Time series raster data in PostgreSQL with the TimescaleDB and postgis_raster

What is timescaleDB?

TimescaleDB is an open-source time-series database (TSDB) designed to handle massive volumes of time-series data efficiently and reliably. It is built as an extension to PostgreSQL, leveraging its robustness and scalability while providing additional functionality specifically tailored for time-series data management.

- Time-series data model
- Automatic data partitioning
- Continuous aggregates
- Advanced indexing
- Compression and data retention policies
- Ecosystem compatibility

What's postgis_raster?

PostGIS Raster is an extension to the PostgreSQL database that adds support for storing, indexing, and processing raster data alongside the existing geospatial capabilities provided by PostGIS. It allows for the integration of raster data, such as satellite imagery, digital elevation models, and other gridded data, into a PostgreSQL database, enabling powerful spatial analysis and data management.

- Storage of raster data
- Spatial indexing
- Raster algebra and analysis
- Integration with vector data
- Raster tiling and pyramids
- Interoperability

Spatial temporal

If we look at snapshot of earth, it’s the spatial part and looking at multiple snapshots will give us the temporal part.

Adding postgis_raster and timescaleDB we can generate some interesting insights and solve some use cases in our friendly neighbourhood database postgres.

Rasters in Database

Rasters in database is not exactly the best thing to do per se because postgres fundamentally is not built to handle raster dataset. Under the hood it actually stores the reference to the raster where most of the functionality is handled by gdal.

For more information check out this amazing talk on Mapscaping where Paul Ramsey explains it better than anyone:

https://mapscaping.com/podcast/rasters-in-a-database/

So why do we still do it?

Because it can be done.
Managing multiple technologies like rasters/COGS in s3 and having python or any other wrapper around will add more complexity to your tech stack, if the use case can be solved within your existing technologies then let’s keep it simple.

Although what we are gonna do is anything but simple!

Getting things done

Use cases:

- Loading raster in the DB using smaller tiles
- Build continuous aggregates to store inferences from the raster data
- Data retention and compression

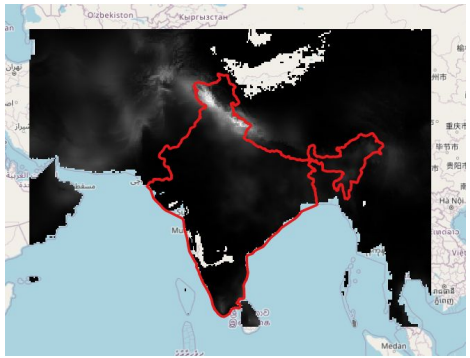
Tools:

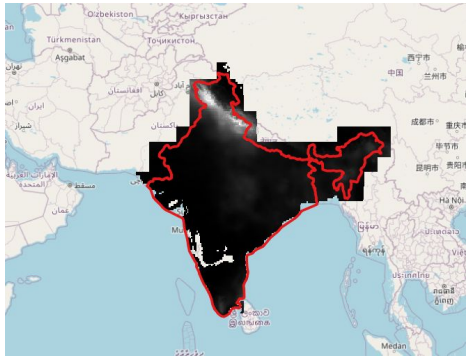
- raster2pgsql
- QGIS

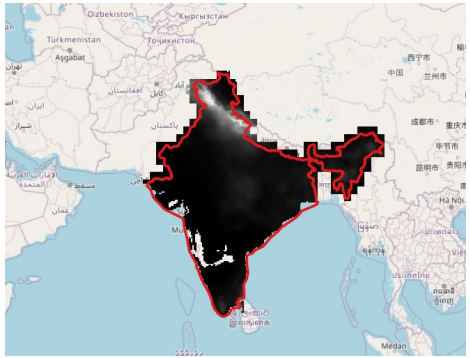
Loading raster in the DB using smaller tiles

We can load the in different tile sizes for all rasters using the *-t TILE_SIZE*, where different use cases require different values the smaller tile size will create many rows and larger tile size will create less rows which can affect the ability to select elements when querying the rasters. e.g .

