Actividad práctica 5: SQL Espacial

Antes de importar los datos, deberás crear una base de datos espacial siguiendo las indicaciones del punto 7 en la *Guía de Usuario de PgAdmin*. Para el desarrollo de este ejercicio vamos a llamar a nuestra base de datos *PRACTICA5.*

Para importar los datos a la base de datos *PRACTICA5* vamos a utilizar la herramienta osm2pgsql que se trata en la lectura de importación/exportación a bases de datos PostgreSQL/PostGIS.

Para empezar con la importación abrimos una consola de comandos y accedemos a la carpeta donde previamente tenemos instalada la aplicación *osm2pgsql.*

Aplicamos el siguiente comando:

osm2pgsql -d PRACTICA5 -U postgres -W -P 5432 -S default.style -s -b 2.7719,41.9566,2.8642,42.0075 "C:\spain\spain-latest.osm"

Este proceso puede tardar varios minutos dependiendo de la capacidad de cada ordenador.

Uso de parámetros:

-d: Nombre de la base de datos donde se van a importar los datos

-U: Nombre del usuario con permisos para acceder a la base de datos.

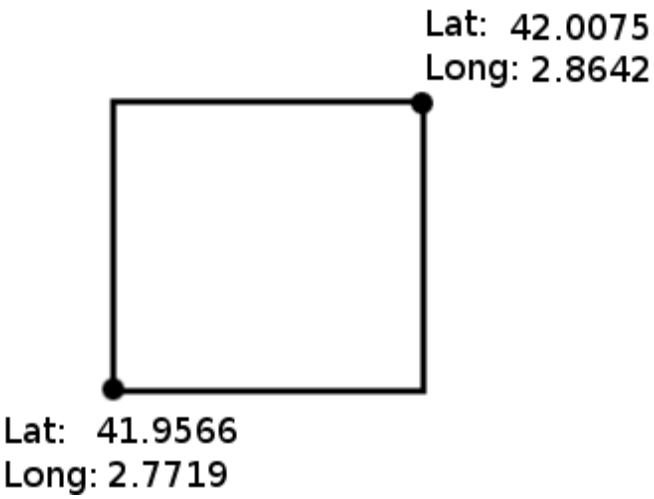
-W: Indica al sistema que solicite la contraseña del usuario indicado en el parámetro -U

-P: Puerto sobre el que se está ejecutando PosrgreSQL. Por defecto el puerto es 5432

-S: Indica que la importación se debe llevar a cabo siguiendo las indicaciones del fichero especificado.

-s: Parámetro opcional. Este parámetro indica que se debe controlar el uso de la memoria durante el proceso de importación (modo *slim*) y es especialmente útil para sistemas informáticos con pocos recursos.

-b: Sirve para importar únicamente la cartografía comprendida dentro de las coordenadas indicadas a modo de *bounding box* (ver lectura *Tipos de datos espaciales* de este mismo tema). Estas coordenadas se corresponden con la esquina inferior izquierda y con la esquina superior derecha tal y como aparece en la siguiente imagen.



Importar el shp barrios

Antes de importar el fichero shape *barrios* deberemos descargarlo desde la plataforma, concretamente desde la descripción de la Practica 3: SQL Espacial. Una vez descargado lo descomprimimos en la carpeta *bin* de PostgreSQL. Lo más probable es que la carpeta *bin* se encuentre en C:\Archivos de programa\PostgreSQL\9.x\bin, (donde x depende de la versión de PostgreSQL) aunque la ubicación concreta dependerá de la versión de nuestro sistema operativo.

Una vez descargado y descomprimido el fichero de barrios, abrimos una consola de comandos y accedemos a la misma carpeta *bin* donde se encuentran tanto los ficheros del shape barrios como la aplicación shp2pgsql.

Para acceder a la carpeta *bin* desde la consola de comandos vamos a utilizar el comando *cd* (*C*hange *D*irectory) del siguiente modo:

cd "C:\Archivos de programa\PostgreSQL\9.x\bin" (ENTER)

Si la carpeta *bin* se encuentra en otra ubicación deberemos indicar esa otra ubicación en el

comando anterior.

Antes de importar el shape a la base de datos deberemos convertirlo a sentencias SQL de tipo INSERT INTO. Para ello desde la consola de comandos ejecutamos el siguiente comando.

shp2pgsql barrios barrio > barrios.sql (ENTER)

Donde:

barrios es el nombre del fichero shape.

barrio: es el nombre de la tabla que se creará durante la ejecución, desde pgAdmin, del fichero barrio.sql.

barrios.sql: es el nombre del fichero que contienen los comandos SQL que llevaran a cabo la creación de la tabla y la inserción de los datos.

Una vez ejecutado el comando anterior, solo nos faltará cargar y ejecutar desde pgAdmin el fichero *barrios.sql.* En caso de dudas podéis consultar la *Guía de usuario de pgAdmin*.

Preparación de los datos

Llegados a este punto, si echamos un vistazo a la tabla vista *geometry_columns* de nuestra base de datos veremos que las tablas *planet_osm_point, planet_osm_line, planet_osm_polygon, planet_osm_roads* contienen geometrías con un srid igual a 900193 mientras que las geometrías de la tabla barrio contienen un srid igual a 0.

f_table_catalog character varying(256)	f_table_schema character varying(256)	f_table_name character varying(256)	f_geometry_column character varying(256)	coord_dimension integer	srid integer	type character varying(30)
	public	planet_osm_point	way	2	900913	POINT
	public	planet_osm_line	way	2	900913	LINESTRING
	public	planet_osm_polygon	way	2	900913	GEOMETRY
	public	planet_osm_roads	way	2	900913	LINESTRING
	public	barrio	the_geom	2	0	MULTIPOLYGON

Para poder utilizar las funciones espaciales de PostGIS es obligatorio que todas las entidades implicadas en el análisis estén representadas en el mismo sistema de referencia. Por este motivo, antes de empezar el análisis deberemos modificar la tabla barrio para que sus geometrías estén representadas utilizando el sistema de referencia con srid igual a 900913.

