

Criando curvas de nível e relevo no OSM usando dados da SRTM

Neste artigo vou mostrar como dar um aspecto mais elegante aos seus [mapas importados do OpenStreetMap](#), aplicando um efeito de relevo com curvas de nível, muito úteis quando a topografia for uma informação relevante em seus mapas. Usarei os arquivos DEM (Digital Elevation Model) fornecidos pela Shuttle Radar Topography Mission ([SRTM](#)) da NASA. Estes arquivos são distribuídos no formato HGT.

A versão que usarei (SRTMGL1) é a versão 3 (void filled) de 1 arco-segundo (30 metros) e cobertura global. Esta versão requer um [cadastro no site do EarthData](#) para ter acesso aos arquivos. Cada arquivo possui resolução de 3601 x 3601 pixels.

O primeiro passo é instalar o programa [phyghtmap](#), que fará todo o trabalho de download e conversão dos arquivos.

```
apt-get install python-matplotlib
```

```
wget  
http://katze.tfiu.de/projects/phyghtmap/phyghtmap_1.80-1_all.d  
eb
```

```
dpkg -i phyghtmap_1.80-1_all.deb
```

Agora, defina uma área para criar os relevos. Você precisará de um par de coordenadas delimitando uma caixa (bounding box) da área desejada. Eu costumo ir no site do [OpenStreetMap](#), colocar o mapa na área desejada e selecionar “exportar”. As coordenadas da caixa aparecerão no lado esquerdo da tela. Use-as na sequência esquerda-baixo-direita-cima.

Atenção! Não defina uma área muito grande ou você poderá esperar dias até todos os arquivos HGT serem baixados da NASA. Vá compondo seu mapa aos poucos. Para ter uma ideia de quantos arquivos sua área vai precisar, use esta ferramenta: <http://dwtkns.com/srtm30m/>.

Para baixar e converter os arquivos HGT para um formato que possa ser

importado para seu OpenStreetMap local, use este comando:

```
phyghtmap --pbf --no-zero-contour --line-cat=500,100 --step=10  
--jobs=8 --srtm=1 --a -44.978:-23.383:-40.902:-20.705 --  
earthdata-user=USUARIO --earthdata-password=SENHA
```

Onde:

--pbf significa que queremos arquivos PBF (formato do OSM).

--no-zero-contour significa que não queremos curvas de nível para altitude zero.

--line-cat=500,100 significa que queremos as curvas "major" a cada 500 metros e as curvas "medium" a cada 100 metros. Explico mais tarde a utilidade disso.

--step=10 significa que queremos um espaço de 10 metros entre as curvas.

--jobs=8 vai utilizar 8 threads.

--srtm=1 é para usar a resolução de 1 arcseg.

--a é a área desejada, separada por dois pontos (":").

--earthdata-user=USUARIO é o usuário criado no site da NASA.

--earthdata-password=SENHA é a senha do usuário criado no site da NASA.

Cada “ladrilho” baixado da NASA (arquivo HGT) vai ser convertido em um arquivo do formato OSM (PBF), então precisaremos consolidar todos eles em um só, ou passaremos nossa vida toda fazendo a migração para o banco. Para fundir arquivos PBF, precisaremos do programa “osmium”.

Debian Jessie, adicionar ao /etc/apt/sources.list:

```
deb http://ftp.debian.org/debian jessie-backports main
```

```
apt-get install osmium-tool
```

```
osmium merge --verbose *.osm.pbf -o consolidado.osm.pbf --
```

overwrite

Isso vai criar um arquivo PBF chamado consolidado.osm.pbf, pronto para ser importado para seu OpenStreetMap local. Obviamente, caso você execute esta linha de comando novamente, certifique-se de ter apagado o arquivo consolidado anterior, ou ele será incluído no novo consolidado, pois ordenamos uma fusão de todos os arquivos PBF encontrados no diretório ("*.osm.pbf").

Eu gosto de deixar as curvas de nível em um banco de dados separado dos dados do OSM, embora eles sejam completamente compatíveis. Esta decisão fica para você.

Crie um banco de dados "contour" e adicione a extensão "PostGIS" a ele e importe os dados gerados para ele.

```
osm2pgsql --latlong --verbose --create --style ./srtm.style -  
-database contour --username postgres -W --host 127.0.0.1  
consolidado.osm.pbf
```

Para criar tabelas mais enxutas, eu preparei um arquivo de estilo do osm2pgsql somente com os dados das curvas de nível. Eis meu arquivo "srtm.style":

#	OsmType	Tag	DataType	Flags
	node,way	contour	text	linear
	node,way	contour_ext	text	linear
	node,way	ele	int4	linear

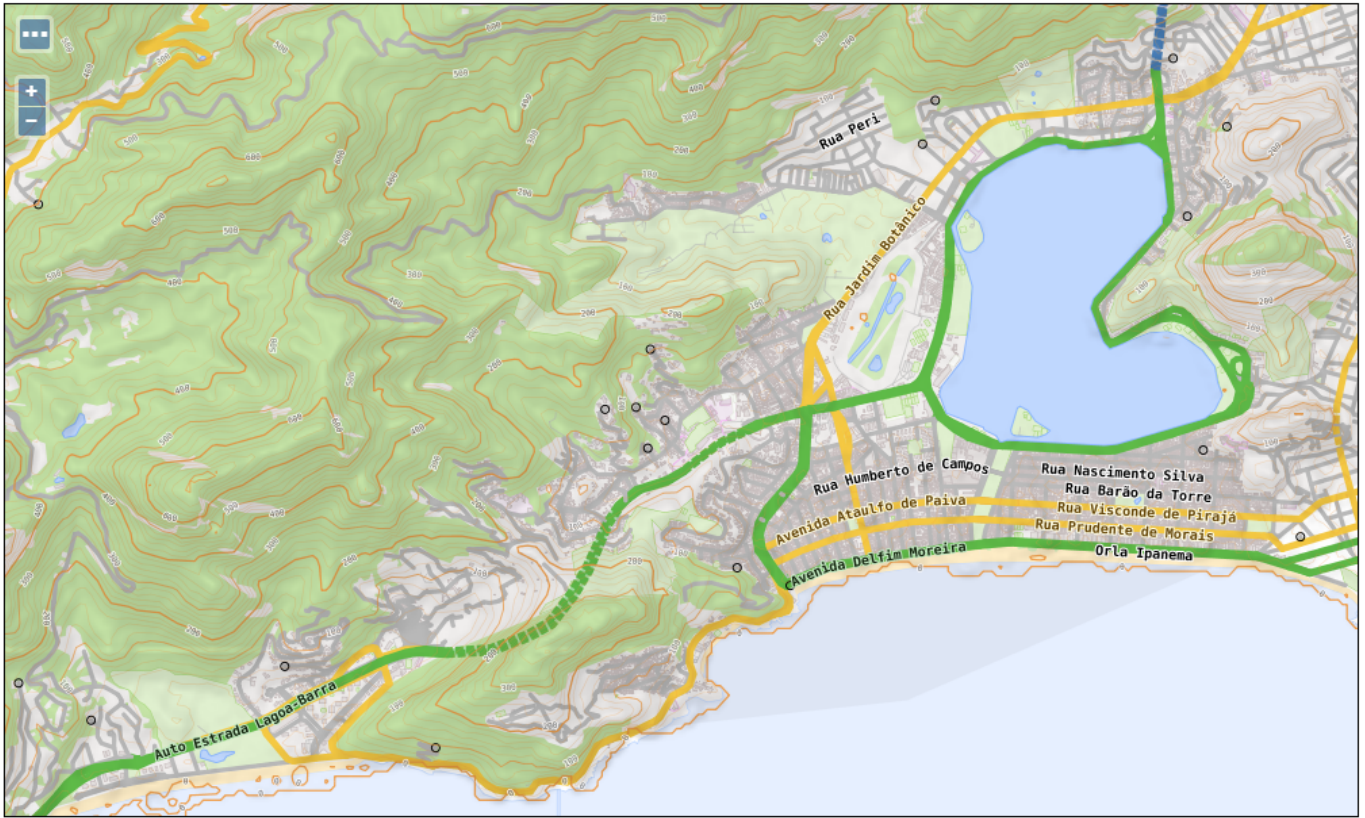
Com isso você terá dados somente na tabela "planet_osm_line", contendo as colunas "contour", "contour_ext" e "ele", além da coluna de índice "osm_id".

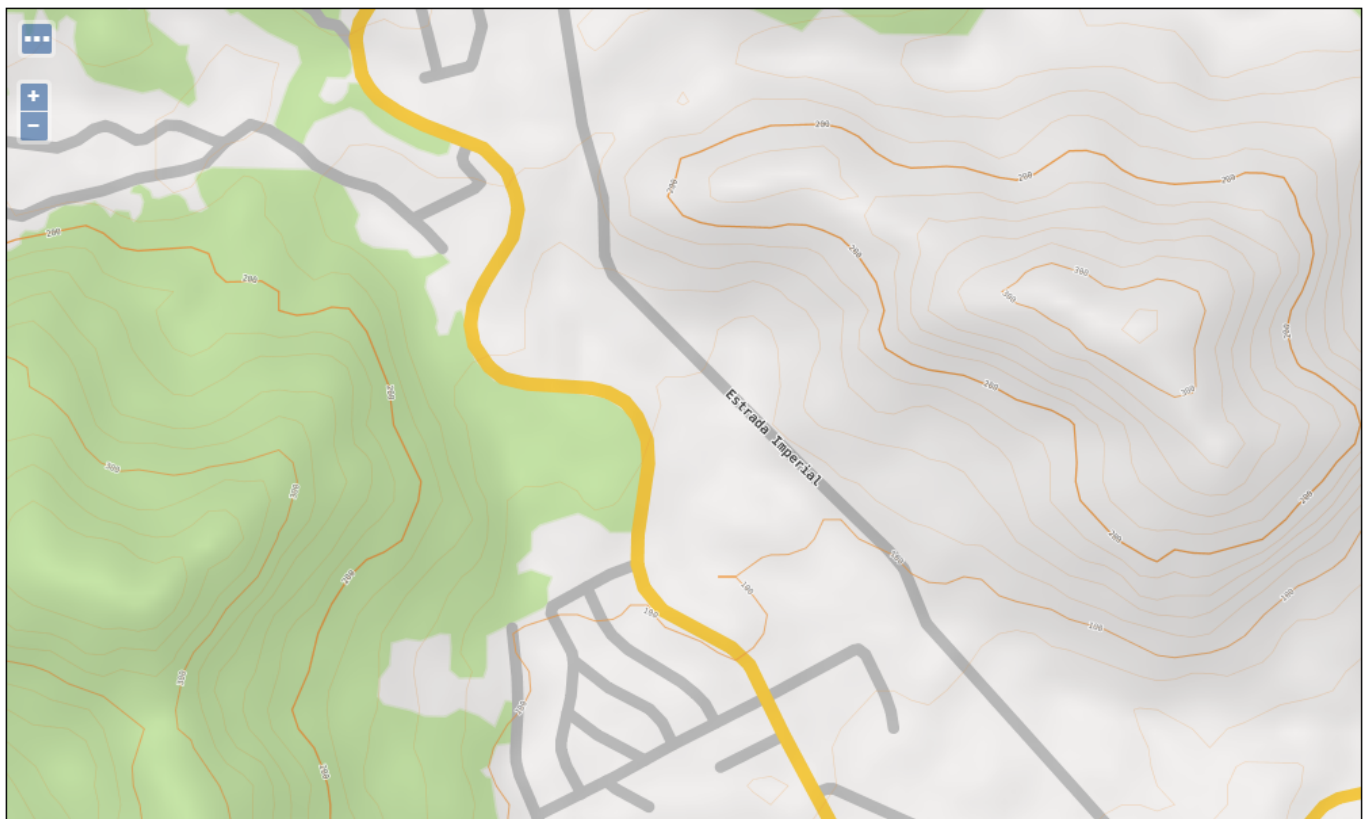
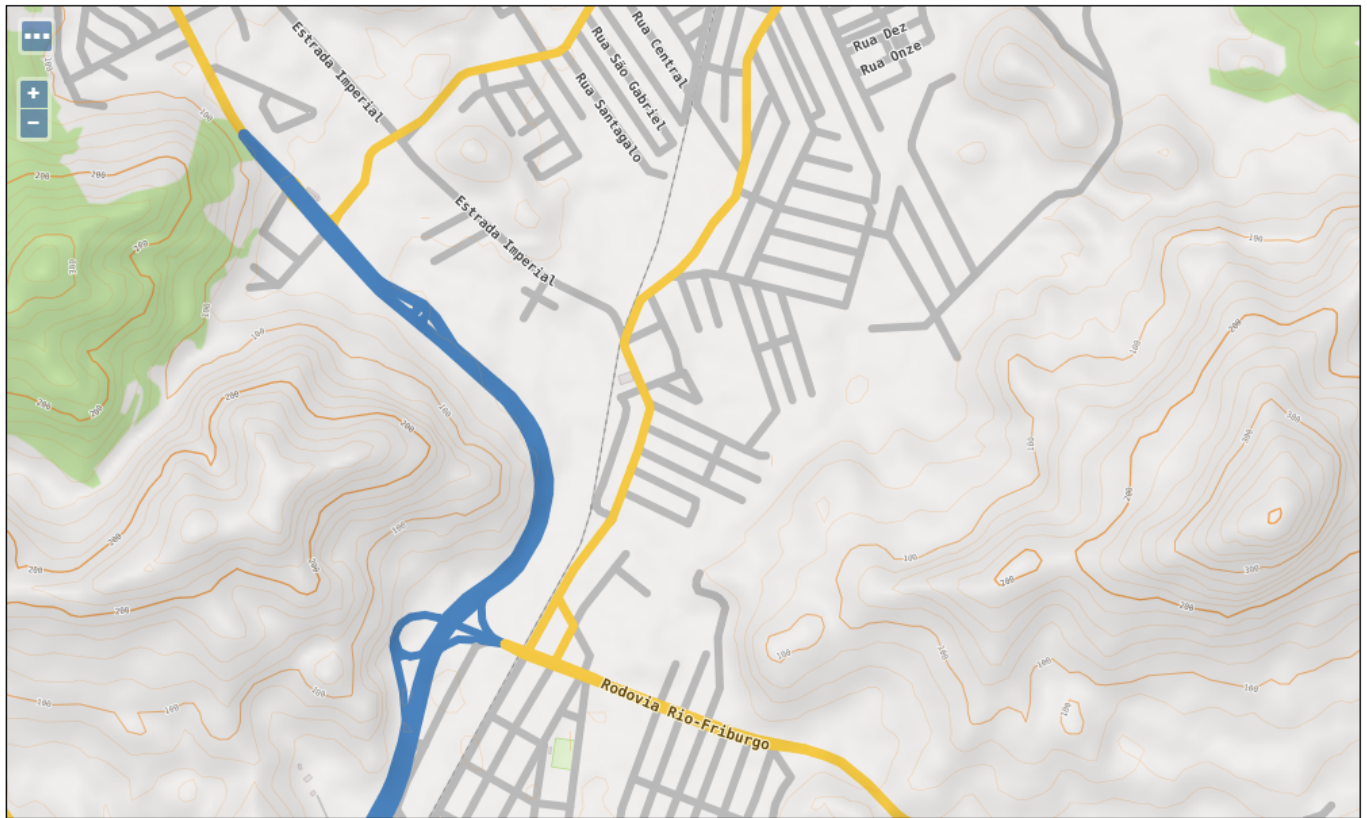
Agora você já poderá criar sua camada no Geoserver, apontando um Coverage para o banco "contour" e usando a tabela "planet_osm_line". Se não quiser usar este nome, crie uma visão:

```
drop view if exists "contours_line";  
create view "contours_line" AS (  
    SELECT osm_id, way,ele as elevation, contour_ext as cont_ext  
    FROM planet_osm_line  
);
```

Perceba que mudei o nome de algumas colunas. Você poderá criar um bom estilo para sua camada. O resultado final em meu servidor você pode ver nas imagens a







As linhas mais grossas são os contornos “major” e “medium” e as mais finas são os “step”. No próximo artigo vou mostrar como criar o efeito de relevo, chamado de “hillshading”, usando os mesmos arquivos HGT baixados pelo [phyghtmap](https://github.com/tytooon/phyghtmap).

NASA Jet Propulsion Laboratory (JPL), 2013, NASA Shuttle Radar Topography Mission United States 1 arc second. Version 3. 6oS, 69oW. NASA EOSDIS Land Processes DAAC, USGS Earth Resources Observation and Science (EROS) Center, Sioux Falls, South Dakota (<https://lpdaac.usgs.gov>)

O OpenStreetMap como repositório de dados abertos - Miguel Penteado

Este é um excelente vídeo que explica em detalhes o OpenStreetMap do ponto de vista dos dados.

Appliance do OSM - Rio de Janeiro

Disponibilizei para download uma máquina no VirtualBox com uma instalação do banco de dados do OpenStreetMap no Geoserver contendo a região metropolitana do Rio de Janeiro. É um excelente ambiente para testes. O Arquivo está no formato OVF 2.0.

Qualquer problema instalando a VM, por favor entre em contato.

Características:

Área coberta: Região metropolitana do Rio de Janeiro.

