

# LIDAR in PostgreSQL with PointCloud

Paul Ramsey <[pramsey@opengeo.org](mailto:pramsey@opengeo.org)>



Over the past 5 months, I have been working on the building blocks for storing LIDAR data in PostgreSQL, and leveraging it for analysis with PostGIS. That means LIDAR types and functions in the database, and loading utilities to get data into and out of the database.



This development has been largely funded by Natural Resources Canada, who are planning to use PostgreSQL and their database for managing national LIDAR inventories.

# Motivation

So, why would anyone want to put LIDAR in a database? What's the motivation here?



**Billions and Billions**

First, you can't just stuff LIDAR point clouds into existing PostGIS types like the Point type, there's just too much of it. A county can generate hundreds of millions of points, a state can generate billions.

(X, Y, Z)

Second, LIDAR is multi-dimensional. And not just X,Y,Z.

(X, Y, Z, Intensity,  
ReturnNumber,  
NumberOfReturns,  
Classification,  
ScanAngleRank, Red,  
Green, Blue)

A dozen or more dimensions PER POINT, is not unusual. Unfortunately, the multidimensionality of LIDAR is not fixed either. Some LIDAR clouds have four dimensions. Others have fourteen. So billions of points with many dimensions: you can't stuff this into existing PostGIS or columnar tables.

Everything is related to everything else, but near things are more related than distant things.

But we don't want to just throw up our hands, because LIDAR point clouds have geographic location, which means if we can get them into a spatial database, we can mash them up with other spatial entities, thanks to Tobler's Law. So there's **value** in the exercise.

## Demotivation

On the other hand, I'm on the record as not wanting to put **rasters** into the database, and LIDAR pointclouds share a lot of the features of rasters ...

Column	Type
<b>gid</b>	<b>integer</b>
<b>area</b>	<b>double precision</b>
<b>perimeter</b>	<b>double precision</b>
<b>gunitice_</b>	<b>double precision</b>
<b>gunitice_i</b>	<b>double precision</b>
<b>gunit_id</b>	<b>integer</b>
<b>gunit_labe</b>	<b>character varying(12)</b>
<b>gmap_id</b>	<b>smallint</b>
<b>geom</b>	<b>geometry(Polygon,26910)</b>

LIDAR data is not very relational. Compare the table definition of a PostGIS feature table, which has lots of interesting non-spatial data related to the spatial column.

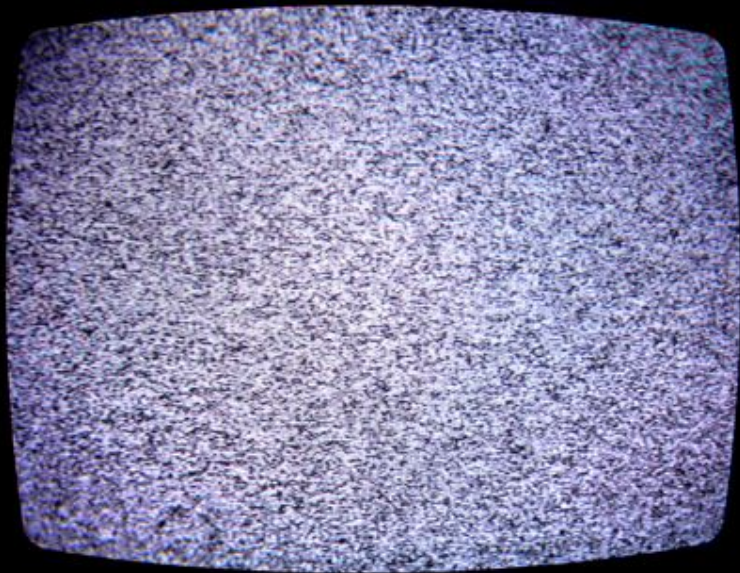
Column	Type
<b>id</b>	<b>integer</b>
<b>pa</b>	<b>pcpatch(1)</b>

With the table of pointcloud data, which is just row upon row of patch blocks, basically blobs in the database. There's not a lot of interesting stuff to query about there!



**Billions and Billions**

Also, LIDAR is really big! Billions and billions of points! That's going to result in really huge tables, which are far more fiddly to manage and back-up in a database than they would be on the filesystem as a bunch of files.



And finally, LIDAR is fairly static. Updates aren't granular and a bit at a time, they tend to be bulk re-surveys, just like raster data.

# Remotivation

Which means, I need some remotivation before I can go on!

