

# 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.

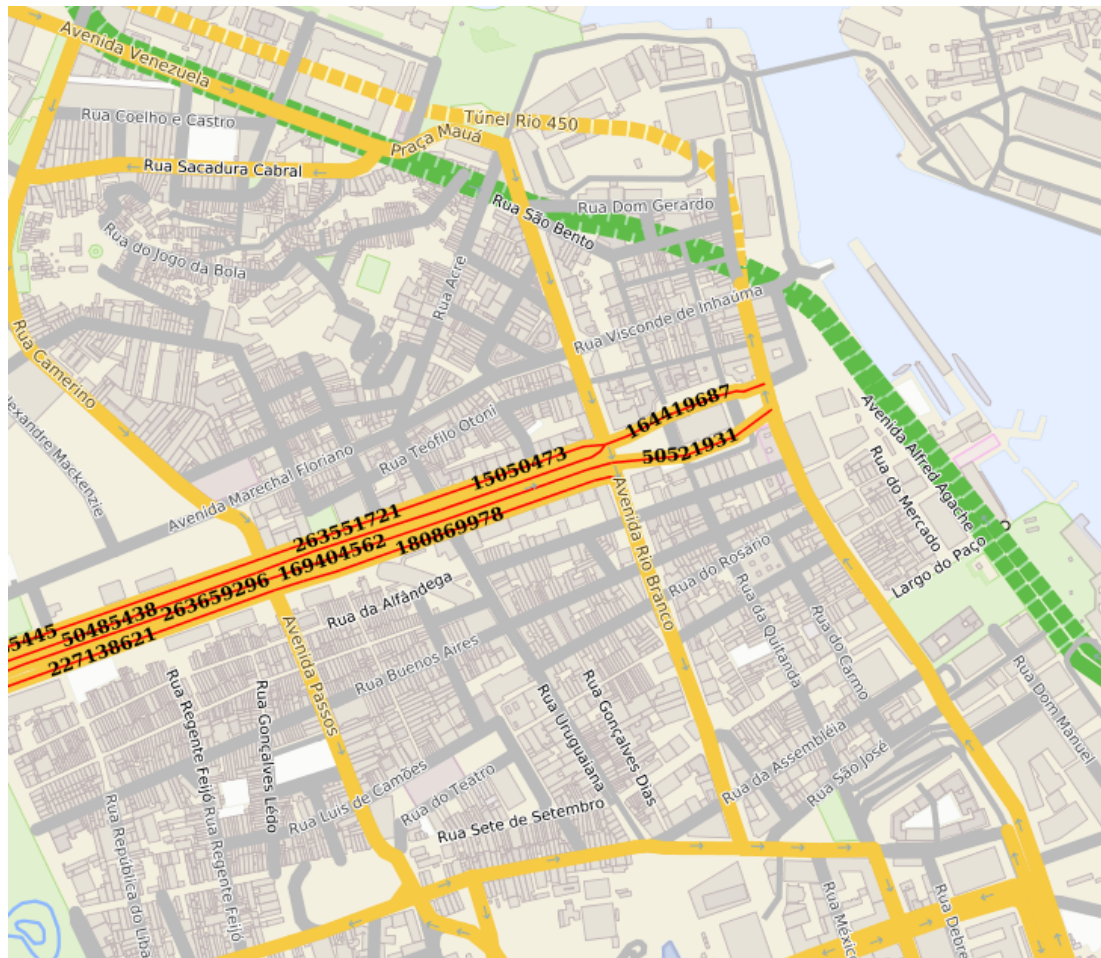
Eu havia mostrado [como criar uma view no banco de dados](#) para ter uma visão gráfica da Rua do Catete (RJ) no GeoServer. Tentar encontrar um caminho numa linha reta seria fácil, então vamos precisar encontrar outra rua um pouco mais longe para servir