Para este fin, podemos ejecutar la función *UpdateGeometrySRID* (en su forma más simple) utilizada para modificar el srid de un objeto espacial.

```
SELECT UpdateGeometrySRID('cp','geom',900913);
```

Indicando el nombre de la tabla, el nombre de la columna y el sistema de referencia que queremos asignar. Al no indicar el schema en el que se encuentra la tabla 'cp', el sistema asume el schema *public* por defecto.

Vamos ahora a abrir un pequeño paréntesis para hablar de versiones de PostGIS anteriores a la 2.x.

Para estas versiones, nos encontraremos con un error parecido al siguiente al intentar modificar el srid de una geometría recién importada utilizando *shp2pgsql*:

```
ERROR: el nuevo registro para la relación «barrio» viola la restricción check «enforce_srid_geom»
```

En este caso, el error se debe a la restricción *enforce_srid_geom* que impide que la tabla barrio contenga geometrías con un srid distinto a 0. Esta restricción se crea, en versiones anteriores a la versión 2.0 de PostGIS, durante la ejecución del fichero barrio.sql.

Para resolver esta hipotética situación debemos eliminar la restricción con el siguiente comando (los siguientes comandos aparecen adaptados a las versiones 1.5.x de PostGIS):

```
ALTER TABLE barrio DROP CONSTRAINT enforce_srid_geom;
```

Una vez eliminada la restricción podemos ejecutar nuevamente el comando

```
UPDATE barrio SET geom=setsrid(geom, 900913);
```

En cualquiera de las dos situaciones anteriores, podemos comprobar los cambios visualizando el contenido de la tabla vista *geometry_columns*.

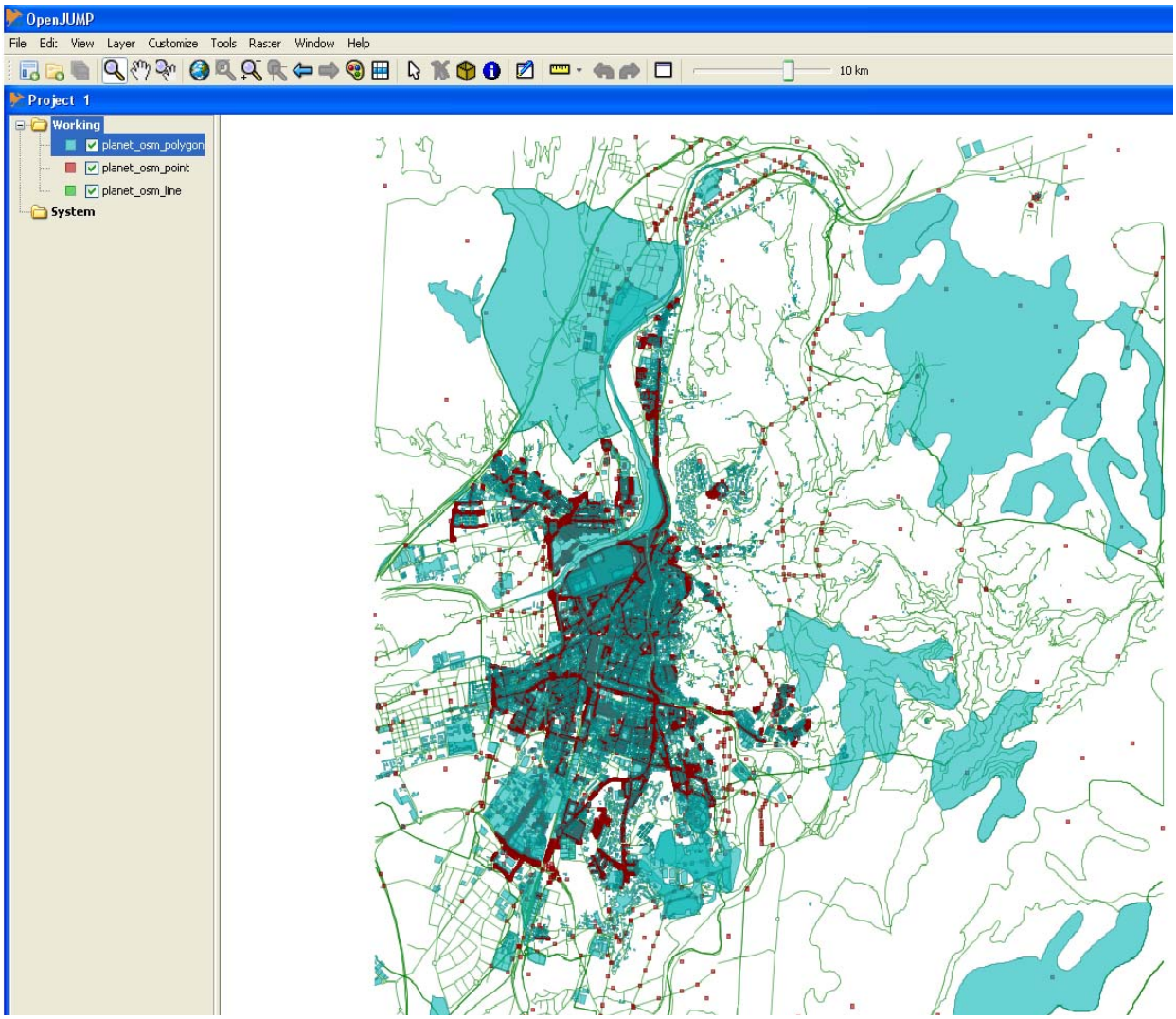
Si los cambios efectuados no aparecen reflejados ejecutaremos el siguiente comando.

```
SELECT populate_geometry_columns();
```