Geoserver pode ser acessado no endereço: `http://<IP>:8080/geoserver/`

Acesso ao Geoserver: Usuário: *admin* Senha: *geoserver*

Hardware:

MAC: 08:00:27:B7:23:D6

IP: Dinâmico (necessita DHCP na rede) via Bridge.

RAM : 10GB (usar menos memória irá comprometer o desempenho)

CPU : 2

HD: 80GB

PAE/NX.

Sistema:

Ubuntu 16.04.2 LTS.

Tomcat 8 na porta 8080.

Java JDK 1.8.0.121.

PostgreSQL 9.5. (Não é possível conexão externa ao banco. Edite o *pg_hba.conf*)

PostGIS 2.2.1 r14555

Geoserver: 2.10.2

Acesso ao sistema: Usuário: *geo* Senha: *geo*

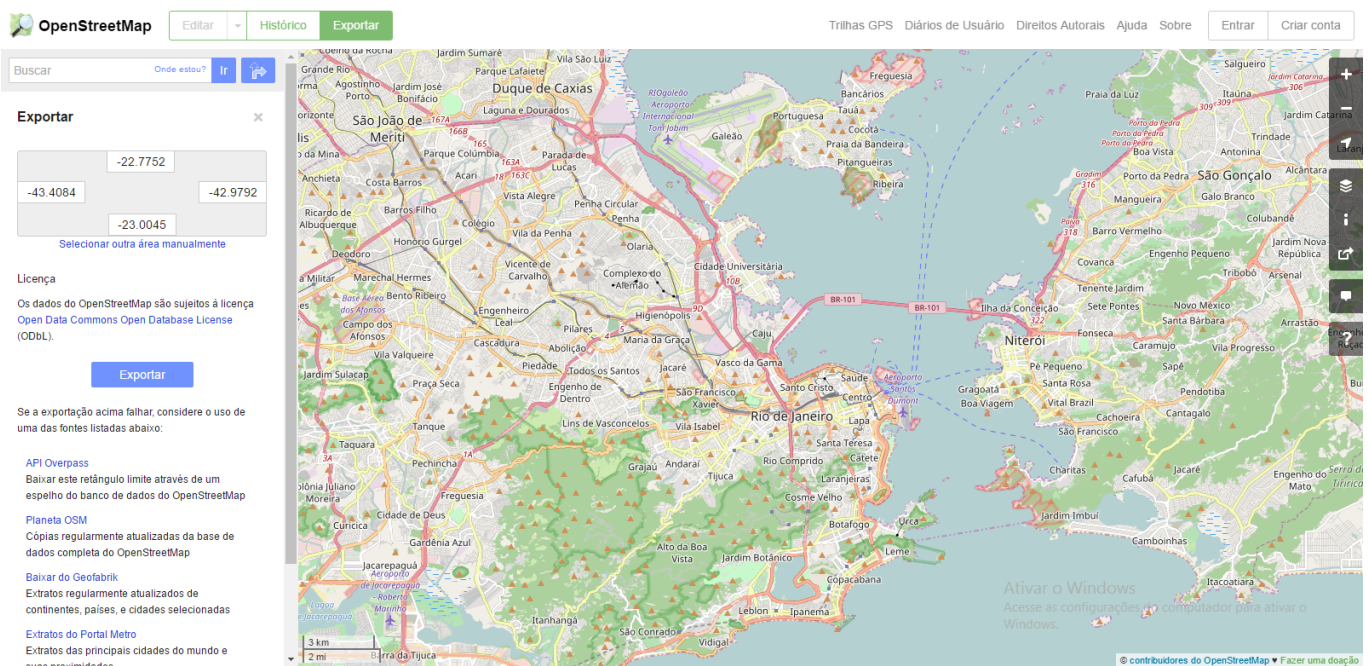
A senha para descompactar o arquivo RAR é

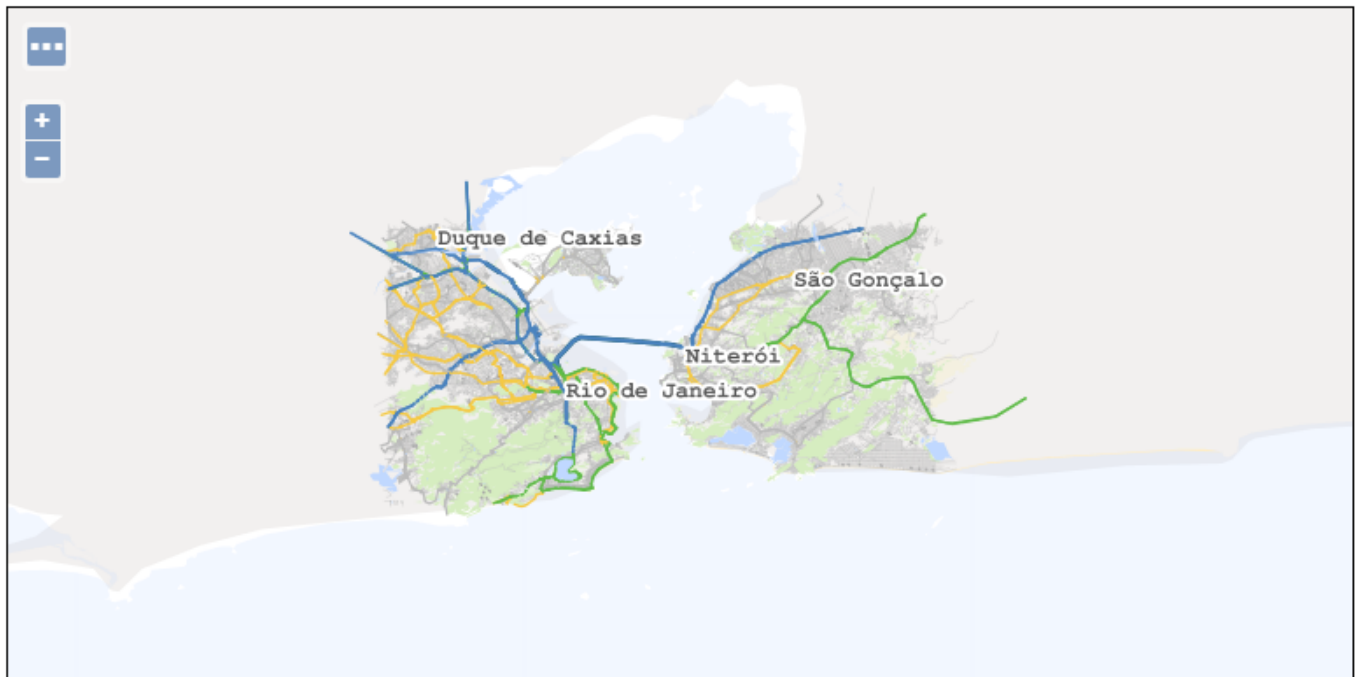
cmabreu.com.br#GEO!@\$123

Link para download

[\[DOWNLOAD\]](#)

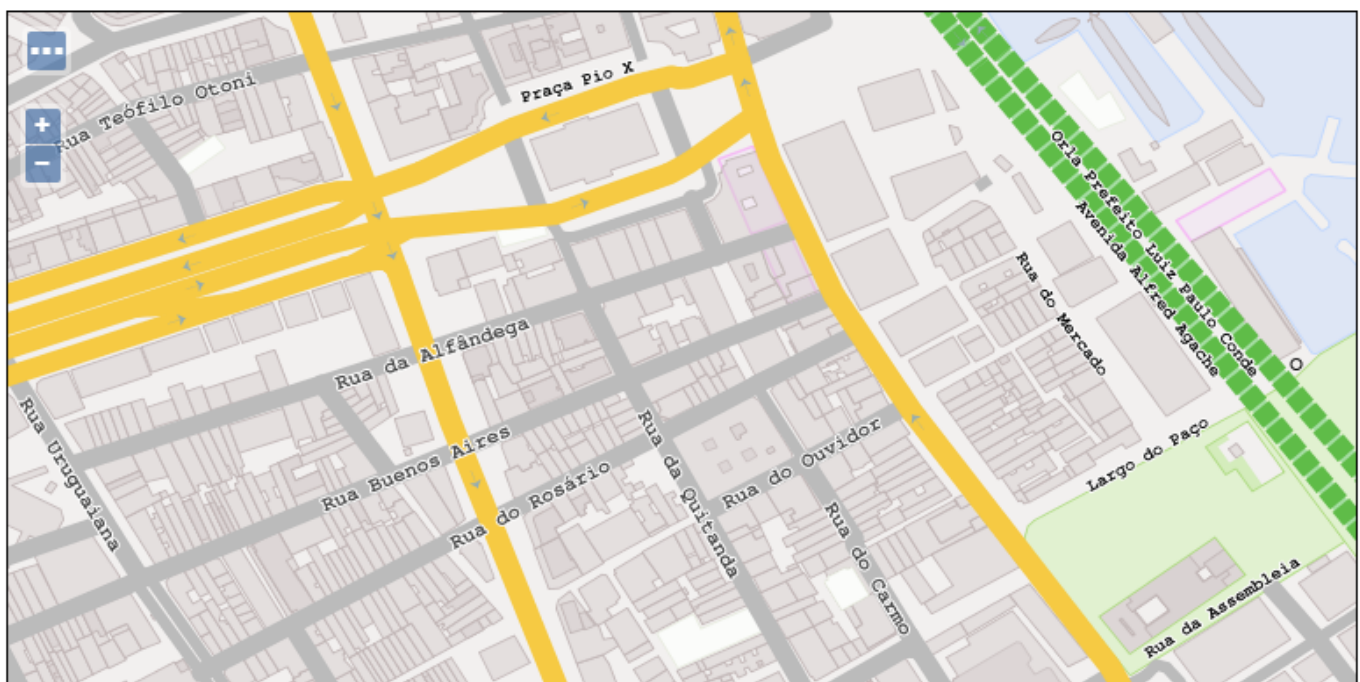
Esta VM foi completamente criada por mim, desde a instalação do SO até a importação dos dados do OSM, incluindo a criação das camadas no Geoserver. Você poderá usá-la apenas em ambientes acadêmicos ou para proveito próprio em sua casa. Se desejar uma instalação completa em um servidor dedicado, entre em contato. Realizo a instalação completa, ofereço suporte ou consultoria.





Scale = 1 : 545K

-43.07190, -22.89688



Scale = 1 : 4261

-43.17624, -22.90299

Trabalhando com rotas nos dados do OpenStreetMap: Parte 4

Neste post vou mostrar como melhorar o desempenho das consultas de rotas. Consulte a [parte 3](#) da série, caso queira.

Nossa função estava demorando muito (cerca de 36 segundos) para mostrar algum resultado. A origem e o destino não estão muito separados geograficamente e até confesso que não existem muitas opções de ruas para ir da Av. Pres. Vargas para a Rua do Catete. Com pontos mais distantes e mais ruas entre eles, a consulta pode se tornar um pesadelo. Se você reparar na consulta inicial, verá que o SQL de seleção de ruas passado para a função de rota *pgr_ksp* não tem nenhum critério. Isso passa tudo que existe na tabela para a seleção. Claro que a própria função possui algum algoritmo que melhora o desempenho, mas nunca é o bastante.

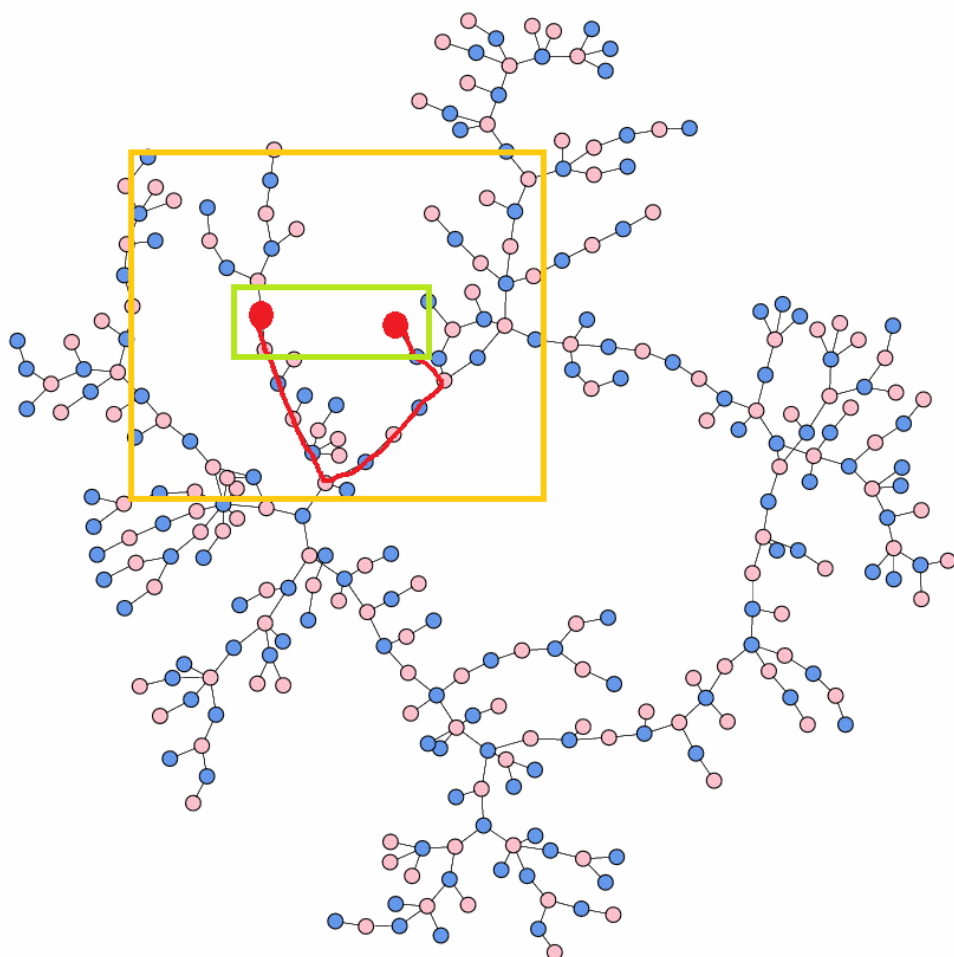
Como vimos antes, o tempo de execução da função de rotas é de cerca de 36 segundos:

```
CREATE OR REPLACE FUNCTION public.calc_rotas(
    IN source integer,
    IN target integer,
    IN k integer,
    IN directed boolean)
RETURNS TABLE(
    seq integer,
    path_id integer,
    path_seq integer,
    node bigint,
    edge bigint,
    cost double precision,
    agg_cost double precision
) AS
$BODY$
SELECT
```

*

```
FROM
  pgr_ksp(
    'SELECT id, source, target, cost, reverse_cost FROM
osm_2po_4pgr',$1, $2, $3, directed:=$4
  )
$BODY$
LANGUAGE sql VOLATILE COST 100;
```

A primeira estratégia é fornecer somente as ruas que estão próximas aos pontos de origem e destino. Existem funções no PostGIS que criam uma área em torno de dois pontos (em verde na figura abaixo). O problema é que podemos perder algumas ruas que estão fora desta caixa, então precisamos criar uma “margem de segurança” para tentar pegar estas ruas. Se esta margem for grande demais, irá prejudicar o desempenho, mas se for pequena demais, poderá causar perda de ruas e por consequência sua rota não irá refletir a realidade. Nos exemplos abaixo, deixarei uma margem de 4 Km, marcado em laranja na figura (parâmetro das funções [ST_Expand](#) e [ST_Buffer](#)). Se você perceber que sua rota dá “saltos” em ruas que não estariam no caminho, tente mexer um pouco neste valor.



