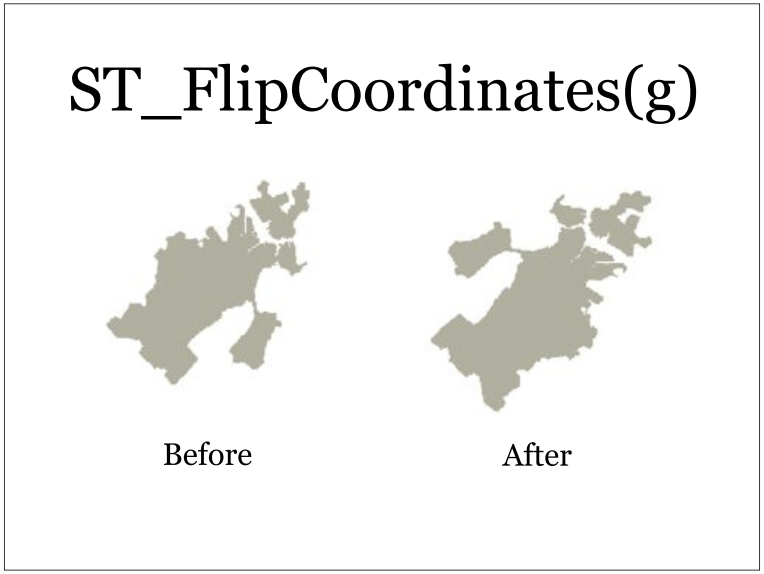


Raster Promise

- Integrated raster/vector analysis is very powerful
- Elevation draping, map algebra, cost surfaces, are all possible from the base type
- Many functions are implemented in PL/PgSQL: performance upside is very high

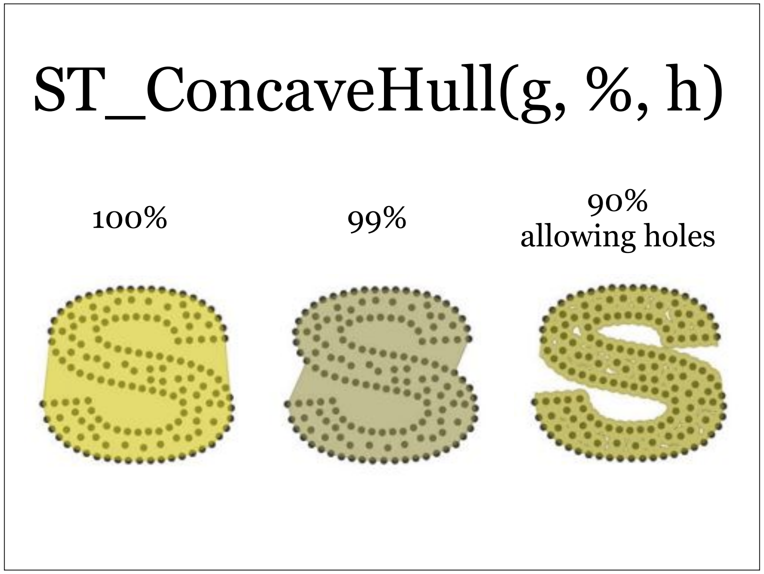
However, the potential is really big. Integrated raster/vector analysis is powerful, new features like draping and cost surfaces can be built on the new type, and the performance enhancements still to be done are not rocket science, they are re-implementations of functions in native C.

But that’s not all! There’s still more new features! Here’s the grab bag of new and gimmicky features.



ST_FlipCoordinates(g)

You can flip your X and Y coordinates. Useful when you load long/lat data as lat/long!