Column	Type
id	integer
pa	pcpatch(1)

But wait, actually, inside those rows and rows of binary blocks there's quite a lot of detailed information,

(X, Y, Z, Intensity,  
ReturnNumber, NumberOfReturns,  
Classification,  
ScanAngleRank,  
Red, Green, Blue)

lots of dimensions per point and, unlike raster, LIDAR use cases do tend to filter and sub-set data using those higher dimensions, the use cases aren't all bulk retrieval.

Everything is related to everything else, but near things are more related than distant things.

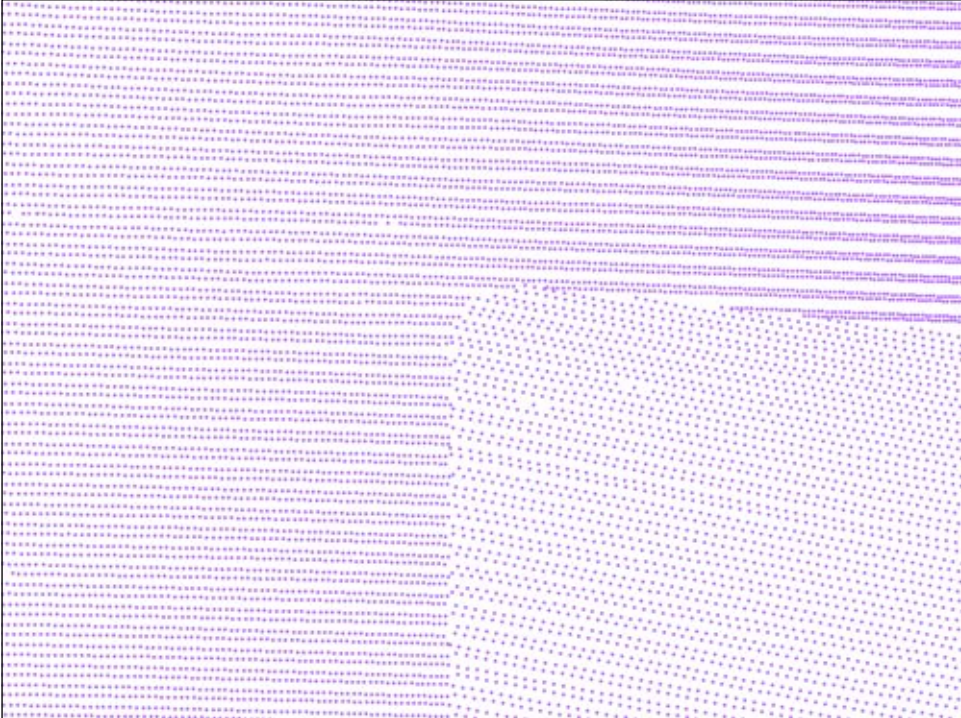
And Tobler's Law is still there, so the same motivation that got me to accept raster in the database applies to LIDAR in the database: once it's **there** you unlock all kinds of cross-type analysis: vector to raster, raster to vector, raster to pointcloud, pointcloud to vector, etc.



