

# 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:

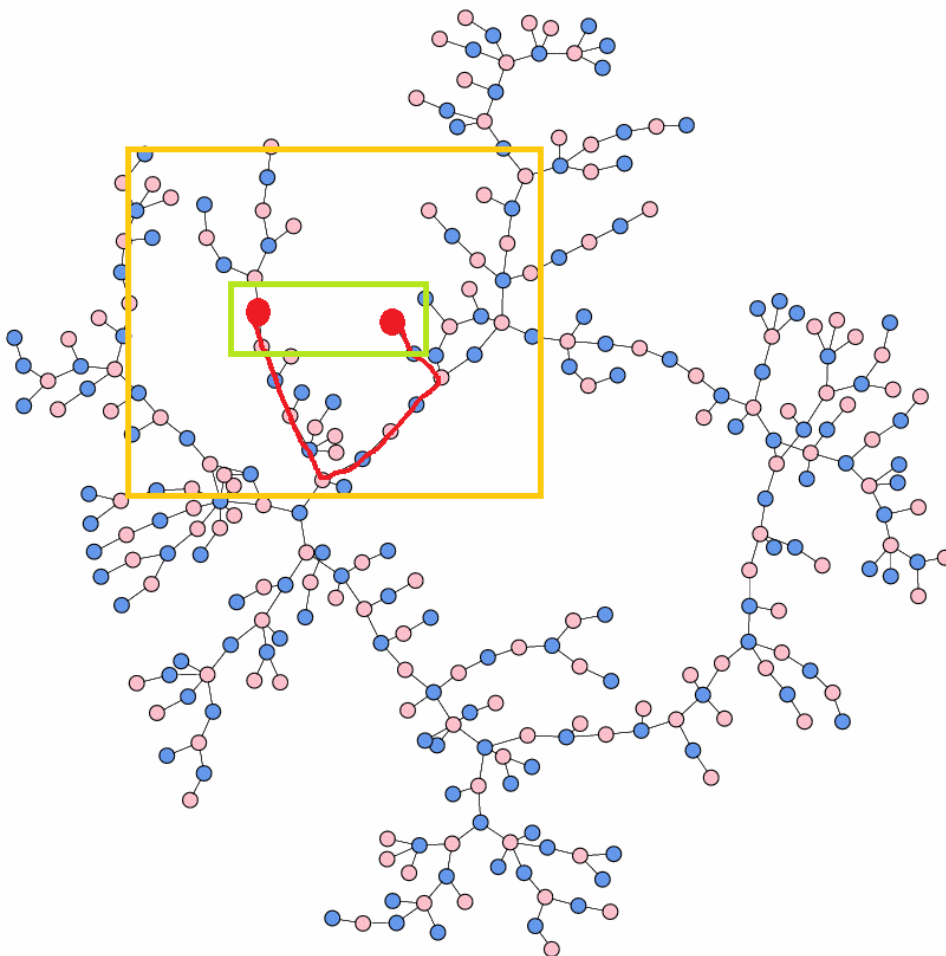
```
CREATE OR REPLACE FUNCTION public.calc_rotas(  
    IN source integer,  
    IN target integer,  
    IN k integer,  
    IN directed boolean)  
RETURNS TABLE(  
    seq integer,  
    path_id integer,  
    path_seq integer,  
    node bigint,  
    edge bigint,  
    cost double precision,  
 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

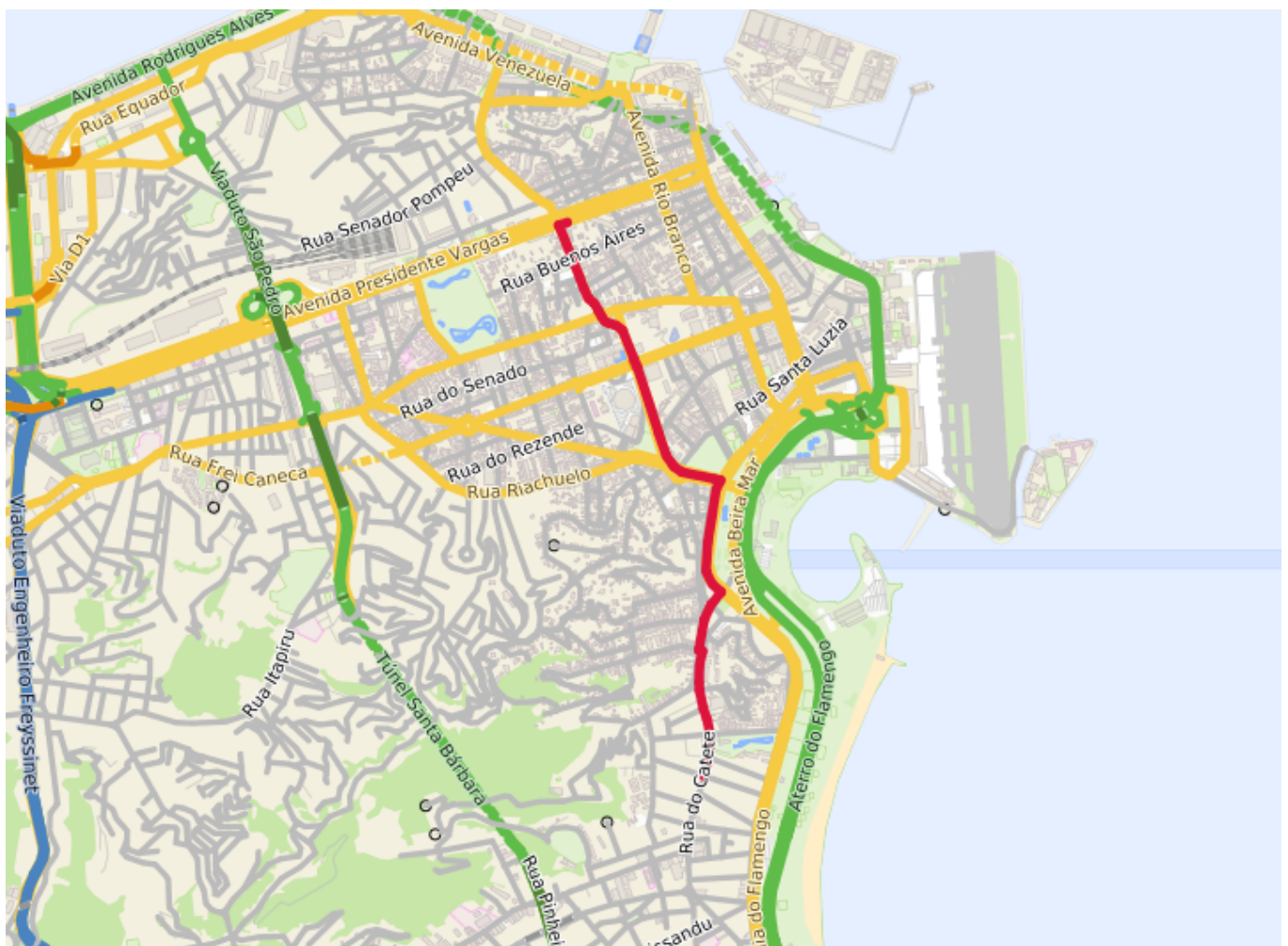
```

        <CssParameter name="stroke-
linecap">round</CssParameter>
    </Stroke>
</LineSymbolizer>
</Rule>