La función *populate_geometry_columns()* hace un repaso exhaustivo de la base de datos y actualiza la tabla vista *geometry_columns.*

Visualización de datos desde OpenJUMP

Cargando desde OpenJUMP las tablas *planet_osm_point*, *planet_osm_line*, *planet_osm_polygon* y *barrios*, obtenemos una vista como la que aparece en la siguiente imagen. En caso de duda, podéis consultar el documento *consultas espaciales con OpenJUMP.*



Tablas y atributos

Tablas

Cuando importamos datos de OpenStreetMap en una base de datos PostgreSQL/PostGIS se generan automáticamente las siguientes tablas: *planet_osm_point*, *planet_osm_line,* *planet_osm_polygon* y *planet_osm_roads*.

Cada tipo de entidad espacial (puntos, líneas y polígonos) queda almacenado en la tabla correspondiente. La única excepción la encontramos en las tablas *planet_osm_roads* y *planet_osm_line* pues ambas contienen geometrías de tipo línea. Si bien *planet_osm_roads* contiene únicamente las carreteras, *planet_osm_line* contiene el resto de estructuras lineales

(líneas de tren, líneas de metro, callejero, etc).

Atributos

Cada objeto dentro de la cartografía de OpenStreetMap se caracteriza por una serie de atributos.

Estos atributos, también conocidos como *tags* sirven para indicar las propiedades de cada entidad espacial. Por ejemplo, sirven para indicar si una entidad de tipo *punto* es un árbol, una farmacia o una cabina telefónica.

Como os podéis imaginar OpenStreetMap tienen una gran variedad de *tags*. Solo hace falta echar un vistazo a las tablas *planet_osm_point*, *planet_osm_line,* etc. para ver los atributos generados. Veréis que la mayoría de esos atributos están vacíos, o mejor dicho, contienen un valor nulo. Podéis ver las etiquetas más utilizadas en este enlace http://taginfo.openstreetmap.org/tags

Las tablas importadas tienen los mismos atributos por lo que podemos mencionarlos conjuntamente, teniendo en cuenta que, un atributo muy utilizado en una tabla puede pasar totalmente desapercibido en otra.

Entre los atributos más habituales encontramos:

- *osm_id*: Atributo de tipo entero que sirve para identificar inequívocamente cada fila de la tabla.
- *way*: contiene la geometría del objeto espacial.
- *highway*: Especialmente utilizado para las entidades lineales. Sirve para describir carreteras y vías peatonales.
- *name*: Utilizado para indicar el nombre del elemento (nombre de la calle, carretera, hotel, etc.)
- *oneway*: Utilizado para indicar si la circulación es en una sola dirección o no.
- *width*: Anchura en metros del objeto espacial

- *foot*: Indica si se trata de un paso peatonal
- *amenity*: Este atributo es muy utilizado para indicar el tipo de establecimiento. Pub, bar, restaurant, parking, pharmacy, etc.
- *natural*: Utilizado para describir elementos naturales. Playas, árboles, acantilados, cumbres, etc.
- *waterway*: Para describir vías de agua como canales, ríos, diques, etc.

Un aspecto importante a tener en cuenta, es el gran número de usuarios que editan cartografía en la comunidad OpenStreetMap. Este hecho provoca, inevitablemente, que durante la creación de la cartografía se sigan criterios y metodologías distintas (propias de cada editor). Por este motivo podemos encontrar regiones que utilizan más unos atributos que otras. Para un mayor detalle os podéis dirigir a http://wiki.openstreetmap.org/wiki/Map_Features

Análisis espacial y comandos SQL

Los comandos SQL que vamos a llevar a cabo a partir de ahora, serán principalmente comandos de consulta (SELECT), por lo tanto, en función de qué tipos de datos devuelvan nuestras consultas será más o menos apropiado visualizarlos en un entorno o en otro.

Las consultas que devuelvan tipos de datos geométricos, por ejemplo, será muy práctico visualizarlos con OpenJUMP para poder ver gráficamente el resultado de la consulta. Por otro lado, las consultas que devuelvan tipos de datos que no se pueden representar gráficamente solo las podremos ejecutar desde el mismo pgAmin. OpenJUMP mostrará un error si la consulta no retorna ninguna columna de tipo geometry.

El siguiente comando SQL devuelve un buffer de 100metros sobre el barrio de L'EIXAMPLE

```
SELECT ST_Buffer(geom, 100) FROM barrio WHERE nombre='EIXAMPLE';
```

Si ejecutamos la consulta desde pgAdmin obtendremos el siguiente resultado:

st_buffer geometry
0103000000010000000CD00000034B69D

Mientras que si ejecutamos el mismo comando desde OpenJUMP obtendremos el siguiente resultado gráfico.

