**Informe** TP Especial

Base de Datos Espaciales y Sistemas de Información Geográfica

**Análisis de Queries con Bases de Datos Espaciales PSQL y MONGODB**

Casagrande, Lucas

Cortés Rodríguez, Kevin

Ingeniería Informática – 2do. Cuatrimestre 2018

**PostgreSQL -** Informe

En esta sección del informe, presentaremos cuáles son las sentencias ejecutadas en una Base de Datos Postgres con PostGIS instalado, en una máquina virtual UBUNTU de 64 bits y 4GB de RAM

**Queries de consulta y update**

**QUERY 1**

with ejemplo as (

SELECT id, "EmpresaId", "Description"

FROM "sube"

ORDER BY geom <-> st\_setsrid(st\_makepoint(-58.401265, -34.640470),4326)

)

select \* from ejemplo order by id asc limit 1000;

**QUERY 2**

SELECT id, "EmpresaId", "Description", ST\_Distance(geom, ref\_geom) AS distance

FROM "sube"

CROSS JOIN (SELECT ST\_MakePoint(-58.401265, -34.640470)::geography AS ref\_geom) AS r

WHERE ST\_DWithin(geom, ref\_geom, 500)

ORDER BY ST\_Distance(geom, ref\_geom);

**QUERY 3**

SELECT id, "EmpresaId", "Description"

from "sube"

where ST\_DWithin(geom::geography, ST\_Buffer(st\_makepoint(-58.401265, -34.640470)::geography,500), 0)

order by id asc;

**QUERY 4**

SELECT id, "EmpresaId", "Description"

FROM "sube"

WHERE ST\_DWithin(geom::geography, st\_makepoint(-58.401265, -34.640470)::geography, 500)

ORDER BY id asc;

Luego, sobre las tablas “sube” y “localidades” se agrega una columna adicional:

El objetivo de agregar una columna adicional en ésta tabla es para consultar la cantidad de puntos de SUBE en dicho punto y contemplando un radio de 500 metros de la tabla “sube”

**ALTER TABLE** localidades **ADD COLUMN** cant\_sube integer;

**UPDATE** localidades **SET** cant\_sube = 0

**UPDATE 1 – ACÁ TRABAJAMOS SOBRE LA GEOM CASTEADA A GEOGRAPHY**

**UPDATE** localidades **AS** l1

**SET** cant\_sube = (

**SELECT** **count**(\*)

**FROM** sube

**WHERE** ST\_DWithin(sube.geom::geography, l1.geom::geography, 500)

)

**UPDATE 2 – AGREGAMOS UNA COLUMNA GEOGRAPHY**

**UPDATE** sube **SET** geog = geom::geography

**UPDATE** localidades **as** l1

**SET** cant\_sube = (

**SELECT** **count**(\*)

**FROM** sube

**WHERE** ST\_DWithin(sube.geog, l1.geog, 500)

)

Mediante la consulta

**select** \* **from** pg\_am;

Podemos ver cuales son las formas/tipos de índice que se pueden crear. En particular, nos centraremos en btree y gist; ya que trabajaremos sobre un campo del tipo geography.

btree

hash

gist

gin

spgist

brin

**CREACIÓN ÍNDICE GEOM GIST**

**create** **index** geom\_spacial\_index **on** sube **using** gist(geom gist\_geometry\_ops\_nd);

**create** **index** geom\_spacial\_index\_localidades **on** localidades **using** gist(geom gist\_geometry\_ops\_nd);

**CREACIÓN ÍNDICE GEOM BTREE**

**create** **index** geom\_spacial\_index **on** sube **using** btree(geom btree\_geometry\_ops);

**create** **index** geom\_spacial\_index\_localidades **on** localidades **using** btree(geom btree\_geometry\_ops);

**CREACIÓN ÍNDICE GEOG GIST**

**create** **index** geog\_spacial\_index **on** sube **using** btree(geog btree\_geography\_ops);

**create** **index** geog\_spacial\_index\_localidades **on** localidades **using** btree(geog btree\_geography\_ops);

**CREACIÓN ÍNDICE GEOG BTREE**

**create** **index** geog\_spacial\_index **on** sube **using** gist(geog gist\_geography\_ops);

**create** **index** geog\_spacial\_index\_localidades **on** localidades **using** gist(geog gist\_geography\_ops);

A su vez, tambien usamos el tipo de indice “brin”. BRIN admite las mismas estrategias que un b-tree y, por lo tanto, necesita el mismo conjunto de operadores.