A versão 2 da nossa função de rotas simplesmente seleciona um container que comporte os pontos de origem e destino ([ST_Extent](#)) e extrapola ele em 4Km para todas as direções ([ST_Expand](#)). Com isso selecionamos somente segmentos que possam estar relacionados com nossa rota e o tempo de execução cai para 19 segundos. Muito bom.

```
CREATE OR REPLACE FUNCTION public.calc_rotas_v2(
    IN source integer,
    IN target integer,
    IN k integer,
    IN directed boolean)
    RETURNS TABLE(seq integer, path_id integer, path_seq integer
, node bigint, edge bigint, cost double precision, agg_cost do
uble precision) AS
$BODY$
SELECT
    *
```

```

FROM
    pgr_ksp(
        'SELECT id, source, target, cost, reverse_cost FROM osm_2
po_4pgr as r,
            (SELECT ST_Expand(ST_Extent(geom_way),4) as box F
ROM osm_2po_4pgr as l1
            WHERE l1.source = ' || $1 || ' OR l1.target = ' ||
$2 || ') as box
            WHERE r.geom_way && box.box', $1, $2, $3, directed:
=$4
    )
$BODY$
LANGUAGE sql VOLATILE
COST 100
ROWS 1000;
ALTER FUNCTION public.calc_rotas(integer, integer, integer, bo
olean)
OWNER TO postgres;

```

Mas pode melhorar. A versão 3 da função utiliza abordagens diferentes na coleta ([ST_Collect](#) e [ST_Envelope](#)) e expansão da área ([ST_Buffer](#)), mas desta vez usando um raio de 4Km (a área resultante é circular). Nosso tempo melhora então para 13 segundos.

```

CREATE OR REPLACE FUNCTION public.calc_rotas_v3(
    IN source integer,
    IN target integer,
    IN k integer,
    IN directed boolean)
    RETURNS TABLE(seq integer, path_id integer, path_seq
integer, node bigint, edge bigint, cost double precision,
agg_cost double precision) AS
$BODY$
SELECT
    *
FROM
    pgr_ksp(
        'SELECT id, source, target, cost, reverse_cost FROM
osm_2po_4pgr as r,
            (SELECT
ST_Buffer(ST_Envelope(ST_Collect(geom_way)), 4) as box FROM
osm_2po_4pgr as l1

```



```

        WHERE l1.source = ' || $1 || ' OR l1.target = ' ||
$2 || ') as box
        WHERE r.geom_way && box.box', $1, $2, $3,
directed:=$4
    )
    $BODY$
    LANGUAGE sql VOLATILE
    COST 100
    ROWS 1000;
ALTER FUNCTION public.calc_rotas(integer, integer, integer,
boolean)
    OWNER TO postgres;

```

Para completar, vamos mexer nos índices. Devemos criar índices *btree* para as colunas *source*, *target* e *ID* e índice *gist* para a coluna de geometria *geom_way*. Siga com um *cluster* para este índice e depois um *analyze*.

```

CREATE INDEX idx_osm_2po_4pgr_source
    ON public.osm_2po_4pgr
    USING btree (source);

```

```

CREATE INDEX idx_osm_2po_4pgr_target
    ON public.osm_2po_4pgr
    USING btree (target);

```

```

CREATE INDEX idx_osm_2po_4pgr_id
    ON public.osm_2po_4pgr
    USING btree (id);

```

```

CREATE INDEX idx_osm_2po_4pgr_geomway
    ON public.osm_2po_4pgr
    USING gist (geom_way);

```

```

CLUSTER idx_osm_2po_4pgr_geomway ON osm_2po_4pgr;

```

```

VACUUM ANALYZE osm_2po_4pgr;

```

Nossa função agora leva 6 segundos para ser executada. Um bom ganho de desempenho. Lembre-se de que o parâmetro de 4Km do retângulo envolvente que selecionamos influencia muito no desempenho. Tente mudar este valor para 0.1 e você verá 6 segundos se transformar em 65 milissegundos! Se o tempo continuar

sendo um problema, você precisará de um HD SSD para seu banco de dados.

Eis alguns exemplos das funções de rotas do PGRouting:

K-Shortest Paths:

```
SELECT * FROM pgr_ksp(  
    'SELECT id, source, target, cost, reverse_cost FROM  
osm_2po_4pgr as r,  
    (SELECT st_buffer(st_envelope(st_collect(geom_way)), 4)  
as box FROM osm_2po_4pgr as l1  
    WHERE l1.source = 1358813 OR l1.target = 6450) as box  
    WHERE r.geom_way && box.box', 1358813, 6450, 1,  
directed:=false  
)
```

A-Star:

```
SELECT *  
    FROM pgr_astar(  
    'SELECT id, source, target, cost, x1,y1,x2,y2 FROM  
osm_2po_4pgr as r,  
    (SELECT ST_Expand(ST_Extent(geom_way),4) as box FROM  
osm_2po_4pgr as l1 WHERE l1.source =1358813 OR l1.target =  
6450) as box  
    WHERE r.geom_way && box.box',  
    1358813, 6450, false, false  
);
```

Dijkstra:

```
SELECT * FROM pgr_dijkstra(  
    'SELECT id, source, target, cost FROM osm_2po_4pgr as r,  
    (SELECT ST_Expand(ST_Extent(geom_way),4) as box FROM  
osm_2po_4pgr as l1 WHERE l1.source =1358813 OR l1.target =  
6450) as box WHERE r.geom_way && box.box', 1358813, 6450,  
false, false  
);
```

Vou criar um estilo para camada do GeoServer para representar a rota. É apenas uma linha vermelha um pouco mais grossa:

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
<StyledLayerDescriptor version="1.0.0"
```

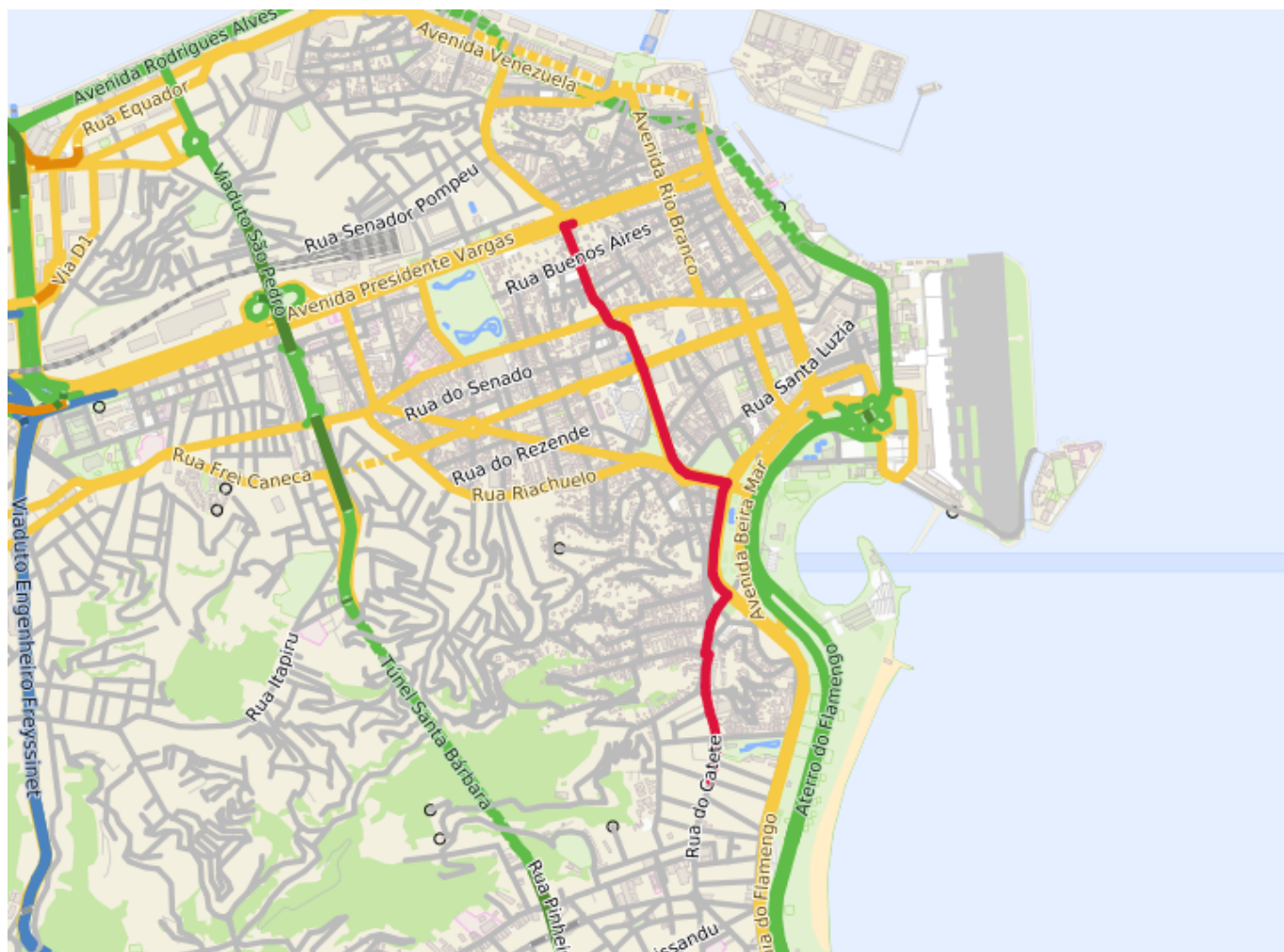
```
        xsi:schemaLocation="http://www.opengis.net/sld
http://schemas.opengis.net/sld/1.0.0/StyledLayerDescriptor.xsd
"
```

```
        xmlns="http://www.opengis.net/sld"
xmlns:ogc="http://www.opengis.net/ogc"
        xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

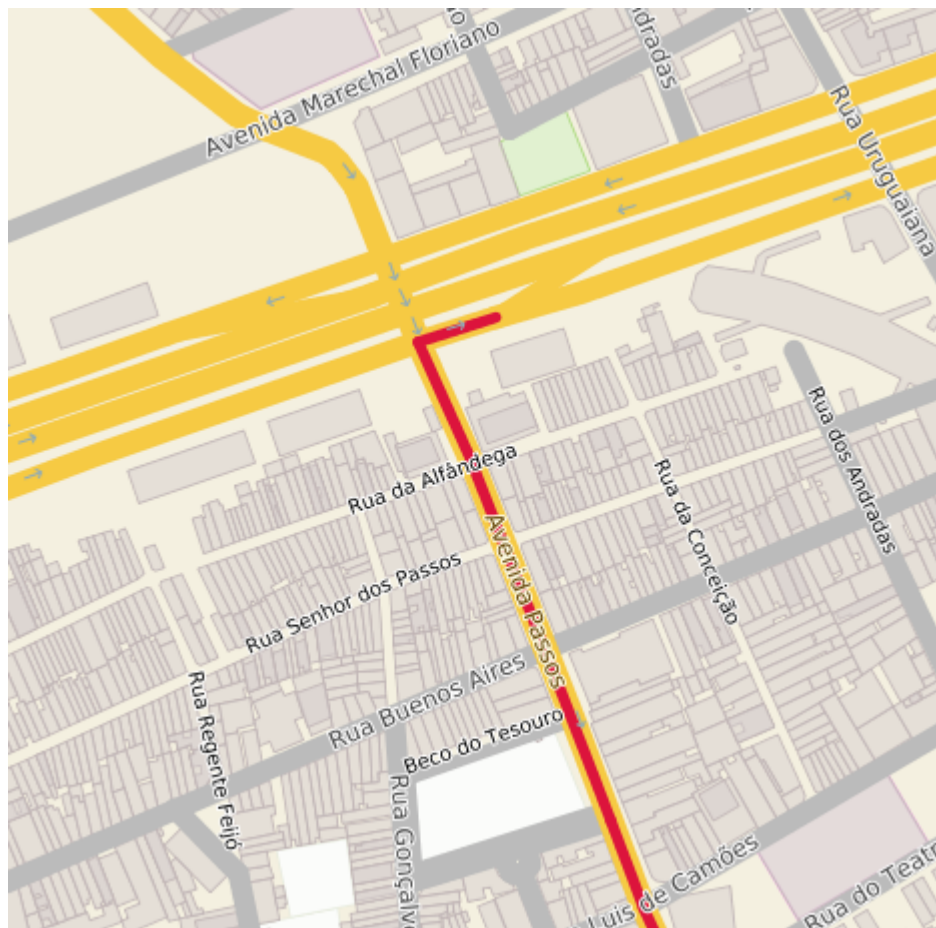
```
    <NamedLayer>
        <Name>rota</Name>
        <UserStyle>
            <Title>Uma linha vermelha para rotas</Title>
            <FeatureTypeStyle>
                <Rule>
                    <Title>A rota vermelha</Title>
                    <LineSymbolizer>
                        <Stroke>
<CssParameter
name="stroke">#DC143C</CssParameter>
                        <CssParameter name="stroke-
width">5</CssParameter>
                        <CssParameter name="stroke-
linecap">round</CssParameter>
                        </Stroke>
                    </LineSymbolizer>
                </Rule>

            </FeatureTypeStyle>
        </UserStyle>
    </NamedLayer>
</StyledLayerDescriptor>
```

E aí está nossa rota representada no mapa:



Repare que a rota não está considerando a direção do tráfego.



Isso é porque no SQL da view nós optamos por colocar o parâmetro *directed* como *false*.

```
select
    osm.osm_id, osm.osm_name, osm.km
from
    calc_rotas_v3( 1358812, 6450, 1, false ) rota
join
    osm_2po_4pgr osm on rota.edge = osm.id
```

Vamos alterar para *true* para ver o resultado:

Servidor

- Status do servidor
- Logs do GeoServer
- Informações de contato
- Sobre o GeoServer

Dados

- Visualizador de Camada
- Espacios de trabajo
- Almacenes
- Camadas
- Grupos de camadas
- Estilos

Servicios

- WMS
- WCS
- WFS

Ajustes

- Global
- JAI

Editar vista SQL

Actualizar la definición de la vista SQL y sus metadatos

Nome da View de Dados

Instrução SQL

```
select osm.*
from calc_rotas_v3( 1358812, 6450, 1, true ) rota
join osm_2po_4pgr osm on rota.edge = osm.id
```

Agora sim! Você pode perceber que a rota sugerida segue a direção do trânsito (pequenas setas nas ruas):



Vamos ver quais são as 3 sugestões de rotas mais curtas que ele dá?

GeoServer

Servidor

- Status do servidor
- Logs do GeoServer
- Informações de contato
- Sobre o GeoServer

Dados

- Visualizador de Camada
- Espacios de trabajo
- Almacenes
- Camadas
- Grupos de camadas
- Estilos

Editar vista SQL

Actualizar la definición de la vista SQL y sus metadatos

Nome da View de Dados

rota_pv_catete

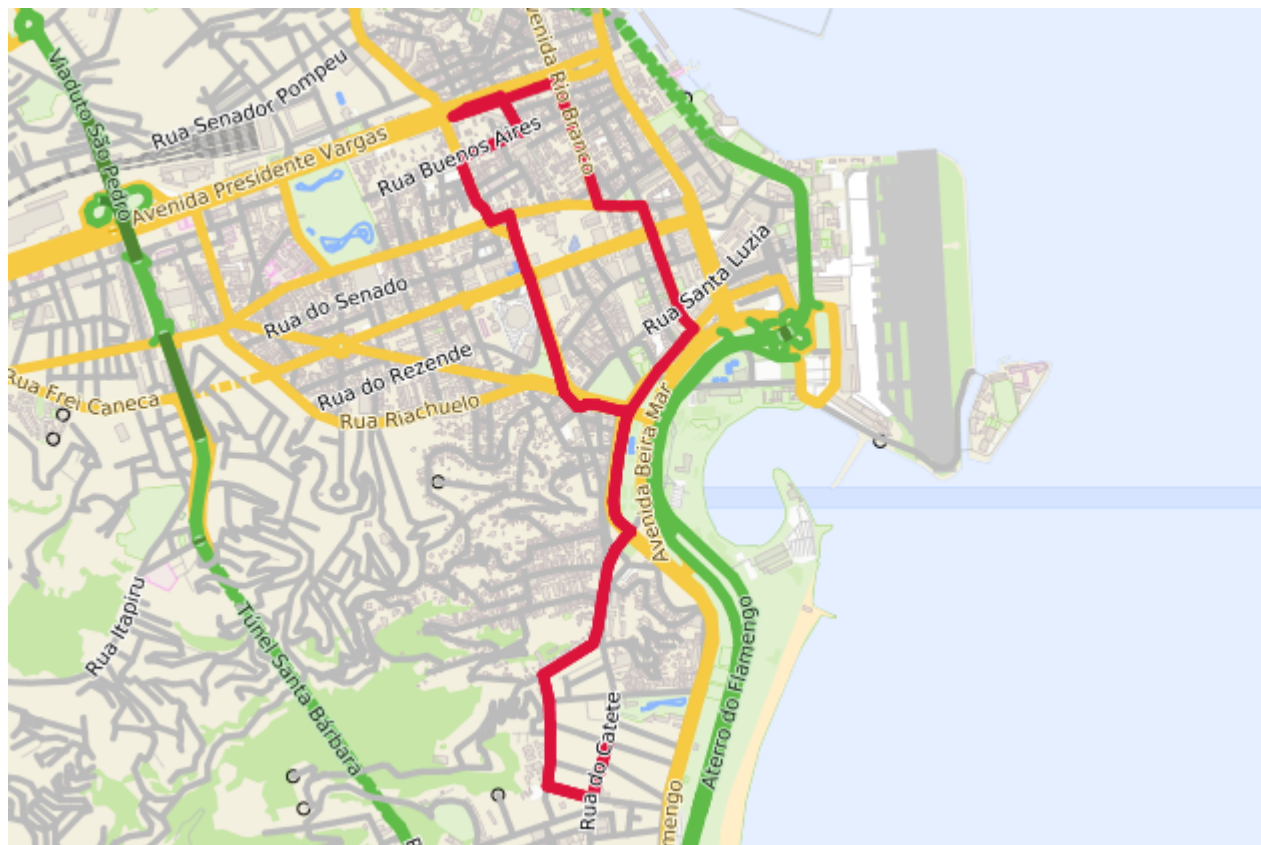
Instrução SQL

```

select osm.*
from calc_rotas_v3( 1358812, 6450, 3, true ) rota
join osm_2po_4pgr osm on rota.edge = osm.id

```

Aí está: Não temos muitas opções para esta rota.



No próximo post: Parametrizando a consulta ao GeoServer. E vem por aí: Criação de uma interface WEB para o calculador de rotas usando o OpenLayers.

[Trabalhando com rotas nos dados do OpenStreetMap: Parte 3](#)

No [post anterior](#) eu mostrei alguns fundamentos básicos no cálculo de rotas usando dados do OSM. É hora de conhecer algumas funções do [pgRouting](#) que

fazem o trabalho pesado para você usando algoritmos eficientes, como o [A Star](#), [Shortest Path](#), [Dijkstra](#), etc.