Ejercicios

Cálculo del área de cada barrio

Al ser el área un valor numérico, vamos a ejecutar esta consulta desde pgAdmin.

```
SELECT ST_Area(geom), nombre FROM barrio;
```

Si queremos seleccionar únicamente el barrio de mayor superficie, ordenamos el resultado de mayor a menor area y lo limitamos a una sola fila (la primera de la lista será la mayor).

```
SELECT ST_Area(geom), nombre FROM barrio ORDER BY ST_Area(geom) DESC LIMIT 1;
```

¿Cuántos metros de frontera contiene cada barrio?

La solución más sencilla consiste en aplicar la función ST_Perimeter sobre el campo que contiene la geometría de los barrios. St_Perimeter solo se puede aplicar sobre geometrías que definan una superficie.

```
SELECT ST_Perimeter(geom) FROM barrio;
```

Otra solución más elaborada consiste en combinar las funciones ST_Boundary y ST_Length. St_Boundary devuelve una geometría de tipo lineal equivalente a la envoltura de la geometría inicial. ST_length devuelve la longitud de una geometría de tipo lineal.

Primero deberemos obtener la envoltura de los barrios. Ello se obtiene con el siguiente comando:

```
SELECT ST_Boundary(geom) FROM barrio;
```

y ahora podemos aplicar la función ST_Length sobre el resultado de ST_Boundary. De este modo obtenemos la longitud de la envoltura.

```
SELECT ST_Length(ST_Boundary(geom)) FROM barrio;
```

Obtener una geometría con la suma de todos los barrios

En PostGIS encontramos dos funciones que nos permiten unir geometrías. Estas son ST_Collect y ST_Union.

```
SELECT ST_Collect(geom) FROM barrio;
```

```
SELECT ST_Union(geom) FROM barrio;
```

La diferencia la encontramos, por supuesto, en el tiempo de respuesta pero además en el tipo de resultado que devuelve cada función. St_Collect nos retorna un solo objeto (una sola fila) que incluye los polígonos de todos los barrios. Inserta las geometrías en un mismo conjunto pero no las disuelve. St_Union retorna un solo objeto con un solo polígono. Este polígono es el resultado de la disolución de los polígonos de cada barrio. El coste de cálculo, por lo tanto, es mayor.

Listado de las farmacias de cada barrio

En esta ocasión intervienen dos entidades: barrio y *planet_osm_point*. Esta última es donde se almacenan las farmacias (etiqueta *amenity*, valor *pharmacy*).

Si deseamos obtener todas las farmacias podemos aplicar este comando:

```
SELECT * FROM planet_osm_point WHERE amenity='pharmacy';
```

Sin embargo, si deseamos agrupar las farmacias según el barrio al que pertenecen deberemos

añadir una nueva entidad (barrio) al comando anterior.

```
SELECT * FROM barrio, planet_osm_point WHERE amenity='pharmacy';
```

y añadir un join espacial. Este join espacial debe aparecer como condición dentro de la cláusula WHERE. De este modo vamos a unir cada barrio con las *pharmacy* que le pertenecen.

```
SELECT * FROM barrio , planet_osm_point WHERE amenity='pharmacy' AND ST_Contains(geom, way);
```

Las siguientes sentencias son igualmente válidas.

```
SELECT * FROM barrio , planet_osm_point WHERE amenity='pharmacy' AND ST_Within(way, geom);
```

```
SELECT * FROM barrio , planet_osm_point WHERE amenity='pharmacy' AND ST_Intersects(way, geom);
```

Obtener un listado con los bares del barrio EIXAMPLE

En este caso para identificar los bares utilizaremos el mismo atributo *amenity* con el valor *bar.*

```
SELECT * FROM barrio, planet_osm_point WHERE amenity='bar' AND nombre='EIXAMPLE' AND
ST_Intersects(way, geom);
```

Crear una nueva tabla con los ríos de la ciudad

A partir de la cartografía importada vamos a generar una nueva tabla que contenga, únicamente, los ríos de Girona. Inicialmente, estos ríos están contenidos como polígonos. Por lo tanto se encuentran en la tabla *planet_osm_polygon*. Podemos identificarlos fácilmente con el atributo *waterway* y el valor *riverbank*. El comando SQL para seleccionar los ríos es.

```
SELECT * FROM planet_osm_polygon WHERE waterway='riverbank';
```

Para crear una nueva tabla con el resultado de la consulta anterior solo es necesario añadir el comando CREATE TABLE *nombre_tabla* AS *selección*. Resultando el comando:

```
CREATE TABLE rio AS SELECT * FROM planet_osm_polygon WHERE waterway='riverbank';
```

Definir una zona inundable de Girona como un buffer de 100m sobre los ríos de la

ciudad

Para esta consulta nos viene muy bien la tabla *rio* que acabamos de crear. La función ST_Buffer es la encargada de obtener la zona inundable. Dado que la unidad del sistema de referencia (900913) es el metro, podemos aplicar directamente el siguiente comando.

```
SELECT ST_Buffer(way, 100) FROM rio;
```

Opcionalmente podemos guardar el resultado obtenido en una nueva tabla llamada *zona_inundable.*

```
CREATE TABLE zona_inundable AS SELECT ST_Buffer(way, 100) AS way FROM rio;
```

En esta ocasión hemos utilizado la palabra reservada *AS* para indicar que deseamos renombrar el resultado de la función ST_Buffer para que se llame *way*. En caso contrario la columna espacial de la tabla zona_inundable se llamaría ST_Buffer, pues por defecto la columna se renombra al nombre de la función utilizada.

¿Cuáles son las calles que cruzan los ríos?