Para usar estos tipos de indices sobre una geometría, hay que verificar que sus “operator classes” estén instaladas. Para ello, se puede corroborar haciendo la siguiente consulta

**select** \* **from** pg\_opclass (1)

Los “operator clases” identifica los operadores que utilizará el índice para esa columna. Por ejemplo, un índice de árbol B en el tipo int4 usaría la clase int4\_ops; Esta clase de operador incluye funciones de comparación para valores de tipo int4. En la práctica, la clase de operador predeterminada para el tipo de datos de la columna suele ser suficiente.

Para el caso de brin, no tenemos definidos los operator classes para un geometry, por lo que se instaló una extensión de PostgreSQL “brin4postgis” (<https://github.com/2ndquadrant-it/brin4postgis>)

En la documentación del mismo, se encuentra como instalar la extensión en la base de datos.

Una vez realizado estos pasos, podemos hacer un índice de tipo brin sobre una geometría ya que contamos con el operator class de la extensión (brin\_geometry\_inclusion\_ops\_box3d)

**create** **index** brin\_geom\_spacial\_index **on** sube **using** brin(box3d(geom) brin\_geometry\_inclusion\_ops\_box3d);

**create** **index** brin\_geom\_spacial\_index\_localidades **on** localidades **using** brin(box3d(geom) brin\_geometry\_inclusion\_ops\_box3d);

Puede revisar los operadores nuevos hacienda la consulta 1

Datos obtenidos

|  |  |  |  |
| --- | --- | --- | --- |
|  | SIN ÍNDICE | CON ÍNDICE BTREE | CON ÍNDICE GIST |
| **QUERY 1** | ≈ 56 ms | ≈ 58 ms | ≈ 59 ms |
| **QUERY 2** | ≈ 26 ms | ≈ 27 ms | ≈ 26 ms |
| **QUERY 3** | ≈ 36 ms | ≈ 30 ms | ≈ 30 ms |
| **QUERY 4** | ≈ 30 ms | ≈ 30 ms | ≈ 30 ms |
| **UPDATE 1** | ≈ 9’ 38’’ | ≈ 10’ 32’’ | ≈ 8’ 55’’ |
| **UPDATE 2** | ≈ 3’ 39’’ | ≈ 2’ 32’’ | ≈ 712 ms |

Dada la diferencia de tiempos notoria en el Update 2, nos fijaremos como es la secuencia llamados dentro del motor de la base de datos

**SIN INDICE**

Update on localidades l1 (cost=0.00..173233368.96 rows=10956 width=154)

-> Seq Scan on localidades l1 (cost=0.00..173233368.96 rows=10956 width=154)

SubPlan 1

-> Aggregate (cost=15811.64..15811.65 rows=1 width=0)

-> Seq Scan on sube (cost=0.00..15811.64 rows=1 width=0)

Filter: ((geog && \_st\_expand(l1.geog, '500'::double precision)) AND (l1.geog && \_st\_expand(geog, '500'::double precision)) AND \_st\_dwithin(geog, l1.geog, '500'::double precision, true))

**CON INDICE btree**

Update on localidades l1 (cost=0.00..170094009.33 rows=10956 width=154)

-> Seq Scan on localidades l1 (cost=0.00..170094009.33 rows=10956 width=154)

SubPlan 1

-> Aggregate (cost=15525.10..15525.11 rows=1 width=0)

-> Index **Only Scan using geog\_spacial\_index on sube** (cost=0.41..15525.09 rows=1 width=0)

Filter: ((geog && \_st\_expand(l1.geog, '500'::double precision)) AND (l1.geog && \_st\_expand(geog, '500'::double precision)) AND \_st\_dwithin(geog, l1.geog, '500'::double precision, true))

**CON INDICE gist**

Update on localidades l1 (cost=0.00..193784.55 rows=10956 width=154)