como o outro ponto da nossa rota. Ter uma visão 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:

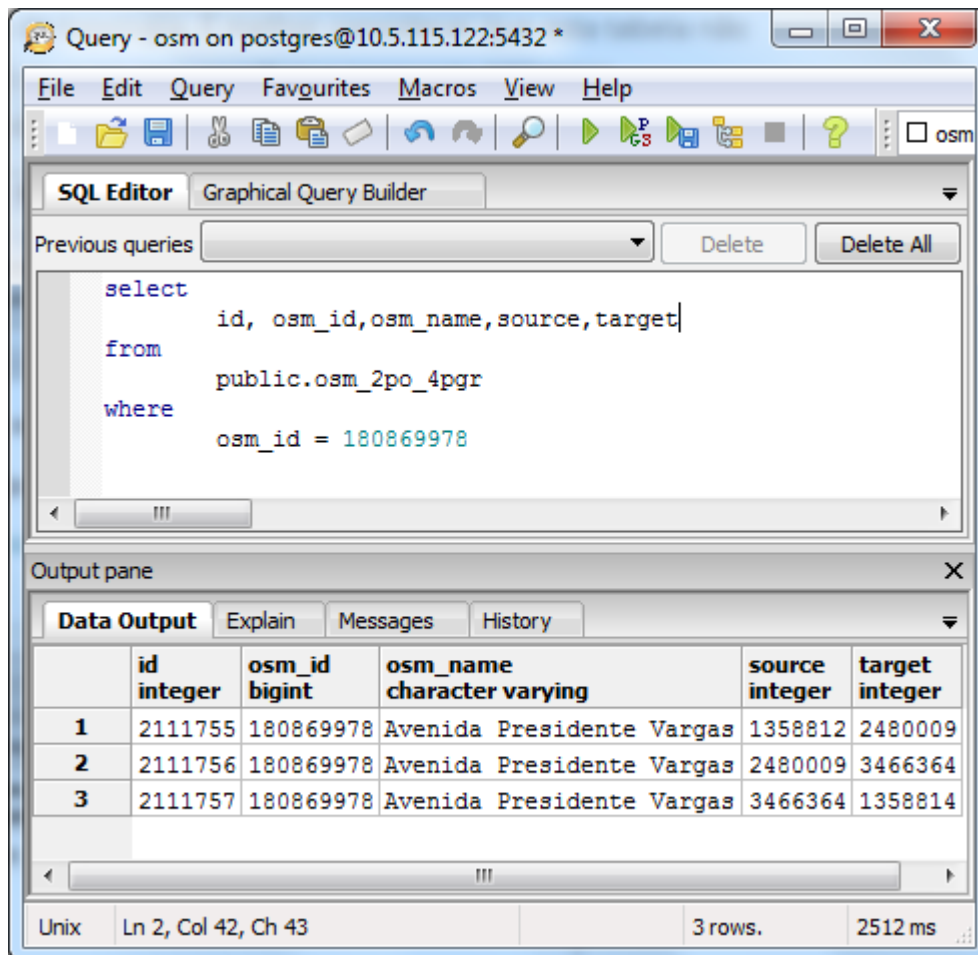
```
create or replace view av_pres_vargas as
select
    *
FROM
    planet_osm_line
WHERE
    planet_osm_line.name = 'Avenida Presidente Vargas'
```

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.