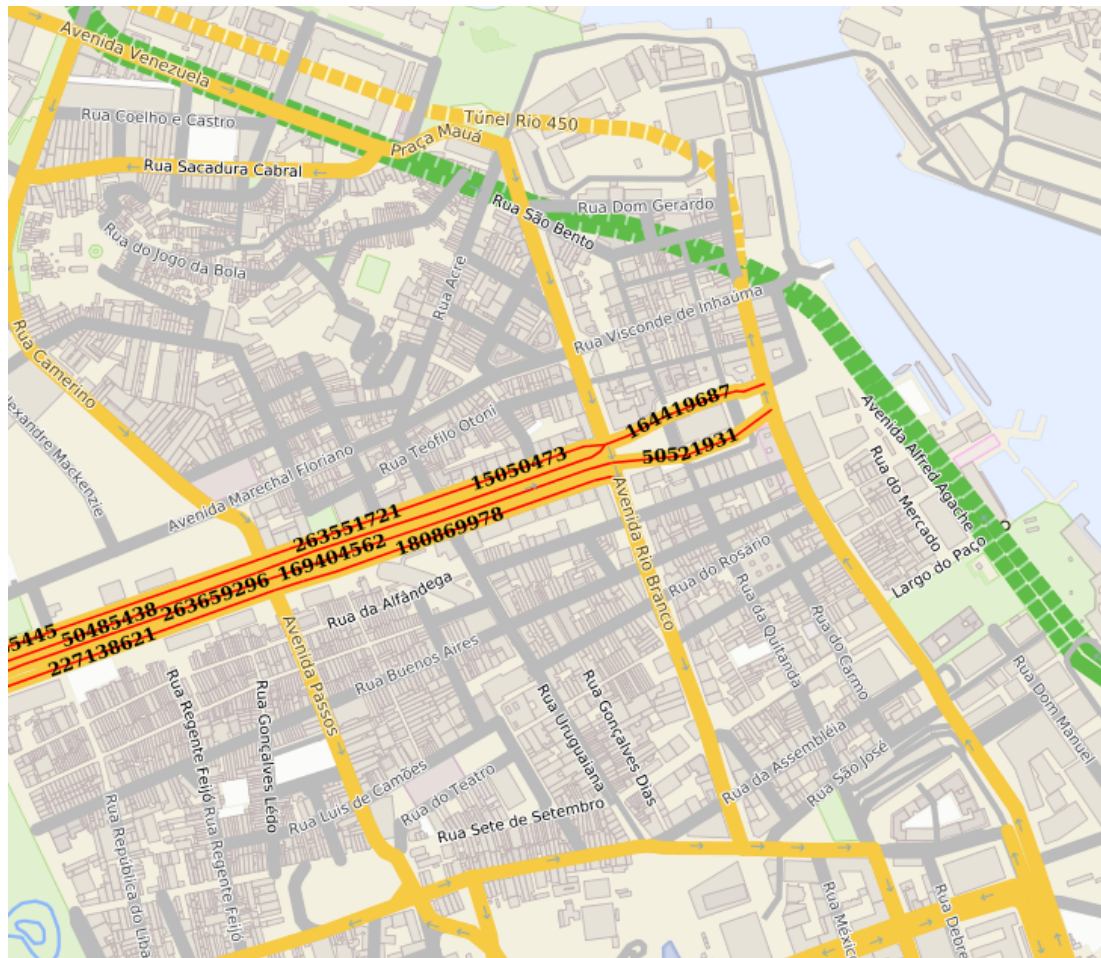
Eu havia mostrado [como criar uma view no banco de dados](#) para ter uma visão gráfica da Rua do Catete (RJ) no GeoServer. Tentar encontrar um caminho numa linha reta seria fácil, então vamos precisar encontrar outra rua um pouco mais longe para servir como o outro ponto da nossa rota. Ter uma visão gráfica da rua ajuda a entender melhor o processo, mas se você não quiser ou achar difícil usar o GeoServer, não tem problema: pode acompanhar as figuras que eu postei ou simplesmente usar a mesma instrução SQL da *view* e usar os dados obtidos.

Vamos usar uma avenida famosa no centro do Rio de Janeiro: a Avenida Presidente Vargas. Eis a *view* para encontrá-la:

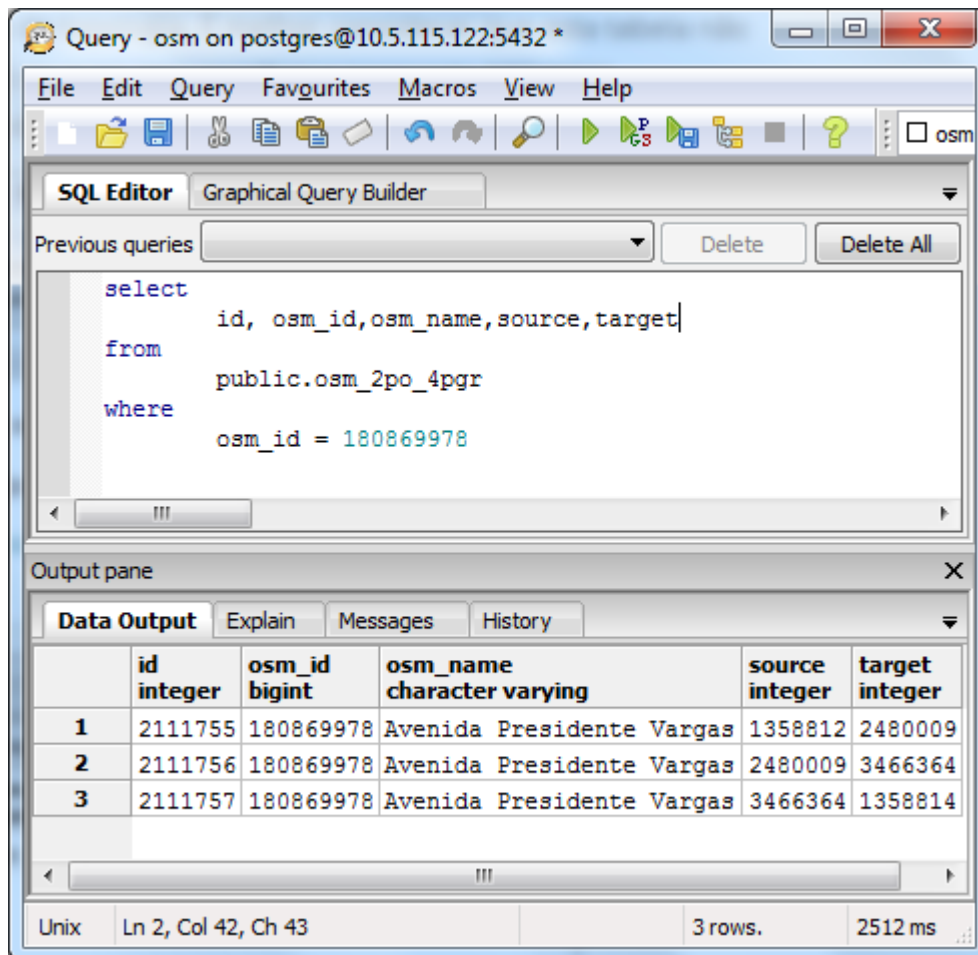
```
create or replace view av_pres_vargas as
select
    *
FROM
    planet_osm_line
WHERE
    planet_osm_line.name = 'Avenida Presidente Vargas'
```

Agora você perceberá a utilidade da visão da rua no mapa: existem várias avenidas com este nome no país. Para pegar somente a do Rio de Janeiro, seria necessário conhecer as coordenadas geográficas do centro da cidade e filtrar a geometria dos dados encontrados pela *view*. Além do mais, você precisará prestar atenção no *source* e *target* para saber qual segmento de rua se conecta com o outro. Eu acho mais fácil usar o mapa.

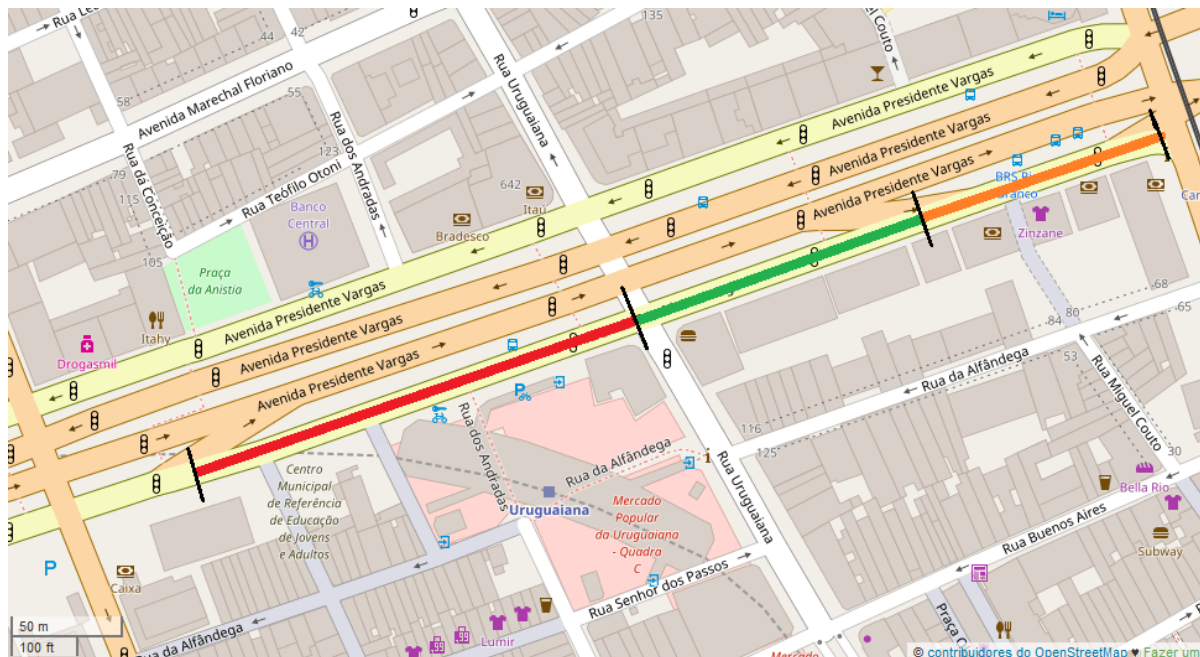
Após criar a *view* no banco de dados, usaremos exatamente o [mesmo processo do post anterior](#) para criar a camada do mapa no Geoserver. Eis o resultado:



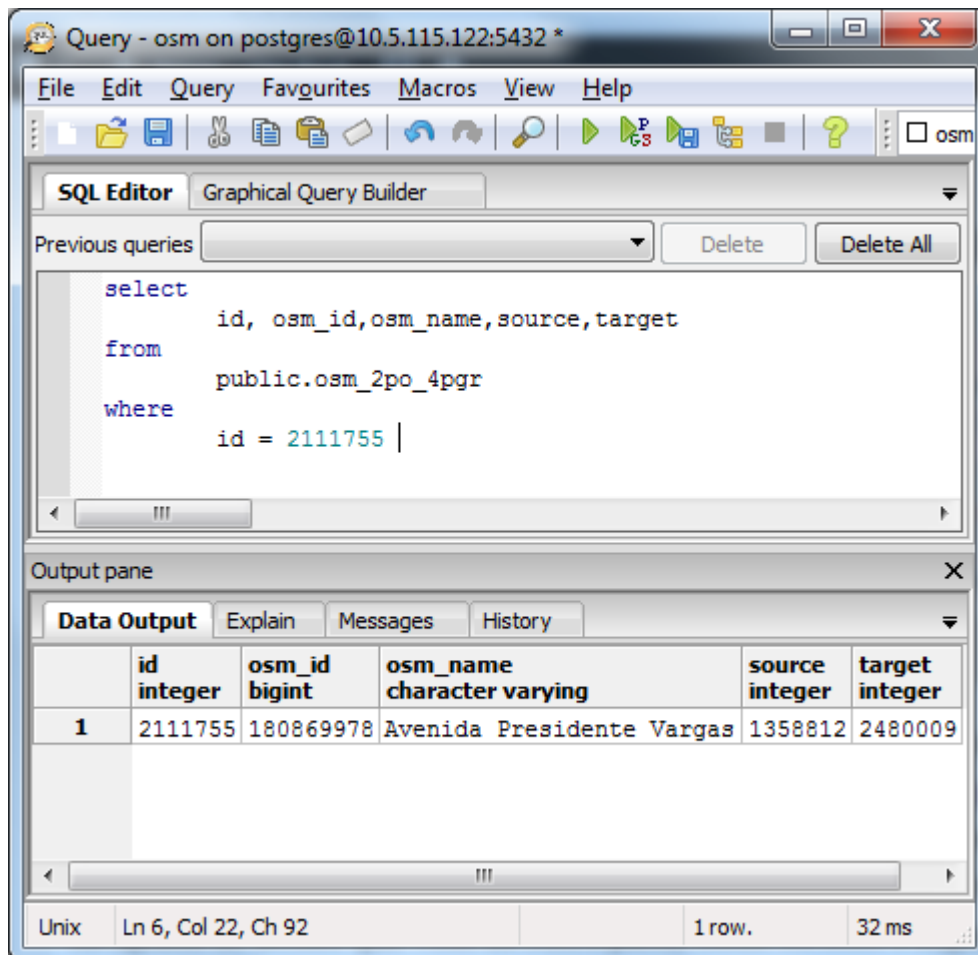
Como da outra vez, escolheremos um segmento qualquer. Vamos usar o segmento 180869978.



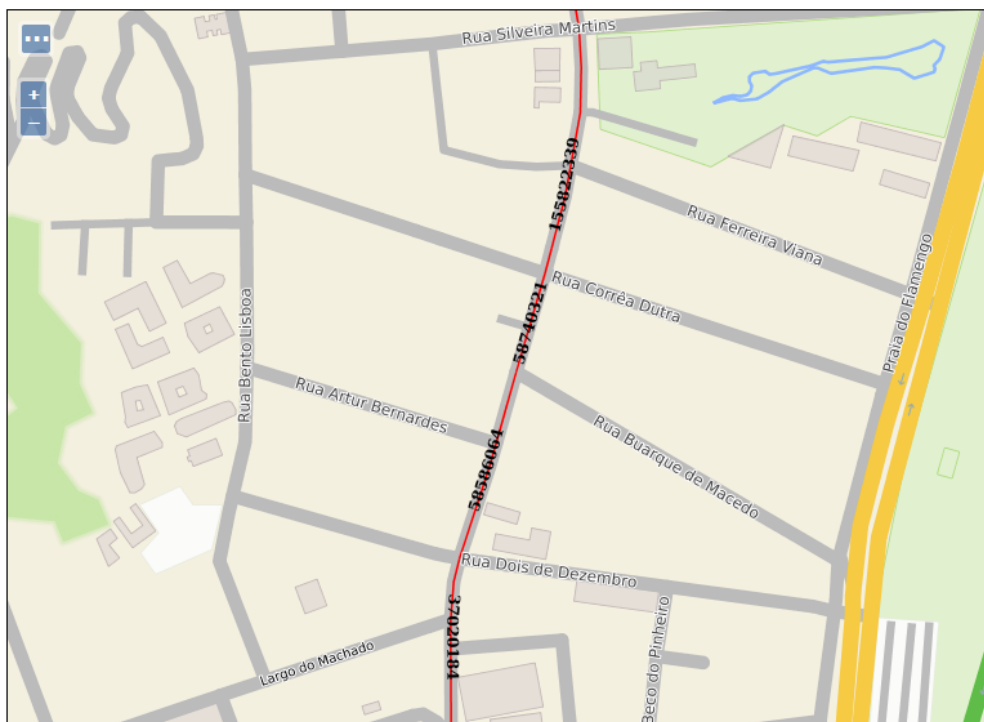
Opa! Encontramos 3 segmentos que apontam para a mesma rua nos dados originais do OSM (*planet_osm_line.osm_id*). Quando criamos a topologia (tabela *osm_2po_4pgr*), toda junção da rua com outras ruas é quebrada em um segmento. Sempre que uma rua “entra” em outra (é possível que o tráfego flua para esta rua), então um novo segmento é criado. Estamos então diante de dois tipos diferentes de segmento: o primeiro é como o próprio OSM entende a Avenida Pres. Vargas. Este entendimento é representado pelo segmento que existe na tabela *planet_osm_line* e possui o *osm_id* = 180869978. O segundo entendimento é o da topologia de rotas, na tabela *osm_2po_4pgr*, representado pelos 3 segmentos encontrados na consulta e que apontam para o mesmo segmento do OSM (mesmo *osm_id*). Vamos ampliar o mapa para ver o que houve. Eis o nosso segmento OSM 180869978 da Av. Pres. Vargas:



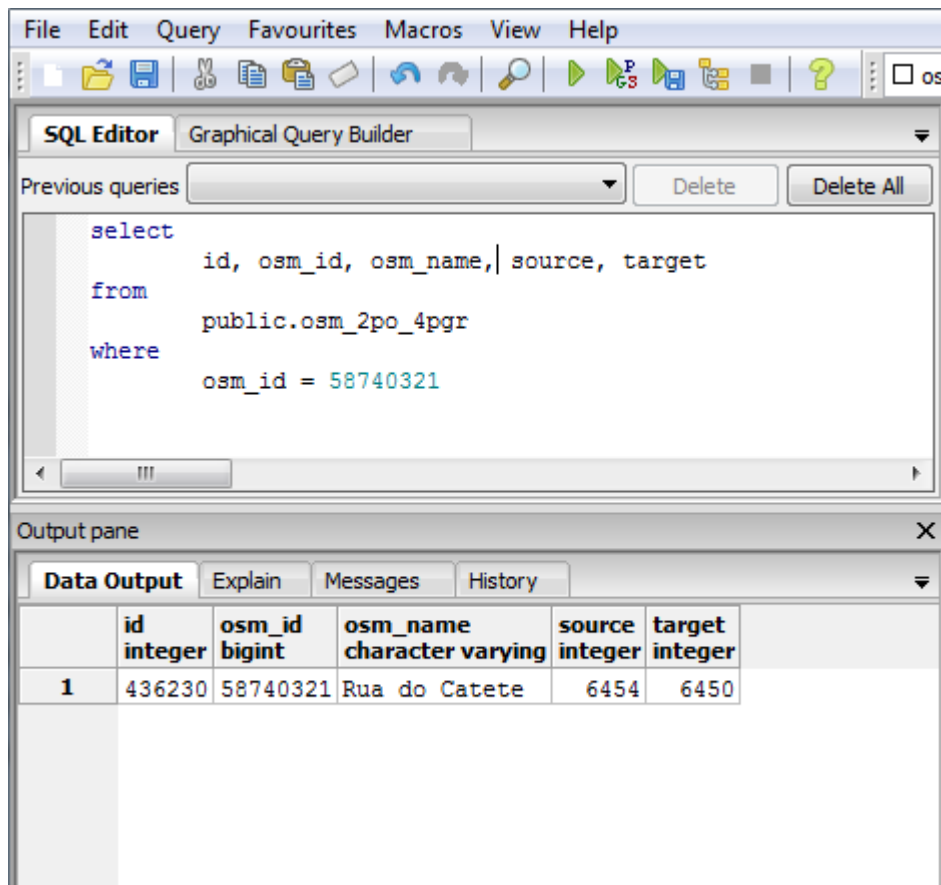
Repare que ele começa em uma saída de acesso para a pista lateral e termina na esquina com a Av. Rio Branco (no canto superior direito do mapa). Isso é [como o OSM percebe este segmento](#). Para efeito de rotas, é possível sair desta via e entrar na Rua Uruguaiana ou chegar pela Rua Uruguaiana e seguir nesta via, então o programa que criou a topologia fragmentou este segmento como eu marquei na cor vermelha. Também é possível chegar nesta via a partir da agulha de acesso que vem da outra pista, então foi feito o segundo fragmento, como eu marquei na cor laranja. O pedaço que liga os dois eu marquei em verde. Agora está explicado porque nosso segmento 180869978 possui 3 registros na tabela de rotas. Perceba também como seus valores de *source* e *target* os conectam perfeitamente. Dito isso, além de escolher um segmento, como fizemos com a Rua do Catete no post anterior, precisaremos decidir qual fragmento dele iremos usar. Vou escolher o primeiro (em vermelho), que corresponde ao ID 2111755 na listagem que conseguimos com o SQL.



Agora sim acabamos com a ambiguidade. Para o destino, vamos ficar com o nosso segmento 58740321 da Rua do Catete, mostrado no post anterior.



Selecionando no banco:



Já temos a origem na Av. Pres. Vargas (*source* = 1358812) e o destino na Rua do Catete (*target* = 6450). Podemos continuar. Vou dar como exemplo o algoritmo [K-Shortest Paths](#) (KSP), que seleciona as *k* rotas mais curtas, mas os outros algoritmos funcionam de forma semelhante. Não está no escopo deste artigo discutir sobre qual deles é o melhor, sendo que eu escolhi este simplesmente porque solucionou um problema de logística que eu tinha.

Quase todas as funções de rotas do *pgRouting* possuem a mesma estrutura: o valor *target* do segmento de destino, o valor *source* do segmento de origem, se vai obedecer a “mão” da rua e uma instrução SQL que vai fornecer o conjunto de ruas (universo de busca). Os parâmetros adicionais vão depender de cada função, sendo que no caso do KSP, é necessário ainda informar a quantidade de rotas que se deseja obter (valor do *K*).

A função K-Shortest Paths no *pgRouting* chama-se [pgr_ksp](#) e esta é a sua assinatura ([imagens do manual do pgRouting 2.3](#)):

```
pgr_ksp(edges_sql, start_vid, end_vid, k, directed, heap_paths)
```

onde:

Column	Type	Description
<code>edges_sql</code>	TEXT	SQL query as described above.
<code>start_vid</code>	BIGINT	Identifier of the starting vertex.
<code>end_vid</code>	BIGINT	Identifier of the ending vertex.
<code>k</code>	INTEGER	The desired number of paths.
<code>directed</code>	BOOLEAN	(optional). When <code>false</code> the graph is considered as Undirected. Default is <code>true</code> which considers the graph as Directed.
<code>heap_paths</code>	BOOLEAN	(optional). When <code>true</code> returns all the paths stored in the process heap. Default is <code>false</code> which only returns <code>k</code> paths.

O SQL que vai fornecer os dados de entrada para a busca (parâmetro `edges_sql`) deve possuir como retorno as seguintes colunas:

Column	Type	Default	Description
<code>id</code>	ANY-INTEGER		Identifier of the edge.
<code>source</code>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<code>target</code>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<code>cost</code>	ANY-NUMERICAL		Weight of the edge (<i>source</i> , <i>target</i>) <ul style="list-style-type: none">When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.
<code>reverse_cost</code>	ANY-NUMERICAL	-1	Weight of the edge (<i>target</i> , <i>source</i>), <ul style="list-style-type: none">When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.

No nosso caso, este SQL será

```
SELECT id, source, target, cost, reverse_cost FROM
osm_2po_4pgr
```

que retornará todo o conteúdo da tabela de topologias como universo de busca. Não se preocupe com isso por enquanto, pois pretendo mostrar como otimizar as buscas mais adiante. O parâmetro `cost` representa o custo de travessia do segmento, nesse caso, seu comprimento, já `reverse_cost` é o custo de travessia do segmento na “mão” contrária à direção do segmento caso decidirmos por usar o parâmetro `directed`, que informa se desejamos obedecer a “mão” das ruas ou não. Vou falar sobre isso mais adiante. Não vou me preocupar com o parâmetro `heap_paths` por não julgar importante para o escopo do artigo.

A função irá retornar uma relação (uma tabela) com a seguinte estrutura:

Column	Type	Description
<code>seq</code>	INTEGER	Sequential value starting from 1.
<code>path_seq</code>	INTEGER	Relative position in the path of <code>node</code> and <code>edge</code> . Has value 1 for the beginning of a path.
<code>path_id</code>	BIGINT	Path identifier. The ordering of the paths For two paths <i>i</i> , <i>j</i> if <i>i</i> < <i>j</i> then <code>agg_cost(i)</code> <= <code>agg_cost(j)</code> .
<code>node</code>	BIGINT	Identifier of the node in the path.
<code>edge</code>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. -1 for the last node of the route.
<code>cost</code>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<code>agg_cost</code>	FLOAT	Aggregate cost from <code>start_vid</code> to <code>node</code> .

O valor do atributo *path_seq* representa a sequencia de vias em uma determinada rota e o valor de *path_id* separa o conjunto de vias de cada rota. Se você optou por receber as 5 rotas mais curtas, *path_id* vai variar de 1 até 5 (ou até o número de caminhos encontrados). Os parâmetros *cost* e *agg_cost* representam respectivamente o comprimento em Km de um segmento e o comprimento total acumulado em Km do início da rota até o segmento selecionado. Tendo apresentado a função *pgr_ksp*, vamos construir uma função *wrapper* para facilitar nossa vida mais um pouco:

```
CREATE OR REPLACE FUNCTION public.calc_rotas(
    IN source integer,
    IN target integer,
    IN k integer,
    IN directed boolean)
RETURNS TABLE(
    seq integer,
    path_id integer,
    path_seq integer,
    node bigint,
    edge bigint,
    cost double precision,
    agg_cost double precision
) AS
$BODY$
SELECT
    *
FROM
    pgr_ksp(
        'SELECT id, source, target, cost, reverse_cost FROM
osm_2po_4pgr',$1, $2, $3, directed:=$4
    )
$BODY$
LANGUAGE sql VOLATILE COST 100;
```

Vamos executar a função. Vou pedir as 5 rotas mais curtas com início na Av. Pres. Vargas de término na Rua do Catete, sem me importar com a direção do tráfego (como pedestre, talvez):

Query - osm on postgres@10.5.115.122:5432 *

File Edit Query Favurites Macros View Help

SQL Editor Graphical Query Builder

Previous queries [v] Delete Delete All

```
select * from calc_rotas( 1358812, 6450, 5, false )
```

Output pane

Data Output Explain Messages History

	seq integer	path_id integer	path_seq integer	node bigint	edge bigint	cost double precision	agg_cost double precision
1	1	1	1	1358812	2111755	0.00381	0
2	2	1	2	2480009	3811819	0.0017831	0.00381
3	3	1	3	32674	3811820	0.0011176	0.0055931
4	4	1	4	333596	3811821	0.0010996	0.0067107
5	5	1	5	2480010	3812154	0.0019902	0.0078103
6	6	1	6	2480161	3812155	0.0038822	0.0098005
7	7	1	7	1611782	4394684	0.0014339	0.0136827
8	8	1	8	1912226	4394685	0.0007281	0.0151166
9	9	1	9	2877462	4394686	0.0020117	0.0158447
10	10	1	10	2441197	4394687	0.0003354	0.0178564
11	11	1	11	1233186	6250115	0.001214	0.0181918
12	12	1	12	70335	108342	0.001556	0.0194058
13	13	1	13	29839	43656	0.0021185	0.0209618
14	14	1	14	29840	108344	0.000723	0.0230803
15	15	1	15	70338	108345	0.002162	0.0238033

OK. Unix Ln 1, Col 53, Ch 53 234 rows. 116140 ms

Como você pode notar, a pesquisa demorou 116.140 ms para ser executada em um servidor dedicado, com 16G de RAM e 8 núcleos. Nada bom, mas como eu disse, ainda dá para melhorar muito este número com alguns truques. Além disso, vale lembrar que ele selecionou as 5 melhores rotas possíveis em um universo de, no meu caso, 11.574.907 de segmentos de ruas. Se optarmos por obedecer a direção do tráfego, este número aumenta consideravelmente.

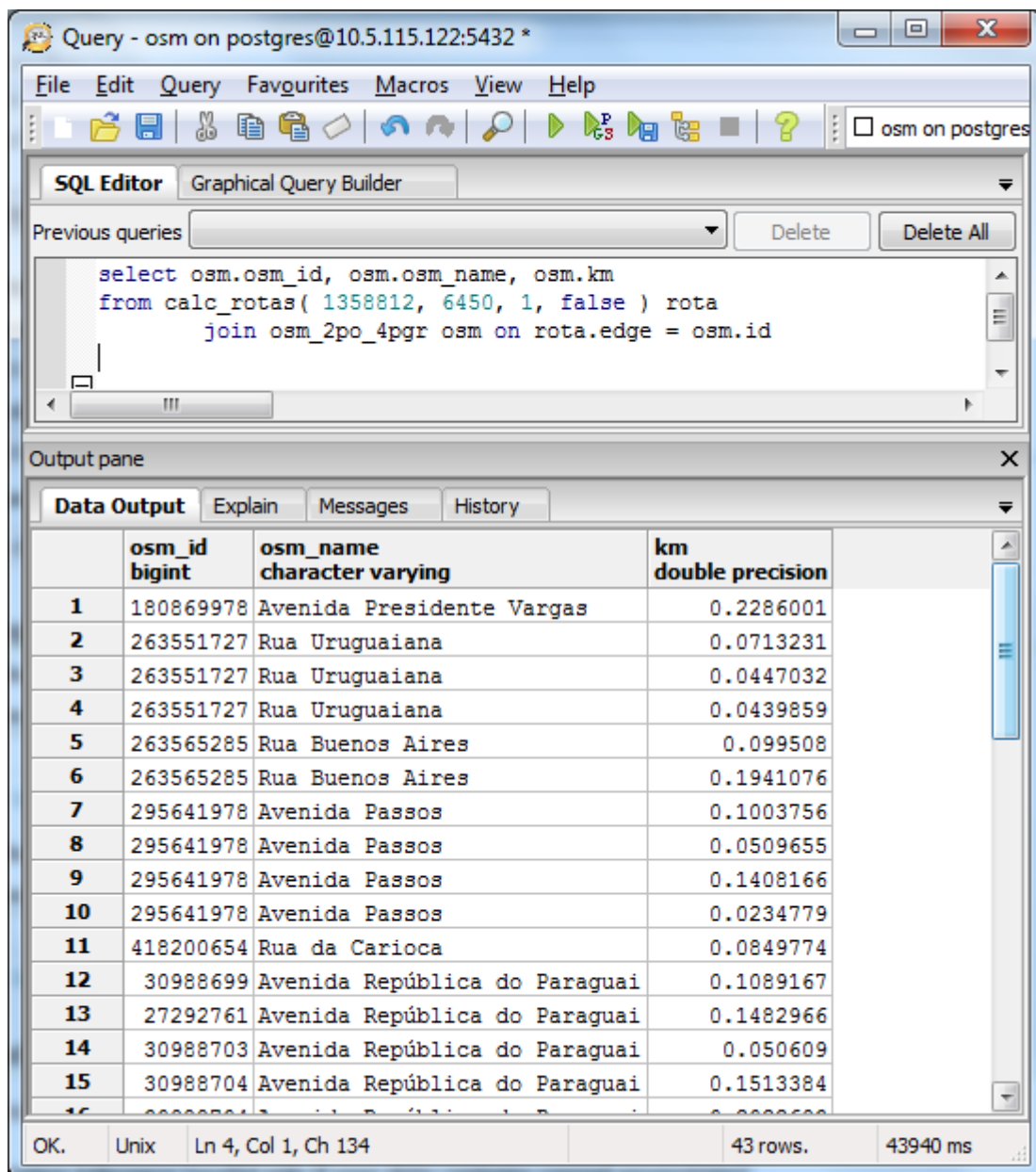
Mas este resultado não me disse muita coisa. Vamos melhorar um pouco mais com uma junção extra. Vou reduzir minhas opções para apenas uma rota para otimizar meu tempo:

```
select
    osm.osm_id, osm.osm_name, osm.km
from
    calc_rotas( 1358812, 6450, 1, false ) rota
```

join

```
osm_2po_4pgr osm on rota.edge = osm.id
```

Resultado:



The screenshot shows a PostgreSQL query editor window titled "Query - osm on postgres@10.5.115.122:5432 *". The SQL Editor tab is active, displaying the following query:

```
select osm.osm_id, osm.osm_name, osm.km
from calc_rotas( 1358812, 6450, 1, false ) rota
join osm_2po_4pgr osm on rota.edge = osm.id
```

The Output pane is visible below the SQL Editor, showing the "Data Output" tab. It displays a table with 4 columns: **osm_id** (bigint), **osm_name** (character varying), and **km** (double precision). The table contains 15 rows of data, numbered 1 to 15. The status bar at the bottom indicates "OK.", "Unix", "Ln 4, Col 1, Ch 134", "43 rows.", and "43940 ms".

	osm_id bigint	osm_name character varying	km double precision
1	180869978	Avenida Presidente Vargas	0.2286001
2	263551727	Rua Uruguaiana	0.0713231
3	263551727	Rua Uruguaiana	0.0447032
4	263551727	Rua Uruguaiana	0.0439859
5	263565285	Rua Buenos Aires	0.099508
6	263565285	Rua Buenos Aires	0.1941076
7	295641978	Avenida Passos	0.1003756
8	295641978	Avenida Passos	0.0509655
9	295641978	Avenida Passos	0.1408166
10	295641978	Avenida Passos	0.0234779
11	418200654	Rua da Carioca	0.0849774
12	30988699	Avenida República do Paraguai	0.1089167
13	27292761	Avenida República do Paraguai	0.1482966
14	30988703	Avenida República do Paraguai	0.050609
15	30988704	Avenida República do Paraguai	0.1513384

O que fiz foi pegar o ID do segmento que veio no resultado da rota (valor de *edge*) e procurar este segmento na tabela de topologias *osm_2po_4pgr*. Assim eu pude saber o nome da rua e seu comprimento, bem como seu *osm_id*, caso eu precise de mais detalhes que existem somente na tabela original do OSM (*planet_osm_line*). O tempo da consulta também reduziu bastante quando optei por receber somente uma rota. Bem melhor.

Caso você deseje ver isso no mapa, proceda criando uma camada tipo SQL View, como mostrado no [post anterior](#). Para o SQL de seleção, coloque por enquanto:

```
select osm.*  
from calc_rotas( 1358812, 6450, 1, false ) rota  
  join osm_2po_4pgr osm on rota.edge = osm.id
```

Dependendo do seu hardware, deve dar um pouco de trabalho para criar esta camada porque a consulta é muito demorada sem a otimização necessária.

No próximo post: [otimização da consulta](#), passagem de parâmetro para o GeoServer e início da construção da interface.

