*E1.1*

***Diagnóstico de Datos y Sistema Actual***

*v1.2*

02 Septiembre 2019

Para cualquier consulta o cuestión pendiente sobre la oferta la persona de contacto es:

|  |  |  |
| --- | --- | --- |
| **Nombre** | **Teléfono** | **e-mail** |
| José Vicente Higón Valero | +34 600 55 44 74 | [jvhigon@gvsig.com](mailto:jvhigon@gvsig.com) |

Información general:

Asociación gvSIG

CIF G98125362

Calle Ángel Guimerá 61, puerta 3

[http://www.gvsig.com](http://www.gvsig.com/)

**Control de cambios**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Versión** | **Responsable** | **Organismo** | **Descripción del cambio** | **Fecha** |
| v.1.0 | Francisco José Peñarrubia | Asoc gvSIG | Documento inicial | 07/18/19 |
| v.1.1 | Francisco José Peñarrubia | Asoc. gvSIG | Ventajas modificaciones BD | 23/08/2019 |
| v.1.2 | Francisco José Peñarrubia | Asoc. gvSIG | Descripción tablas más importantes y explicación mejoras optimización de consultas | 02/09/2019 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Índice**

[1. Introducción](#_gjdgxs) **4**

[2. Análisis de la base de datos existente](#_30j0zll) **5**

[2.1 Datos duplicados](#_1fob9te) 8

[2.2 Ejes de calles vs direcciones en esa calle](#_3znysh7) 9

[2.3 Direcciones en localidad/departamento vs geometría de localidad/departamento](#_2et92p0) 10

[2.4 Campos vacíos](#_tyjcwt) 12

[2.5 Inconsistencias mayores](#_3dy6vkm) 12

[2.6 Coherencia en el crecimiento/decrecimiento de los números de puerta](#_1t3h5sf) 13

[2.5 Inventario de capas geométricas y su cobertura](#_4d34og8) 14

[2.6.- Otras consideraciones](#_2s8eyo1) 16

[**3. Detalle de campos de las tablas más importantes**](#_17dp8vu) **17**

[4.1. Descripción de los campos de “punto”](#_3rdcrjn) 18

[4.1. Descripción de los campos de “calle”](#_26in1rg) 20

[4. Análisis del código fuente](#_lnxbz9) **20**

[**5. Optimización de la sugerencia de calles.**](#_35nkun2) **33**

[**6. Consideraciones Finales**](#_1ksv4uv) **37**

# 1. Introducción

El siguiente documento hace referencia al plan de trabajo definido en el documento *Sección VI. Lista de Bienes y Servicios* de la licitación **Abreviada Nº 112/2018**:

[...]

*4. Entregables*

*Durante la ejecución del proyecto, la empresa adjudicataria deberá cumplir con*

*los siguientes entregables:*

***1) Documento de diagnóstico de Datos y Sistema Actual***

*2) Propuesta de diseño de base de datos y migración.*

*3) Plan de Trabajo*

*4) Documento de Requerimientos.*

[…]

Se trata de realizar un análisis de la solución actual incluyendo:

* Un diagnóstico que contenga el análisis de los datos existentes y la estructura de la base de datos que los almacena.
* El análisis del código fuente (algoritmos de búsqueda, funcionalidades ofrecidas, etc.), considerando criterios y decisiones que estén programados a la hora de dar los servicios actuales.

# 2. Análisis de la base de datos existente

El esquema evaluado es TSUBASA, alojado en una base de datos PostgreSQL e importado a partir de un fichero comprimido facilitado por Agesic.

Estas son las tablas y el número de registros de cada una:

|  |  |
| --- | --- |
| **Tabla** | **Num. Registros Estimado** |
| alias | 37.410 |
| alias\_departamento | 1 |
| alias\_entidad\_colectiva | 1 |
| alias\_localidad\_geo | 758 |
| alias\_numero\_puerta | 2.990 |
| alias\_tramo\_ruta | 183 |
| bloque | 0 |
| calle | 26.496 |
| calle\_departamento | 26.579 |
| calle\_localidad | 47.839 |
| calles\_aux | 123.467 |
| calles\_unificadas\_aux | 88.058 |
| cambios\_direccion2 | 2.228.457 |
| chivo\_cp\_libro | 2.863 |
| chivo\_pk\_libro | 807 |
| chivo\_rc\_libro | 20.737 |
| codigos\_clasificacion | 1.153 |
| codigos\_postales\_vigentes | 121 |
| departamento | 20 |
| dir\_encontradas | 868.265 |
| dir\_no\_encontradas | 82.488 |
| direccion\_interna | 0 |
| ejes\_modi | 53.886 |
| entidad\_colectiva | 5 |
| esquinas | 34.930 |
| estadisticas | 0 |
| eventos\_puntos | 1.260 |
| forma\_punto | 21 |
| historico | 2.048.390 |
| iso\_metadata | 0 |
| iso\_metadata\_reference | 0 |
| loc\_cp\_borrar | 1.883 |
| localidades\_no\_oficiales | 1.916 |
| log | 16.214 |
| log\_base | 13.435.461 |
| manzana | 54.647 |
| manzana\_old | 37.123 |
| origen | 4 |
| origen\_alias | 14 |
| palabra | 9.564 |
| plugin\_clientes\_observaciones | 1 |
| pta\_del\_este\_auxiliar | 972 |
| punto | 1.596.041 |
| punto\_notable | 1.114 |
| puntos\_notables\_a\_procesar | 0 |
| puntos\_quiebre | 177.781 |
| resto | 4 |
| ruta | 115 |
| sesion | 1.168 |
| sinonimo | 896 |
| sinonimo\_entidad | 2 |
| sinonimo\_localidad | 10 |
| sistema | 2 |
| solar | 841.003 |
| spatial\_ref\_sys | 5.757 |
| tipo\_alias | 5 |
| tipo\_entidad\_colectiva | 13 |
| tipo\_punto\_notable | 4 |
| tipo\_tramo\_ruta | 0 |
| tipo\_vialidad | 18 |
| torre | 0 |
| tramo\_cp\_libro | 28.341 |
| tramo\_pk\_libro | 10.385 |
| tramo\_rc\_libro | 44.886 |
| tramo\_ruta | 186 |
| tramos | 6.251 |
| validacion\_calle | 262 |
| validacion\_punto | 164 |
| zonas\_paquetes | 17 |

Podemos ver que hay varias tablas que seguramente no se usan, ya que están vacías o tienen muy pocos registros.

En cuanto a las tablas que mayor número de registros, encontramos **log\_base**, con 13 millones de registros que seguramente no vamos a necesitar.

