Postgres Tutorials

Easy PostgreSQL Time Bins

Paul Ramsey

Mar 16, 2023 · 7 min read

It's the easiest thing in the world to put a timestamp on a column and track when events like new records or recent changes happen, but what about reporting?

Binning data for large data sets like time series is a great way to let you group data sets by obvious groups and then use SQL to pull out a query that easily works in a graph.

Here's some PostgreSQL secrets that you can use to build up complete reports of time-based data.

Earthquake Data

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more

Earthquakes are a natural source of time-stamped data, and Crunchy Bridge gives us access to PL/Python. This data has a geometry column, so I'll also add PostGIS.

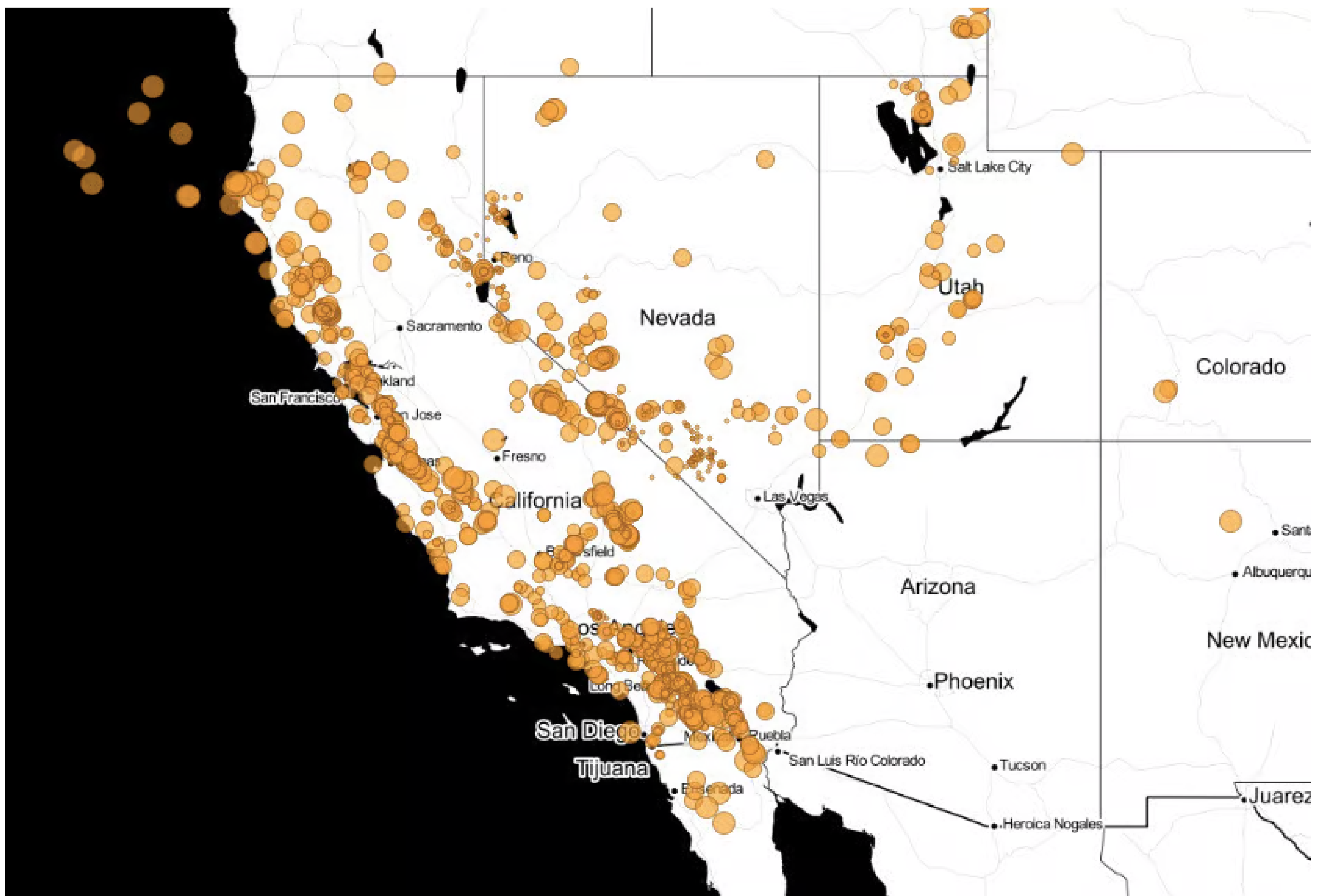
```
CREATE EXTENSION plpython3u;
CREATE EXTENSION postgis;
```

Our target table is just a few interesting columns for each quake.

```
CREATE TABLE quakes (
    mag   float8,
    place text,
    ts    timestamptz,
    url   text,
    id    text,
    geom  geometry(pointz, 4326)
);
```

To populate the table, we pull the live earthquake feed published by the USGS.