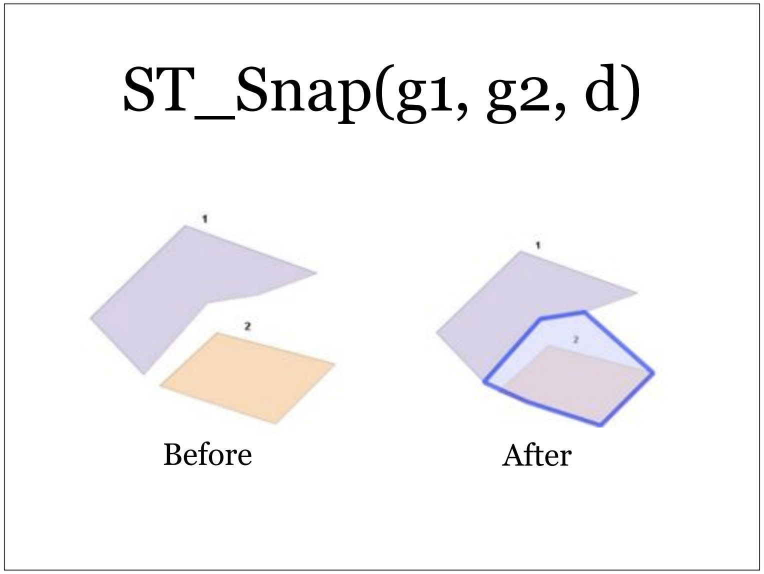
ST_ConcaveHull(g, %, h)

100%

99%

90%
allowing holes

You can generate hulls that shrinkwrap the input features, a concave rather than convex hull.

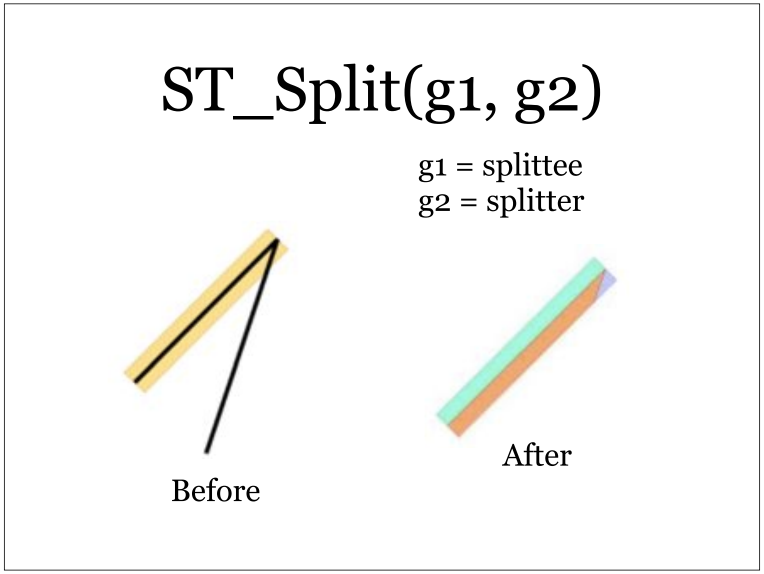


ST_Snap(g1, g2, d)

Before

After

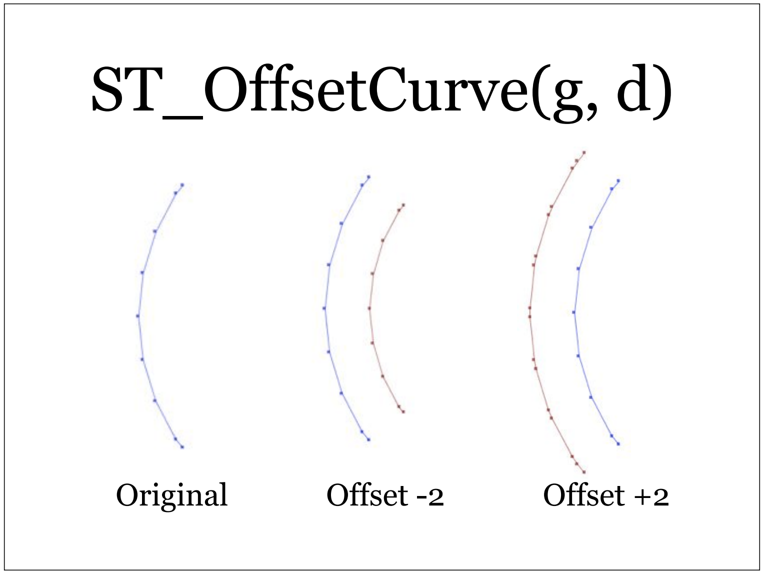
You can snap nearby features together (work in progress).