En este caso vamos a utilizar la función espacial ST_Crosses aplicada a las entidades (o tablas) *planet_osm_line* (que contiene las estructuras lineales como las calles) y *rio.*

```
SELECT planet_osm_line.* FROM planet_osm_line, rio
WHERE ST_Crosses(planet_osm_line.way, rio.way);
```

Si visualizamos el resultado en OpenJUMP veremos que el resultado obtenido muestra otras entidades además de las calles. Podemos saber de qué entidades de se trata si añadimos a la vista los atributos de la consulta pulsando sobre el icono.

De este modo podemos ver como en la consulta anterior se han añadido, además de las calles, los límites administrativos, el propio cauce del río Ter y la autopista AP-7. Todas ellas entidades lineales que se encuentran también en la tabla *planet_osm_line*.

Para obtener únicamente las calles, modificamos el comando anterior del siguiente modo:

```
SELECT planet_osm_line.* FROM planet_osm_line, rio
WHERE ST_Crosses(planet_osm_line.way, rio.way) AND planet_osm_line.ref IS NULL AND
planet_osm_line.boundary IS NULL AND planet_osm_line.waterway IS NULL
```

Elementos de la tabla *planet_osm_point* localizados dentro de las zonas inundables

En esta ocasión podemos utilizar varias funciones espaciales. Concretamente ST_Intersects, ST_Within o ST_Contains. Todas ellas nos permiten establecer las entidades de la tabla *planet_osm_point* que se encuentran dentro de la zona inundable.

Si anteriormente hemos optado por crear la tabla *zona_inundable,* ahora podemos aplicar directamente.

```
SELECT planet_osm_point.* FROM planet_osm_point, zona_inundable WHERE
ST_Intersects(zona_inundable.way, planet_osm_point.way);
```

Utilizando el resto de funciones podemos modificar la cláusula *WHERE* del siguiente modo.

```
WHERE ST_Contains(zona_inundable.way, planet_osm_point.way);
```

o bien,

```
WHERE ST_Within(planet_osm_point.way, zona_inundable.way);
```

Fíjate que el orden de las entidades es distinto según si utilizamos ST_Contains o ST_Within.

En caso de no disponer de la tabla zona_inundable, usaremos el siguiente comando.

```
SELECT planet_osm_point.* FROM planet_osm_point, rio
WHERE ST_Intersects(ST_Buffer(rio.way,100), planet_osm_point.way);
```

En esta ocasión hemos utilizado la función ST_*Intersects* aunque, lógicamente, siguen siendo válidas ST_*Contains* y ST_*Within*. En el comando anterior, antes de aplicar la función ST_*Intersects* se calcula el buffer para el atributo *way* de la tabla *rio*. Una vez obtenido el buffer se aplica la función ST_*Intersects* para obtener el resultado deseado. La ejecución de

este comando requiere mucho más tiempo que si utilizamos la tabla zona_inundable ya que se deben realizar más cálculos.

Crear una nueva tabla de puntos donde no se incluyan los árboles

Si habéis observado con atención el listado de puntos del apartado anterior, os habréis dado cuenta que la mayoría de puntos son de tipo *natural* y más concretamente árboles (*tree*). Para seleccionar los árboles.

```
SELECT * FROM planet_osm_point WHERE "natural"='tree';
```

Para excluir los árboles:

```
SELECT * FROM planet_osm_point WHERE "natural" <> 'tree' OR "natural" IS NULL;
```

Para crear la tabla:

```
CREATE TABLE sin_arboles AS SELECT * FROM planet_osm_point WHERE "natural" <> 'tree' OR "natural" IS NULL;
```

En esta ocasión nos vemos obligados a indicar el atributo *natural* utilizando las comillas dobles ("). Esto es debido a que la palabra *natural* tiene un sentido propio en el sistema gestor de bases de datos PostgreSQL. Algo así como si quisiéramos dar el nombre *SELECT* a un atributo de una tabla.

Agrupar (disolver) los ríos de la tabla *rio* que tengan el mismo nombre. El resultado obtenido debe reemplazar a la tabla *rio*

Como ya hemos visto, la función de PostGIS que disuelve geometrías es *St_Union*. Esta función pertenece al grupo de funciones denominadas de agregado. Es decir, agrega las entidades espaciales de distintas filas siguiendo el criterio especificado en el comando SQL. Si no indicamos ningún criterio, St_Union fusionará todas los objetos espaciales de una tabla en un solo objeto tal y como hemos hecho en un ejercicio anterior. El comando,

```
SELECT ST_Union(way) FROM rio;
```

une todos los polígonos de la tabla *rio* en un solo polígono. Es decir, genera un nuevo polígono con todos los tramos de la tabla *rio*. Si lo que queremos es unir los tramos que tengan el mismo nombre, es decir, los que pertenecen al mismo río entonces utilizaremos la cláusula *GROUP BY* aplicada a la columna *name.*

```
SELECT ST_Union(way) FROM rio GROUP BY name;
```

La solución correcta no termina aquí, ya que ahora tenemos los polígonos agrupados por nombre pero no sabemos el nombre de cada polígono, pues el comando anterior no devuelve el atributo *name*.

El comando final, ahora sí, es:

```
SELECT ST_Union(way), name FROM rio GROUP BY name;
```

Como se nos pide que substituyamos la tabla *rio* deberemos, antes de eliminar la actual tabla *rio,* generar una nueva tabla.

```
CREATE TABLE rio2 AS SELECT ST_Union(way) AS way, name FROM rio GROUP BY name;
```

De nuevo hemos aplicado la palabra AS para renombrar el resultado de la función ST_Union.