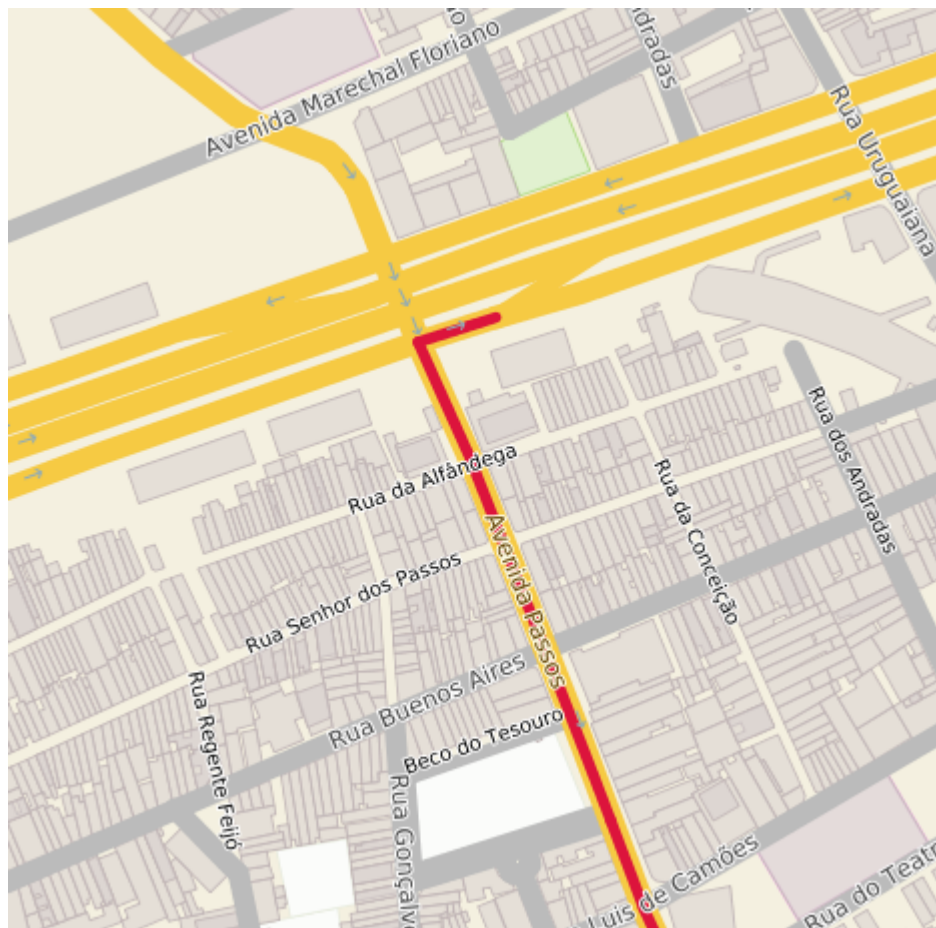
</FeatureTypeStyle>
</UserStyle>
</NamedLayer>
</StyledLayerDescriptor>

```

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

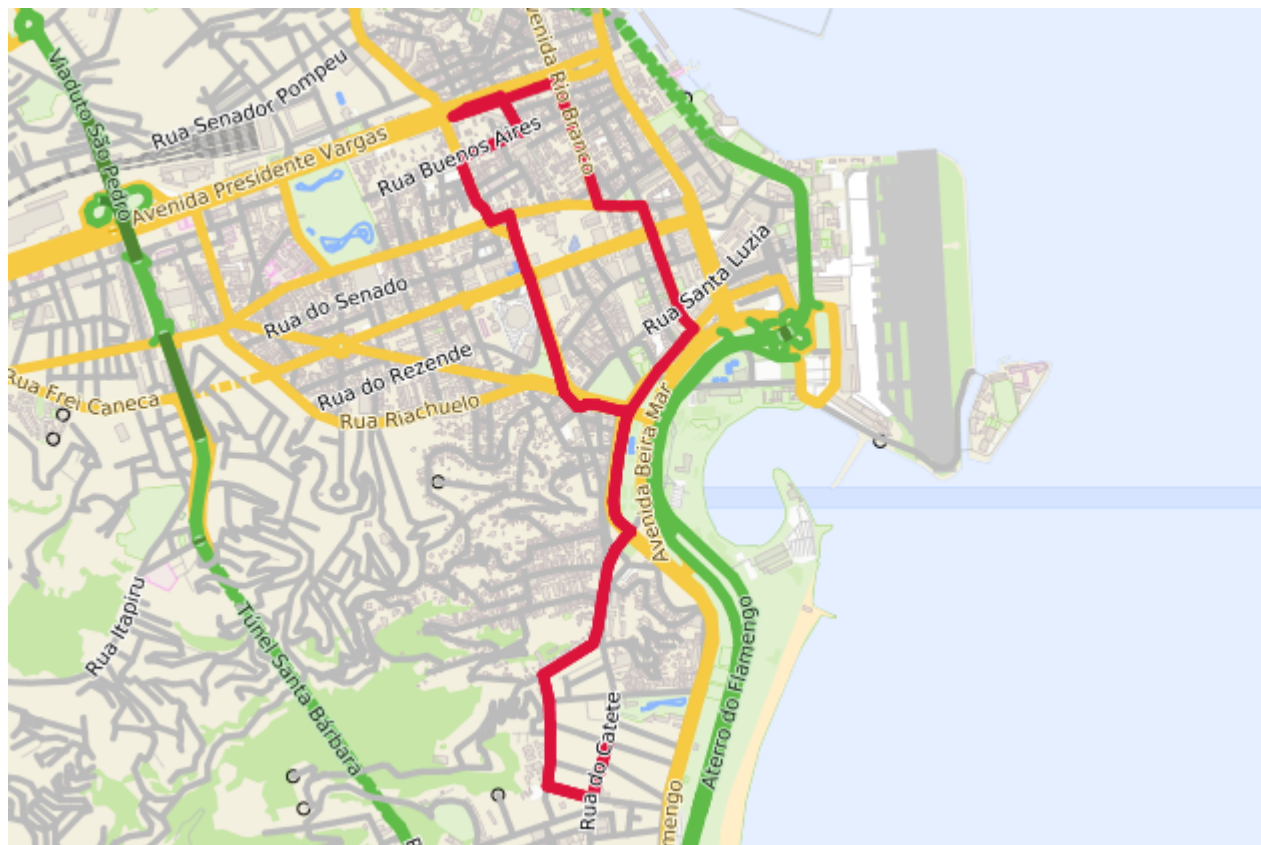
**Nome da View de Dados**

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.

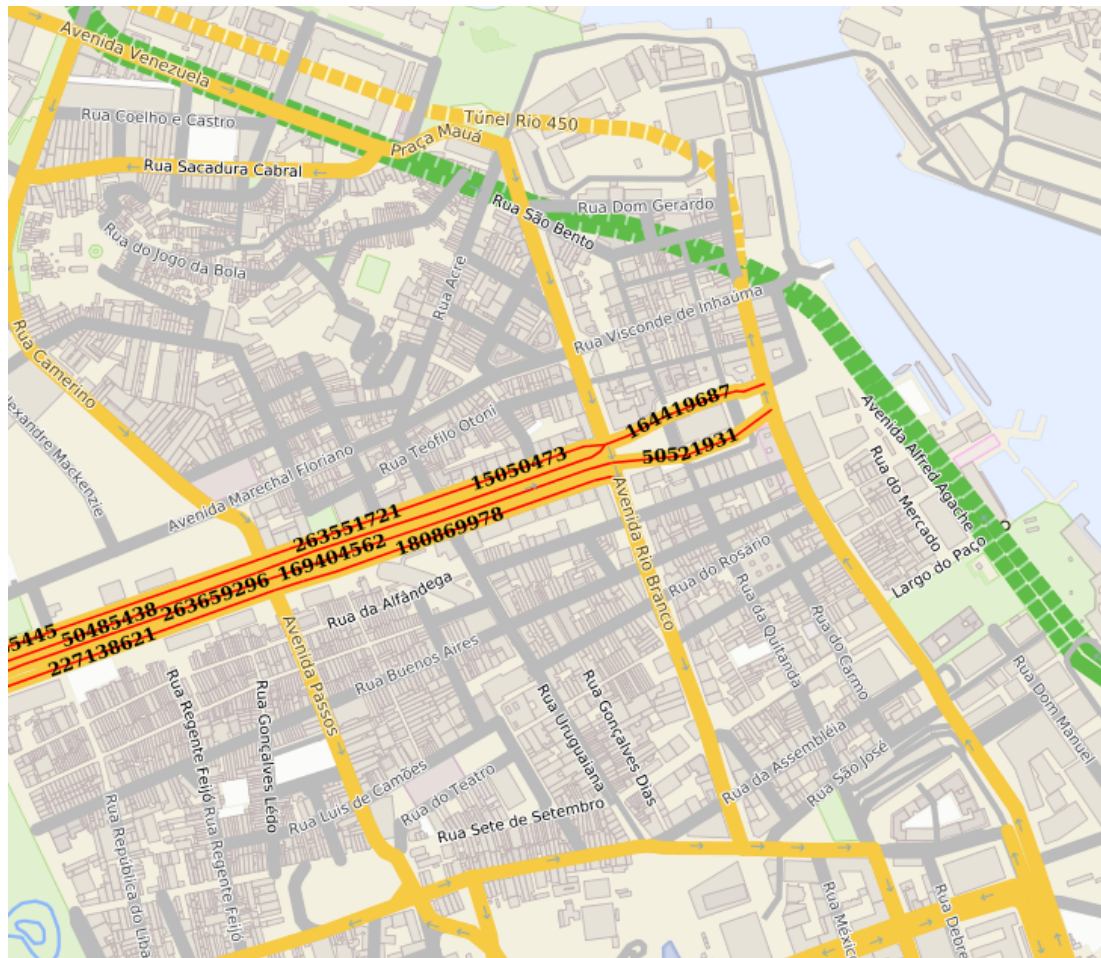
Vamos usar uma avenida famosa no centro do Rio de Janeiro: a Avenida Presidente Vargas. Eis a *view* para encontrá-la:

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

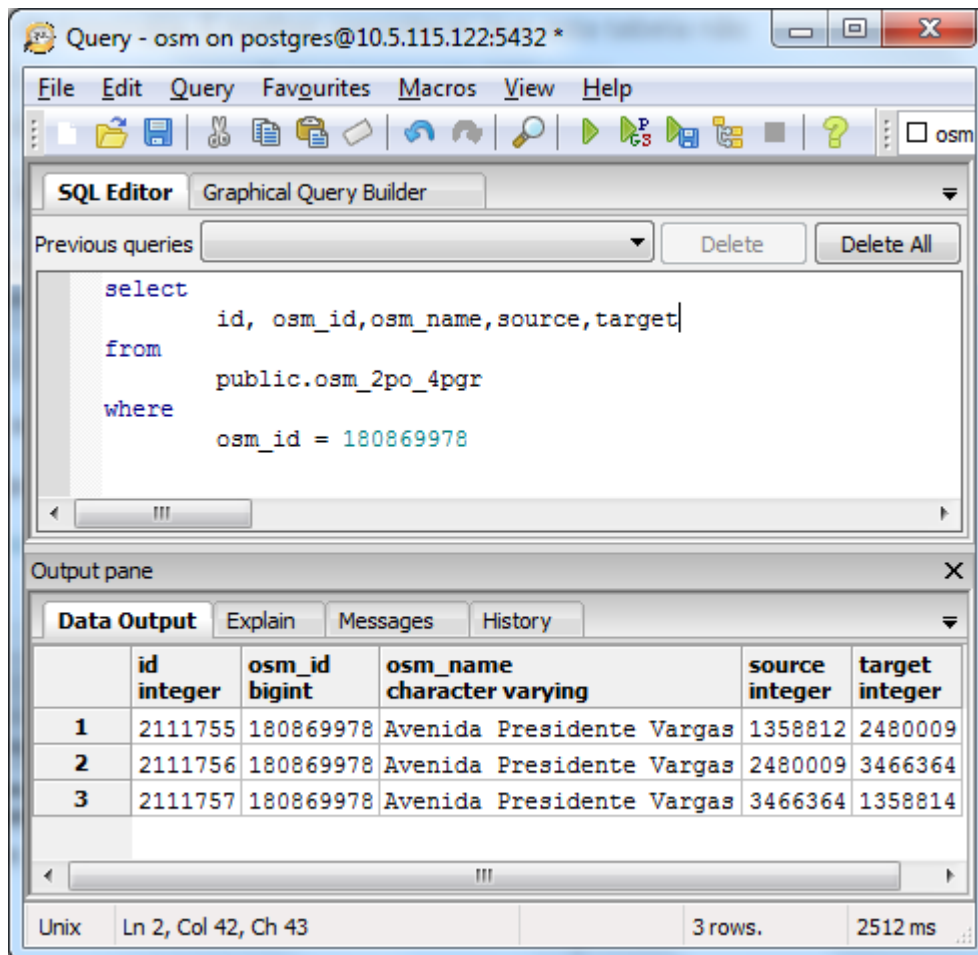
Agora você perceberá a utilidade da visão da rua no mapa: existem várias avenidas com este nome no país. Para pegar somente a do Rio de Janeiro, seria necessário conhecer as coordenadas