ST_Split(g1, g2)

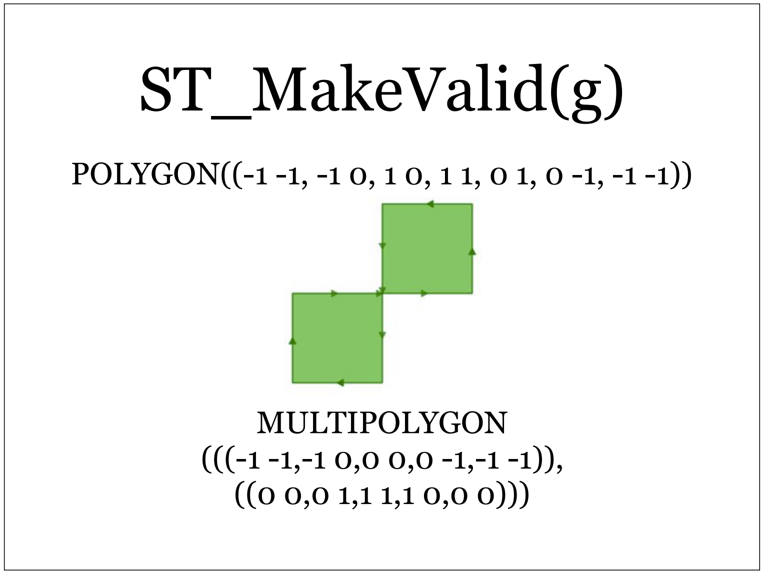
g1 = splittee
g2 = splitter

You can split a polygon using an input line.



ST_OffsetCurve(g, d)

You can generate offset curves to the left or right of input lines.

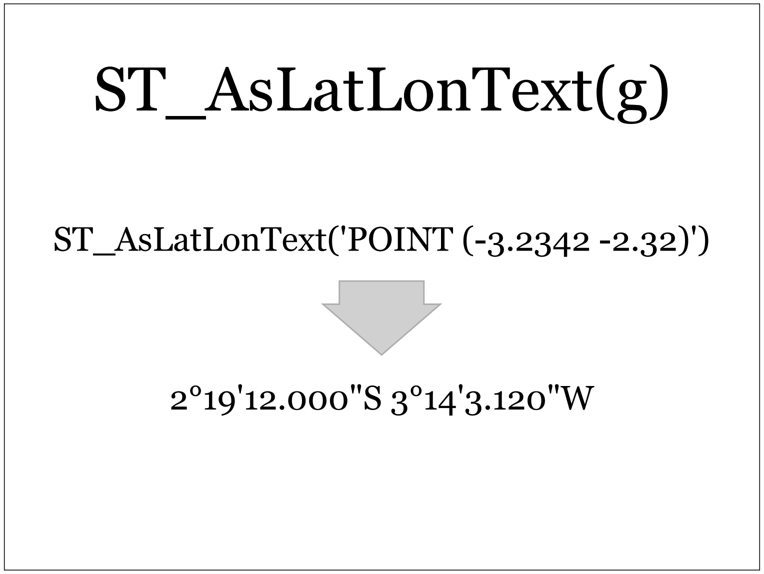


ST_MakeValid(g)

POLYGON((-1 -1, -1 0, 1 0, 1 1, 0 1, 0 -1, -1 -1))

MULTIPOLYGON
(((-1 -1,-1 0,0 0,0 -1,-1 -1)),
((0 0,0 1,1 1,1 0,0 0)))

And you can finally do something about invalid features! ST_MakeValid will even fix my favorite invalid polygon, the figure eight.