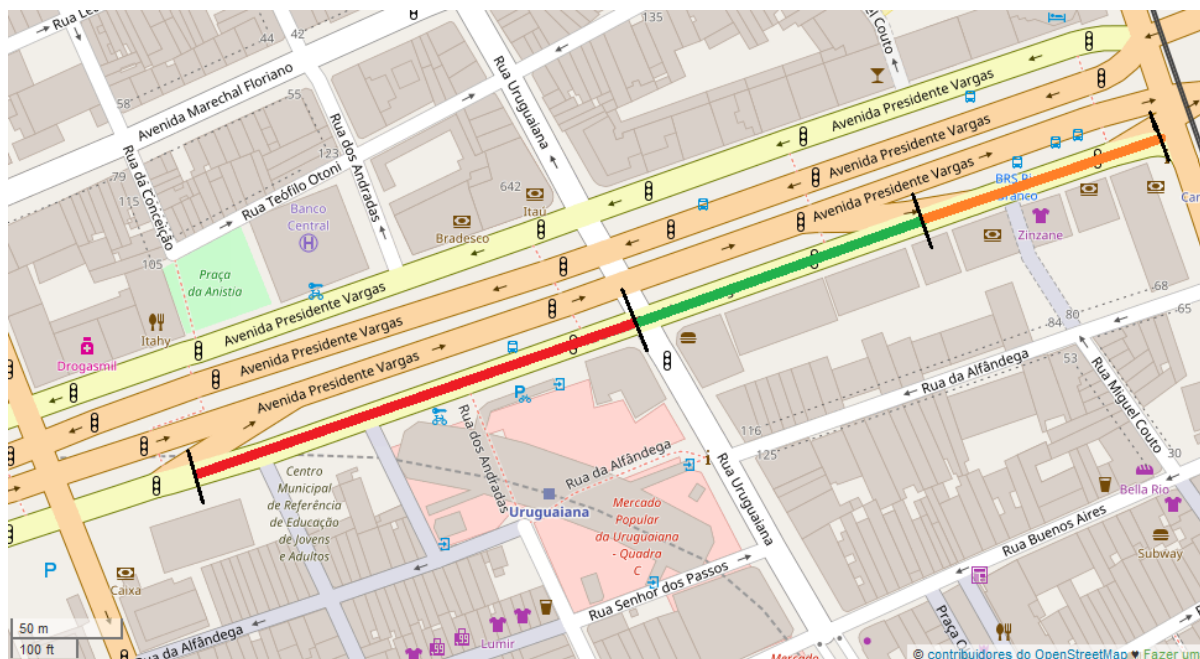
Após criar a *view* no banco de dados, usaremos exatamente o [mesmo processo do post anterior](#) para criar a camada do mapa no Geoserver. Eis o resultado:



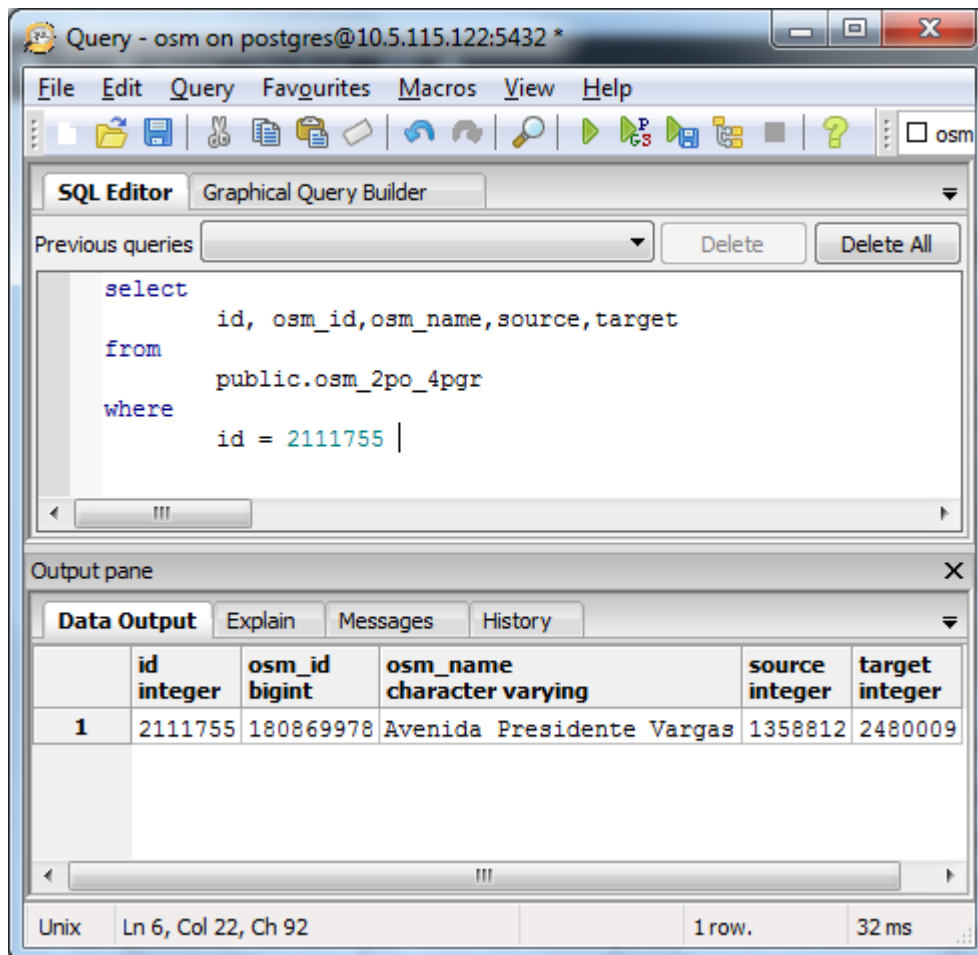
Como da outra vez, escolheremos um segmento qualquer. Vamos usar o segmento 180869978.