geográficas do centro da cidade e filtrar a geometria dos dados encontrados pela *view*. Além do mais, você precisará 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:

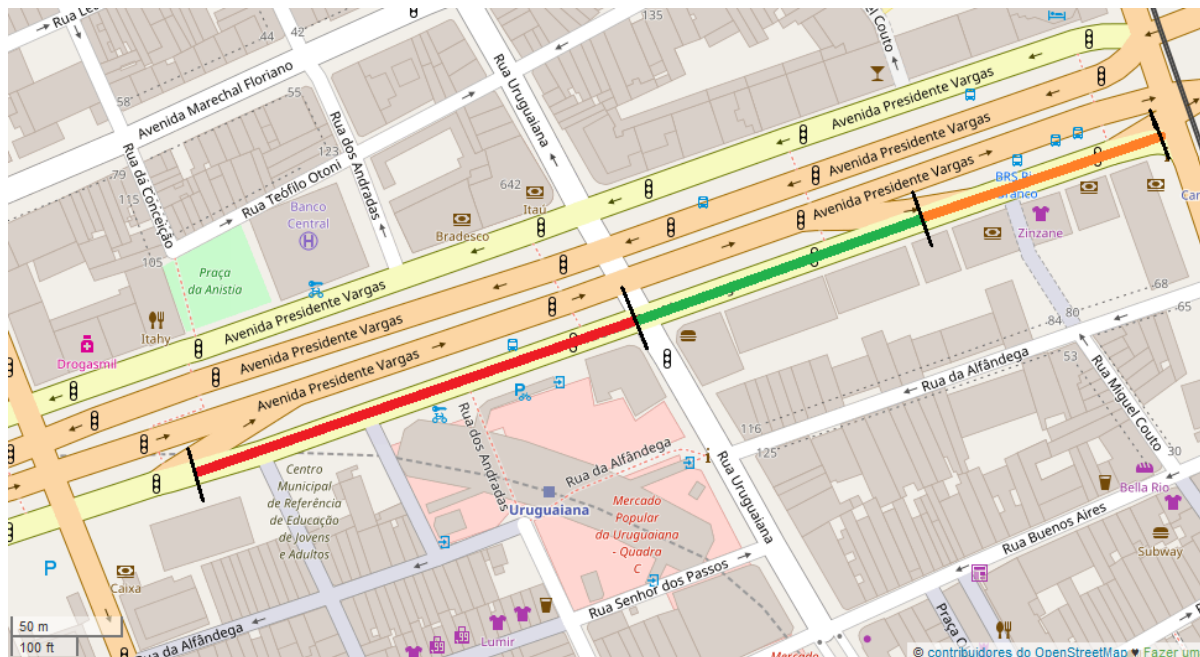




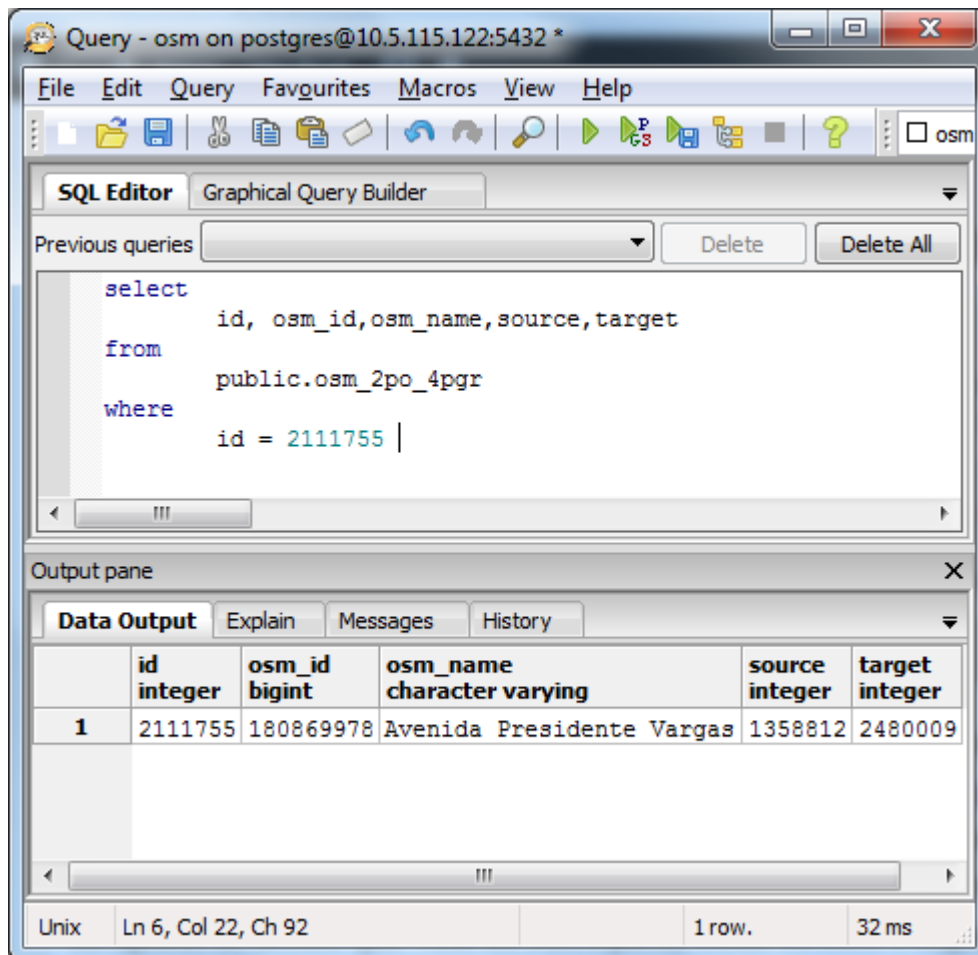
Como da outra vez, escolheremos um segmento qualquer. Vamos usar o segmento 180869978.



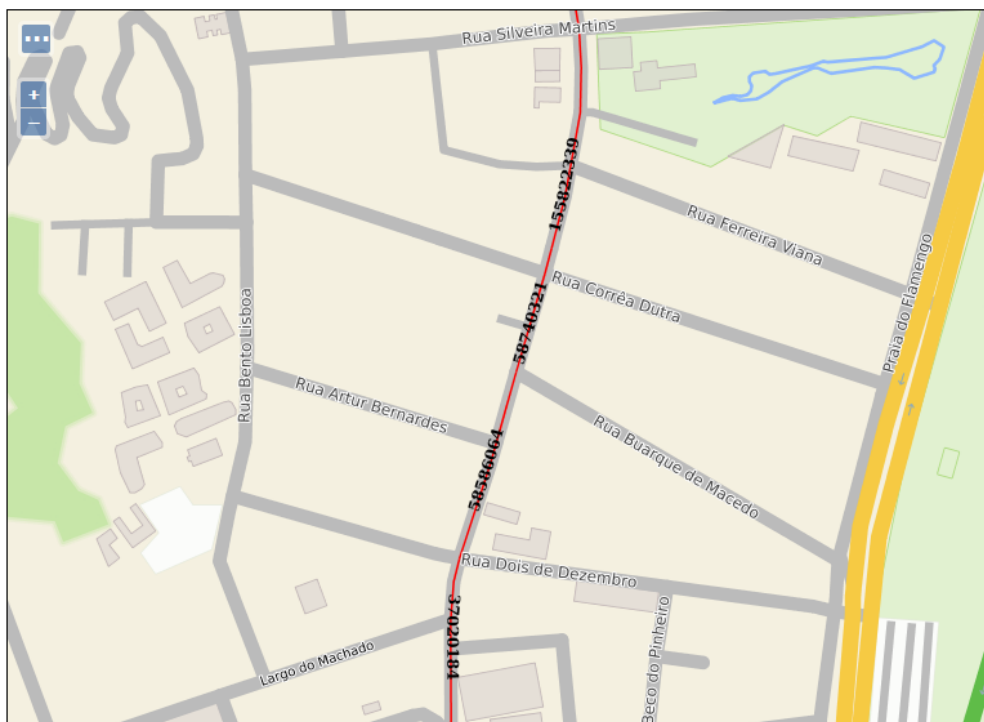
Opa! Encontramos 3 segmentos que apontam para a mesma rua nos dados originais do OSM (*planet\_osm\_line.osm\_id*). Quando criamos a topologia (tabela *osm\_2po\_4pgr*), toda junção da rua com outras ruas é quebrada em um segmento. Sempre que uma rua “entra” em outra (é possível que o tráfego flua para 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.