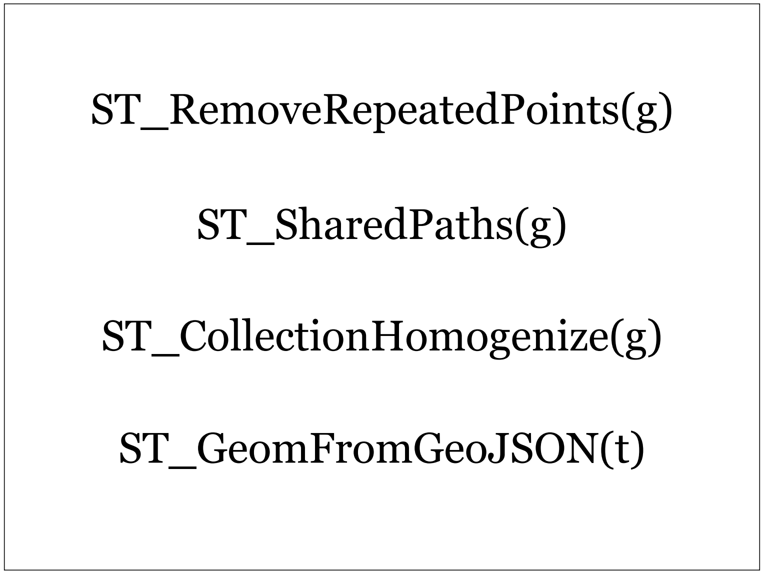


ST_AsLatLonText(g)

ST_AsLatLonText('POINT (-3.2342 -2.32)')

2°19'12.000"S 3°14'3.120"W

No more writing lat/lon output functions in PHP, you can use an in-built function to get all kinds of standard formatting for lat/lon coordinates.



ST_RemoveRepeatedPoints(g)

ST_SharedPaths(g)

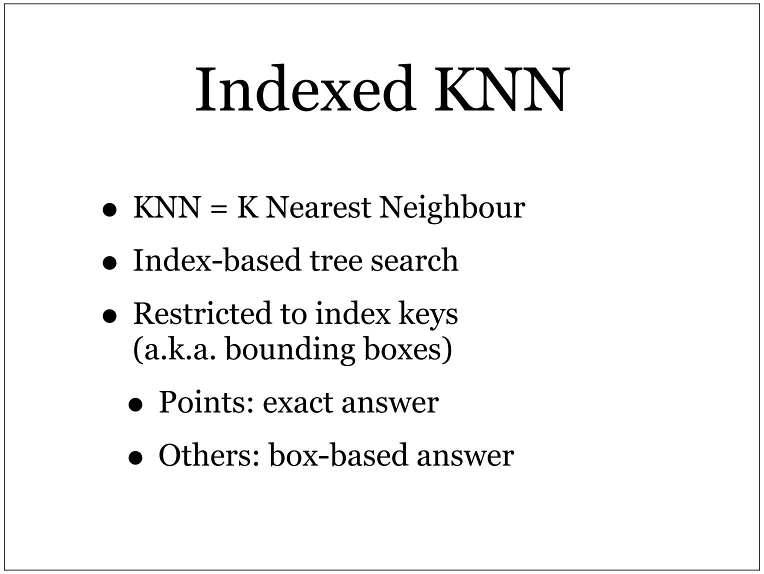
ST_CollectionHomogenize(g)

ST_GeomFromGeoJSON(t)

And that’s not even mentioning removing repeated points, or finding co-joint lines, or cleaning collections, or creating geometries from JSON inputs. Wow!

Are we done? Hell no!