Referências:

https://en.wikipedia.org/wiki/Shortest_path_problem

https://en.wikipedia.org/wiki/K_shortest_path_routing

http://docs.pgrouting.org/2.3/en/src/ksp/doc/pgr_ksp.html

<http://docs.pgrouting.org/2.3/en/doc/src/developer/sampledata.html#sampledata>

<http://pgrouting.org/docs/howto/one-way.html>

[First taste of routing in PostGIS using pgRouting](#)

Trabalhando com rotas nos dados do OpenStreetMap

Para esta série de artigos, vou pressupor que você já possui um [ambiente OSM instalado](#) e ainda guarda com você o arquivo **south-america-latest.osm.pbf**. Se você não possui nada disso, acompanhe primeiro [esta série](#) antes de prosseguir.

Vou mostrar como criar uma tabela de vértices (topologia) dos dados de ruas do

OSM para então calcular rotas com eficiência.

Vamos precisar baixar o programa *Osm2Po*, que faz todo o trabalho de criação da topologia sem que você precise de muito esforço. De quebra ele ainda oferece um serviço WEB onde você já poderá calcular suas rotas, mas não é minha intenção usar serviços de terceiros! vamos criar nosso próprio sistema de cálculo de rotas com o [Geoserver](#) como servidor de mapas e o [OpenLayers](#) como interface com o usuário.

```
$ wget http://osm2po.de/releases/osm2po-5.1.0.zip
```

```
$ unzip osm2po-5.1.0.zip
```

Após baixar e descompactar o arquivo, edite o arquivo *osm2po.config* e retire os comentários das seguintes linhas:

```
postp.0.class = de.cm.osm2po.plugins.postp.PgRoutingWriter
postp.0.writeMultiLineStrings = true
postp.1.class = de.cm.osm2po.plugins.postp.PgVertexWriter
postp.2.class = de.cm.osm2po.plugins.postp.PgPolyWayWriter
postp.3.class = de.cm.osm2po.plugins.postp.PgPolyRelWriter
```

```
postp.4.class = de.cm.osm2po.postp.GeoExtensionBuilder
postp.5.class = de.cm.osm2po.postp.MlgExtensionBuilder
postp.5.id = 0
postp.5.maxLevel = 3, 1.0
```

```
postp.6.class = de.cm.osm2po.sd.postp.SdGraphBuilder
```

```
# Pg*Writer usually create sql files. Enable the following
# parameter to redirect them to stdout (console)
```

```
postp.pipeOut = false
```

Minha máquina possui 8GB de RAM, então eu serei generoso separando 5GB para a execução do programa. Se você quiser modificar isso, altere o parâmetro *-Xmx5g* no comando de execução abaixo.

```
java -Xmx5g -jar osm2po-core-5.1.0-signed.jar
tileSize=30x60,10 south-america-latest.osm.pbf
```

Digite “yes” para aceitar a licença. Só precisará fazer isso uma vez. Dependendo

da sua quantidade de memória, isso poderá demorar um pouco. Ao final da conversão, se tudo correr bem, o programa continuará rodando e você deverá ter um servidor web ouvindo na porta 8888:

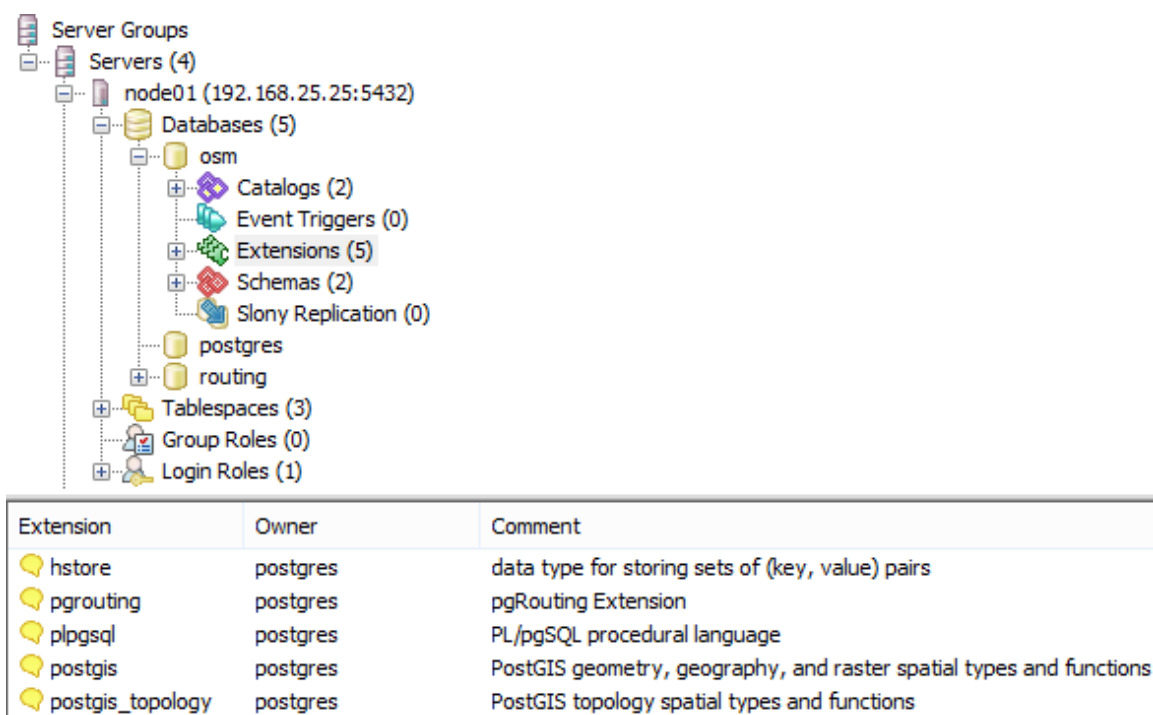
`http://localhost:8888/0sm2poService`

Pode parar o serviço usando CTRL+C. O que nos interessa é o conteúdo da pasta *osm* que foi criada. Dentro dela, entre outros arquivos, deverá conter o arquivo **osm_2po_4pgr.sql** (de tamanho um pouco exagerado para um SQL, mas é isso mesmo).

Vamos importar este SQL para o [banco de dados osm](#) do nosso OpenStreetMap:

```
$ su postgres (opcional, dependendo de seu ambiente. Você  
precisará estar em sudo -i para fazer isso)  
$ psql -U postgres -d osm -q -f "osm_2po_4pgr.sql" ( a  
conexão local precisa estar como "trust" )
```

Onde *postgres* é o usuário do servidor e *osm* é o nome do banco de dados. [Mais detalhes em como configurar a conexão local como trust](#). Certifique-se de que todas as extensões necessárias estão instaladas antes de executar a importação:



The screenshot shows a tree view of a PostgreSQL database instance. Under the 'Servers' section, 'node01 (192.168.25.25:5432)' is expanded, showing 'Databases (5)'. The 'osm' database is selected, showing its internal structure: 'Catalogs (2)', 'Event Triggers (0)', 'Extensions (5)', 'Schemas (2)', and 'Slony Replication (0)'. Below this, other databases like 'postgres' and 'routing' are listed, along with 'Tablespaces (3)', 'Group Roles (0)', and 'Login Roles (1)'. At the bottom, a table lists the installed extensions for the 'osm' database.

Extension	Owner	Comment
hstore	postgres	data type for storing sets of (key, value) pairs
pgrouting	postgres	pgRouting Extension
plpgsql	postgres	PL/pgSQL procedural language
postgis	postgres	PostGIS geometry, geography, and raster spatial types and functions
postgis_topology	postgres	PostGIS topology spatial types and functions

Banco de dados OSM

Ao final da importação teremos uma tabela chamada *osm_2po_4pgr* em nosso banco de dados *osm*. Pode apagar a pasta “*osm*” agora, caso tenha problemas de espaço em disco. Esta tabela contém basicamente os vértices de todas as ruas e estradas do banco de dados do OSM. Mas como funciona isso?

Inicialmente você deverá escolher as ruas de partida e destino da sua rota na tabela *osm_2po_4pgr* usando o nome da rua. O importador copia a geometria e o nome das ruas da tabela *planet_osm_line*. Se você precisar realizar uma busca usando outros critérios diretamente na tabela *planet_osm_line*, você poderá usar o atributo *osm_id* das duas para realizar o *join*.

Conforme foi dito, a tabela de topologia *osm_2po_4pgr* armazena os vértices que une as ruas. O que importa para o calculador de rotas é: “*dado um vértice, quais outros vértices eu consigo alcançar?*”. Para responder esta pergunta, a tabela contém dois atributos (campos) chamados *source* e *target*, que representam os vértices que conectam as ruas umas com as outras ou as segmentações da própria rua. É fácil concluir então que, dado o registro de uma rua qualquer, o seu *target* aponta para uma rua a qual ela está conectada (*source* com o mesmo valor) e seu *source* aponta para uma rua que se conecta nela (*target* com o mesmo valor). Basta então seguir a trilha de *sources* e *targets* da origem até o destino, encontrando quais ruas possuem o *source* igual ao *target* da rua anterior. Como o OSM fragmenta as ruas, isso também vale para os segmentos da mesma rua. Na imagem abaixo podemos ver um trecho da Rua do Catete (no Rio de Janeiro) fragmentada e com seus vértices marcados em vermelho. Cada círculo vermelho é o *source* de um segmento e o *target* do segmento seguinte.

para o PostgreSQL: Parte 1

Após [preparar o ambiente do PostgrSQL](#), é hora de importar os dados do OSM. No site Geofabrik existem [vários arquivos de dados](#) onde você pode escolher a área de cobertura desejada, desde todo o planeta até países. Além disso, acessando <http://www.openstreetmap.org> você poderá baixar os dados de uma área a sua escolha.

Eu vou importar os dados da América do Sul, mas se as coisas ficarem difíceis devido a requisitos de memória e processamento, escolha um [arquivo menor](#). Baixe sempre arquivos tipo PBF.

```
wget http://download.geofabrik.de/south-america-latest.osm.pbf
```

Você também vai precisar de um arquivo chamado *default.style*. Este arquivo vai informar ao programa de importação o que você quer e o que não quer mandar para o banco de dados. Eu não sei o motivo deste arquivo se chamar *style*, já que não tem nada a ver com estilos, e sim com filtro de dados.

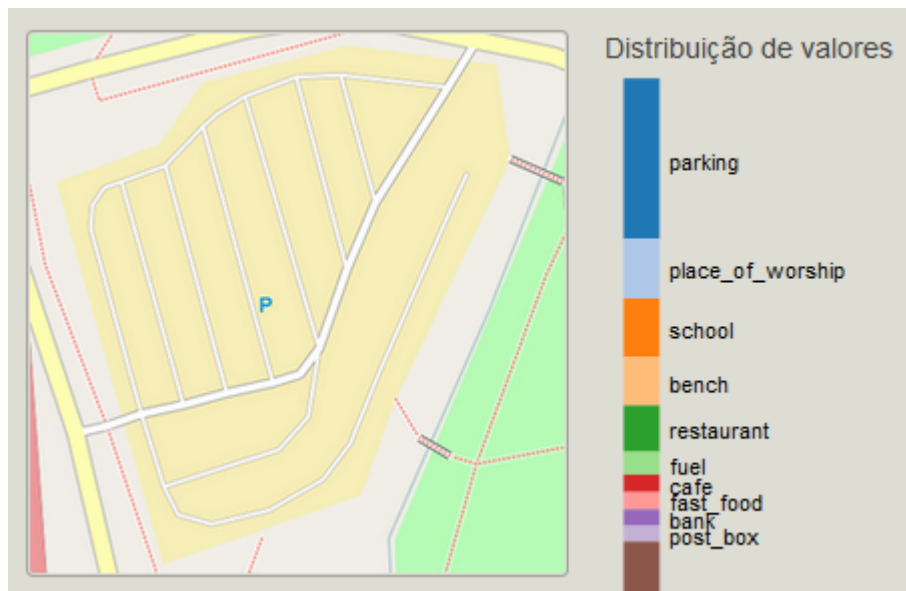
Antes de puxar o gatilho vou explicar uma coisa: o resultado disso tudo será a criação de 4 tabelas em seu banco de dados (vai criar um pouco mais, porém as tabelas do OSM são só 4):

```
planet_osm_point  
planet_osm_polygon  
planet_osm_line  
planet_osm_roads
```

Os nomes são intuitivos. O que eu quero explicar é o seguinte: todas as tabelas terão a mesma estrutura, baseado no que você decidir no arquivo *default.style*. A estrutura do OSM funciona baseada em elementos chamados *tags*. Cada tag pode possuir uma certa quantidade de valores. O site <http://taginfo.openstreetmap.org/> permite consultar todas as tags e seus possíveis valores. A [tag amenity](#), por exemplo, possui a seguinte descrição no site:

For describing useful and important facilities for visitors and residents.

Esta tag possui valores como *parking*, *place_of_worship*, *school*, *bank*, *fuel*, etc...



O site descreve em detalhes o que significa cada um destes valores para todas as tags e como eles se relacionam entre si quando agrupados com outras tags para dar sentido à informação.

amenity=fuel

A retail-type facility where vehicles can be refueled.

Visão geral

Combinações

Mapa

Wiki

Projetos

Visão geral

Tipo	Número de objetos	
✱ Todos	300 581	0.01%
○ Ponto	217 693	0.19%
▣ Linha	82 015	0.02%
▢ Relação	873	0.02%



Podemos encontrar também a informação de onde esta tag é mais frequente observando a “Visão Geral”. O exemplo da imagem acima significa que esta informação pode ocorrer mais frequentemente como pontos. Após a importação dos dados, cada tag irá se tornar uma coluna nas tabelas do OSM e seus valores serão as linhas destas tabelas. Daí pode-se concluir que as tabelas do OSM não são normalizadas e possuem uma vasta quantidade de valores nulos, dependendo da consulta que você fizer.

Query - osm on postgres@10.5.115.122:5432 *

File Edit Query Favurites Macros View Help

SQL Editor Graphical Query Builder

Previous queries Delete Delete All

```
select
    "addr:street", amenity, "name"
from
    planet_osm_point
where
    amenity = 'bank' and
    "addr:street" = 'Rua das Laranjeiras'
```

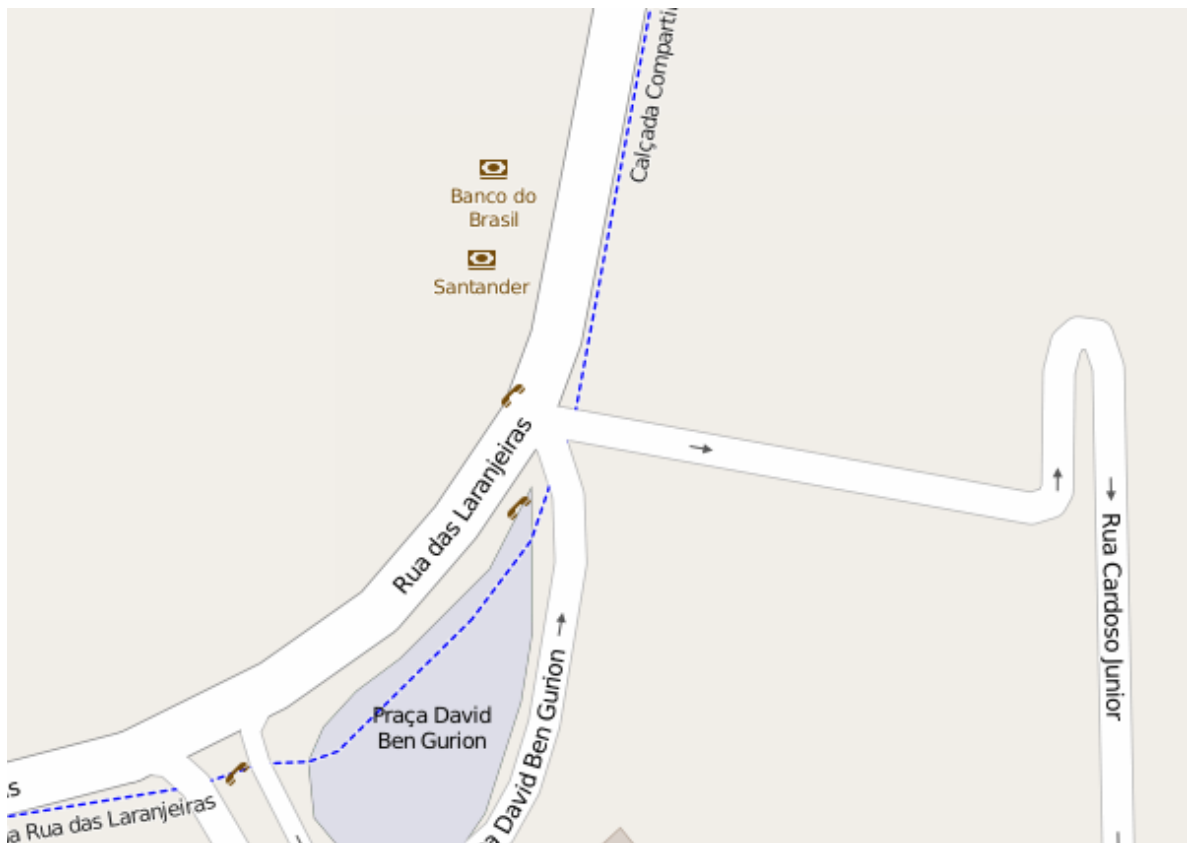
Output pane

Data Output Explain Messages History

	addr:street text	amenity text	name text
1	Rua das Laranjeiras	bank	Santander
2	Rua das Laranjeiras	bank	Banco do Brasil