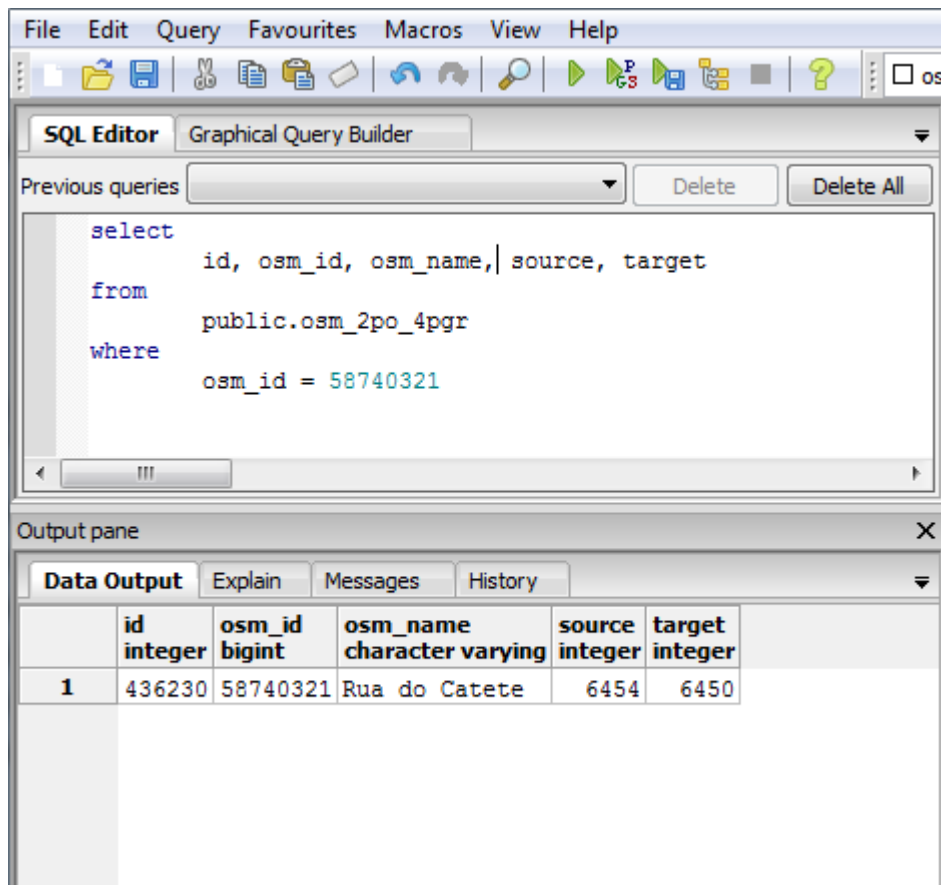


Agora sim acabamos com a ambiguidade. Para o destino, vamos ficar com o nosso segmento 58740321 da Rua do Catete, mostrado no post anterior.





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"><li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li></ul>
<code>reverse_cost</code>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"><li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li></ul>

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.

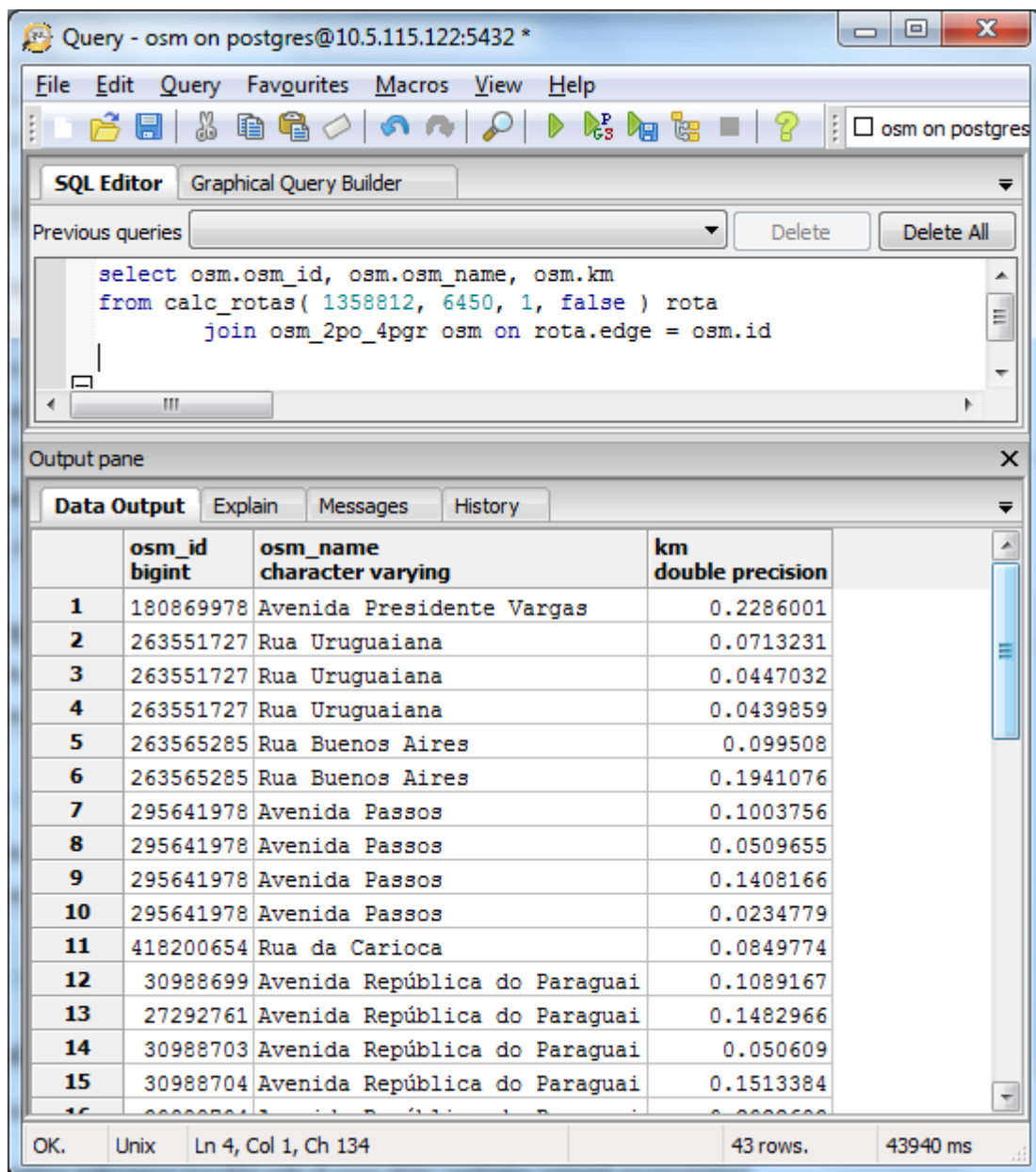
Mas este resultado não me disse muita coisa. Vamos melhorar um pouco mais com uma junção extra. Vou reduzir minhas opções para apenas uma rota para otimizar meu tempo:

```
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 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, with the first 15 rows visible in the screenshot.

	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

The status bar at the bottom indicates "OK.", "Unix", "Ln 4, Col 1, Ch 134", "43 rows.", and "43940 ms".

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/Shortest_path_problem)

[https://en.wikipedia.org/wiki/K\\_shortest\\_path\\_routing](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/src/ksp/doc/pgr_ksp.html)

[http://docs.pgrouting.org/2.3/en/doc/src/developer/sampled\\_data.html#sample\\_data](http://docs.pgrouting.org/2.3/en/doc/src/developer/sampled_data.html#sample_data)

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

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

---

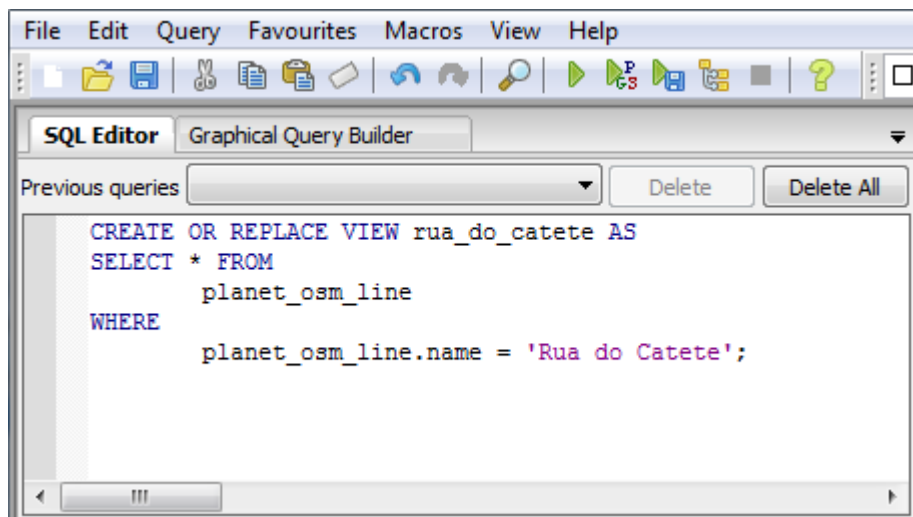
## **Trabalhando com rotas nos dados do OpenStreetMap: Parte 2**

No [artigo anterior](#), criamos a tabela de topologia e aprendemos alguma coisa sobre ela e um pouco sobre como as rotas são armazenadas. Neste artigo vou me aprofundar um pouco mais nestes conceitos antes de prosseguir com a prática.