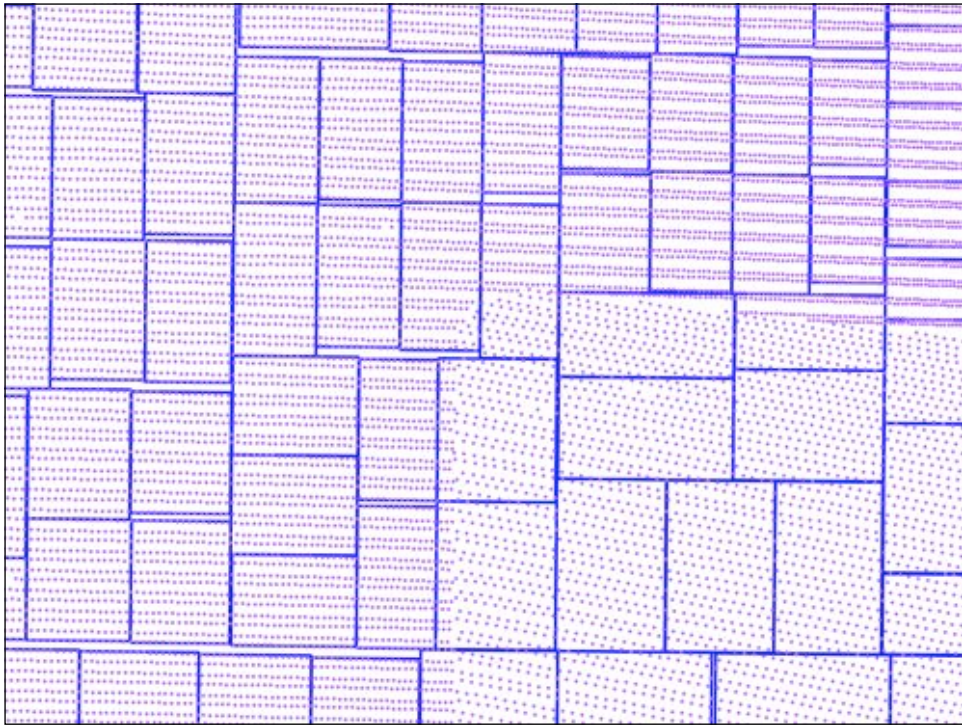


# Just Do It

OK, how do we store LIDAR in the database?



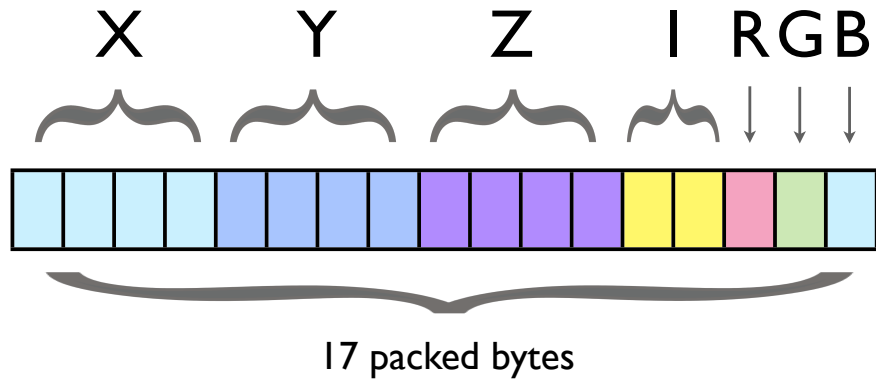
First, we can't store one point per row, because a table with billions of rows will be too big to use practically: the index will be too big, the table size will be very large with one dimension per column, in general there is a cost for a query iterating over a row, which we want to minimize.



So, for storage, we organize the points into **patches** of several hundred points each. This reduces a table of billions into a table of 10s of millions, which is more tractable.

```
PcPoint(pcid)  
PcPatch(pcid)
```

Practically, we need two new types: the pointcloud point, and the pointcloud patch. PcPoint and PcPatch. The point type is for filtering and for casts to PostGIS. The patch type is what we use to store data in tables.



$7 \times 8 = 56$  bytes as doubles

The goal of LIDAR storage is to try and keep things small, because there's **so much data**. So data are packed into a byte array, using as few bytes as possible to represent each value. Compare a packed form to a form that uses doubles for every dimension: there's no comparison.

```
<pc:dimension>
  <pc:position>1</pc:position>
  <pc:size>4</pc:size>
  <pc:description>
    X coordinate as a long integer.
    You must use the scale and offset
    information of the header to determine
    the double value.
  </pc:description>
  <pc:name>X</pc:name>
  <pc:interpretation>int32_t</pc:interpretation>
  <pc:scale>0.01</pc:scale>
  <pc:offset>0</pc:offset>
  <pc:active>true</pc:active>
</pc:dimension>
```

The description of the how bytes are packed into a point is done using an XML schema document, which uses the same format adopted by the open source PDAL project. This is an “X” dimension...

Note the “scale” and “offset” values, which allow data to be more efficiently packed into narrower byte space. Multiple dimensions are

## POINTCLOUD\_FORMATS

Column	Type
pcid	integer
srid	integer
schema	text

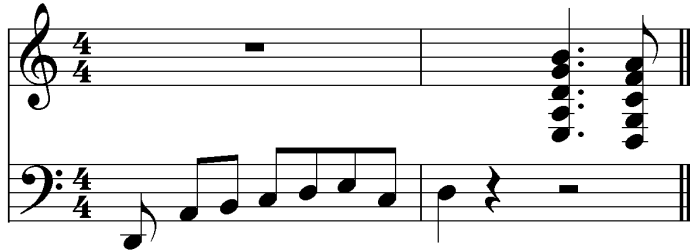
Each schema document is stored in a row in the pointcloud\_formats table, which assigns every schema and spatial reference system a unique “point cloud id”, “pcid”. So....

- **PcPatches** are collections of...
- **PcPoints** which are packing of dimensions...
- Described in XML **schema** documents...
- Stored in the **pointcloud\_formats** table...
- Tied together with a “**pcid**” that relates patches and points to the schemas necessary to interpret them!

to recap...!