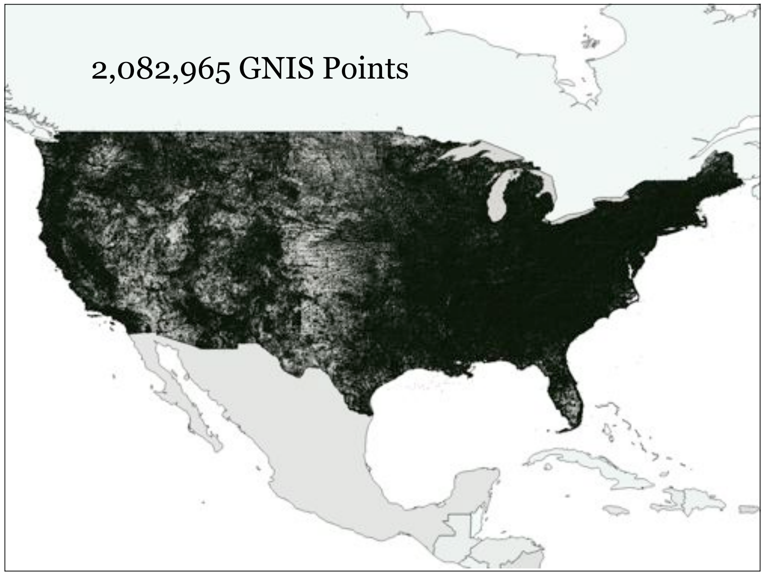
Having a good neighbor is important, and knowing your nearest neighbors is very very useful too!



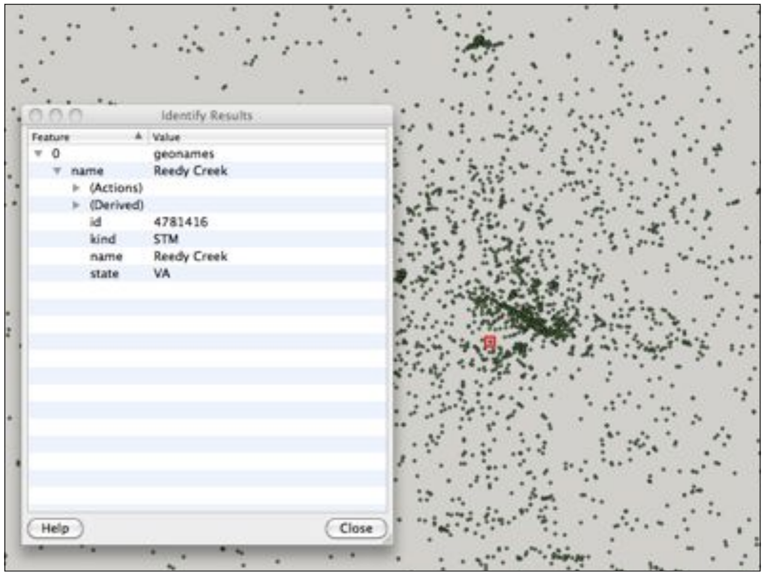
Indexed KNN

- KNN = K Nearest Neighbour
- Index-based tree search
- Restricted to index keys (a.k.a. bounding boxes)
    - Points: exact answer
    - Others: box-based answer

PostGIS 2.0 how has support for nearest-neighbor indexed searching. For very large tables, with irregular densities, this can be a huge performance win.

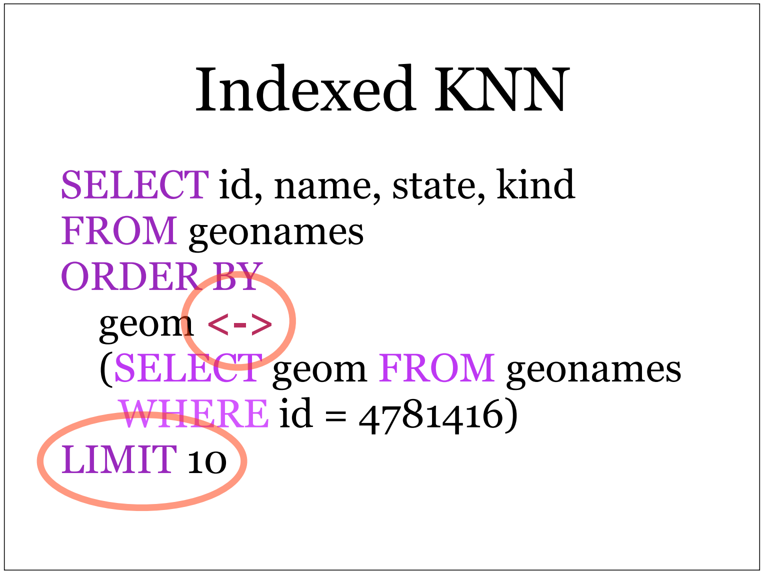


So, here’s an example I put together, loading all the USA named geographic points, 2M of them.