Vou usar o GeoServer para criar a visualização de certos dados da tabela

*osm\_2po\_4pgr*. Infelizmente não é o escopo deste artigo ensinar a usar o GeoServer. Por sorte, tudo que vou exibir neste artigo é ilustrativo e servirá apenas para você acompanhar minhas explicações. Não há necessidade de executar nada por enquanto.

Vou começar criando uma *view* no nosso banco de dados *osm* somente para exibir a Rua do Catete (no Rio de Janeiro). Esta *view* não tem finalidade prática no cálculo de rotas e só servirá como ilustração dos conceitos mostrados no artigo:



Agora, é necessário criar uma SQL View no GeoServer. Vá em “Camadas”, selecione “Adicionar Novo Recurso”, escolha o *workspace* do nosso banco de dados OSM e clique em “Configure New SQL View”.

nd types. [Create new feature type...](#)  
statement. [Configure new SQL view...](#)  
a capa que desea configurar



Pesquisa
Acción
Publicar novamente
Publicar novamente
Publicar novamente

Na instrução SQL, preencha com

```
select * from public.rua_do_catete
```

clique em “Atualizar” e modifique os valores do atributo “way” conforme figura abaixo (tipo “LineString” e SRID “900913”:

way_area	Float
abandoned:aeroway	String
abandoned:amenity	String
abandoned:building	String
abandoned:landuse	String
abandoned:power	String
area:highway	String

way	<input type="text" value="LineString"/>	<input type="text" value="900913"/>
-----	---	-------------------------------------

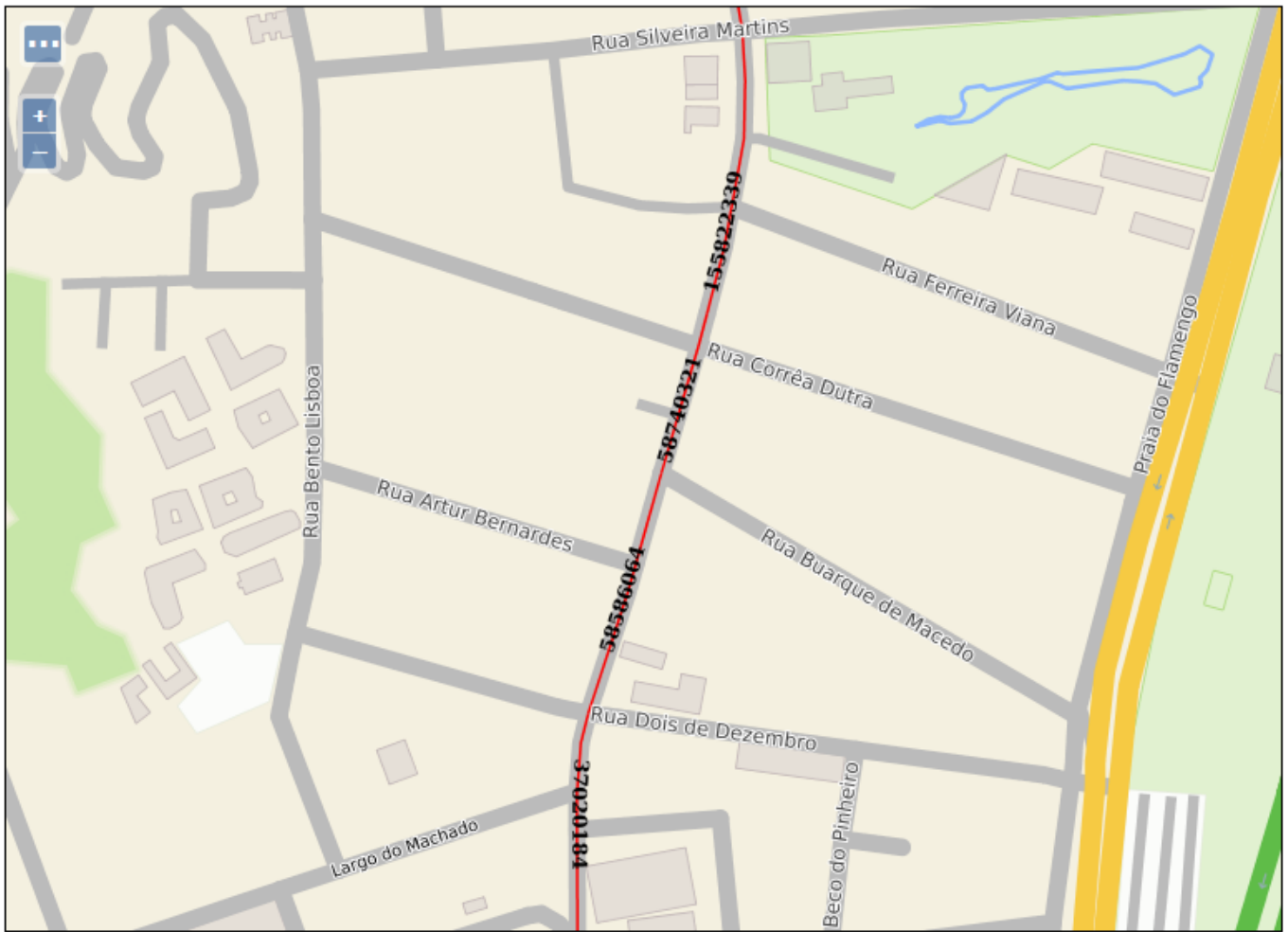
Atualize o “*Bounding Box*” (Retângulos Envolventes) e salve. Eu criei um estilo para esta camada somente para que eu possa acompanhar os identificadores dos segmentos da rua.

```
<NamedLayer>
  <Name>catete_line</Name>
  <UserStyle>
    <Title>Estilo para a Rua do Catete</Title>
  <FeatureTypeStyle>
    <Rule>
      <LineSymbolizer>
        <Stroke>
          <CssParameter name="stroke">#FF0000</CssParameter>
        </Stroke>
      </LineSymbolizer>
      <TextSymbolizer>
        <Label>
          <ogc:PropertyName>osm_id</ogc:PropertyName>
        </Label>
        <LabelPlacement>
          <LinePlacement />
        </LabelPlacement>
      </TextSymbolizer>
    </Rule>
  </FeatureTypeStyle>
</NamedLayer>
```



```
    <CssParameter name="fill">#000000</CssParameter>
  </Fill>
  <Font>
    <CssParameter name="font-family">Arial</CssParameter>
    <CssParameter name="font-size">12</CssParameter>
    <CssParameter name="font-style">normal</CssParameter>
    <CssParameter name="font-weight">bold</CssParameter>
  </Font>
  <VendorOption name="followLine">true</VendorOption>
  <VendorOption name="maxAngleDelta">90</VendorOption>
  <VendorOption name="maxDisplacement">400</VendorOption>
  <VendorOption name="repeat">150</VendorOption>
</TextSymbolizer>
</Rule>
</FeatureTypeStyle>
</UserStyle>
</NamedLayer>
```