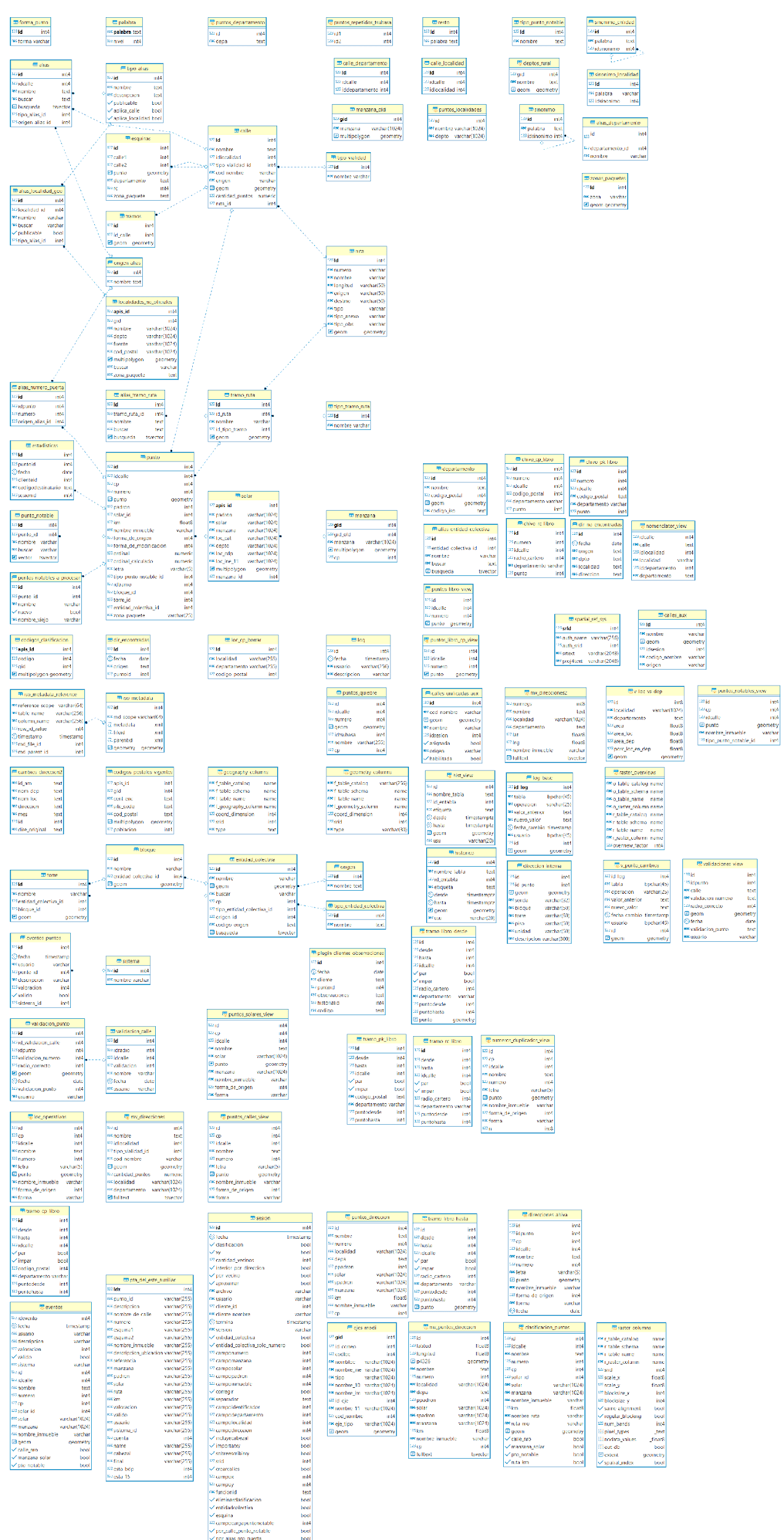
Luego tenemos **cambios\_direccion2** con 2.2 millones de registros, y la tabla **historico** con2 millones de registros. Parecen demasiados cambios de dirección, y viendo la tabla, lo que cambia es el campo id\_am y el campo id. (Preguntar el propósito y la funcionalidad de la tabla).

La siguiente tabla con más registros es **punto**, y contiene una geometría de tipo punto en el sistema de referencia EPSG:32721 (WGS 84 / UTM zone 21).

Esta tabla es la que hemos utilizado como base para el piloto de búsqueda de direcciones, porque parece la más completa. Sin embargo, contiene muchos campos con valores nulos, probablemente porque contiene datos relativos a conceptos distintos. Hay puntos que provienen de las calles, otros de los puntos notables de interés, y otros de manzana y solar.

Quizás deberíamos plantearnos separarla en varias tablas, y buscar de manera separada y ofrecer los datos relevantes de cada consulta.

Diagrama Entidad-Relación del esquema Tsubasa



## 2.1 Datos duplicados

Existe una vista en Tsubasa llamada **numeros\_duplicados\_view** con la siguiente definición:

**CREATE** **OR** **REPLACE** **VIEW** public.numeros\_duplicados\_view

**AS** **SELECT** a.id,

a.cp,

a.idcalle,

a.nombre,

a.numero,

a.letra,

a.punto,

a.nombre\_inmueble,

a.forma\_de\_origen,

a.forma,

a.n

**FROM** ( **SELECT** puntos\_calles\_view.id,

puntos\_calles\_view.cp,

puntos\_calles\_view.idcalle,

puntos\_calles\_view.nombre,

puntos\_calles\_view.numero,

puntos\_calles\_view.letra,

puntos\_calles\_view.punto,

puntos\_calles\_view.nombre\_inmueble,

puntos\_calles\_view.forma\_de\_origen,

puntos\_calles\_view.forma,

**count**(\*) **OVER** (**PARTITION** **BY** puntos\_calles\_view.idcalle, puntos\_calles\_view.numero) **AS** n

**FROM** puntos\_calles\_view) a

**WHERE** a.n > 1;

La vista generada tiene 181.510 registros, sin embargo, habría que incluir en la cláusula OVER también el campo letra, porque muchos puntos no están en realidad repetidos (tiene distinta letra de portal). Con esa modificación se obtienen 160.912 registros.

Otra consulta que puede ser útil es saber los puntos que están unos encima de otros:

**WITH** **data** **AS** (

**SELECT** id, idcalle, numero, letra, punto, nombre\_inmueble, **count**(\*) **OVER** (**PARTITION** **BY** punto)

**FROM** punto

)

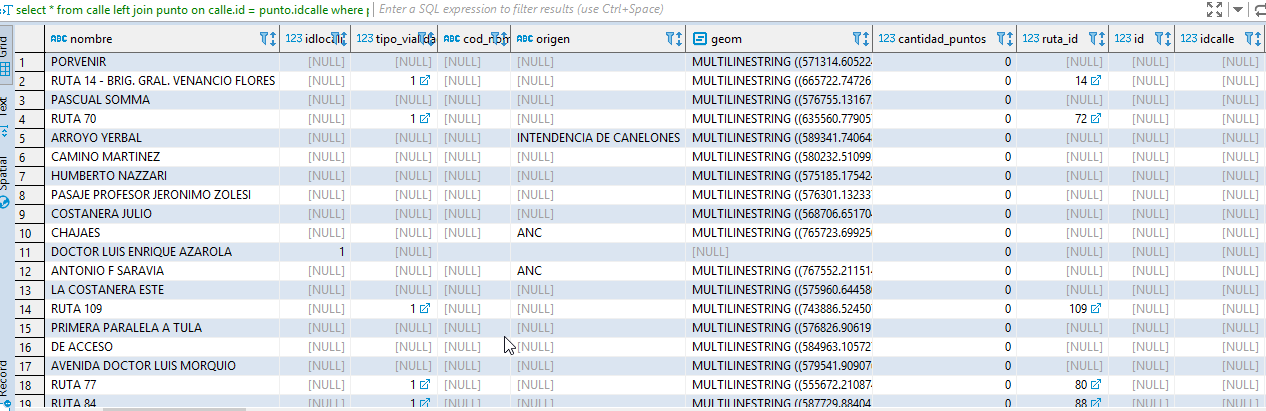
**SELECT** \* **FROM** **data** **WHERE** **count** > 1;

Con esto obtenemos un total de 11.433 puntos que coinciden sus coordenadas exactamente.

## 2.2 Ejes de calles vs direcciones en esa calle

1. Si analizamos las tablas de calle y punto encontramos algunos problemas, como por ejemplo calles que no tienen asociado ningún punto:
2. select count(\*)
3. from calle
4. left join punto on calle.id = punto.idcalle
5. where punto.idcalle is null

Obtenemos 272 registros (calles con 0 puntos asociados)



En el otro sentido, puntos que no tienen ninguna calle asociada, y no son registros de padrón, manzana, solar o nombre\_inmueble:

select \*

from calle

right join punto on calle.id = punto.idcalle

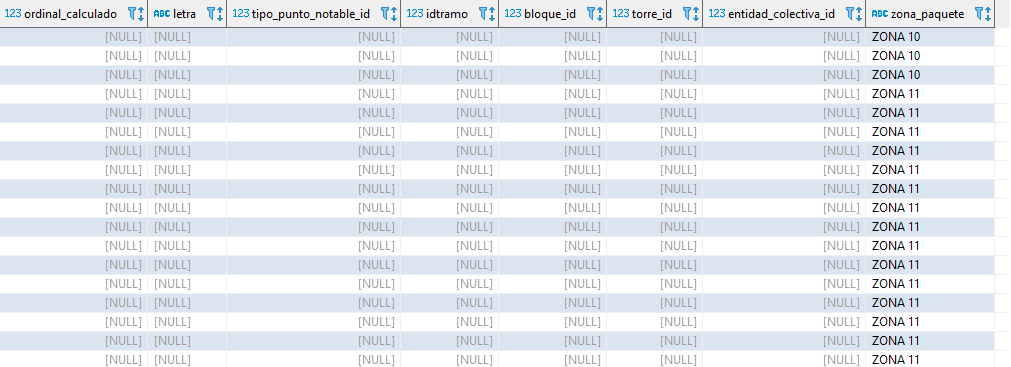
where calle.id is null

and padron is null

and nombre\_inmueble is null

arroja un total de 1331 registros que parece que tienen todos los campos nulos excepto el

campo zona\_paquete:



También es interesante la siguiente consulta:

select calle.id, calle.nombre, min(numero) as minPortal, max(numero)

from calle

left join punto on calle.id = punto.idcalle

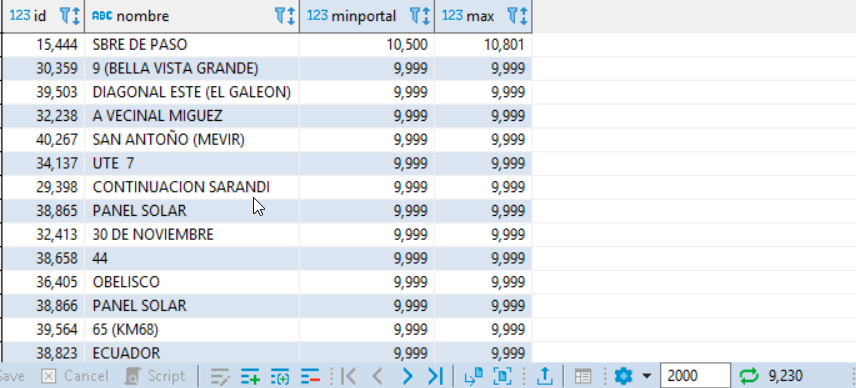
group by calle.id, calle.nombre, punto.idcalle

having punto.idcalle is not null and min(numero) is not null and min(numero) > 1

and calle.geom is null

order by minPortal desc

que no sirve para seleccionar las calles que tienen direcciones (registros en tabla punto) y enlazan con calle, pero no tienen geometría (9.230 registros):



## 2.3 Direcciones en localidad/departamento vs geometría de localidad/departamento

* En estas consultas se está enlazando de manera geométrica (usando ST\_INTERSECTS a pesar de que es más lento) porque hay muchos registros en la tabla punto que no tienen rellenado el campo id de entidad o de departamento. Para acelerar estas consultas, se recomienda calcular correctamente ese campo y luego enlazar po id.

La siguiente consulta sirve para seleccionar las direcciones que no tienen un departamento asignado (798 registros):

select \*

from punto

left join calle\_departamento on punto.idcalle = calle\_departamento.idcalle

left join departamento on calle\_departamento.iddepartamento = departamento.id

where punto.idcalle is not null

and departamento.id is null

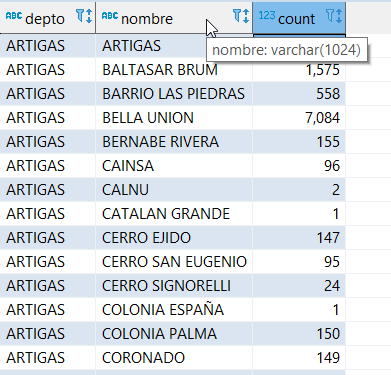
y aquí podemos ver los puntos que hay dentro de cada localidad\_no\_oficial:

SELECT l.depto,l.nombre, count(\*)

FROM punto as p

left join localidades\_no\_oficiales as l on st\_contains(l.multipolygon,p.punto)

group by l.depto, l.nombre



y se ven 81.808 registros fuera de los polígonos de localidad\_no\_oficial.



## 2.4 Campos vacíos

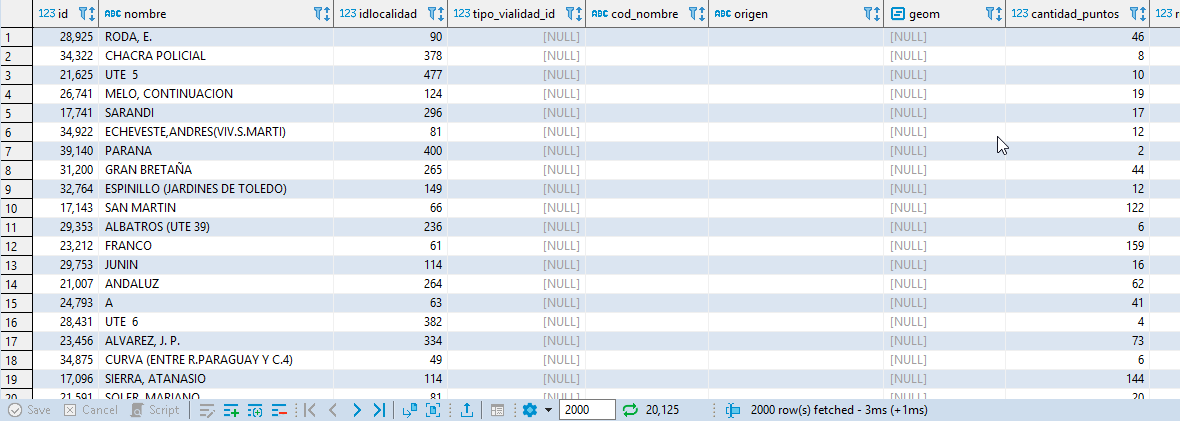
* El campo “tipo\_via” no se emplea en la base de datos. En la tabla “calle” está a menudo sin rellenar, y sería útil a la hora de buscar y categorizar las calles.
* Además, muchos registros de la tabla “calle” no tienen geometría:

select \*

from calle

where geom is null

nos da un total de 20.125 registros sin geometría de calle:



## 2.5 Inconsistencias mayores

También encontramos algunos problemas al enlazar por departamento:

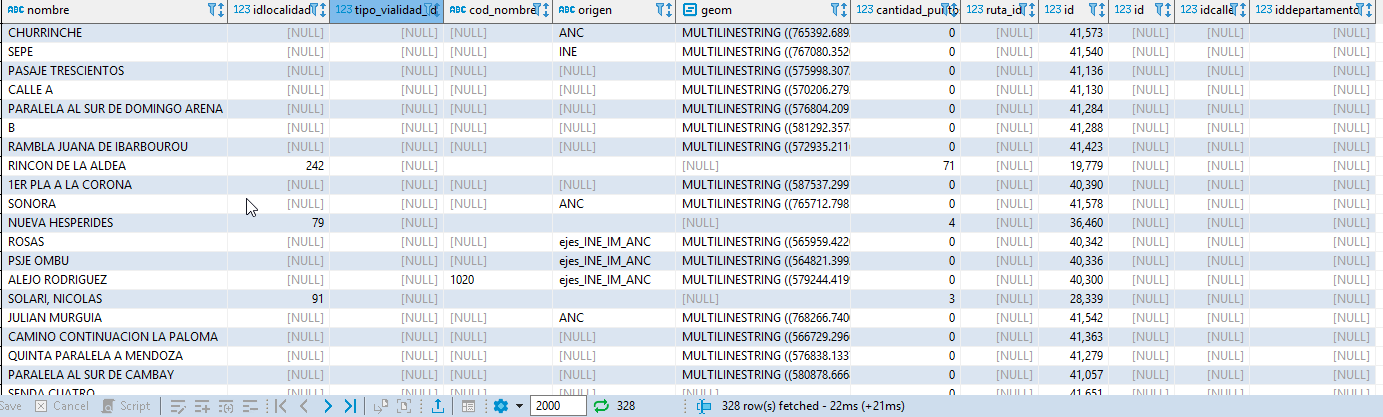
select \*

from calle

left join calle\_departamento on calle.id = calle\_departamento.idcalle

where iddepartamento is null

nos da un total de 328 registros de calle que no tienen un id de departamento válido:



A revisar: El número de registros en la tabla “punto” con “solar\_id” not null es 603.261, en cambio, el número de registros en la tabla “solar” es 841.003 registros. Eso quiere decir que esas dos tablas seguramente no están sincronizadas, y es un argumento más para no incluir los centroides de solar en la tabla “punto” (o añadir algún mecanismo que sincronice los registros).