Feature	Value
▼ 0	geonames
▼ name	Reedy Creek
▶ (Actions)	
▶ (Derived)	
id	4781416
kind	STM
name	Reedy Creek
state	VA

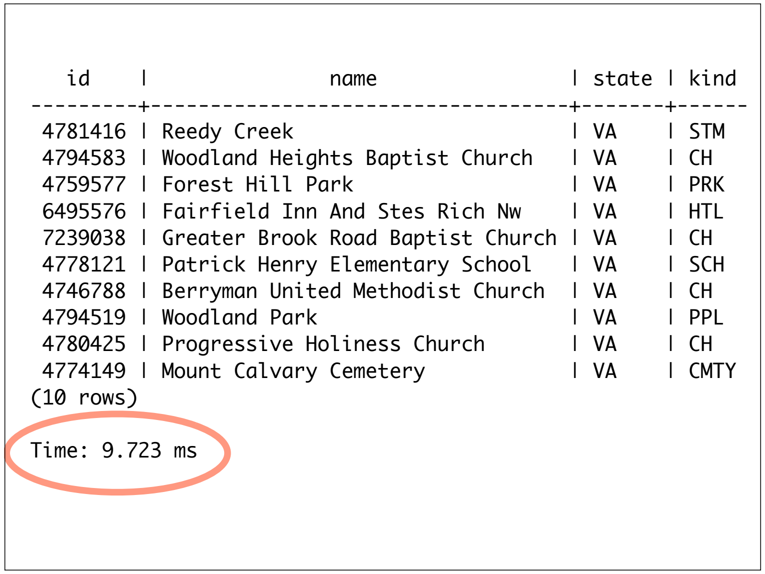
Find one point, in this case Reedy Creek.



Indexed KNN

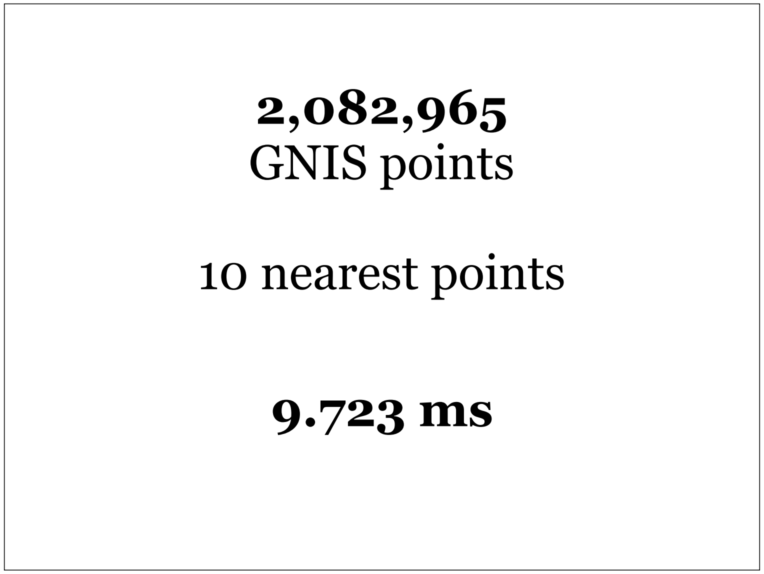
```
SELECT id, name, state, kind
FROM geonames
ORDER BY
   geom <->
   (SELECT geom FROM geonames
    WHERE id = 4781416)
LIMIT 10
```