But enough about internals, how do we work with pointcloud data in SQL?

*Miles Davis,* **So What**



**CREATE EXTENSION  
pointcloud;**

**CREATE EXTENSION  
postgis;**

**CREATE EXTENSION  
pointcloud\_postgis;**

depends on

depends on

Pointcloud only supports PostgreSQL 9.1 and up, so we only support installation via the “extension” method. Enable the pointcloud extension. If you want to do PostGIS integration, enable PostGIS, then enable pointcloud\_postgis. Rather than having pointcloud depend on PostGIS, point cloud is independent, and the pointcloud\_postgis extension



Schema	Name	Type
public	geography_columns	view
public	geometry_columns	view
public	pointcloud_columns	view
public	pointcloud_formats	table
public	raster_columns	view
public	raster_overviews	view
public	spatial_ref_sys	table

We have a lot of tables and views after enabling those extensions, most of which are from PostGIS, but there are two from pointcloud. The **pointcloud\_formats** table, as we mentioned, holds the schema information for the points. The **pointcloud\_columns** view acts like the geometry\_columns view, showing an up-to-date list of what tables have pointcloud data in

```
INSERT INTO pointcloud_formats (pcid, srid, schema)
VALUES (1, 0,
'<?xml version="1.0" encoding="UTF-8"?>
<pc:PointCloudSchema
  xmlns:pc="http://pointcloud.org/schemas/PC/1.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <pc:dimension>
    <pc:position>1</pc:position>
    <pc:size>4</pc:size>
    <pc:description>X coordinate.</pc:description>
    <pc:name>X</pc:name>
    <pc:interpretation>int32_t</pc:interpretation>
    <pc:scale>0.01</pc:scale>
  </pc:dimension>
  <pc:dimension>
    <pc:position>2</pc:position>
    <pc:size>4</pc:size>
    <pc:description>Y coordinate.</pc:description>
    <pc:name>Y</pc:name>
    <pc:interpretation>int32_t</pc:interpretation>
    <pc:scale>0.01</pc:scale>
  </pc:dimension>
```

Before we can create an points or patches, we need a schema to describe the dimensions we are going to hold. This is a simple four-dimensional schema, with X, Y, Z as 32-bit integers and Intensity as a 16-bit integer. We assign it PCID = 1.

```
CREATE TABLE pcpoints (  
    id SERIAL PRIMARY KEY,  
    pt PcPoint(1)  
);
```

Now we can create a points table (note, this is just for demonstration, we will store patches, not points, when in production).

```
INSERT INTO pcpoints (pt)  
VALUES (  
    PC_MakePoint(1,  
        ARRAY[-126, 45, 34, 4]  
    )  
);
```

And we can create a new point to insert into the table. The PC\_MakePoint function lets you turn an array of doubles into a PcPoint.

```
SELECT pt FROM pcpoints;
```

```
0101000000C8CEFFFF94110000  
480D00000400
```

And if we select the point back out of the table, we get the well-known binary format, which looks obscure as usual,

```
SELECT pt FROM pcpoints;
```

01	<i>endian</i>
01000000	<i>pcid</i>
C8CEFFFF	<i>x</i>
94110000	<i>y</i>
480D0000	<i>z</i>
0400	<i>intensity</i>

But actually just has a short header, giving the endianness and the pcid, and then the data itself. This is little endian data (intel processor), with the least significant bit first.



```
SELECT PC_AsText(pt)
FROM pcpoints;

{"pcid":1,
 "pt": [-126, 45, 34, 4]}
```

But the “as text” function returns a more obviously human readable format (or at least a computer interchangeable one). Rather than ape OGC well-known text, I’ve decided the emitting JSON is more likely to allow people to use pre-existing parsing functions.

```
SELECT PC_Get(pt, 'z')
FROM pcpoints;
```

34

You can pull any dimension from a point using the dimension name. This feature is the gateway to point filtering, as we’ll see.

```
SELECT
    ST_AsText(pt::geometry)
FROM pcpoints;

POINT Z (-126 45 34)
```

If you have the pointcloud\_postgis extension enabled, you can **cast** pcpoints to postgis points, which is useful for visualization or integration analysis.

```
INSERT INTO pcpoints (pt)
VALUES (
    PC_MakePoint(1,
        ARRAY[-127, 46, 35, 5]
    )
);
```