## 2.6 Coherencia en el crecimiento/decrecimiento de los números de puerta

Esta consulta

select calle.id, calle.nombre, min(numero) as minPortal, max(numero)

from calle

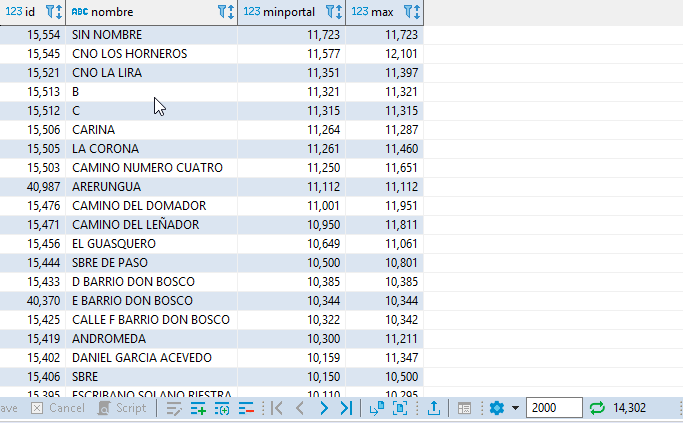
left join punto on calle.id = punto.idcalle

group by calle.id, calle.nombre, punto.idcalle

having punto.idcalle is not null and min(numero) is not null and min(numero) > 1

order by minPortal desc

nos sirve para ver que muchas calles parece que no tienen completos sus números de portal (no empiezan en 1) (14302 registros):



## 2.5 Inventario de capas geométricas y su cobertura

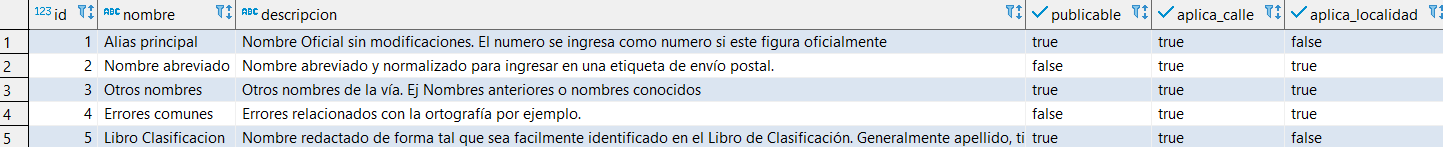
* Inventario de capas geométricas y su cobertura (Ej. Falta de ejes de calle en Localidades).

|  |  |  |
| --- | --- | --- |
| **Capa** | **Tipo geometría** | **Cobertura** |
| bloque | MULTIPOLYGON | sin datos |
| calle | MULTILINESTRING |  |
| calles\_aux \* | MULTILINESTRING |  |
| calles\_unificadas\_aux \* | MULTILINESTRING |  |
| manzana\_old \* | MULTIPOLYGON |  |
| punto | POINT | todo el país |
| ruta | MULTILINESTRING | todo el país |
| solar | MULTIPOLYGON | todo el país |
| tramo\_ruta |  |  |
| clasificacion\_puntos | GEOMETRY |  |
| codigos\_clasificacion \* |  |  |
| codigos\_postales\_vigentes | MULTIPOLYGON | todo el país |
| departamento | MULTIPOLYGON | todo el país |
| deptos\_rural | GEOMETRY |  |
| direccion\_interna | POINT | sin datos |
| puntos\_calles\_view | GEOMETRY |  |
| direcciones\_ahiva \* | GEOMETRY |  |
| ejes\_modi \* | MULTILINESTRING |  |
| entidad\_colectiva | MULTIPOLYGON |  |
| esquinas | GEOMETRY |  |
| eventos \* | GEOMETRY |  |
| historico \* | GEOMETRY |  |
| hist\_view \* | GEOMETRY |  |
| iso\_metadata \* | GEOMETRY |  |
| loc\_operativos \* | GEOMETRY |  |
| localidades\_no\_oficiales | MULTIPOLYGON | todo el país |
| log\_base \* | GEOMETRY |  |
| manzana | MULTIPOLYGON | todo el país |
| numeros\_duplicados\_view | GEOMETRY |  |
| puntos\_libro\_cp\_view \* | GEOMETRY |  |
| puntos\_libro\_view \* | GEOMETRY |  |
| puntos\_notables\_view | GEOMETRY |  |
| puntos\_quiebre | GEOMETRY |  |
| puntos\_solares\_view | GEOMETRY |  |
| torre | MULTIPOLYGON |  |
| tramo\_libro\_desde \* | GEOMETRY |  |
| tramo\_libro\_hasta \* | GEOMETRY |  |
| tramos | MULTILINESTRING |  |
| v\_loc\_vs\_dep \* | GEOMETRY |  |
| v\_punto\_cambios \* | GEOMETRY |  |
| validacion\_punto | POINT |  |
| validaciones\_view | POINT |  |
| zonas\_paquetes | GEOMETRY |  |
| mv\_puntos\_direccion | GEOMETRY |  |
| mv\_direcciones | MULTILINESTRING |  |

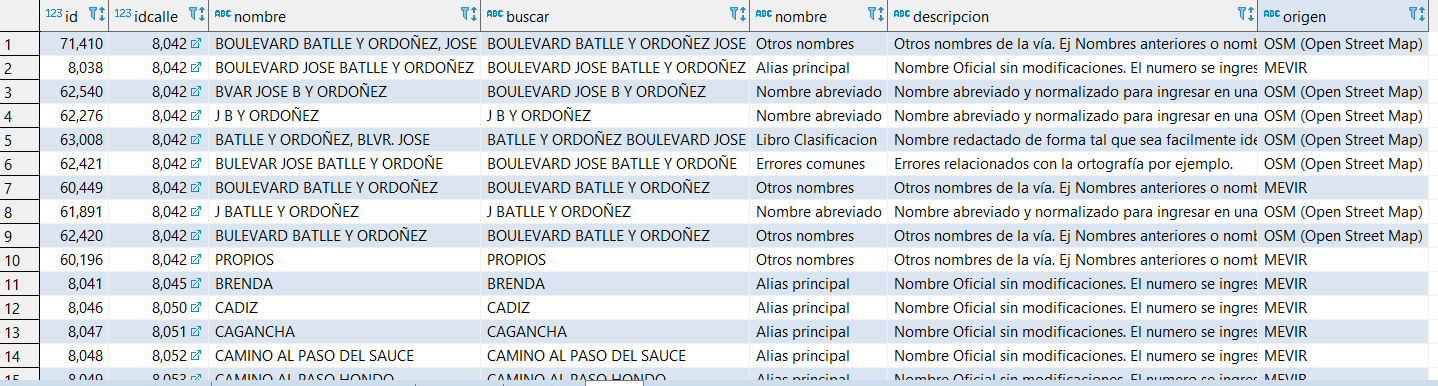
\* tablas/vistas que no corresponden a el proyecto

## 2.6.- Otras consideraciones

La tabla de alias es importante para mejorar las búsquedas. Los alias están tipificados usando la tabla “tipo\_alias”:



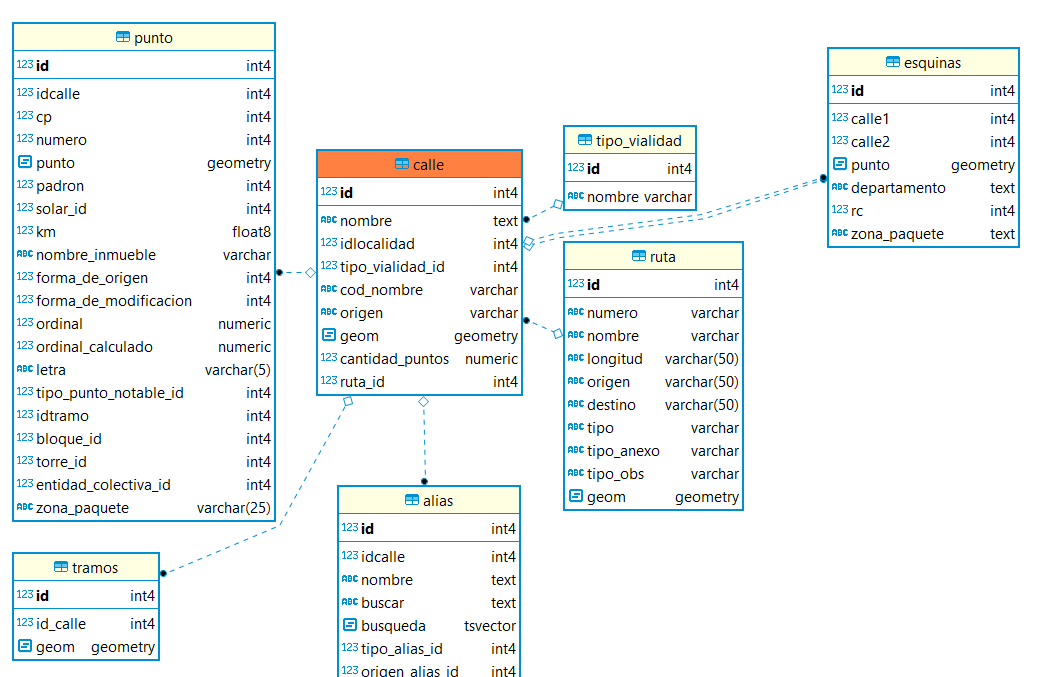
Algunos registros de la tabla “alias”:



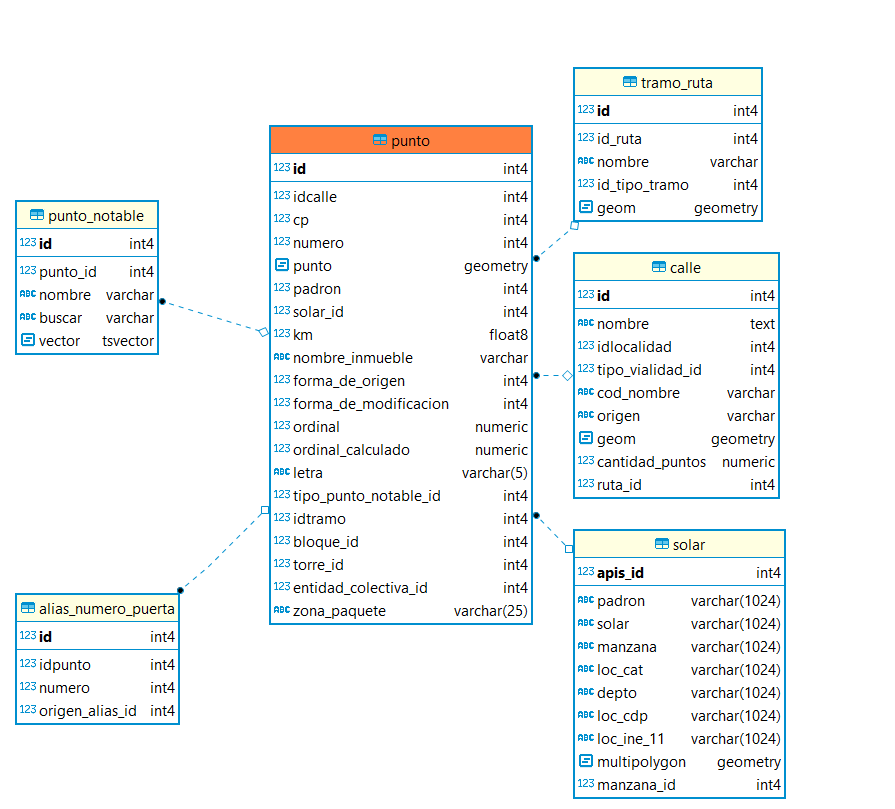
# 3. Detalle de campos de las tablas más importantes

La tabla más importante es “punto”, tal y como nos han explicado desde Correos. Es la que se utilizó para empezar a ubicar las direcciones, y tiene una mezcla de portales, centroides de solares, manzanas y puntos de interés (inmueble).

La tabla de “calle” y “alias” se relacionan por el campo idcalle, al igual que la tabla “punto”:



En cuanto a la tabla “punto”, estos son sus campos y sus relaciones principales:



Para terminar, la tabla “ruta” enlaza con “calle” y con “tramo\_ruta” con los campos ruta\_id y id\_ruta respectivamente:

## 4.1. Descripción de los campos de “punto”

|  |  |  |
| --- | --- | --- |
| Campo | Descripción | Tipo |
| id | Identificador único | int4 |
| idcalle | Identificador de calle | int4 |
| cp | Código postal | int4 |
| numero | Número de portal | int4 |
| punto | Punto en EPSG 32721 (WGS84 UTM Zone 21S) | Geom |
| padron | Identificador de Padron | int4 |
| solar\_id | Identificador de Solar | int4 |
| km | Punto Km (si aplica) | float8 |
| nombre\_inmueble | Punto singular o de interés (si aplica) | varchar |
| forma\_de\_origen | Id de origen (enlaza con tabla origen). Indica de dónde ha salido la información | int4 |
| forma\_de\_modificacion | Id de forma\_punto. Enlaza con la tabla forma\_punto para indicar cómo se modificó ese punto (Desde vista, desde Tsubasa, etc) | int4 |
| ordinal | Utilidad desconocida | numeric |
| ordinal\_calculado | Utilidad desconocida | numeric |
| letra | Letra del número de portal | varchar(5) |
| tipo\_punto\_notable\_id | Tipo de punto notable. Enlaza con la tabla tipo\_punto\_notable | int4 |
| idtramo | Identificador de tramo. Enlaza con la tabla tramo. | int4 |
| bloque\_id | Enlaza con la tabla bloque | int4 |
| torre\_id | Enlaza con la tabla torre | int4 |
| entidad\_colectiva\_id | Enlaza con la tabla entidad\_colectiva | int4 |
| zona\_paquete | Utilidad desconocida | varchar(25) |

## 4.1. Descripción de los campos de “calle”

|  |  |  |
| --- | --- | --- |
| Campo | Descripción | Tipo |
| id | Identificador único (Se usa para enlazar con otras tablas, por ejemplo calle\_localidad y calle\_departamento) | int4 |
| nombre | Nombre oficial de la calle (En mayúsculas) | varchar |
| idlocalidad | Identificador de Localidad. Enlaza con la tabla localidades\_no\_oficiales | int4 |
| tipo\_vialidad\_id | Id de tipo\_vialidad. No se usa en la práctica. | int4 |
| cod\_nombre | El código de la calle cuando proviene de un origen externo, por ejemplo ejes del INE. | varchar |
| origen | Origen de donde se han obtenido los datos | varchar |
| geom | Geometría de la calle (Multilinestring en EPSG 32721) | geom |
| cantidad\_puntos | Cantidad de puntos asociados a esa calle (calculados por algún proceso interno) | int4 |
| ruta\_id | Identificador de ruta (si es parte de una ruta) | int4 |

# 4. Análisis del código fuente

Junto con el código fuente, venía esta información:

“El tsubasa es una interfaz de escritorio que usan para cargar direcciones y además se utiliza como librería para los servicios web. Al momento de cargar direcciones, a través de tsubasa, se establecen qué conceptos se refieren para poder después implementar la búsqueda. Por las dudas, lo que se carga son direcciones para buscar y no nuevas direcciones a agregar a la base.

Proyecto IsisBúsquedaDireccionPlugin contiene y expone los servicios web.

Cuando agregan funcionalidad, la prueban en escritorio. Para determinar desde los servicios web que características o funcionalidad utilizan, habilitan o deshabilitan las mismas desde el componente de servidor CONTROLADOR, operación BUSCAR DIRECCIONES, ejemplo:

DataConfiguracion dc = new DataConfiguracion();

dc.setAproximar(true);

dc.setPorVecino(true);

dc.setPorAliasNroPuerta(true);

dc.setPorEsquina(true);

dc.setPorEntidadesColectivas(true);

dc.setPorCallePuntoNotable(false);

dc.setUsarCorrector(true);

dc.setFiltroGeografico(filtroGeografico);

dc.setServicioWeb(true);

dc.setPorInmuebleAproximado(true);

dc.setPorPtoNotable(true);

dc.setCantVecino(99);

Otras clases importantes del sistema son:

-Codificador, que arma las entradas para realizar la búsqueda, según el tipo de forma canónica en que se pase la dirección

-Corrector, corrige elementos de entrada.

-Controlador, recibe los pedidos web.Si no obtiene resultados al llamar a las búsquedas ejecuta estrategias de aproximación

-DriverPostgre, es el que arma las consultas sql, para las consultas geográficas, guarda en la base la dirección completa, normalizada y vectorizada (con frecuencia de palabras, esto se configura en postgres),

-AliasSinonimo, obtiene alias de las calles

AliasSinonimos también normaliza los parámetros de búsqueda utilizando la tabla de sinónimos. (AV por AVDA por ejemplo).

Otras consideraciones:

-Para la búsqueda por esquinas, todas las noches,se crea una tabla con las esquinas posibles, para no calcularlas a pedido que es muy costoso, los criterios para ser considerado esquina es que dos ejes de calle se corten, o que tengan determinado grado de aproximación.

El caso de puntos cercanos no se está utilizando más. Si se encontraran con que tienen un alto número de calles sin geometría o con geometrías incompletas podía ser una opción.