Here’s how we find the 10 nearest names to Reedy Creek. Note the use of the funny arrow-like operator in the ORDER BY clause and the LIMIT. You have to use ORDER BY and you have to LIMIT.



```
   id    |                name                | state | kind
---------+------------------------------------+-------+------
 4781416 | Reedy Creek                        | VA    | STM
 4794583 | Woodland Heights Baptist Church    | VA    | CH
 4759577 | Forest Hill Park                   | VA    | PRK
 6495576 | Fairfield Inn And Stes Rich Nw     | VA    | HTL
 7239038 | Greater Brook Road Baptist Church  | VA    | CH
 4778121 | Patrick Henry Elementary School    | VA    | SCH
 4746788 | Berryman United Methodist Church   | VA    | CH
 4794519 | Woodland Park                      | VA    | PPL
 4780425 | Progressive Holiness Church        | VA    | CH
 4774149 | Mount Calvary Cemetery             | VA    | CMTY
(10 rows)

Time: 9.723 ms
```

But most importantly, note how fast we get back the 10 nearest entries from this 2M record table.

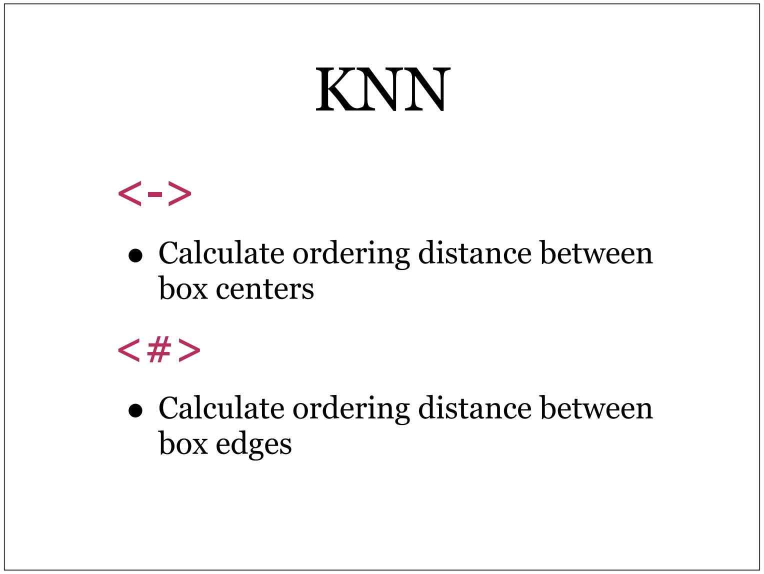


2,082,965
GNIS points

10 nearest points

9.723 ms

Again for emphases. 2M points. 10 nearest. 9.7ms.



KNN

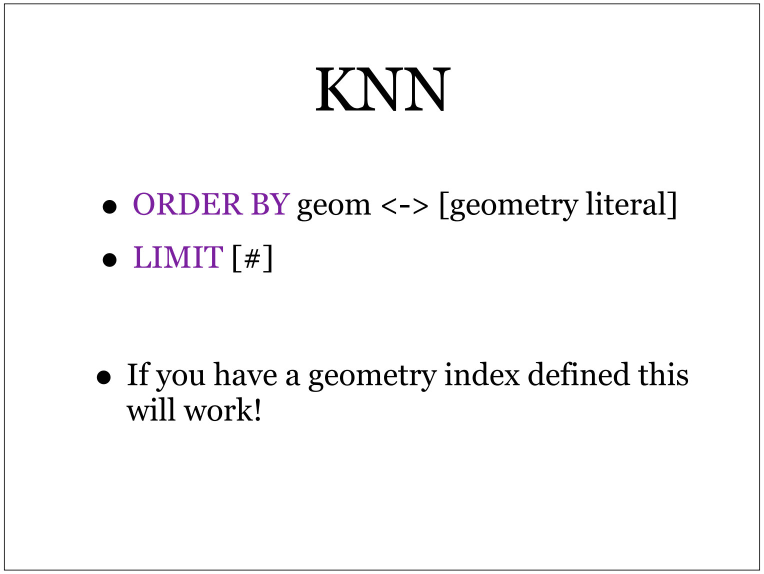
<->

- Calculate ordering distance between box centers

<#>

- Calculate ordering distance between box edges

Because KNN searches the index, and the index is bounding box based, the operators work on box distances. There are two ways to measure distance between two boxes: the distance between the box centers (the arrow operator), and the distance between the nearest box edges (the box operator).