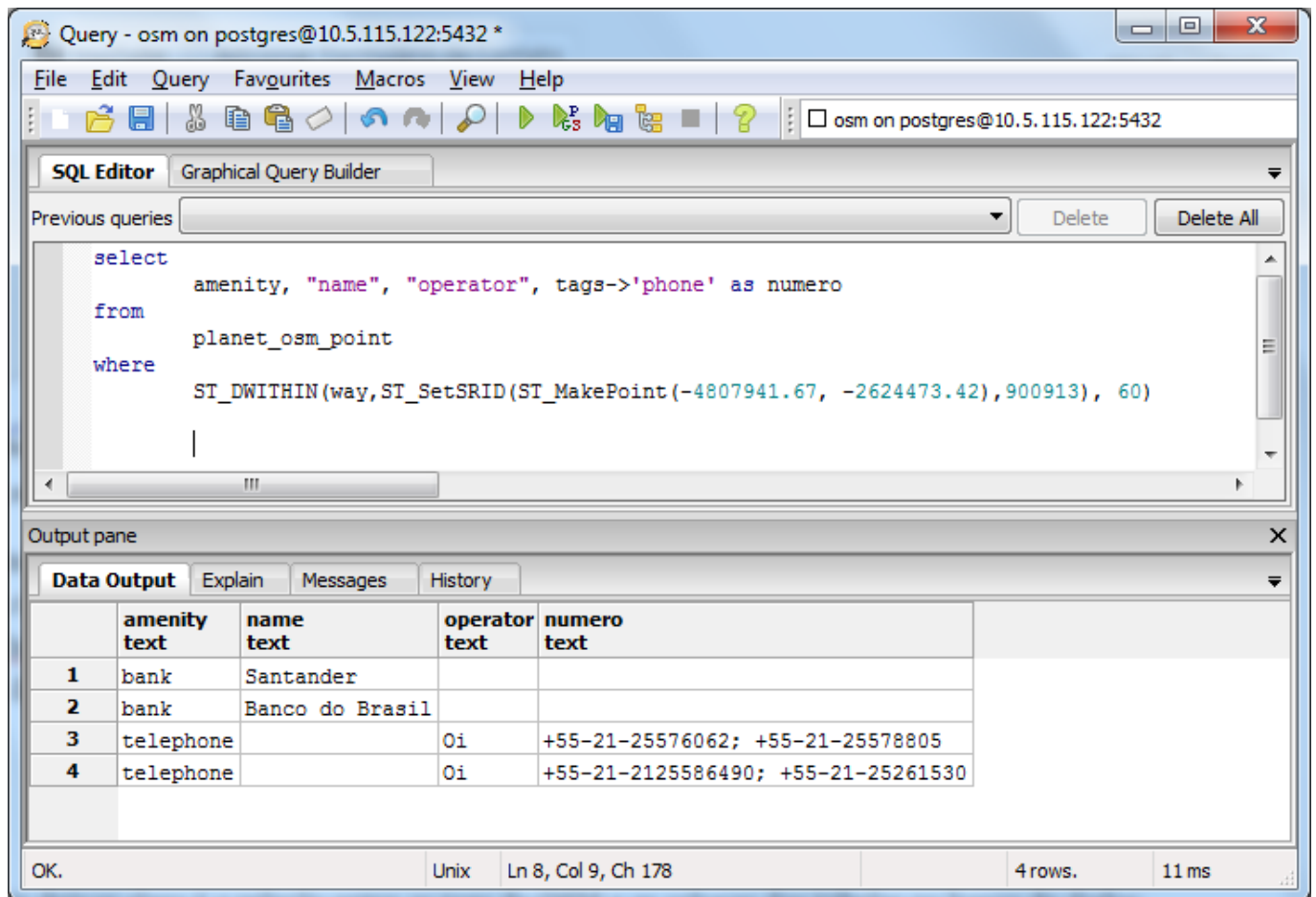
OK. Unix Ln 4, Col 25, Ch 65 2 rows. 422 ms

Consultando estes bancos da tag “amenity”...



...resulta neste mapa

Obviamente existem muito mais bancos na Rua das Laranjeiras (RJ). Isso depende da combinação de valores com outras tags. Nem todos os bancos possuem a tag *"addr:street"* preenchidas. Nesse caso eu deveria consultar os bancos usando o *bounding box* das coordenadas geográficas de Laranjeiras. Perto dos nossos bancos existem quatro telefones públicos (dois *postes* com dois telefones cada). Um na praça e outro perto do Santander. Vamos ligar para eles?

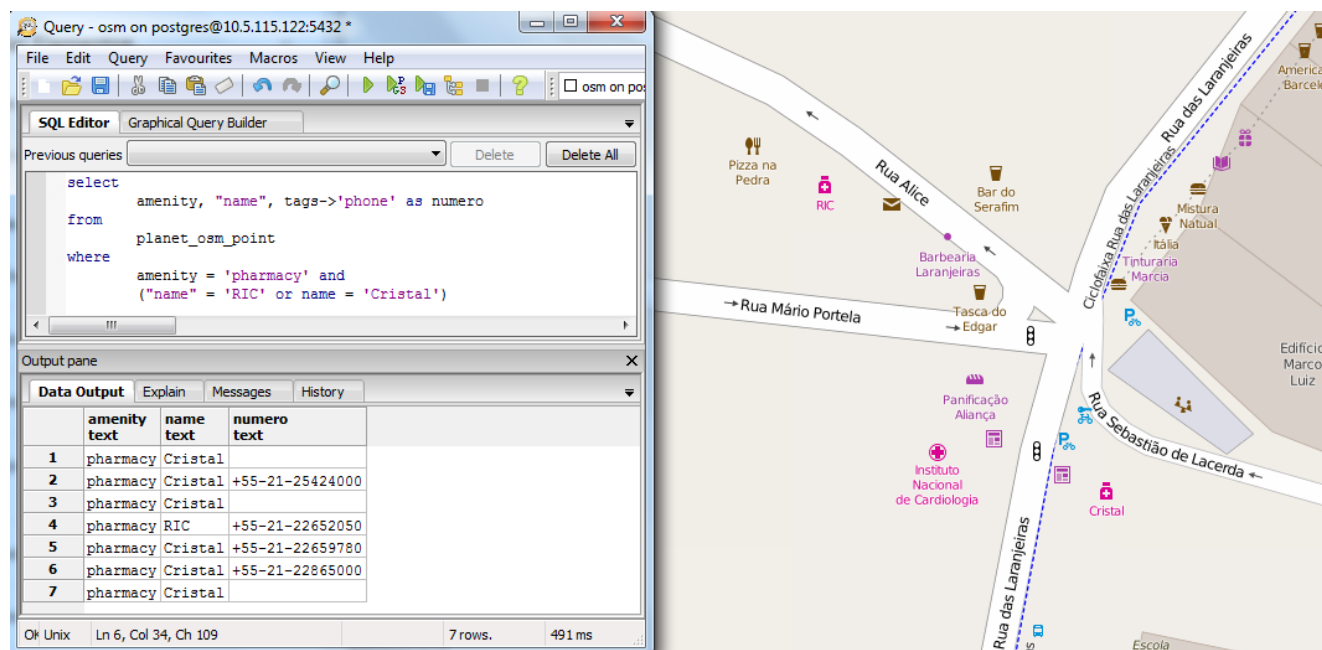


Nesta consulta eu mandei pesquisar tudo da tabela de pontos (a rua e a ciclovia em azul estão na tabela de linhas e a praça na tabela de polígonos) dentro de um raio de 60 metros das coordenadas do Banco do Brasil (serve do Santander também) . Para conseguir estas coordenadas, modifique a consulta dos bancos para:

```
select
    "addr:street", amenity, "name" , ST_AsText(way) as
coordenadas
from
    planet_osm_point
where
    amenity = 'bank' and
    "addr:street" = 'Rua das Laranjeiras'
```

Isso vai mostrar as coordenadas dos elementos encontrados. Bem, o que eu quis deixar claro é a relação entre as tags do OSM e as colunas das tabelas no banco de dados, bem como mostrar que vão existir colunas que não são preenchidas porque não fazem sentido junto com outras informações (o banco não tem operador e o telefone não tem nome). Já deu para notar que, se você não quiser

mapas, tudo bem: você terá uma boa massa de dados para fazer todo tipo de consultas interessantes em seus aplicativos.



Algo mais útil: Telefones de farmácias próximas.

Um arquivo default.style irá se parecer com isso:

node,way	admin_level	text	linear
node,way	aerialway	text	linear
node,way	aeroway	text	polygon
node,way	amenity	text	polygon
node,way	barrier	text	linear
node,way	bicycle	text	linear
node,way	brand	text	linear
node,way	bridge	text	linear
node,way	boundary	text	linear
node,way	electrified	text	linear
node,way	building	text	polygon

O [arquivo original de exemplo](#) contém uma boa descrição do que significam estes valores e como consultar o site [taginfo](#) para criar seus próprios critérios de importação.

Como este post já ficou bem grande, vou deixar a importação dos dados do OSM para o próximo.

Criando um Servidor de Mapas: Instalando o PostgreSQL

Após [instalar o GeoServer](#), é necessário instalar o gerenciador de banco de dados PostgreSQL juntamente com o PostGIS. Existe um pacote contendo tudo o que você precisa, mas não está no repositório oficial do Ubuntu. Mais informações no [site do PostGIS](#). Não é o escopo deste artigo ensinar como instalar o PostgreSQL, pois o método de instalação envolve muitos passos que podem mudar drasticamente com as versões, o que tornaria meu artigo obsoleto rapidamente. Você pode acompanhar [este tutorial para realizar a instalação](#). Basicamente, é necessário ver a versão do Ubuntu que você está usando (*codename*), adicionar a URL do repositório na sua lista do *apt* e efetuar a instalação. Como estamos em um terminal sem interface gráfica (pelo menos eu recomendo dessa forma) não é necessário instalar o *PgAdmin*.

Não execute o passo a seguir sem antes ter [adicionado o repositório do apt.postgresql.org](#) ([alternativo](#)) na sua lista do *apt*. Existem outras alternativas, como instalar o PostgreSQL e depois instalar o PostGIS e o pgRouting, mas eu acho esta forma mais fácil. [Neste site](#) você encontra informações úteis sobre instalação e atualização.

```
apt-get install postgresql-9.5-postgis-2.2
```

Já instalei com o PostGIS, que é uma extensão que permite usar dados georreferenciados. Verifique se foi tudo bem com o PostgreSQL. Vou puxar um *sudo* interativo para esta situação, mas você não deve fazer disso um hábito para não causar acidentes:

```
$ sudo -i (não faça disso um hábito)
```

```
$ su postgres
```

```
$ psql
```

```
$> \l ( barra + letra L minúscula : lista os bancos de dados existentes )
```

```
$> \q ( barra + letra Q minúscula : sair do psql )
```



```
$ exit
```

```
$ exit ( do sudo )
```

Aproveita que está no `psql` e já troca logo a senha do usuário *postgres*. Troque esta senha fantástica pela sua própria (e lembre-se dela!).

```
$ su postgres
```

```
$ psql
```

```
$> ALTER USER Postgres WITH PASSWORD 'zebrasemlistra';
```

```
$\q
```

Se tudo foi bem, você deverá ter o PostgreSQL instalado em sua máquina.

Eu alterei a configuração do PostgreSQL para permitir o acesso externo ao servidor e o acesso local sem precisar fornecer senha. Edite o seguinte arquivo :

```
$ vi /etc/postgresql/9.3/main/pg_hba.conf
```

Altere os itens conforme a seguir:

Troque:

```
local      all      all      peer
```

Por:

```
local      all      all      trust
```

Troque:

```
host       all      all      peer
```

Por:

```
host       all      all      trust
```

Para melhorar o desempenho durante a importação e a exibição de mapas, modifique estes parâmetros na configuração do PostgreSQL:

```
$ vi /etc/postgresql/9.3/main/postgresql.conf
```

option	default	recommended
shared_buffers	24 MB	4 GB
work_mem	1 MB	100MB

maintenance_work_mem	16 MB	4096 MB
fsync	on	off
autovacuum	on	off (*)
checkpoint_segments		60
random_page_cost	4.0	1.1
effective_io_concurrency	1	2
temp_tablespace	''	'tablespace_1'
listen_address	'localhost'	'*'

Vamos precisar de uma pasta para o *table space*: um banco de dados para o OpenStreetMap (OSM) e as extensões *postgis*, *postgis_topology* e *hstore*.

```
$ mkdir -p /media/osm/postgres/tablespace_1
$ chown postgres /media/osm/postgres/tablespace_1
$ su postgres
$ psql
```

```
$> create tablespace tablespace_1 location
'/media/osm/postgres/tablespace_1';
$>
CREATE DATABASE osm WITH OWNER boundless tablespace tablespace
_1;
$> connect osm;
$> CREATE EXTENSION postgis;
$> CREATE EXTENSION postgis_topology;
$> CREATE EXTENSION hstore;
$> SELECT postgis_full_version();
$> \q
$ exit
```

Reinicie o PostgreSQL:

```
$ service postgresql restart
```

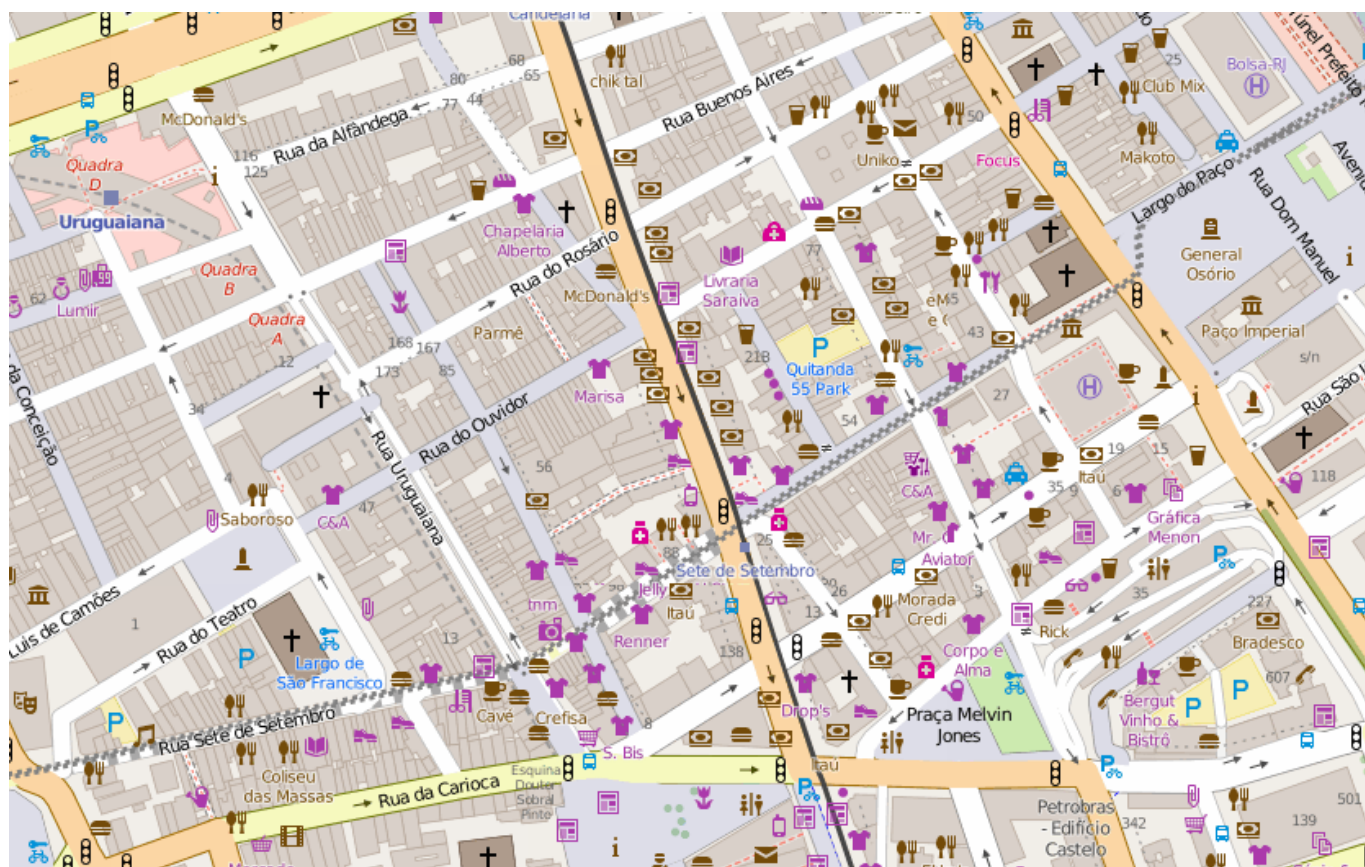
A ferramenta de importação dos dados do OpenStreetMap para o banco de dados do PostgreSQL é o *osm2pgsql*:

```
apt-get install osm2pgsql
```

No próximo post: [baixando o arquivo de dados do OpenStreetMap](#).

Criando um servidor de Mapas

Nesta série de posts vou mostrar os passos que segui para criar um servidor de mapas [OpenStreetMap](#) em uma máquina local.