If we add one more point to our points table, we can use the two points in there to make a two-point patch, with a patch aggregate, So first we add a second point,

```
CREATE TABLE pcpatches
AS
SELECT
    PC_Patch(pt) AS pa
FROM pcpoints;
```

And then here we can use the PC\_Patch function to aggregate our two points into a new patch, in a new table.

```
SELECT PC_AsText(pa)
FROM pcpatches;
```

```
{ "pcid": 1,
  "pts": [
    [-126, 45, 34, 4],
    [-127, 46, 35, 5]
  ] }
```

And this is what it looks like in JSON. So we've taken a set of points and aggregated them into a patch. And we can also do the reverse, taking a patch and turning it into a tuple set of points,

```
SELECT
```

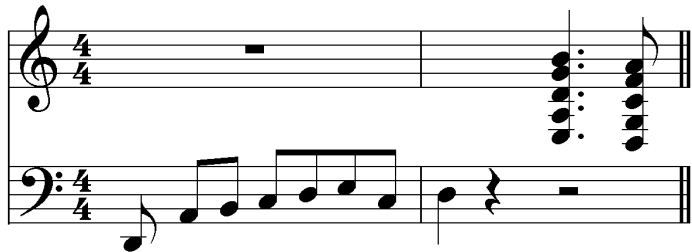
```
  PC_AsText(  
    PC_Explode(pa) )
```

```
FROM pcpatches;
```

```
{ "pcid":1,  
  "pt":[-126,45,34,4] }  
{ "pcid":1,  
  "pt":[-127,46,35,5] }
```

using the **PC\_Explode** function. You can use this facility of exploding patches into points, and then aggregating them back into patches, to filter by any attribute or pass the points into PostGIS for spatial filtering too.

*Miles Davis,* **So What**



Now it's **possible.....** that you aren't the kind of person to be excited by SQL examples, so here's some real world data loading and processing using PostgreSQL pointcloud!



In order to real-world data loaded into the database, we use the PDAL open source LIDAR processing tools, which let us handle multiple input formats (in this case LAS files) and output formats (in this case PostgreSQL Pointcloud) and also apply processing chains to the points on the way through.

## PDAL

`drivers.pgpointcloud.writer`  
`filters.chipper`  
`drivers.pgpointcloud.reader`

For the Natural Resources Canada project, I wrote a PostgreSQL Pointcloud driver for PDAL, which is now available in the PDAL source repository.

In addition to the reader and writer, we have to use the chipper to break the input file into small chips suitable for database storage.

```

<?xml version="1.0" encoding="utf-8"?>
<Pipeline version="1.0">
  <Writer type="drivers.pgpointcloud.writer">
    <Option name="connection">
      host='localhost' dbname='pc' user='pramsey'</Option>
    <Option name="table">mtsthelens</Option>
    <Option name="srid">26910</Option>
    <Filter type="filters.chipper">
      <Option name="capacity">400</Option>
    <Filter type="filters.cache">
      <Option name="max_cache_blocks">3</Option>
      <Option name="cache_block_size">32184</Option>
    <Reader type="drivers.las.reader">
      <Option name="filename">st-helens.las</Option>
      <Option name="spatialreference">EPSG:26910</Option>
    </Reader>
  </Filter>
</Filter>
</Writer>
</Pipeline>

```

The example data was a 420Mb LAS file of Mt St Helens, with 12 million points in it.

I used this PDAL pipeline file for the load. Note that it uses a chipping filter (in yellow) between the reader and the writer to break the file up into smaller patches for database storage. The writer driver (in blue) needs a connection string and destination table name etc

**Table "public.mtsthelens"**

Column	Type
id	integer
pa	pcpatch(1)

**Indexes:**

"mtsthelens\_pkey"  
PRIMARY KEY, btree (id)

When the load is done we have a table like this, with a primary key and patch data,

```
SELECT
  Count(*)
  Sum(PC_NumPoints(pa))
FROM mtsthelens;
```

count	sum
30971	12388139

And we can confirm all 12 million points are loaded, and they are stored in 30971 patches, which look



like this. Kind of hard to see what's going on, the physical context



is this,  
and we can look a bit closer



and see the patch lines, like this. In this load, the chipper ensured that each patch holds about 400 points, though we could go higher, up to about 600 points without passing the PostgreSQL page size. Now let's do some analysis!







Mount St Helens is a bit of an odd mountain, because it doesn't have a nice pointy summit. It's got a rim, around a caldera.