Finalmente eliminamos la tabla rio y renombramos rio2 a rio.

```
DROP TABLE rio;
ALTER TABLE rio2 RENAME TO rio;
```

Obtener la geometría de la isla que hay en el río Ter (Riu Ter)

Si observamos la tabla nueva tabla *rio* desde OpenJUMP, veremos que la única isla que aparece está en el río Ter. Esta isla no es más que un agujero dentro de otro polígono. Por lo tanto, podemos utilizar las funciones ST_Numinteriorrings y ST_Dumprings para extraer la geometría de la isla.

ST_Numineriorrings nos permitirá distinguir los polígonos que tienen algún anillo interior, es decir, los polígonos que tienen algún agujero (isla). La segunda función, ST_Dumprings (igual que la función ST_Dump) sirve para extraer, a partir de una geometría dada, el anillo exterior

y los anillos interiores que la componen. Para ver los tramos de la tabla *rio* que tienen algún agujero aplicamos el siguiente comando:

```
SELECT * FROM rio WHERE ST_Numinteriorrings(way) > 0 AND name ='Riu Ter';
```

Vemos que solo hay un polígono, por lo que solo habrá un tramo de la tabla *rio* que contenga alguna isla. Combinando ST_*Numinteriorrings* con ST_*Dumprings* obtenemos las geometrías de los polígonos interiores y exteriores de los tramos del río que contienen algún agujero.

```
SELECT (ST_Dumprings(way)).geom FROM rio WHERE ST_Numinteriorrings(way)>0;
```

Como las islas (agujeros) siempre serán los polígonos internos y por lo tanto de menor superficie que el polígono en el que se encuentran, podemos ordenar el resultado anterior en función del área de cada polígono (ascendentemente) y limitar el resultado a una sola fila.

```
SELECT (ST_Dumprings(way)).geom FROM rio WHERE ST_Numinteriorrings(way)>0
ORDER BY ST_Area((ST_Dumprings(way)).geom ) ASC LIMIT 1;
```

Obtener un listado con los tramos de los ríos que no tengan nombre y que además tengan algún punto de contacto con otros tramos que sí tengan nombre

La función espacial que vamos a utilizar en esta ocasión es ST_*Touches(A, B)*. Esta función espacial, como tantas otras, es fácilmente aplicable cuando los atributos A y B se encuentran en tablas distintas. Por ejemplo, si queremos obtener los objetos de la tabla *planet_osm_roads* que tocan a los objetos de la tabla *planet_osm_line* podemos ejecutar el siguiente comando.

```
SELECT planet_osm_line.* FROM planet_osm_line, planet_osm_roads
WHERE ST_Touches(planet_osm_roads.way, planet_osm_line.way);
```

En esta consulta, sin embargo, debemos operar espacialmente con objetos que pertenecen a la misma tabla. Es decir, debemos obtener los tramos de la tabla *rio* que tocan a otros tramos de la misma tabla *rio*. Probablemente lo primero que nos venga a la cabeza será un comando como el siguiente:

```
SELECT * FROM rio WHERE ST_Touches(way, way);
```

Sin embargo este comando no devolverá ningún valor válido ya que al estar implicada una sola tabla (*rio*) no se van a evaluar todas las posibles combinaciones entre tramos del río. Para que se evalúen todas las posibles combinaciones entre tramos deberemos simular que los atributos *way y way* -ST_Touches (way, way)- pertenecen a dos tablas distintas. De este modo, sí se combinarán todas las filas de la tabla *rio* con todas las filas de la tabla *rio* y podremos obtener el resultado deseado.

Para simular que estamos utilizando dos tablas distintas vamos a dar a cada tabla un alias. El siguiente comando utiliza la tabla *rio* en dos ocasiones. En la primera se le da el alias *r* mientras que en la segunda se le aplica el alias *r2.* Como podéis ver esto es tan sencillo como indicar tras el nombre de la tabla la palabra *'AS'* seguida del nombre de nuestro alias.

```
SELECT r.* FROM rio AS r, rio AS r2 WHERE ST_Touches(r.way, r2.way) ;
```

Cuando aplicamos un alias a una tabla, deberemos utilizar ese mismo alias para referirnos a todos los atributos de esa tabla. De ahí que debamos indicar r.*, r.way y r2.way.

En el enunciado del ejercicio se pide los tramos que **no** tengan nombre y que estén en contacto con otros tramos que **sí** tengan nombre asignado. Por lo tanto debemos modificar la cláusula *WHERE* y añadir las condiciones correspondientes.

```
SELECT r.* FROM rio AS r, rio AS r2
WHERE ST_Touches(r.way, r2.way) AND r.name IS NULL AND r2.name IS NOT NULL;
```

Localizar el estacionamiento más cercano al ayuntamiento

Tanto el ayuntamiento como los estacionamientos están representados como puntos y por lo tanto se encuentran en la tabla *planet_osm_points*. Con el atributo *amenity* podemos saber qué punto es el ayuntamiento y qué puntos son estacionamientos.