-Para determinadas operaciones, registran el tiempo que demora en ejecutar, para disponer de métricas de tiempos de ejecución, y ver qué operaciones son las más costosas.

-Disponen de una versión para wildfly, que no está en producción que utiliza pool de conexiones, para esto, se deben configurar los fuentes para la utilización del pool.

-Cuando la búsqueda de calle, la calle contiene un nombre que consta de una fecha y año, no puede buscar.

-Las Capas geográficas de la base son:

Entidades Colectivas

Punto de Interés

Localidades\_no\_oficiales

Calle

Departamento

Direcciones

“

La clase principal de donde podemos sacar más información es **DriverPostgres**

En esta clase encontramos las funciones del tipo “obtenerXXXXX:

* obtenerEntidadColectiva
* obtenerDepartamento
* obtenerLocalidad
* obtenerCodigoPostalDeDepto
* ObtenerCallesPorBuscar
* obtenerCallesUtilizandoAlias
* obtenerEsquinas
* obtenerPuntoCercanoACalle (¿Funciona??)
* obtenerPuntosPorCalleYNumero
* obtenerPuntosPorCalleYPuntoNotable
* obtenerPuntosPorEntidadColectivaTorreBloque
* obtenerPuntosPorManzana
* obtenerPuntosPorManzanaYSolar
* obtenerPuntosPorNombreInmueble
* obtenerPuntosPorNombreInmuebleAproximar
* obtenerPuntosPorPadron
* obtenerPuntosPorRutaYKm

También es importante la función normalizarCalle, que llama a la función homónima de la clase AliasSinonimos:

public String **normalizarCalle**(String calle) throws Exception {

if(calle==null){

return calle;

}

if (calle.**startsWith**("CL ")){

String aux[] = calle.**split**("CL ");

if (aux.length> 0){

calle=(aux[1]);

}

}

String cadena = calle.**trim**().**toUpperCase**();

cadena = StringUtils.**reemplazarCaracteresRaros**(cadena);

boolean seguir = true;

while (seguir) {

seguir = false;

for (String sinonimo : this.**obtenerHashDeSinonimos**().**keySet**()) {

String nueva = **sustituirPrimero**(cadena, sinonimo, this.**obtenerHashDeSinonimos**().**get**(sinonimo));

if (!nueva.**equals**(cadena)) {

seguir = true;

cadena = nueva;

}

}

}

cadena = cadena.**trim**();

return cadena;

}

La función obtenerHashDeSinonimos está en la clase DriverPostgres:

public HashMap<String, String> **obtenerHashDeSinonimos**() throws Exception {

if (Testing.TESTING) {

Testing.**empiezaFuncion**("obtenerHashDeSinonimos");

}

HashMap<String, String> salida = new HashMap<String, String>();

String consulta = "select s1.palabra, s2.palabra "

+ "from sinonimo s1 join sinonimo s2 on s1.idsinonimo=s2.id";

PreparedStatement ps = this.**prepareStatement**(consulta);

if (TEST) {

logger.**info**(ps.**toString**());

}

ResultSet rs = ps.**executeQuery**();

while (rs.**next**()) {

salida.**put**(**getString**(rs, 1), **getString**(rs, 2));

}

rs.**close**();

ps.**close**();

if (Testing.TESTING) {

Testing.**terminaFuncion**("obtenerHashDeSinonimos");

}

return salida;

}

Esto quiere decir, que se está llamando a esta función cada vez que se busca una calle (la clase AliasSinonimos se crea cada vez, así que el servicio web ganaría velocidad si se cachea el hash de sinónimos.

Se ha detectado también una incongruencia entre la documentación y el código fuente del servicio de direcciones (SugerenciaCalleCompleta).

En la documentación (serviciosdedireccionesv.pdf) se usa el parámetro todos=todos.

Sin embargo, para obtener todos los valores, el valor debe ser “true”, y cualquier otra cosa para obtener los 6 resultados con más puntos asociados (se ordena por la suma del campo cantidad\_puntos de la tabla “calle”).

Otro posible cuello de botella es que la clase DriverPostgres se crea en cada petición, y se conecta y desconecta de la base de datos (aunque probablemente la principal pérdida de tiempo está en la generación de la tabla hash con los sinónimos).

El código fuente donde se hace la petición está en la clase SugerenciaCalleCompleta y es el siguiente:

protected void **processRequest**(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

try {

*/// response.setContentType("application/json;charset=UTF-8");*

Boolean todos = true;

try {

if (!request.**getParameter**("todos").**equals**("true")) {

todos = false;

}

} catch (Exception e) {

}

String entrada = request.**getParameter**("entrada");

if (entrada == null) {

entrada = request.**getParameter**("query");

}

try {

entrada = StringUtils.**codificarUTF**(entrada);

} catch (Exception ex) {

}

String tipoRespuesta = request.**getParameter**("tipoRespuesta");

DriverPostgres driver = new **DriverPostgres**();

driver.**conectar**();

AliasSinonimos aliasSinonimos = new **AliasSinonimos**(driver);

List<String[]> sugerencias = aliasSinonimos.**obtenerSugerenciasDeCallesCompleta**(entrada, todos);

driver.**desconectar**();

if (tipoRespuesta != null && tipoRespuesta.**equals**("json")) {

Gson gson = new **Gson**();

List<ItemSalida> toRet = new LinkedList<>();

for (String[] aux : sugerencias) {

toRet.**add**(new **ItemSalida**(aux[0], aux[1], aux[2]));

}

String respuesta = gson.**toJson**(toRet);

*// response.getOutputStream().write(respuesta.getBytes());*

response.**getOutputStream**().**write**(respuesta.**getBytes**("UTF-8"));

} else {

String respuesta = "<ul>";

Integer maximo = sugerencias.**size**();

if (!todos) {

maximo = Math.**min**(5, sugerencias.**size**());

}

for (int i = 0; i < maximo; i++) {

String[] sug = sugerencias.**get**(i);

*// String textoParaMostrar = "<b>" + StringEscapeUtils.escapeHtml(sug[0]) + "</b>, " + StringEscapeUtils.escapeHtml(sug[1]) + ", " + StringEscapeUtils.escapeHtml(sug[2]);*

*// String titleParaMostrar = StringEscapeUtils.escapeHtml(sug[0]) + ", " + StringEscapeUtils.escapeHtml(sug[1]) + ", " + StringEscapeUtils.escapeHtml(sug[2]);*

String textoParaMostrar = "<b>" + (sug[0]) + "</b>, " + (sug[1]) + ", " + (sug[2]);

String titleParaMostrar = (sug[0]) + ", " + (sug[1]) + ", " + (sug[2]);

respuesta += "<li title='" + titleParaMostrar + "'><p class='sugerencia'>" + textoParaMostrar + "</p></li>";

}

*// if (sugerencias.size() == 6){*

*// respuesta += "<li></li>";*

*// }*

if (!todos && sugerencias.**size**() > 5) {

respuesta += "</ul><div class='verMasResultadosDiv'> <a href='javascript:void(0)' onclick='barraBusquedaDIRPlugin.verMasResultados();'>" + StringEscapeUtils.**escapeHtml**("Ver más resultados") + " </a> </div>";

} else {

respuesta += "</ul>";

}

*//response.setCharacterEncoding("UTF-8");*

response.**getOutputStream**().**write**(respuesta.**getBytes**());

*// response.getOutputStream().write(respuesta.getBytes("UTF-8"));*

}

} catch (Exception e) {

e.**printStackTrace**();

}

}

En la clase AliasSinonimos se intenta obtener la localidad y el departamento (separados por comas), y sustituir las cadenas de sinónimos (esto quizás se pueda mejorar en el futuro).