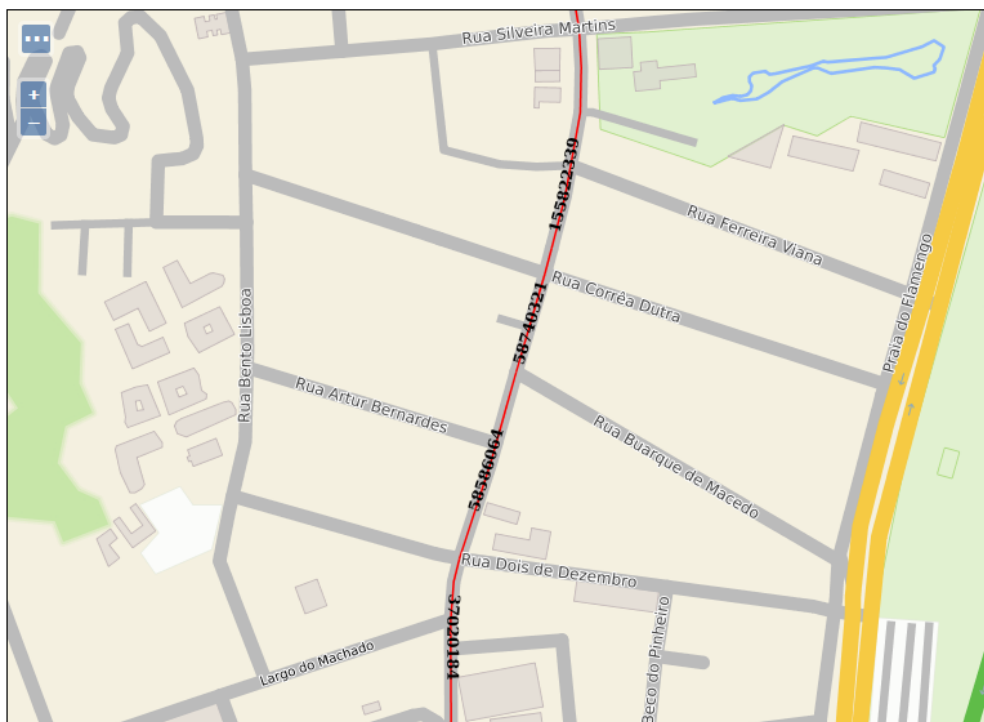
Opa! Encontramos 3 segmentos que apontam para a mesma rua nos dados originais do OSM (*planet\_osm\_line.osm\_id*). Quando criamos a topologia (tabela *osm\_2po\_4pgr*), toda junção da rua com outras ruas é quebrada em um segmento. Sempre que uma rua “entra” em outra (é possível que o tráfego flua para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



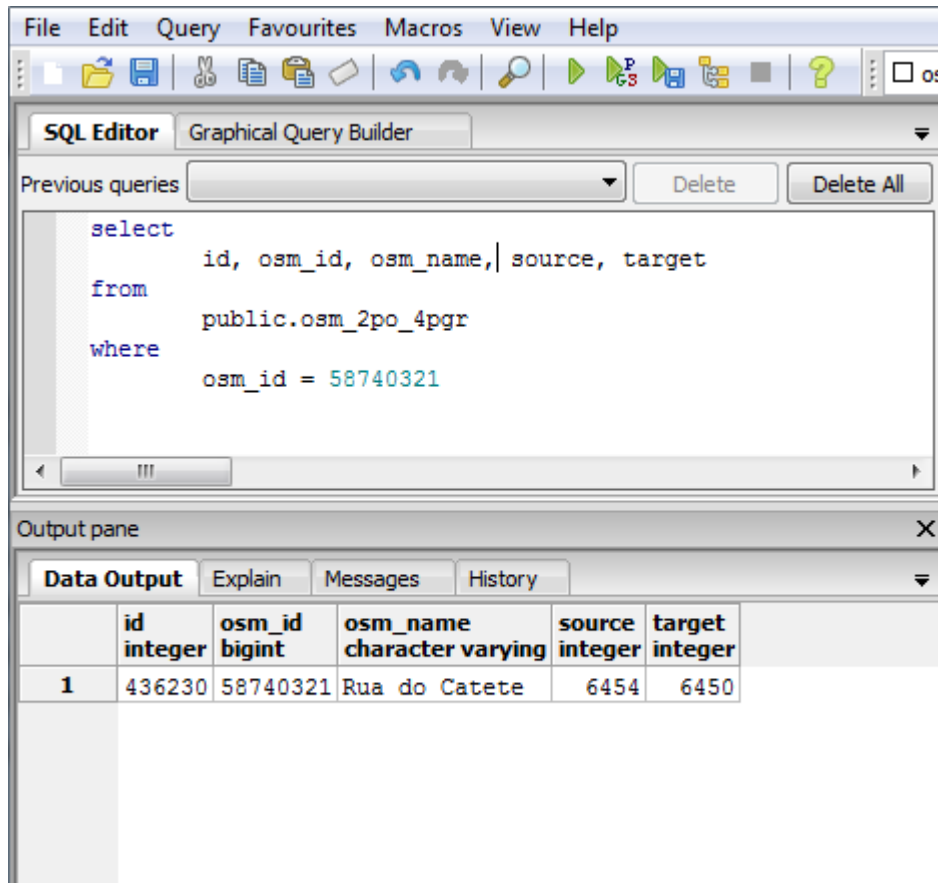
Repare que ele começa em uma saída de acesso para a pista lateral e termina na esquina com a Av. Rio Branco (no canto superior direito do mapa). Isso é [como o OSM percebe este segmento](#). Para efeito de rotas, é possível sair desta via e entrar na Rua Uruguaiana ou chegar pela Rua Uruguaiana e seguir 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>	<code>TEXT</code>	SQL query as described above.
<code>start_vid</code>	<code>BIGINT</code>	Identifier of the starting vertex.
<code>end_vid</code>	<code>BIGINT</code>	Identifier of the ending vertex.
<code>k</code>	<code>INTEGER</code>	The desired number of paths.
<code>directed</code>	<code>BOOLEAN</code>	(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>	<code>BOOLEAN</code>	(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>	<code>ANY-INTEGER</code>		Identifier of the edge.
<code>source</code>	<code>ANY-INTEGER</code>		Identifier of the first end point vertex of the edge.
<code>target</code>	<code>ANY-INTEGER</code>		Identifier of the second end point vertex of the edge.
<code>cost</code>	<code>ANY-NUMERICAL</code>		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>	<code>ANY-NUMERICAL</code>	-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>	<code>INTEGER</code>	Sequential value starting from 1.
<code>path_seq</code>	<code>INTEGER</code>	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>	<code>BIGINT</code>	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>	<code>BIGINT</code>	Identifier of the node in the path.
<code>edge</code>	<code>BIGINT</code>	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>	<code>FLOAT</code>	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<code>agg_cost</code>	<code>FLOAT</code>	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.