O próximo passo foi criar um grupo de camadas com a camada da Rua do Catete e o grupo de camadas do OSM (grupo que eu criei com todas as camadas do OSM). O resultado disso tudo você vê na figura abaixo:



É claro que você não precisaria de nada disso. Bastava selecionar os registros na tabela com o SQL usado pela *view* e pronto, mas a princípio não daria para saber a sequência correta dos segmentos e quem conecta com quem, pois esta informação está na geometria (e na tabela de topologia, mas para achar alguma coisa lá precisamos saber o que estamos procurando). Já temos os segmentos da Rua do Catete com seus respectivos identificadores. Agora podemos procurar um segmento da rua na tabela de topologia. É melhor considerar que esta tabela não entende de ruas, mas sim de segmentos de ruas. Vou escolher o segmento 58740321 (segmento da Rua do Catete entre a Corrêa Dutra e a Buarque de Macedo):

The screenshot shows a PostgreSQL SQL Editor window. The menu bar includes File, Edit, Query, Favourites, Macros, View, and Help. The toolbar contains icons for file operations, query execution, and help. The SQL Editor tab is active, showing the following query:

```
select
    id, osm_id, osm_name, source, target
from
    public.osm_2po_4pgr
where
    osm_id = 58740321
```

Below the query editor is the Output pane, which is currently displaying the Data Output tab. The results are shown in a table with the following columns: id, osm\_id, osm\_name, source, and target.

	id integer	osm_id bigint	osm_name character varying	source integer	target integer
1	436230	58740321	Rua do Catete	6454	6450

Agora que achamos os dados do segmento, podemos perguntar quem chega até ele e para onde ele vai. Observe o seu valor *source* 6454. Vamos perguntar: “quem tem como destino o segmento cujo *source* é 6454?” ou, “dado o segmento com origem 6454, quem chega até ele?”

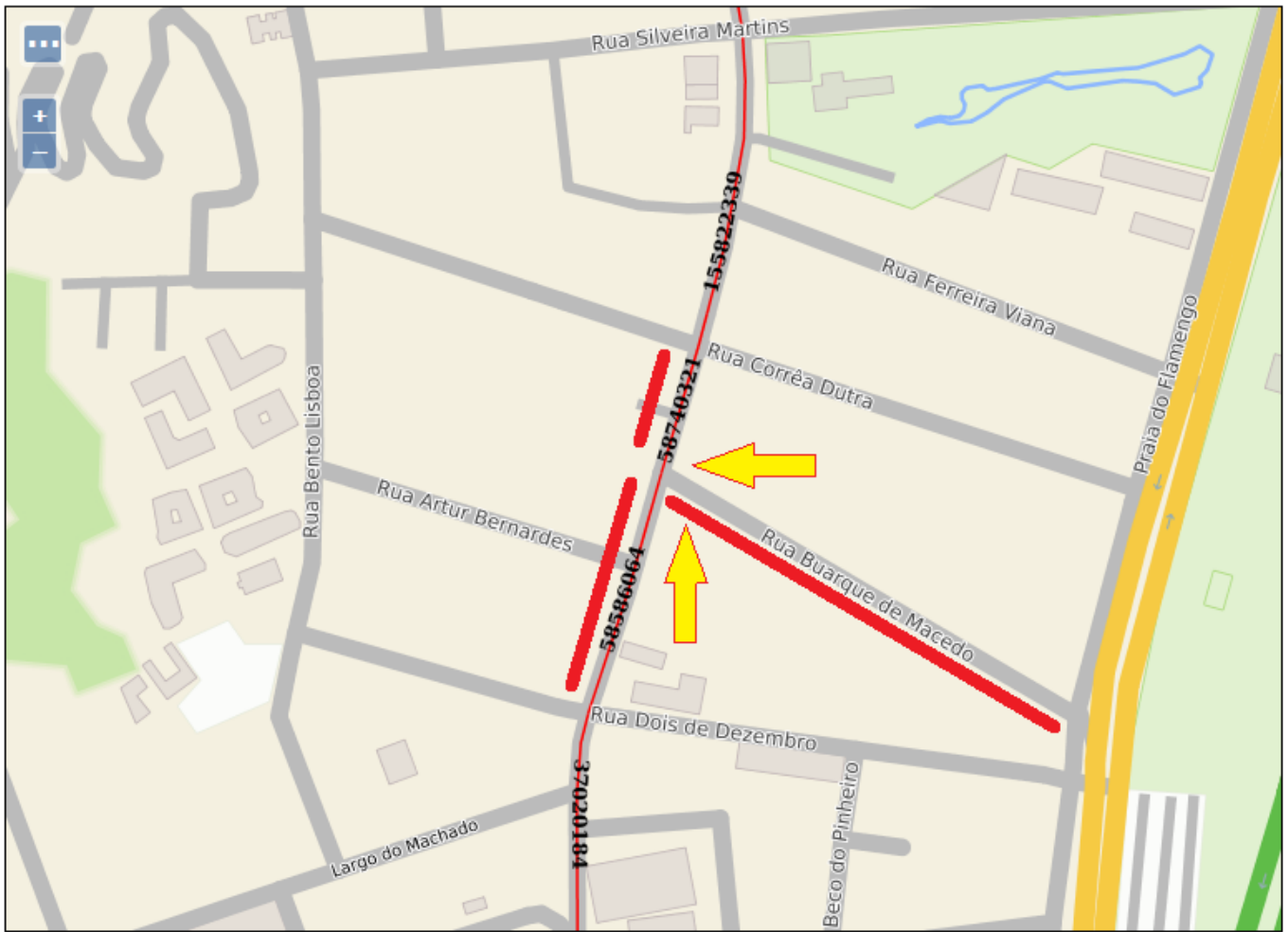
The screenshot shows a PostgreSQL SQL Editor window. The menu bar includes File, Edit, Query, Favourites, Macros, View, and Help. The toolbar contains icons for file operations, query execution, and help. The SQL Editor tab is active, showing a query in the SQL Editor pane. The query is:

```
select
  id, osm_id, osm_name, source, target
from
  public.osm_2po_4pgr
where
  target = 6454
```

Below the SQL Editor is the Output pane, which is currently showing the Data Output tab. The results are displayed in a table with 6 columns: id, osm\_id, osm\_name, source, and target. The table contains two rows of data.

	id integer	osm_id bigint	osm_name character varying	source integer	target integer
1	4915	5136172	Rua Buarque de Macedo	6453	6454
2	435938	58586064	Rua do Catete	6533	6454

Encontramos duas ruas cujo *target* é a rua com source 6454: o segmento anterior da Rua do Catete (não por coincidência o segmento 58586064), e a Rua Buarque de Macedo.