*/\*\**

*\* Retorna la sugerencia de calle para una entrada dada*

*\**

*\* @param calle*

*\* @return*

*\* @throws* **Exception**

*\*/*

public List<String[]> **obtenerSugerenciasDeCallesCompleta**(String entrada, Boolean todos) throws Exception {

String[] partes = entrada.**split**(",");

String calle;

String localidad = null;

String departamento = null;

String localidadODepto = null;

String numero = "";

if (partes.length == 1) {

calle = partes[0];

} else if (partes.length == 2) {

calle = partes[0];

localidadODepto = partes[1];

} else {

*// partes.length > 2*

calle = "";

for (int i = 0; i < partes.length - 2; i++) {

calle += partes[i];

}

AliasSinonimos as = new **AliasSinonimos**(driver);

*// driver.conectar();*

*//localidad = partes[partes.length - 2];*

localidad = as.**normalizarNombreLocalidad**(partes[partes.length - 2]);

*// driver.desconectar();*

departamento = partes[partes.length - 1];

}

String calleIncompleta = null;

if (!calle.**endsWith**(" ") */\*&& partes.length == 1 \*/*) { *// Si se termino de escribir la palabra pero no se escribio localidad o depto*

String[] palabras = calle.**split**(" ");

calle = "";

for (int i = 0; i < palabras.length - 1; i++) {

calle += palabras[i] + " ";

}

calleIncompleta = palabras[palabras.length - 1];

try {

*// Si la ultima palabra es un numero no le doy bola*

new **Integer**(calleIncompleta);

numero = calleIncompleta;

calleIncompleta = null;

} catch (Exception e) {

}

}

*// driver.conectar();*

String calleNormalizada = driver.**normalizarCalle**(calle);

List<String[]> toRet = driver.**obtenerSugerenciasDeCallesCompleta**(calleNormalizada, calleIncompleta, localidad, departamento, localidadODepto, numero, todos);

*// driver.desconectar();*

return toRet;

}

Y por último, se llama a la función obtenerSugerenciasDeCallesCompleta con la calleNormalizada y calleIncompleta, además del resto de parámetros. En esa función es donde se monta la consulta SQL que se lanza a la base de datos:

*//La localidad se pasa normalizada para matchear con el campo buscar de la base*

public List<String[]> **obtenerSugerenciasDeCallesCompleta**(String calle, String calleIncompleta, String localidad, String departamento, String localidadODepto, String numero, Boolean todas) throws Exception {

if (Testing.TESTING) {

Testing.**empiezaFuncion**("obtenerSugerenciasDeCallesCompleta");

}

calle = **convertirParaTsQuery**(calle);

if (calleIncompleta != null) {

calleIncompleta = calleIncompleta.**trim**();

}

if (localidad != null) {

localidad = localidad.**trim**();

}

if (departamento != null) {

departamento = departamento.**trim**();

}

if (localidadODepto != null) {

localidadODepto = localidadODepto.**trim**();

}

if (!numero.**equals**("")) {

numero = " " + numero;

}

String consulta = ""

+ " select "

+ " c.nombre || '" + numero + "' as nombre , coalesce(l.nombre, 'SIN LOCALIDAD') as localidad, coalesce(d.nombre, 'SIN DEPARTAMENTO') as departamento, sum(c.cantidad\_puntos) as cantidad\_puntos "

+ " from "

+ " calle c "

+ " join alias a on a.idcalle = c.id "

+ " join calle\_localidad cl on (c.id = cl.idcalle) "

+ " join localidades\_no\_oficiales l on (l.apis\_id = cl.idlocalidad) "

+ " left join alias\_localidad\_geo al on al.localidad\_id = l.apis\_id "

+ " join calle\_departamento cd on (c.id = cd.idcalle) "

+ " join departamento d on (d.id = cd.iddepartamento and st\_intersects(l.multipolygon, d.geom)) "

+ " where 1=1 ";

if (calle != null && !calle.**equals**("")) {

consulta += " and a.buscar @@ to\_tsquery(?) ";

}

if (localidad != null) {

consulta += " and (l.buscar ilike '%' || ? ||'%' "

+ " or "

+ " al.buscar ilike '%' || ? ||'%' )";

}

if (departamento != null) {

consulta += " and d.nombre ilike '%' || ? ||'%' ";

}

if (localidadODepto != null) {

consulta += " and (d.nombre ilike '%' || ? ||'%' or l.buscar ilike '%' || ? ||'%' or al.buscar ilike '%' || ? ||'%' ) ";

}

if (calleIncompleta != null) {

consulta += " and a.buscar ilike '%' || ? ||'%' ";

}

consulta += " group by c.nombre, l.nombre, d.nombre ";

if (!todas) {

consulta += " order by cantidad\_puntos desc limit 6 ";

} else {

consulta += " order by departamento, localidad, nombre ";

}

PreparedStatement ps = this.**prepareStatement**(consulta);

int pos = 1;

if (calle != null && !calle.**equals**("")) {

ps.**setString**(1, calle);

pos++;

}

if (localidad != null) {

ps.**setString**(pos, localidad);

pos++;

ps.**setString**(pos, localidad);

pos++;

}

if (departamento != null) {

ps.**setString**(pos, departamento);

pos++;

}

if (localidadODepto != null) {

ps.**setString**(pos, localidadODepto);

pos++;

ps.**setString**(pos, localidadODepto);

pos++;

ps.**setString**(pos, localidadODepto);

pos++;

}

if (calleIncompleta != null) {

ps.**setString**(pos, calleIncompleta);

pos++;

}

if (TEST) {

logger.**info**(ps.**toString**());

}

ResultSet rs = ps.**executeQuery**();

List<String[]> toRet = new LinkedList<>();

while (rs.**next**()) {

String nombre = rs.**getString**("nombre");

String loc = rs.**getString**("localidad");

String dep = rs.**getString**("departamento");

String[] sugerencia = {nombre, loc, dep};

toRet.**add**(sugerencia);

}

ps.**close**();

if (Testing.TESTING) {

Testing.**terminaFuncion**("obtenerSugerenciasDeCallesCompleta");

}

return toRet;

}

Para terminar, otra de las posibles mejoras es esta parte de la consulta:

join departamento d on (d.id = cd.iddepartamento and st\_intersects(l.multipolygon, d.geom)) "

y que es algo que se repite en otras consultas. No se debería hacer un join utilizando st\_intersects. Por un lado, se pierden puntos fuera de los departamentos, como Isla de Flores, y por otro y más importante, el join es mucho más lento que si utilizamos un campo como el idDepartamento. Si ese campo no existe o no es de fiar, lo podemos calcular con una sentencia SQL de tipo UPDATE, y a partir de ahí, utilizar el id para hacer los JOIN.

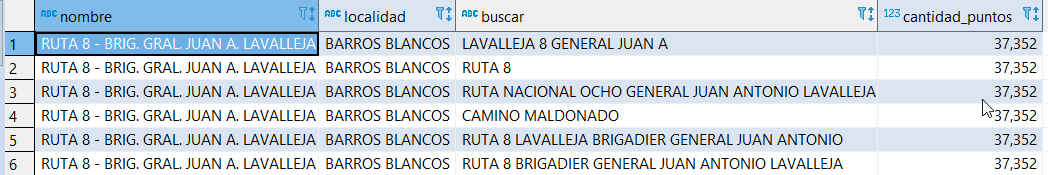
Es MUY IMPORTANTE darse cuenta del JOIN entre “calle” y “alias”:

from calle c

join alias a on a.idcalle = c.id

join calle\_localidad cl on (c.id = cl.idcalle)

Este JOIN hace que existan varios registros dentro de los cuales se busca por varios nombres de calle. Ejemplo para la Ruta 8 - BRIG. GRAL. JUAN. A. LAVALLEJA:

El resultado con el JOIN tiene 84.743 registros. Sin el JOIN (tabla calle solo), tiene 57.997 registros. No supone un incremento demasiado grande, y permite encontrar muchas calles con distintas variaciones.

Hay otro fallo importante, y que explica la lentitud en las búsquedas. La consulta hace una búsqueda fulltext en el campo a.buscar:

if (calle != null && !calle.**equals**("")) {

consulta += " and a.buscar @@ to\_tsquery(?) ";

}

pero en realidad el campo preparado para fulltext search es “a.busqueda”.

Los registros que encuentra se encuentran gracias a una cláusula posterior:

if (calleIncompleta != null) {

consulta += " and a.buscar ilike '%' || ? ||'%' ";

}

que es mucho más lenta.

# 5. Optimización de la sugerencia de calles.

Para permitir que se utilice el servicio de sugerencia de calles mientras el usuario teclea, se necesita que sea rápido.

Para mejorar la rapidez, hemos creado 2 vistas materializadas en base a la tabla punto, que es la más completa, y varios índices.