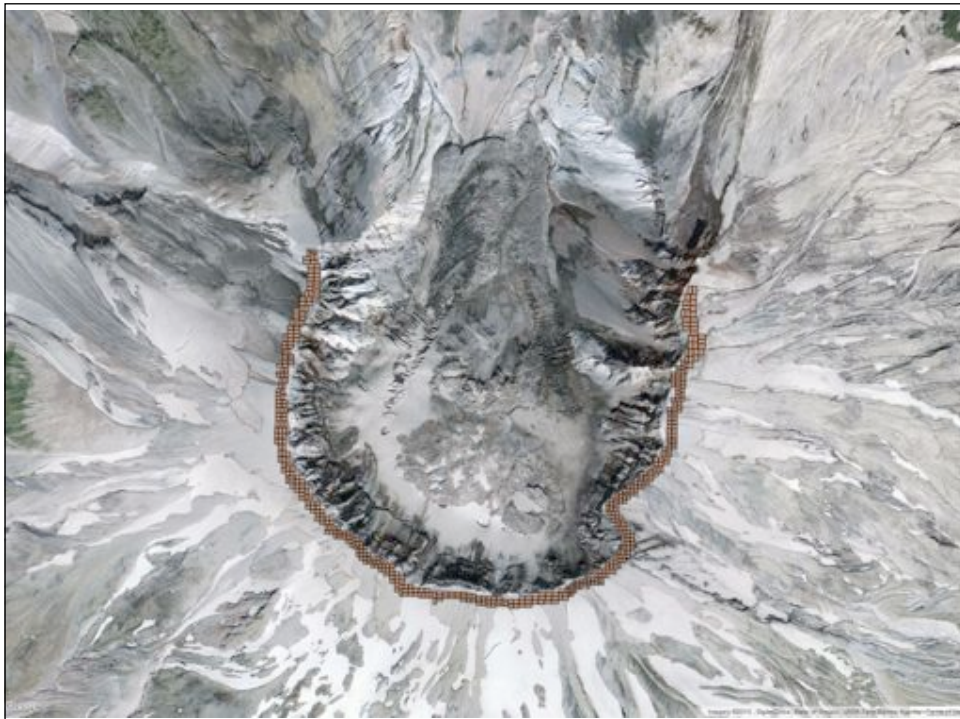
Question: **How tall is the rim?**



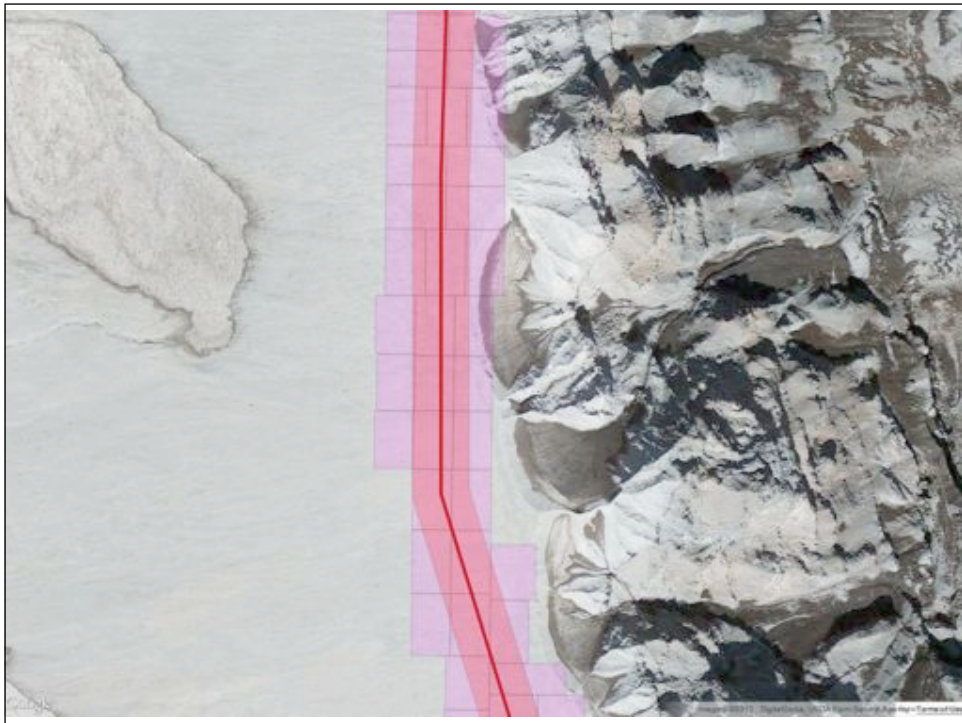
I digitized a line around the rim, so I can calculate an average elevation. I'm going to get the average elevation of every point within 15 meters of my line.

```
WITH patches AS (  
  SELECT pa  
  FROM  
    mtsthelens m,  
    mtsthelens_rim r  
  WHERE  
    PC_Intersects(m.pa,  
    ST_Buffer(r.geom, 15))  
) ,
```

It's a multi-stage query, so I use my favourite SQL syntactic sugar, the “**WITH**” clause to do it in an understandable order... first I get all the patches that intersect my 15 meter buffer of the rim line...