O OpenStreetMap (OSM) possui um incrível banco de dados com várias informações georreferenciadas do mundo inteiro, atualizado diariamente pela comunidade mundial de voluntários. Existe ainda um [vasto catálogo de informações](#) com ilustrações, onde é possível ver os tipos de elementos

disponíveis e como consultá-los no banco.

Para possuir um desses, você precisará de um bom hardware. É imprescindível ter um linux (preferencialmente o Ubuntu).

Mínimo:

i5 ou equivalente.

16 GB RAM (8 até roda, mas vai ficar lento).

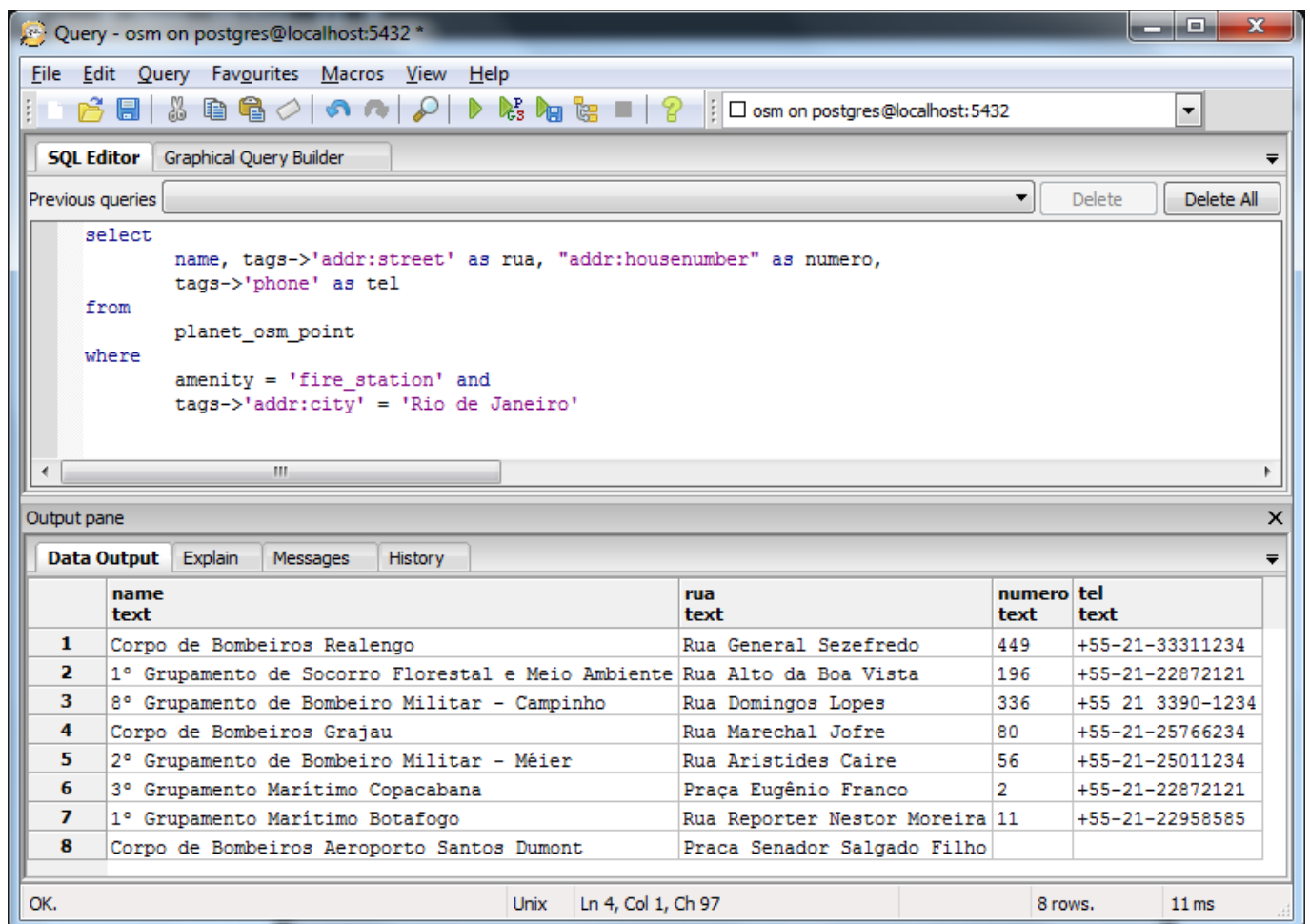
60 GB de HD livres.

Ubuntu 14.04

Internet

Na consulta a seguir eu usei a informação na página 11 do catálogo:

amenity | fire_station



The screenshot shows a PostgreSQL query editor window titled "Query - osm on postgres@localhost:5432 *". The SQL Editor tab is active, displaying the following query:

```
select
    name, tags->'addr:street' as rua, "addr:housenumber" as numero,
    tags->'phone' as tel
from
    planet_osm_point
where
    amenity = 'fire_station' and
    tags->'addr:city' = 'Rio de Janeiro'
```

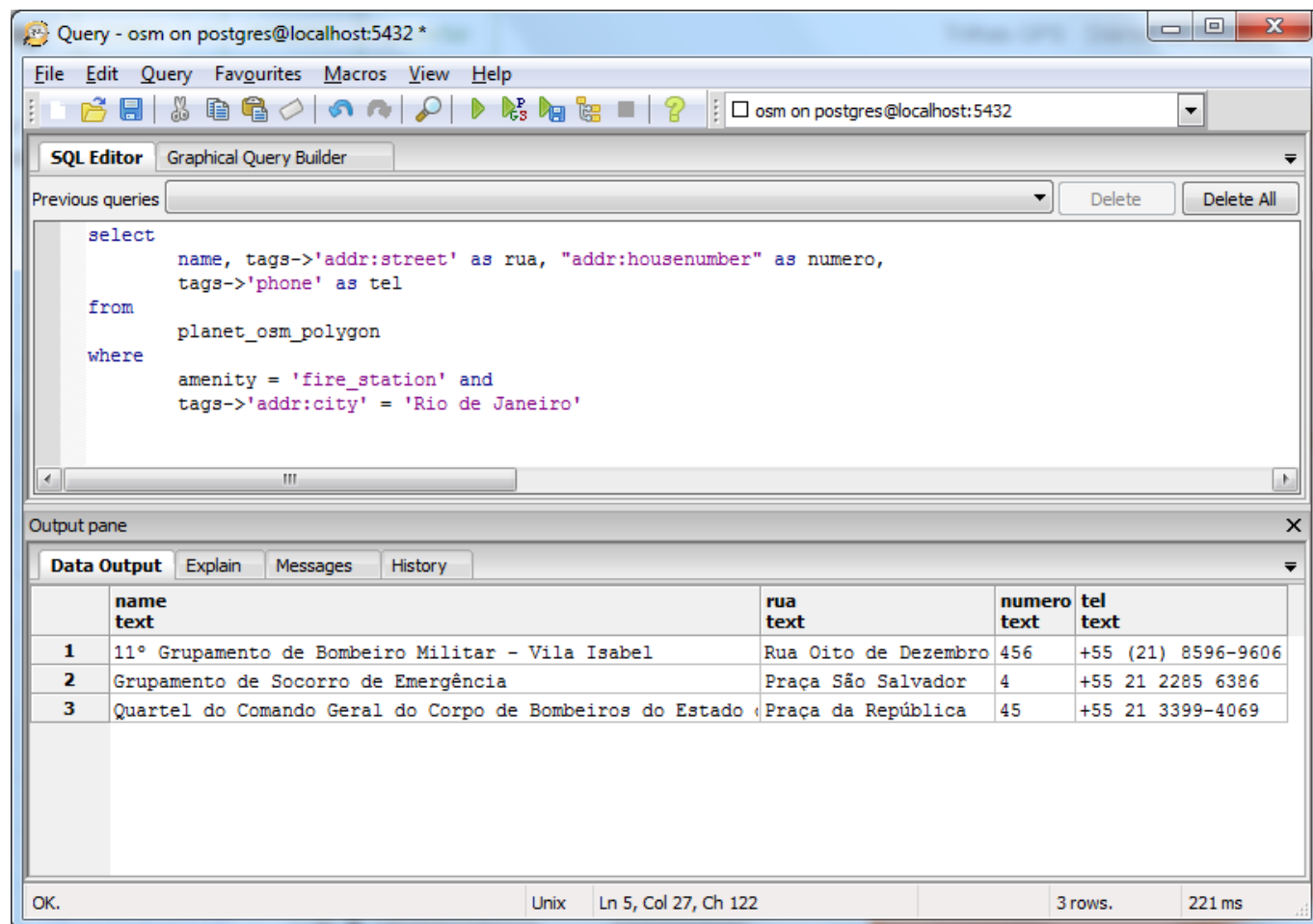
The Output pane at the bottom shows the results of the query in a table with 5 columns: name, rua, numero, and tel. The results are as follows:

	name text	rua text	numero text	tel text
1	Corpo de Bombeiros Realengo	Rua General Sezefredo	449	+55-21-33311234
2	1º Grupamento de Socorro Florestal e Meio Ambiente	Rua Alto da Boa Vista	196	+55-21-22872121
3	8º Grupamento de Bombeiro Militar - Campinho	Rua Domingos Lopes	336	+55 21 3390-1234
4	Corpo de Bombeiros Grajau	Rua Marechal Jofre	80	+55-21-25766234
5	2º Grupamento de Bombeiro Militar - Méier	Rua Aristides Caire	56	+55-21-25011234
6	3º Grupamento Marítimo Copacabana	Praça Eugênio Franco	2	+55-21-22872121
7	1º Grupamento Marítimo Botafogo	Rua Reporter Nestor Moreira	11	+55-21-22958585
8	Corpo de Bombeiros Aeroporto Santos Dumont	Praca Senador Salgado Filho		

The status bar at the bottom indicates "OK.", "Unix", "Ln 4, Col 1, Ch 97", "8 rows.", and "11 ms".

As 3 tabelas do OSM (*planet_osm_line*, *planet_osm_point* e *planet_osm_polygon*) possuem a mesma estrutura, o que significa que a mesma consulta pode ser aplicada para pontos, polígonos e linhas. Mas como saber onde está o que eu

preciso? no catálogo, existe um símbolo que indica em quais tabelas a mesma informação está presente. No caso de fire_station (página 11), posso encontrar um quartel de bombeiros como um ponto (latitude, longitude) ou como um polígono (área que o quartel ocupa no mapa).



The screenshot shows a PostgreSQL query editor window titled "Query - osm on postgres@localhost:5432 *". The window has a menu bar (File, Edit, Query, Favourites, Macros, View, Help) and a toolbar. The "SQL Editor" tab is active, displaying the following SQL query:

```
select
    name, tags->'addr:street' as rua, "addr:housenumber" as numero,
    tags->'phone' as tel
from
    planet_osm_polygon
where
    amenity = 'fire_station' and
    tags->'addr:city' = 'Rio de Janeiro'
```

Below the query editor is the "Output pane" with tabs for "Data Output", "Explain", "Messages", and "History". The "Data Output" tab is selected, showing a table with 5 columns: an index column, "name text", "rua text", "numero text", and "tel text". The table contains 3 rows of data:

	name text	rua text	numero text	tel text
1	11º Grupamento de Bombeiro Militar - Vila Isabel	Rua Oito de Dezembro	456	+55 (21) 8596-9606
2	Grupamento de Socorro de Emergência	Praça São Salvador	4	+55 21 2285 6386
3	Quartel do Comando Geral do Corpo de Bombeiros do Estado	Praça da República	45	+55 21 3399-4069

The status bar at the bottom of the window displays "OK.", "Unix", "Ln 5, Col 27, Ch 122", "3 rows.", and "221 ms".



Oportunamente, vou mostrar como usar este mesmo banco de dados para criar um sistema de cálculo de rotas.

Recomendo que você possua uma instalação limpa do Ubuntu com o Java instalado. Se você não lembra como instalar o Java:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

```
magno@AKRAB:~$ java -version
```

```
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed
mode)
```

```
$ sudo apt-get install oracle-java8-set-default
```

O próximo passo é [instalar o Tomcat](#):

```
sudo apt-get install tomcat7
```


Agora é só instalar o GeoServer, que é, em curtas palavras, um software capaz de transformar dados georreferenciados em imagens de mapas que podem ser acessadas usando um navegador de internet.

Vá no site e baixe a versão mais atual:

<http://geoserver.org/release/stable/>

Existem algumas opções de download que você deve considerar:

- Web Archive : É um arquivo Java WAR que você pode instalar em um servidor web (Tomcat, por exemplo). É a minha preferida.
- Windows Installer: É o famoso “deixa que eu faço, mas não te digo o que fiz.”
- Platform Independent Binary: A opção para quem não tem um servidor Tomcat já pronto e também não quer ter trabalho. Ele executa seu próprio servidor web.

Vamos escolher o WAR. Baixe o arquivo e copie para a pasta `/var/lib/tomcat7/webapps/`.

```
$ wget http://sourceforge.net/projects/geoserver/files/GeoServer/2.9.0/geoserver-2.9.0-war.zip
```

URL alternativa:

```
http://ufpr.dl.sourceforge.net/project/geoserver/GeoServer/2.9.0/geoserver-2.9.0-war.zip
```

Versão mais recente (não testada):

```
http://sourceforge.net/projects/geoserver/files/GeoServer/2.10.0/geoserver-2.10.0-war.zip
```

```
http://ufpr.dl.sourceforge.net/project/geoserver/GeoServer/2.10.0/geoserver-2.10.0-war.zip
```

```
$ unzip geoserver-2.9.0-war.zip
$ cp geoserver.war /var/lib/tomcat7/webapps
```

Reinicie o Tomcat

```
sudo service tomcat7 restart
```

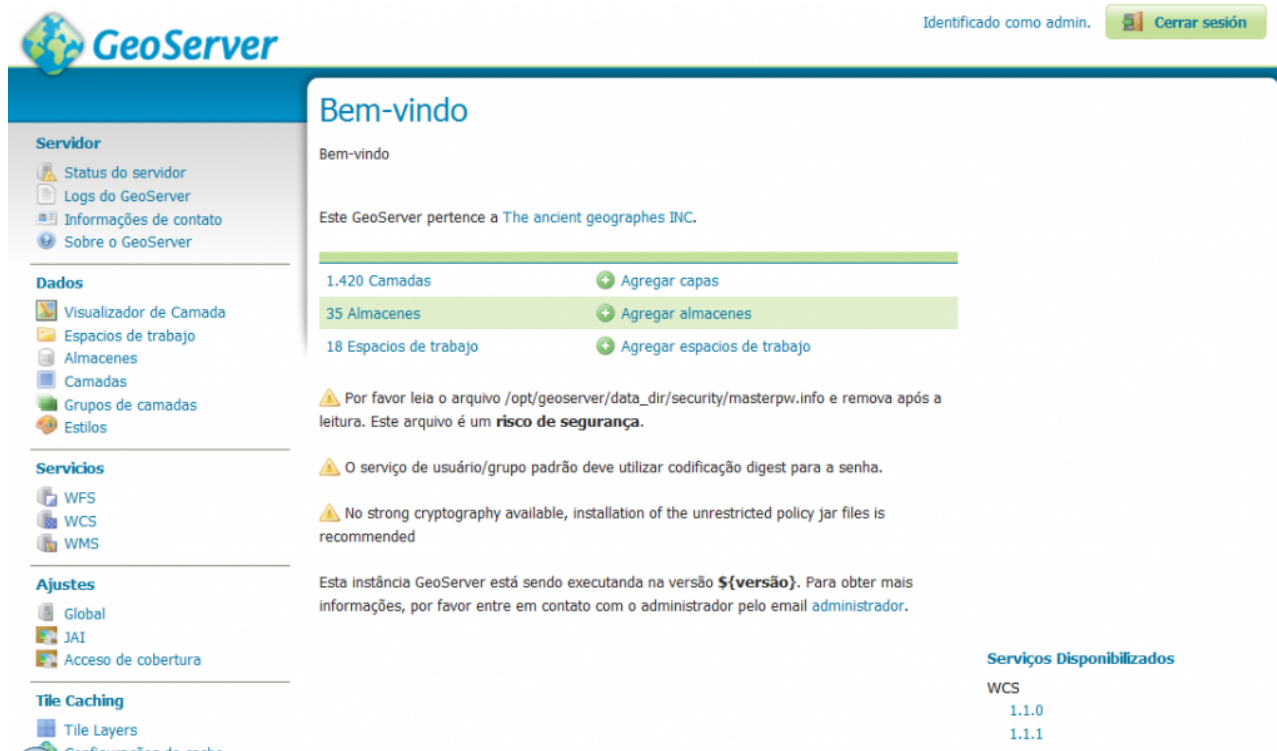
Se tudo deu certo, você pode apontar seu navegador para

<http://localhost:8080/geoserver>

Faça login em seu novo servidor:

Usuário: admin

Senha: geoserver



Não, você não pode trocar o idioma. Ele detecta do seu browser e o mais perto do português que chegará é o espanhol.

Não é o escopo deste post ensinar a usar o GeoServer, mas você estiver curioso, poderá ir em “Visualizador de Camada” e clicar em “OpenLayers” ao lado de cada nome para ver um dos mapas de exemplo.

[No próximo post](#) vou mostrar como instalar o banco de dados PostgreSQL e como colocar todos os quiosques de cachorro-quente da América do Sul na palma de sua mão.