La consulta con la que se generan estas vistas es la siguiente:

-- CREATE EXTENSION unaccent;

-- View: public.puntos\_direccion

CREATE MATERIALIZED VIEW public.mv\_puntos\_direccion AS

SELECT punto.id,

st\_y(st\_transform(punto.punto, 4326)) AS latitud,

st\_x(st\_transform(punto.punto, 4326)) AS longitud,

st\_transform(punto.punto, 4326) AS p4326,

punto.idcalle,

calle.nombre,

punto.numero,

puntos\_localidades.nombre AS localidad,

puntos\_departamento.depa,

punto.padron AS ppadron,

solar.solar,

solar.padron AS spadron,

solar.manzana,

punto.km,

punto.nombre\_inmueble,

punto.cp,

to\_tsvector('spanish'::regconfig, (((((unaccent(COALESCE(calle.nombre, ' '::text)) || ' '::text) || unaccent(COALESCE(punto.nombre\_inmueble, ' '::text::character varying)::text)) || ' '::text) || unaccent(COALESCE(puntos\_localidades.nombre, ' '::text::character varying)::text)) || ' '::text) || unaccent(COALESCE(puntos\_departamento.depa, ''::text))) AS fulltext

FROM punto

LEFT JOIN calle ON punto.idcalle = calle.id

LEFT JOIN puntos\_localidades ON punto.id = puntos\_localidades.id

LEFT JOIN solar ON punto.solar\_id = solar.apis\_id

LEFT JOIN puntos\_departamento ON punto.id = puntos\_departamento.id;

-- limit 100;

CREATE INDEX idx\_fulltext\_mv\_puntos\_direccion

ON public.mv\_puntos\_direccion USING gin

(fulltext);

CREATE MATERIALIZED VIEW public.mv\_direcciones2

AS

SELECT DISTINCT count(\*) AS numregs,

mv\_puntos\_direccion.idcalle,

mv\_puntos\_direccion.nombre,

alias.buscar,

mv\_puntos\_direccion.localidad,

mv\_puntos\_direccion.depa AS departamento,

avg(mv\_puntos\_direccion.latitud) AS lat,

avg(mv\_puntos\_direccion.longitud) AS lng,

mv\_puntos\_direccion.nombre\_inmueble,

to\_tsvector('spanish'::regconfig, (((((unaccent(COALESCE(alias.buscar, ' '::text)) || ' '::text) || unaccent(COALESCE(mv\_puntos\_direccion.nombre\_inmueble, ' '::text::character varying)::text)) || ' '::text) || unaccent(COALESCE(mv\_puntos\_direccion.localidad, ' '::text::character varying)::text)) || ' '::text) || unaccent(COALESCE(mv\_puntos\_direccion.depa, ''::text))) AS fulltext

FROM mv\_puntos\_direccion

JOIN alias ON alias.idcalle = mv\_puntos\_direccion.idcalle

GROUP BY mv\_puntos\_direccion.nombre, alias.buscar, mv\_puntos\_direccion.idcalle, mv\_puntos\_direccion.localidad, mv\_puntos\_direccion.depa, mv\_puntos\_direccion.fulltext, mv\_puntos\_direccion.manzana, mv\_puntos\_direccion.nombre\_inmueble

HAVING mv\_puntos\_direccion.manzana IS NULL

WITH DATA;

CREATE INDEX idx\_fulltext\_mv\_direcciones2

ON public.mv\_direcciones2 USING gin

(fulltext)

TABLESPACE pg\_default;

CREATE INDEX idx\_puntos\_direccion\_p4326 ON public.mv\_puntos\_direccion USING GIST(p4326);

CREATE INDEX idx\_depa\_localidad\_nombre ON mv\_puntos\_direccion (depa, localidad, nombre);

CREATE INDEX mv\_direcciones2\_idcalle\_idx ON public.mv\_direcciones2 USING btree (idcalle);

La primera vista materializada (mv\_puntos\_direccion) se genera a partir de la tabla “punto” y sus enlaces (joins) con las tablas de “calle”, “puntos\_localidades”, “solar” y “puntos\_departamento”. Esta es la vista de la que obtendremos la ubicación final.

A partir de esta vista, generamos otra vista materializada: “mv\_direcciones2”, que es la que vamos a emplear para la búsqueda de candidatos. La vista la generamos con una consulta de tipo DISTINCT sobre los campos ya existentes de “mv\_puntos\_direccion”, y hacemos un JOIN con la tabla “alias” para buscar todas los variantes de una calle.

Esta tabla es la que usamos para hacer la búsqueda de candidatos. Una vez que el usuario selecciona una calle, entonces usamos el campo idcalle para obtener todos los números de portal de la vista “mv\_puntos\_direccion”, y buscar ahí el portal en concreto si el usuario lo ha especificado en la consulta.

El índice espacial (“idx\_puntos\_direccion\_p4326”) que creamos sirve para que la búsqueda de geocodificación inversa sea muy rápida.

Usando estas vistas materializadas, las consultas son mucho más rápidas, y permiten obtener resultados muy similares a los que se obtienen en el servicio actual, pero con tiempos de respuesta por debajo de los 50 msegs.

# 6. Consideraciones Finales

Se deberían corregir (poco a poco) los problemas detectados en la base de datos y en los nuevos desarrollos, tener en cuenta los fallos y las optimizaciones del punto anterior.

Además, ahora mismo, en la tabla de punto se mezclan los puntos de calles, de inmuebles especiales y puntos de interés (sucursales de correos, farmacias, etc) las rutas y los centroides de solares, manzanas y padrones.

Planteamos la posibilidad de separar las direcciones. Por un lado los portales (asociados a las calles - tramos con el idcalle), y por otro los POIs (Puntos de Interés).

Los centroides de solares, manzanas y padrones, directamente se pueden quitar y calcularlos al vuelo cuando sea necesario a partir de los polígonos de padrón, manzana y solar según un parámetro en el API. Con esto evitaríamos duplicidades y tener que editar el centroide cuando un solar o manzana se segrega.

Hay dos ventajas principales al hacer este cambio.

Por un lado, la mayor parte de las búsquedas se hace sobre calle y número de portal, así que quitar los centroides de la tabla “punto” serviría para acelerar las consultas de tipo calle y nº de portal.

Además, será más sencillo dar de alta nuevos portales y calles, editar y realizar el mantenimiento, ya que no se mezclan conceptos (y en el caso de los centroides de manzana y solar, ni siquiera hay que mantenerlos, ya que el centroide se puede calcular en base a los polígonos de manzana y solar. Si ese punto se originó como centroide, pero a posteriori se ha movido para reflejar mejor el punto de entrada a una manzana o solar, entonces habrá que mantenerlo, pero sigue siendo mejor en otra tabla aparte.

En cuanto a los puntos singulares, inmuebles o POIs, se aplica el mismo principio de mantener tablas separadas para conceptos separados.

Por otro lado, actualmente se calcula por la noche la tabla de esquinas. Esa tabla y ese proceso ya no serán necesarios, ya que las pruebas que hemos hecho han demostrado tiempos bastante buenos con los índices y consultas que hemos hecho (del orden de 30-40 msegs), y por lo tanto seguramente podemos prescindir también de esa tabla.

En cuanto a la tabla ruta, se aconseja seguir un modelo similar al que se usar en Cartociudad, y unificar ruta y calle en una única tabla “vial” (pág. 20 del documento de especificaciones de Cartociudad).

La idea principal es tener una única tabla “tramo” con campo geometría de tipo multilinestring. Los viales (calles o rutas) están formados por colecciones de tramos, que enlazan por el campo idcalle (o id\_vial si unificamos ruta y calle).

La tabla “vial” tendrá un campo “tipo\_vial” con los tipos de viales que podamos necesitar (ruta, calle, autovia, nacional, camino, etc) que podemos utilizar para dibujar con un estilo u otro en función del tipo de vía. La geometría solo se guarda en la tabla de “tramo”, de forma que sea sencillo editar una calle o ruta, y cuando se edita en un sitio, el cambio sirve también para el concepto de calle o ruta.

Para el caso de los portales y puntos kilométricos, en el caso de Cartociudad (pág. 21) también se utiliza una tabla unificada “portalpk” que se describe de la siguiente forma:

“Identificador y acceso exterior a cada construcción,

representado como un punto en el borde de la parcela a la que

pertenece, en caso de que el vial sea urbano. Para carretera,

representa el punto kilométrico, o distancia en kilómetros desde

el origen de la carretera.”

La tabla tiene un campo tipo\_porpk que indica si un registro representa a un portal o a un punto kilométrico.

Si todos estos cambios esbozados se consideran oportunos, en el siguiente documento “Propuesta de diseño de base de datos y migración” se detallarán las tablas necesarias y los scripts de copia para la migración.