Which looks like this, several hundred patches.



And if you look closer, it looks kind of like this, all the patches that touch the buffer.

```
points AS (  
  SELECT PC_Explode(pa)  
        AS pt  
  FROM patches  
) ,
```

Then I take those patches and explode them into a set of points...



```
filtered_points AS (  
  SELECT PC_Get(pt, 'z')  
         AS z  
  FROM points,  
       mtsthelens_rim r  
 WHERE  
       ST_Intersects  
       (pt::geometry,  
        ST_Buffer(r.geom, 15))  
)
```

And filter those points using the buffer shape, so here I'm pushing a cast into PostGIS to run the ST\_Intersects() filter



Which gives us just the points inside the buffer,

```
SELECT avg(z),  
       count(z)  
FROM filtered_points;
```

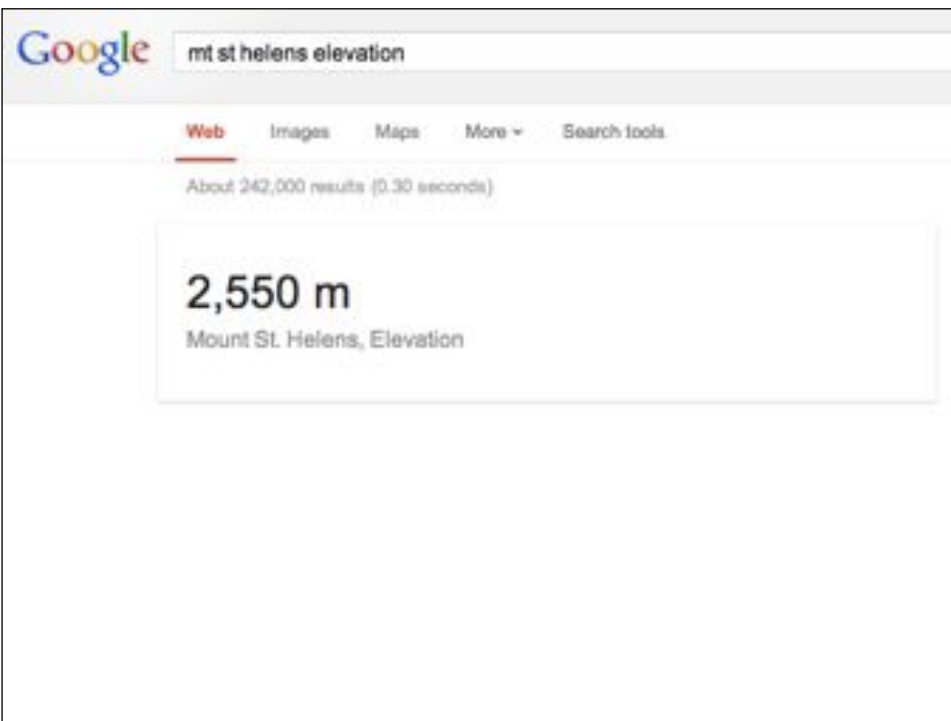
avg	count
2425.086	108736

And finally summarize the elevation of the points, which tells us we had 108 thousand points, and an average elevation of 2425 meters.

Which is odd,

because Google says, that the elevation of Mt St Helens is 2550 meters...

so, what's going on? Let's look at all the points in our database that are above 2500 meters



```
WITH points AS (  
  SELECT PC_Explode(pa)  
         AS pt  
  FROM mtsthelens  
)  
SELECT pt::geometry  
FROM points  
WHERE  
PC_Get(pt, 'z') > 2500;
```

exploding all the patches and  
finding just the points that are  
higher than our elevation threshold