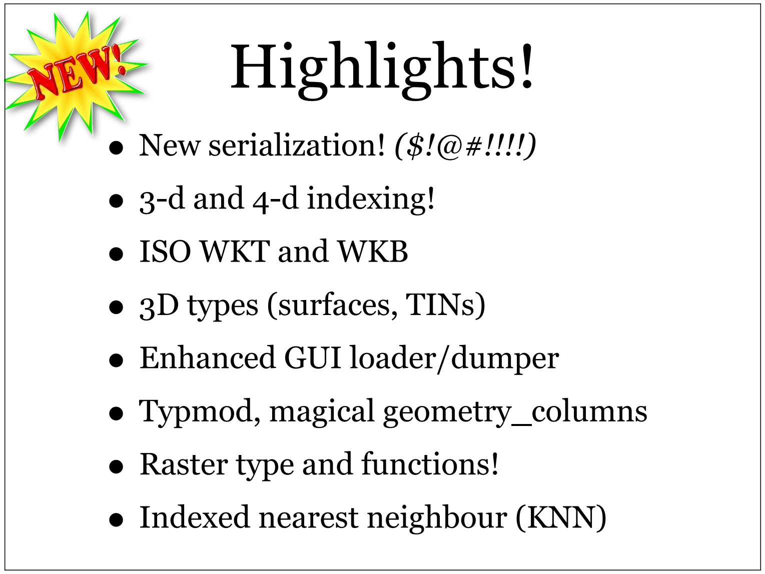
KNN

- ORDER BY geom <-> [geometry literal]
- LIMIT [#]

- If you have a geometry index defined this will work!

As long as you have a geometry index defined, and PostgreSQL >= 9.0 this will work!

Thanks neighbor!

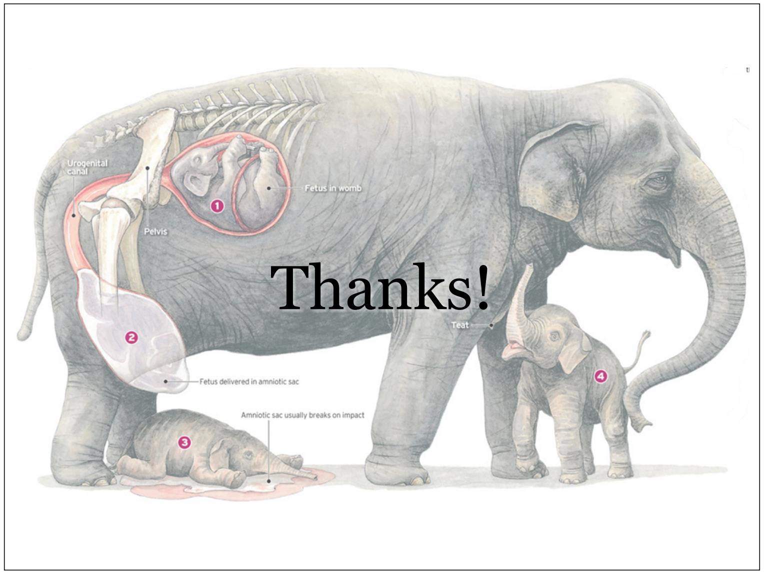


Highlights!

- New serialization! *($!@#!!!!)*
- 3-d and 4-d indexing!
- ISO WKT and WKB
- 3D types (surfaces, TINs)
- Enhanced GUI loader/dumper
- Typmod, magical geometry_columns
- Raster type and functions!
- Indexed nearest neighbour (KNN)

So to recap!
....
And the very best part of PostGIS 2.0?

It comes with a 100% money back guarantee!



Thanks!

Urogenital canal
Fetus in womb
1
Pelvis
2
Teat
4
Fetus delivered in amniotic sac
Amniotic sac usually breaks on impact
3

Thanks, and I promise our next release will come faster than a baby elephant!