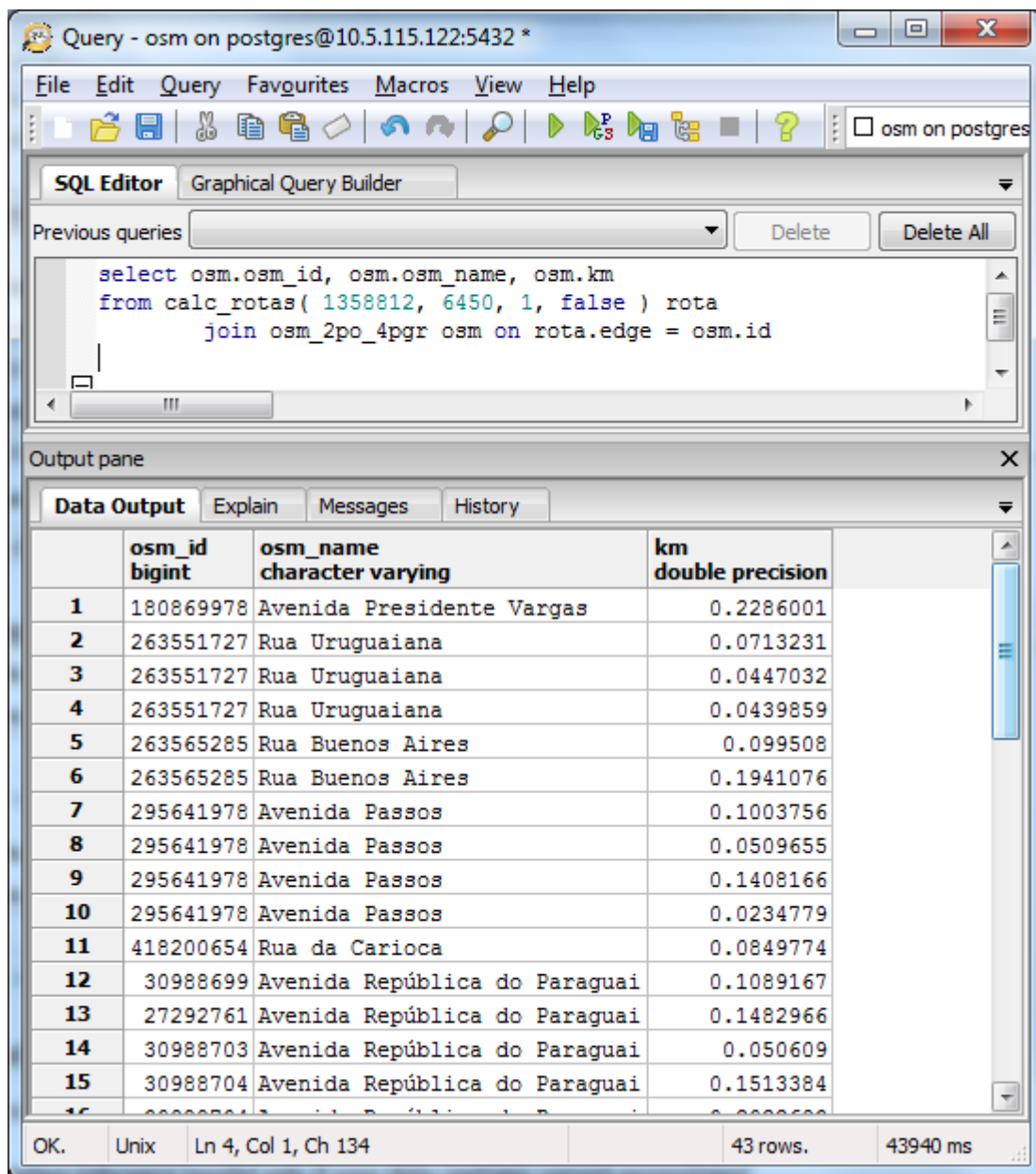
Mas este resultado não me disse muita coisa. Vamos melhorar um pouco mais com uma junção extra. Vou reduzir minhas opções para apenas uma rota para otimizar meu tempo:

```
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 also visible, showing the results of the query in a table format. The table has four columns: **osm\_id** (bigint), **osm\_name** (character varying), and **km** (double precision). The results are as follows:

	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" and "43 rows. 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/sampleddata.html#sampledata>

<http://pgrouting.org/docs/howto/oneway.html>

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