Ótimo! Descobrimos de onde viemos, mas para onde vamos? Precisamos procurar as ruas com *source* igual ao *target* (6450) do nosso segmento inicial (58740321):

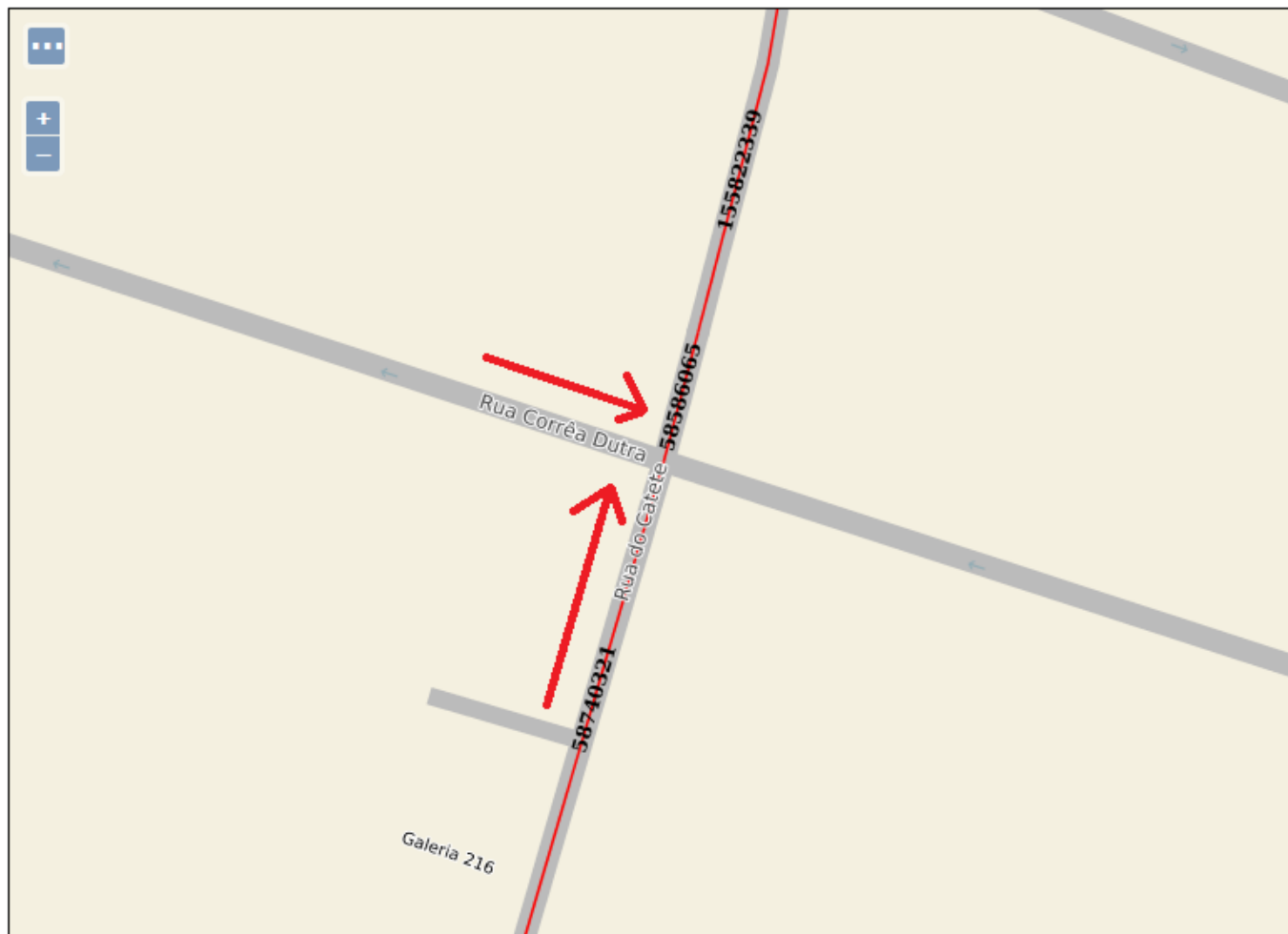
The screenshot shows a SQL Editor window with a menu bar (File, Edit, Query, Favourites, Macros, View, Help) and a toolbar. The 'SQL Editor' tab is active, displaying a query in the 'Previous queries' list. The query is:

```
select
  id, osm_id, osm_name, source, target
from
  public.osm_2po_4pgr
where
  source = 6450
```

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 6 columns: id, osm\_id, osm\_name, source, and target. The table contains two rows of data:

	id integer	osm_id bigint	osm_name character varying	source integer	target integer
1	433896	56178677	Rua Corrêa Dutra	6450	267805
2	435939	58586065	Rua do Catete	6450	268919

Encontramos a Rua Corrêa Dutra e outro segmento da Rua do Catete (58586065), mas este segmento não está no mapa! Depois do segmento 58740321 vem o 155822339. Bom, isso é um problema de zoom. O GeoServer “sabe” que precisa omitir algumas coisas para manter a ordem na tela. Então, vamos aproximar um pouco o zoom para ver com mais clareza...



Aí está o segmento perdido bem onde ele deveria estar. Nossa rota então ficou assim: Você pode vir da Rua Buarque de Macedo (segmento 5136172) ou da Rua do Catete (segmento 58586064) , pegar a Rua do Catete (segmento 58740321) e seguir pela Rua Corrêa Dutra (segmento 56178677) ou continuar na Rua do Catete (segmento 58586065) .

Bom, aprendemos a fazer uma rota “na mão”. Na prática, quando você olha para um mapa no Google ou no OpenStreetMap, você não percebe os segmentos de ruas, mas saiba que eles são peças fundamentais no cálculo de rotas. O atributo *osm\_id* serve para vincular as topologias de volta para a tabela de dados do OSM *planet\_osm\_line*, com mais informações sobre o segmento.

Tente agora continuar pela Rua Corrêa Dutra, Praia do Flamengo, Ferreira Viana, retornar pela Rua do Catete, Artur Bernardes e Bento Lisboa.

No próximo artigo, veremos como obedecer a “mão” da rua e começar a preparar o terreno para nosso calculador de rotas.

---

# 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:

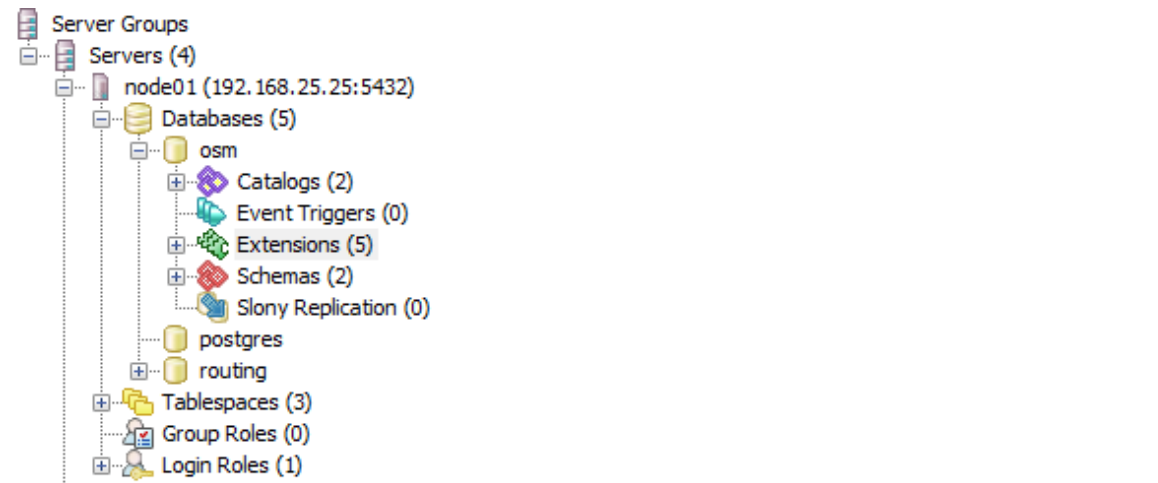
```
http://localhost:8888/Osm2poService
```

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:



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.

Vou terminar este post por aqui. Já criamos a tabela de topologia e vimos alguns conceitos sobre ela. [No próximo post](#) vou continuar explicando os conceitos de rotas para depois partir para a prática.