Para localizar el ayuntamiento podemos utilizar el comando:

```
SELECT * FROM planet_osm_point WHERE amenity='townhall';
```

De forma parecida, para identificar los estacionamientos:

```
SELECT * FROM planet_osm_point WHERE amenity='parking';
```

Igual que hemos hecho en la consulta anterior, debemos utilizar dos alias distintos sobre la tabla *planet_osm_point* para simular que los parking y el *townhall* se encuentran en tablas distintas.

```
SELECT ST_Distance(p1.way, p2.way) FROM planet_osm_point p1, planet_osm_point p2
WHERE p1.amenity='parking' AND p2.amenity='townhall';
```

El siguiente paso consiste en aplicar la función *MIN* sobre el resultado de la función ST_Distance. De este modo obtenemos el estacionamiento más cercano (mínima distancia) al ayuntamiento.

```
SELECT min(ST_Distance(p1.way, p2.way)) FROM planet_osm_point p1, planet_osm_point p2
WHERE p1.amenity='parking' AND p2.amenity='townhall';
```

Distancia media entre todos los estacionamientos

Una vez más debemos hacer uso de los alias. Primero obtenemos todas las distancias:

```
SELECT ST_Distance(p1.way, p2.way) FROM planet_osm_point p1, planet_osm_point p2
WHERE p1.amenity='parking' AND p2.amenity='parking' AND p1.osm_id<>p2.osm_id;
```

En esta ocasión hemos añadido la condición *p1.osm_id<>p2.osm_id* para evitar que se calcule la distancia de un parking a sí mismo.

Añadiendo la función de agregado *AVG* obtenemos el resultado final:

```
SELECT avg(ST_Distance(p1.way, p2.way)) FROM planet_osm_point p1, planet_osm_point p2
WHERE p1.amenity='parking' AND p2.amenity='parking' AND p1.osm_id<>p2.osm_id;
```

Restaurantes que se encuentran a menos de 500m de algún parking

Otro ejemplo del uso de alias:

```
SELECT distinct(p2.*) FROM planet_osm_point p, planet_osm_point p2
WHERE ST_Distance(p.way, p2.way)<500 AND p.amenity='parking' AND p2.amenity='restaurant';
```

En esta ocasión hemos utilizado la cláusula *distinct* para que los restaurantes no aparezcan repetidos. En caso de no incluir *distinct*, un restaurante que tuviera 10 parkings a menos de 500 metros aparecería repetido 10 veces. Una vez para cada parking.

Desde un punto ubicado en las coordenadas 313663 (lat), 5157539 (long) sobre el sistema de referencia con srid=900913, ¿a qué distancia se encuentra la farmacia más próxima?

Lo primero que debemos hacer en esta ocasión es convertir las coordenadas dadas a un objeto espacial de tipo *point*. Para ello vamos a utilizar la función ST_Geomfromtext.

El comando ***SELECT*** *ST_Geomfromtext('POINT(313663, 5157539)', 900913)* devuelve el objeto espacial. Por lo tanto para calcular la distancia entre ese punto y las farmacias podemos ejecutar:

```
SELECT ST_Distance(ST_Geomfromtext('POINT(313663 5157539)', 900913) , way)
FROM planet_osm_point WHERE amenity='pharmacy';
```

Como ya hemos hecho anteriormente, aplicamos la función *MIN* al resultado de las distancias y obtenemos el comando SQL final.

```
SELECT min(ST_Distance(ST_Geomfromtext('POINT(313663 5157539)', 900913) , way))
FROM planet_osm_point WHERE amenity='pharmacy';
```

Una vez conocemos la distancia *X* a la que se encuentra la farmacia más cercana, podemos localizar esa farmacia con un comando como el siguiente:

```
SELECT * FROM planet_osm_point WHERE ST_Distance(ST_Geomfromtext('POINT(313663 5157539)',
900913) , way) = X AND amenity='pharmacy';
```

Donde podemos substituir X por la consulta anterior y obtener el siguiente comando:

```
SELECT * FROM planet_osm_point WHERE ST_Distance(ST_Geomfromtext('POINT(313663 5157539)', 900913) ,
```

```
way) = (
        SELECT min(ST_Distance(ST_Geomfromtext ('POINT(313663 5157539)', 900913) , way))
        FROM planet_osm_point WHERE amenity='pharmacy'
) AND amenity='pharmacy';
```

Repetir la consulta anterior para las coordenadas 2.81742, 41.98552. En esta ocasión las coordenadas pertenecen al sistema de referencia con srid=4326 (coordenadas geográficas)

En primer lugar vamos a crear el objeto espacial con la función st_GeomFROMText.

```
SELECT ST_Geomfromtext ('point(2.81742 41.98552)', 4326);
```

Antes de utilizar cualquier función espacial que implique dos o más objetos, debemos comprobar que los objetos están definidos con el mismo sistema de coordenadas, es decir, todos los objetos implicados en el análisis deben contener el mismo srid. Por este motivo, debemos reproyectar el objeto espacial que acabamos de crear al sistema con srid 900913. Siendo 900913 el mismo srid de la tabla *planet_osm_point.*

Para reproyectar objetos espaciales vamos a utilizar la función *ST_Transform(objeto, nuevo_srid).*

```
SELECT ST_Transform(ST_Geomfromtext ('point(2.81742 41.98552)', 4326), 900913);
```

Añadimos ahora las funciones MIN y st_distance sobre el objeto reproyectado y obtenemos:

```
SELECT min(ST_Distance(ST_Transform(ST_Geomfromtext ('POINT(2.81742 41.98552)', 4326),900913) ,
way)) FROM planet_osm_point WHERE amenity='pharmacy';
```

Obtener la zona de transición entre barrios. Esta zona empieza a 50 metros del límite entre barrios

Una vez más estamos ante una operación espacial que implica objetos geográficos de una misma tabla por lo que deberemos utilizar nuevamente alias. En primer lugar vamos a realizar un buffer de 50 metros sobre los barrios y posteriormente aplicaremos la intersección entre esos barrios. De este modo obtendremos la zona de transición que se pide en este ejercicio

siempre que durante el proceso evitemos la intersección de un barrio consigo mismo. Por ese motivo hemos añadido la cláusula *WHERE b1.gid<>b2.gid.*

```
SELECT ST_Intersection(ST_Buffer(b1.geom, 50), ST_Buffer(b2.geom,50) )
FROM barrio b1, barrio b2 WHERE b1.gid<>b2.gid;
```

Obtener una relación donde aparezca la superficie de cada barrio destinada a parques urbanos

Para obtener el suelo edificado debemos buscar el atributo *landuse* y el valor *village_green*.