Using ST_Intersects to select all rasters in India will yield different results for different tile sizes.







Loading raster in the DB using smaller tiles

Same raster loaded at different tile sizes yields:

- 100x100 - 157 rows
- 10x10 - 7263 rows
- 5x5 - 26,001 rows

Now the next steps add more rasters with temporal dimension so we can have a look at the world as it floats through the universe.

Doing the timeseries stuff

```
prec_data=# \d worldclim;
                                Table "public.worldclim"
  Column   |           Type           | Collation | Nullable |              Default
-----------+--------------------------+-----------+----------+-----------------------------------
 rid       | integer                  |           | not null | nextval('worldclim_rid_seq'::regclass)
 rast      | raster                   |           |          |
 timestamp | timestamp without time zone |           |          |
Indexes:
    "worldclim_pkey" PRIMARY KEY, btree (rid)
    "worldclim_st_convexhull_idx" gist (st_convexhull(rast))
    "worldclim_timestamp_idx" btree ("timestamp")
```

```
                                        Table "public.worldclim_hypertable"
  Column   |           Type           | Collation | Nullable | Default | Storage  | Compression | Stats target | Description
-----------+--------------------------+-----------+----------+---------+----------+-------------+--------------+-------------
 rid       | integer                  |           |          |         | plain    |             |              |
 rast      | raster                   |           |          |         | extended |             |              |
 timestamp | timestamp without time zone |           | not null |         | plain    |             |              |
Indexes:
    "worldclim_hypertable_st_convexhull_idx" gist (st_convexhull(rast))
    "worldclim_hypertable_timestamp_idx" btree ("timestamp" DESC)
Triggers:
    ts_insert_blocker BEFORE INSERT ON worldclim_hypertable FOR EACH ROW EXECUTE FUNCTION _timescaledb_internal.insert_blocker()
Child tables: _timescaledb_internal._hyper_1_100_chunk,
              _timescaledb_internal._hyper_1_101_chunk,
              _timescaledb_internal._hyper_1_102_chunk,
              _timescaledb_internal._hyper_1_103_chunk,
              _timescaledb_internal._hyper_1_104_chunk,
              _timescaledb_internal._hyper_1_105_chunk,
              _timescaledb_internal._hyper_1_106_chunk,
              _timescaledb_internal._hyper_1_107_chunk,
              _timescaledb_internal._hyper_1_108_chunk,
              _timescaledb_internal._hyper_1_109_chunk,
```

Continuous Aggregates

```sql
-- Get summary stats for an year
SELECT ST_SummaryStatsAgg(worldclim.rast, true, 1)
FROM worldclim
where timestamp ≥ '2001-01-01' AND timestamp < '2002-01-01';
```

```sql
-- Let's automate this step
-- yearly aggregates of all the rasters
CREATE MATERIALIZED VIEW worldclim_continous_aggregates_yearly(st_summarystatsagg)
WITH (timescaledb.continuous) AS
  SELECT ST_SummaryStatsAgg(worldclim_hypertable.rast, true, 1)
    FROM worldclim_hypertable
    group by time_bucket('1year', timestamp);
-- Boom      You, 1 second ago • Uncommitted changes
select * from worldclim_continous_aggregates_yearly;
```

More things to try:

- Add indexing on rasters using h3 and create zonal aggregates
- Use retention policy to remove old data
- Use compression policy to compress not-so-much accessed data
- Actions and automation using timescale to create workflows

Sources

- https://www.worldclim.org/data/monthlywth.html#
- https://docs.timescale.com/api/latest
- https://postgis.net/docs/using_raster_dataman.html
- https://postgis.net/docs/RT_reference.html
- https://github.com/jashanbhullar/foss4g-2023-timescaledb-postgis_rasters-in-postgres

Thank You

You can reach me out on:


https://twitter.com/json_singh


https://www.linkedin.com/in/jsonsingh/

All the code is at:
https://github.com/jashanbhullar/foss4g-2023-timescaledb-postgis_rasters-in-postgres

Look for `json singh` !