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more




```
CREATE OR REPLACE FUNCTION fetch_quakes()
RETURNS setof quakes
AS $$
    import requests
    import json
    url = 'https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_month
    r = requests.get(url)
    quakes = r.json()

    for q in quakes['features']:
        q_id      = q['id']
        props    = q['properties']
        geojson = json.dumps(q['geometry'])
        epoch    = props['time']
        q_ts      = plpy.execute(f"SELECT to_timestamp({epoch}/1000.0) AS t")[0
        q_geom    = plpy.execute(f"SELECT st_geomfromgeojson('{geojson}') AS g'
        q_mag     = props['mag']
        q_url     = props['url']
        q_place   = props['place']
```

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more

```
        yield (q_mag, q_place, q_ts, q_url, q_id, q_geom)

$$
LANGUAGE 'plpython3u';
```

And populating the table is then just a simple refresh and load.

```
TRUNCATE quakes;
INSERT INTO quakes SELECT * FROM fetch_quakes();
```

Simple Summaries

One month of quakes of all sizes is a table of a few thousand records. (The answer will vary depending on when you run the query, since the input is live.)

```
-- 11791
SELECT Count(*)
FROM quakes;
```

Where the data support it, running grouped aggregates off of rounded results is a good way to generate a summary. Here's the summary of magnitudes, using the `floor()` function to turn the distinct floating point magnitudes into groupable integers.

```
SELECT floor(mag) AS mags,
       Count(*)
FROM quakes
GROUP BY mags
ORDER BY mags;
```

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more

```
 mags | count
------+-------
   -2 |     4
   -1 |   719
    0 |  2779
    1 |  5252
    2 |  1676
    3 |   342
    4 |   849
    5 |   142
    6 |    13
```

Histogram Summaries

Let's look at magnitude 6 quakes.

```
SELECT ts::date AS date, count(*)
FROM quakes q
WHERE q.mag > 6
GROUP BY date
```

```
    date    | count
------------+-------
 2023-02-15 |     1
 2023-02-17 |     1
 2023-02-20 |     1
 2023-02-23 |     2
```

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more

```
 2023-03-01 |     1
 2023-03-02 |     1
 2023-03-04 |     1
 2023-03-14 |     1
```

To build a good histogram, you need a value for every category in your binning of the raw data. Unfortunately, the quake data are sparse: there isn't a result for every day of the last month.

There's a couple ways to solve this problem.

Since we are binning by date, we can take a list of all dates in our range, and left join the counts to that list. Dates without counts will get **`NULL`** counts, but we can use **`Coalesce()`** to convert those to zeroes.

```
WITH counts AS (
    SELECT ts::date AS date, count(*)
    FROM quakes q
    WHERE q.mag > 6
    GROUP BY date
)
SELECT series::date, coalesce(counts.count, 0)
FROM generate_series('2023-02-13'::date, '2023-03-14'::date, '1 day'::interval
LEFT JOIN counts
ON counts.date = series;
```

Your result will start like this:

```
   series   | coalesce
------------+----------
 2023-02-13 |        0
 2023-02-14 |        0
```

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more

```
 2023-02-16 |        0
 2023-02-17 |        1
```

The magic ingredient here is the `generate_series()` function. It is usually used to generate sets of integers, but it will also generate sets of timestamps, or dates, or floats, as long as you provide a third parameter, the distance between each element.

In this example, we generated using a one day interval.

Timestamp Bins

In PostgreSQL 14 and higher, there is a new date_bin() function for rounding timestamps to any stride, so you aren't restricted to just rounding to days or years or months.

Replacing the cast to date with `date_bin()` and ensuring that `generate_series()` shares the same stride and start time as `date_bin()` our SQL looks almost the same.

```
WITH counts AS (
    SELECT date_bin('2.5 days'::interval, ts, '2023-02-13'::timestamp), count(
    FROM quakes q
    WHERE q.mag > 6
    GROUP BY date_bin
)
SELECT series, coalesce(counts.count, 0) AS count
FROM generate_series('2023-02-13'::timestamp, '2023-03-14'::timestamp, '2.5 da
LEFT JOIN counts
ON counts.date_bin = series;
```

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more

```
2023-02-13 00:00:00 |       1
2023-02-15 12:00:00 |       2
2023-02-18 00:00:00 |       0
2023-02-20 12:00:00 |       1
2023-02-23 00:00:00 |       2
2023-02-25 12:00:00 |       1
2023-02-28 00:00:00 |       1
2023-03-02 12:00:00 |       2
2023-03-05 00:00:00 |       0
2023-03-07 12:00:00 |       0
2023-03-10 00:00:00 |       0
2023-03-12 12:00:00 |       1
```

And the result is a complete set of counts for this add 2.5 day stride.