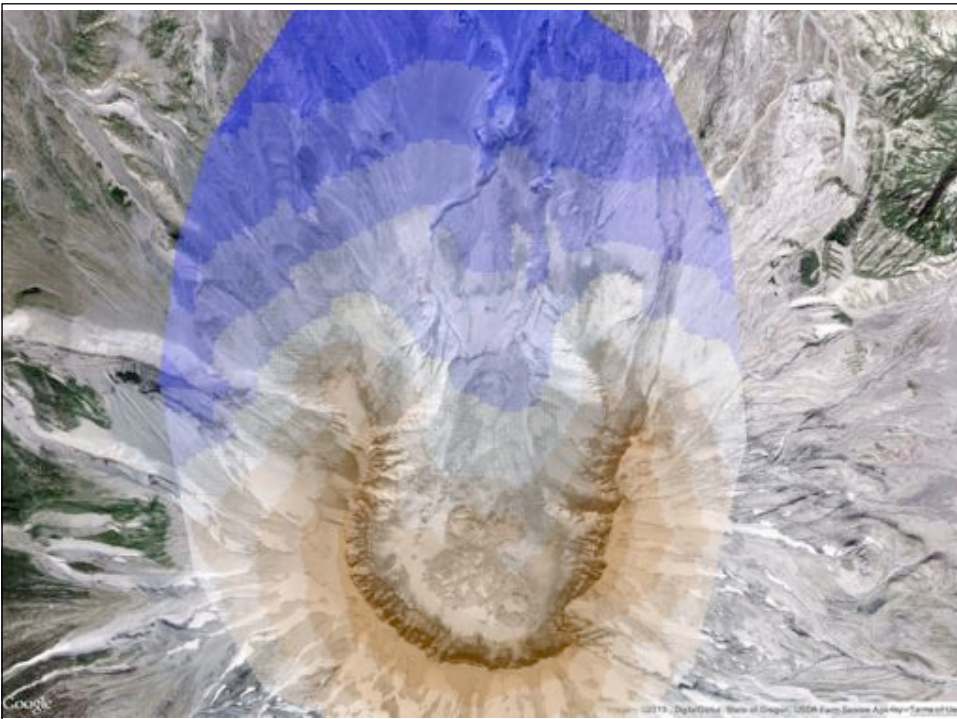
And we see that, actually the rim  
isn't **flat**, it's tallest at the southern  
end and slopes downwards to the  
north,



which we can see by taking the patches, and coloring them thematically by their average point elevation



which shows the rim sloping downwards from a tallest point at the south.



## Compression: None

- `<Metadata name="compression">`  
none  
`</Metadata>`
- Byte-packed points, ordered point-by-point.
- Equivalent to compressed LAS.

In order to lower I/O load, compressing the data a bit is an important concern.

The compression of a PcPatch is determined by the compression entry in the schema document. There are three compression modes right now. Compression of “**none**” stores the data just as packed

## Compression: Dimensional

- `<Metadata name="compression">`  
dimensional  
`</Metadata>`
- Each dimension separately compressed.
  - Run-length, common bits, or zlib.
- 4-5 times smaller than uncompressed LAS.

**Dimensional** compression is the default. Using dimensional compression, it’s possible to pack 400 to 600 points into a single patch without going over the 8KB PostgreSQL page size. Not exceeding the page size can be a performance boost, since larger objects are copied into a side table in page-sized chunks and accessed in a true random access



# Compression: GeohashTree

- `<Metadata name="compression">`  
ght  
`</Metadata>`
- Points sorted into a prefix-tree based on geohash code
- Compression from 2-4 times, depending on parameters

GHT compression is still experimental. The compression aspects are still not as good as dimensional, but because the points are ordered, there's some good possibilities for high speed spatial processing and overview generation.

## PC\_Functions

- `PC_Get(pcpoint, dimension) → numeric`
- `PC_Explode(pcpatch) → pcpoint[]`
- `PC_Union(pcpatch[]) → pcpatch`
- `PC_Patch(pcpoint[]) → pcpatch`
- `PC_Intersects(pcpatch, geometry)`
- `pcpatch::geometry, pcpoint::geometry`

I've shown a number of functions in action, but some of the most important are the  
get function, to interrogate points,  
explode to break patches into points,  
union to merge patches,  
patch to merge points,  
intersects to find spatial overlaps,  
and  
the rest into vector geometry.

## Future

- PC\_Transform(pcpatch, pcid) → pcpatch
- PC\_Intersection(pcpatch, geometry) → pcpatch
- PC\_Raster(pcpatch, raster, dimension) → raster
- PC\_FilterBetween(pcpatch, dim, min, max) → pcpatch
- PDAL writer flexibility  
PDAL reader queries

There's lots of work on the drawing board,

...

...

## Get It

- Pointcloud  
<http://github.com/pramsey/pointcloud>
- PDAL  
<http://github.com/PDAL/PDAL>
- *Questions?*

And it's ready for use and abuse right now.