-> Seq Scan on localidades l1 (cost=0.00..193784.55 rows=10956 width=154)

SubPlan 1

-> Aggregate (cost=17.59..17.60 rows=1 width=0)

-> **Index Scan using geog\_spacial\_index** on sube (cost=0.40..17.59 rows=1 width=0)

**Index Cond: (geog && \_st\_expand(l1.geog, '500'::double precision))**

Filter: ((l1.geog && \_st\_expand(geog, '500'::double precision)) AND \_st\_dwithin(geog, l1.geog, '500'::double precision, true))

**MONGO**

Para hacer las pruebas en Mongo, se utilizan dos tablas distintas, pero con el mismo contenido que la base de PSQL. La base está levantado sobre la misma VM que PSQL.

En la base “sube” de MongoDB, se crean dos tablas con el mismo contenido. Lo único que varía es el índice creado

**> sube**

**> sube2dIndex**

Todos los datos son los que se obtuvieron de la página de SUBE.

A su vez, se crean dos indices espaciales:

Para la tabla “sube” > 2dsphere

Para la tabla “sube2dIndex” > 2d

db.sube.createIndex( { geometry : "2dsphere" } )

db.sube2dIndex.createIndex( { "geometry.coordinates" : "2d" } )

Por otro lado, creamos la tabla “localidades”, que luego se utilizará para hacer las pruebas correspondientes.

mongoimport --db sube -c localidades --file "localidadesCompact.geojson" --jsonArray

Luego, corremos dos scripts:

\*\* Promedio de 3 corridas

**Test1.js – Verifica el tiempo de hacer el update sobre “sube” (similar a UPDATE 2)**

Tiempo total: 19843 ms

**Test2.js – Verifica el tiempo de hacer el update sobre “sube2dIndex” (similar a UPDATE 2)**

Tiempo total: 18267 ms

Ambos scripts toman la tabla localidades, y hacen una consulta a sube y sube2dIndex respectivamente, buscando la cantidad de aciertos dentro del punto (lat, lng) obtenido de “localidades” y en un rango de 500 metros.

La analogía de estas queries es la sentencia de UPDATE 2 de PostgreSQL.

Pruebas en Mongo sin índice no son posibles, ya que requiere de un índice “geoNear” para las consultas espaciales.

2018-11-19T09:26:33.159-0300 E QUERY [js] Error: count failed: {

"ok" : 0,

"errmsg" : "error processing query: ns=sube.sube2dIndexTree: GEONEAR field=geometry.coordinates maxdist=0.005 isNearSphere=0\nSort: {}\nProj: {}\n planner returned error: unable to find index for $geoNear query",

"code" : 2,

"codeName" : "BadValue"

}

**CONSULTA DE DATOS**

**API REST**

<http://server:3000/mongo/lat/lng/metros>

<http://server:3000/psql/lat/lng/metros>

*Disclaimer*

*Devuelve una estructura en formato GeoJSON*

**MAPA**

http://server:3000/mongo/lat/lng/metros?map=true

<http://server:3000/psql/lat/lng/metros>?map=true

Haciendo solo <http://server:3000/> redirige a la ubicación actual

Con un total de 50 conexiones concurrentes a la base (equivalente a 50 clientes conectados simultáneamente), y un total de 11K de request:

|  |  |  |
| --- | --- | --- |
| **MONGO** | Tiempo total (ms) | 51187 |
| Promedio (ms) | 215 |
| **PSQL** | Tiempo total (ms) | 54323 |
| Promedio (ms) | 278 |

Cabe aclarar que los tiempos en PSQL son mayores (no por amplia diferencia) ya que PSQL tiene que hacer una “transformación” de los datos a un GeoJSON con la llamada **st\_asgeojson**

Bibliografía

<https://hub.packtpub.com/6-index-types-in-postgresql-10-you-should-know/>

<https://www.postgresql-sessions.org/_media/8/gbroccolo_jrouhaud_pgsession_brin4postgis.pdf>

<https://github.com/2ndquadrant-it/brin4postgis>

<https://postgis.net/docs/using_postgis_dbmanagement.html#brin_indexes>