Arbitrary Bins of Any Size

What if we want to summarize using a bin layout that doesn't neatly align with the rounding of a particular type? What about magnitude 6 earthquakes by week? Or in an irregular set of bins.

We can generate the bins easily enough with `generate_series()` . **Note that we could also manually construct an array of irregularly spaced bin boundaries if we wanted.**

```
SELECT array_agg(a) AS bins
FROM generate_series(
    '2023-02-13'::date,
    '2023-03-14'::date,
    '1 week'::interval) a;
```

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more

Fortunately there is another PostgreSQL function to make use of the bins array, `width_bucket()`. We can feed our bins into `width_bucket()` as an array to get back counts in each bucket.

```
WITH a AS (
    SELECT array_agg(a) AS bins
    FROM generate_series(
        '2023-02-13'::date,
        '2023-03-14'::date,
        '1 week'::interval) a
),
counts AS (
    SELECT
        width_bucket(ts, a.bins) AS bin,
        Count(*) AS count
    FROM quakes
    CROSS JOIN a
    WHERE mag > 6
    GROUP BY bin
)
SELECT * FROM counts;
```

This is extremely flexible, as the bin widths can be any interval at all, or a mixed collection of widths: a week, 2 days, 47 hours, whatever.

However, the query result isn't very informative.

```
 bin | count
-----+-------
   1 |     3
   2 |     4
   3 |     3
   5 |     1
```

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more

We have the bin number and the count, but we have lost the information about the bin boundaries, and also we have a missing zero count for bin 4.

To get back the bin boundaries, we reach back to the array we initially generated, and **`unnest()`** it. To get the bin numbers at the same time, we use the **`WITH ORDINALITY`** keywords.

```
WITH a AS (
    SELECT array_agg(a) AS bins
    FROM generate_series(
        '2023-02-13'::date,
        '2023-03-14'::date,
        '1 week'::interval) a
),
counts AS (
    SELECT
        width_bucket(ts, a.bins) AS bin,
        Count(*) AS count
    FROM quakes
    CROSS JOIN a
    WHERE mag > 6
    GROUP BY bin
)
SELECT
    b.elem AS bin_min,
    b.bin,
    Coalesce(counts.count, 0) AS count
FROM a
CROSS JOIN unnest(bins) WITH ORDINALITY AS b(elem, bin)
LEFT JOIN counts ON b.bin = counts.bin;
```

The final result is ready for charting!

```
        bin_min         | bin | count
------------------------+-----+-------
 2023-02-13 00:00:00+00 |   1 |     3
 2023-02-20 00:00:00+00 |   2 |     4
```

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more

```
2023-03-06 00:00:00+00 |   4 |     0
2023-03-13 00:00:00+00 |   5 |     1
```

We have a count for every bin, and a bottom value for every bin. Tinker with this query and adjust the bin width at the top, to see how flexible PostgreSQL's dynamic binning tools are.

Conclusions

- PL/Python is a fun tool for dynamic HTTP data access.
- The `generate_series()` function can create sets of floats and timestamps as well as integers.
- The new `date_bin()` function is very handy for grouping timestamps on non-standard intervals.
- The `width_bucket()` function is a powerful tool for creating counts of values in dynamically generated bins.
- Pairing `unnest()` with `ORDINALITY` is a cute trick to generate row numbers to go along with row sets.

Enjoy this article?

You will love our newsletter!

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more

Join The List

WRITTEN BY

Paul Ramsey

March 16, 2023 • More by this author

PRODUCTS

Crunchy Postgres

Crunchy Postgres for Kubernetes

Crunchy Bridge

Crunchy Certified PostgreSQL

Crunchy PostgreSQL for Cloud Foundry

Crunchy MLS PostgreSQL

Crunchy Spatial

SERVICES & SUPPORT

Enterprise PostgreSQL Support

Ansible

Red Hat Partner

Trusted PostgreSQL

Crunchy Data Subscription

RESOURCES

Customer Portal

Software Documentation

Blog

Events

COMPANY

About Crunchy Data

Team

News

Careers

Contact Us

Newsletter

Security

CRUNCHY DATA NEWSLETTER

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more

Enter your email

Join The List

© 2018-2023 Crunchy Data Solutions, Inc.

This site uses cookies for usage analytics to improve our service. By continuing to browse this site, you agree to this use. Learn more