Podemos seleccionar la superficie de todos los parques del siguiente modo:

```
SELECT ST_Area(way) FROM planet_osm_polygon WHERE landuse='village_green';
```

Para relacionar estas superficies con los barrios a las que pertenecen, añadiremos la tabla barrio y utilizaremos la función ST_Intersects .

```
SELECT ST_Area(way) , nombre FROM barrio, planet_osm_polygon
WHERE landuse='village_green' AND ST_Intersects(geom, way);
```

Finalmente si queremos sumar todas las superficies de un mismo barrio aplicaremos la función de agregado *SUM* combinada con la cláusula de agrupación *GROUP BY.*

```
SELECT sum(ST_Area(way)) , nombre FROM barrio, planet_osm_polygon
WHERE landuse='village_green' AND ST_Intersects(geom, way) GROUP BY nombre;
```

Obtener una relación de las calles con su longitud

Para distinguir las calles dentro de la tabla *planet_osm_line* deberemos utilizar el atributo *highway* y el valor *residential*. Además, antes de calcular las longitudes de las calles deberemos unir los distintos tramos de una misma calle mediante la función *St_Union* y la cláusula *GROUP BY*. Así podremos agrupar los tramos que compartan nombre. Por lo tanto,

```
SELECT ST_Union(way) FROM planet_osm_line WHERE highway='residential' GROUP BY name;
```

Finalmente podemos aplicar la función *ST_Length,* sobre el resultado obtenido con *ST_Union*, y añadir el atributo *name* para que se muestre el nombre de la calle en el resultado final.

```
SELECT ST_Length(ST_Union(way)), name FROM planet_osm_line
WHERE highway='residential' GROUP BY name;
```

Obtener una relación con el número de farmacias que tiene cada calle. Solo deben aparecer las calles que tienen alguna farmacia. Las farmacias pertenecen a la calle que tienen más próxima

Vamos a analizar la consulta por partes. En primer lugar vamos a obtener un listado con la distancia mínima que hay entre alguna calle y cada farmacia. Llegados a este punto conoceremos las distancias mínimas entre farmacias y calles pero no conoceremos aun el nombre de esas calles. El siguiente paso será conocer cual es el nombre de la calle que tiene esa distancia mínima con cada farmacia. Finalmente podremos contar el número de farmacias de cada calle. Para obtener el listado que relacione cada farmacia con su distancia mínima ejecutamos el siguiente comando:

Comando versión 1:

```
SELECT planet_osm_point.osm_id, min(ST_Distance(planet_osm_point.way, planet_osm_line.way))
FROM planet_osm_point, planet_osm_line
WHERE planet_osm_point.amenity='pharmacy' AND planet_osm_line.highway='residential'
GROUP BY planet_osm_point.osm_id;
```

El siguiente paso consiste en averiguar cual es la calle que se encuentra a esa distancia mínima que ya conocemos. Para ello debemos utilizar la consulta anterior como si se tratara de una tabla más. Es decir, la próxima versión del comando SQL tendrá una estructura parecida a: *SELECT CAMPOS FROM TABLAS, CONSULTA_VERSIÓN_1 WHERE CONDICIONES.*

Si queremos utilizar el resultado de una consulta como si se tratara de una tabla más estamos obligados a utilizar un alias. A continuación se muestra la segunda versión del comando SQL. En negrita aparece la consulta que hemos visto anteriormente. El alias asignado es '*distancias_minimas*'.

Comando versión 2:

```
SELECT planet_osm_line.name FROM planet_osm_line, planet_osm_point,
    (
    SELECT planet_osm_point.osm_id, min(ST_Distance(planet_osm_point.way,
    planet_osm_line.way)) AS distancia
    FROM planet_osm_point, planet_osm_line
    WHERE planet_osm_point.amenity='pharmacy' AND planet_osm_line.highway='residential'
    GROUP BY planet_osm_point.osm_id
    ) AS distancias_minimas
WHERE ST_Distance(planet_osm_line.way, planet_osm_point.way) = distancias_minimas.distancia
AND planet_osm_point.osm_id=distancias_minimas.osm_id;
```

Finalmente podemos contar el número de veces que aparece cada calle. Para ello vamos a utilizar el comando versión 2 como si fuera una tabla más.

Comando versión 3:

```
SELECT count(*), name FROM
    (
    SELECT planet_osm_line.name FROM planet_osm_line, planet_osm_point,
        (
        SELECT planet_osm_point.osm_id, min(ST_Distance(planet_osm_point.way,
        planet_osm_line.way)) AS distancia
        FROM planet_osm_point, planet_osm_line
        WHERE planet_osm_point.amenity='pharmacy' AND
        planet_osm_line.highway='residential' GROUP BY planet_osm_point.osm_id
        ) AS distancias_minimas
    WHERE ST_Distance(planet_osm_line.way, planet_osm_point.way) = distancias_minimas.distancia
    AND planet_osm_point.osm_id=distancias_minimas.osm_id
    ) AS conteo
```

```
GROUP BY name;
```