

# 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:

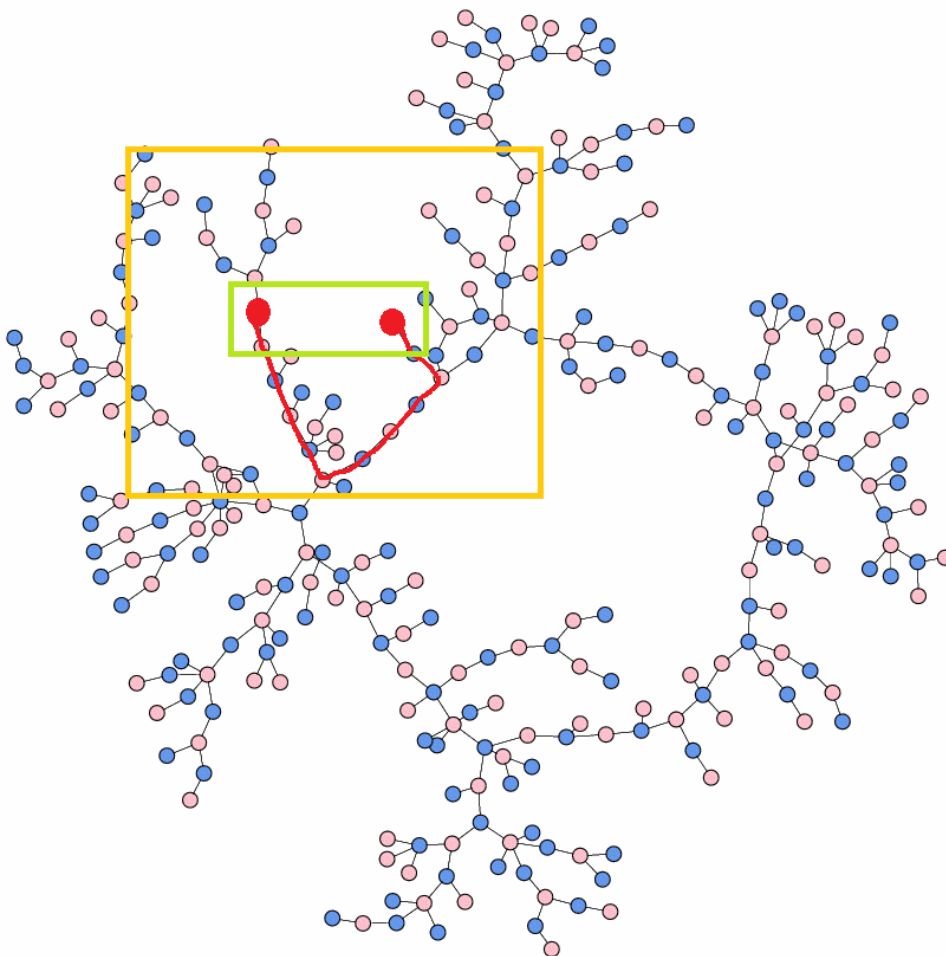
```
CREATE OR REPLACE FUNCTION public.calc_rotas(  
    IN source integer,  
    IN target integer,  
    IN k integer,  
    IN directed boolean)  
RETURNS TABLE(  
    seq integer,  
    path_id integer,  
    path_seq integer,  
    node bigint,  
    edge bigint,  
    cost double precision,  
 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

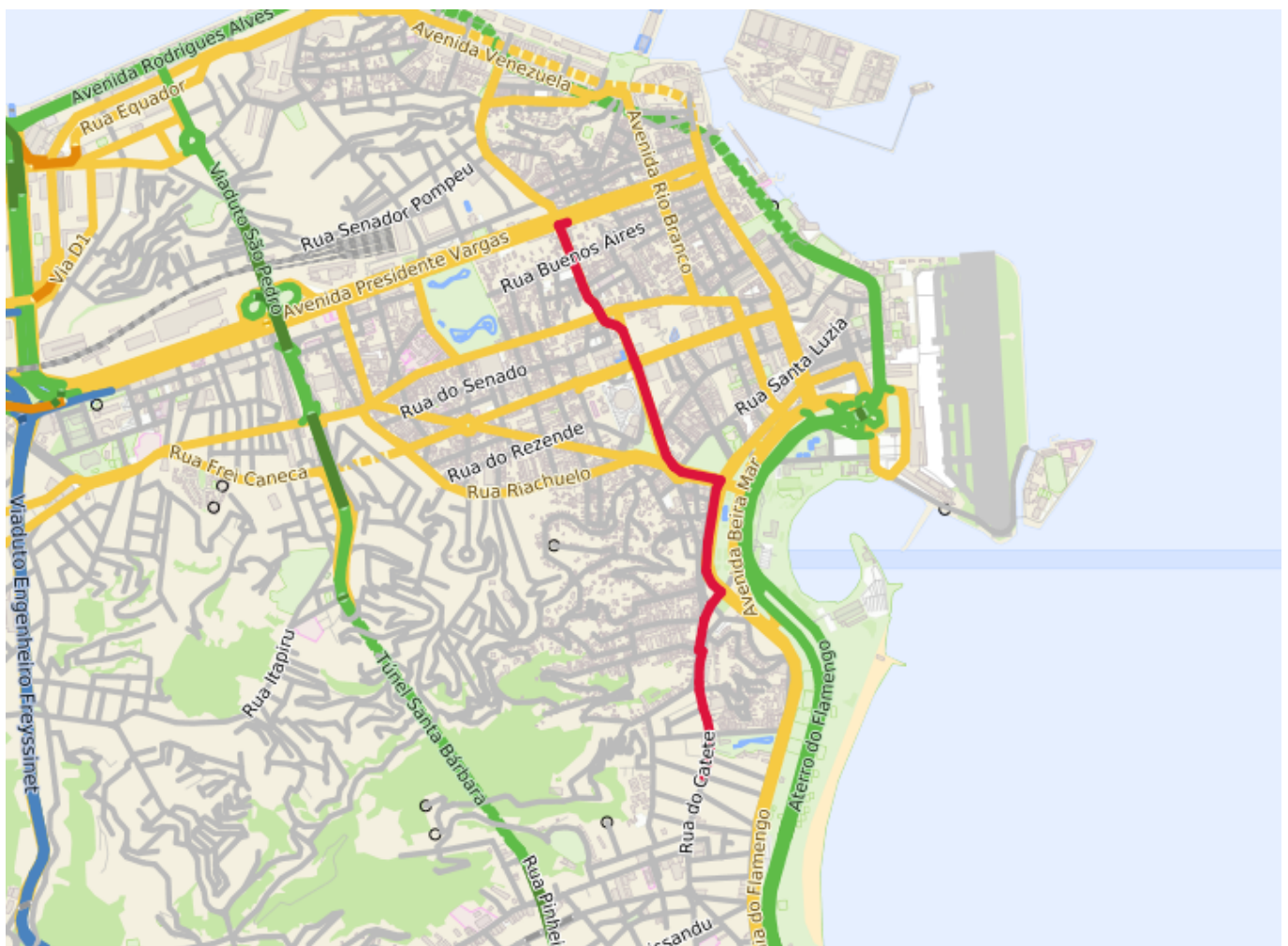
```

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

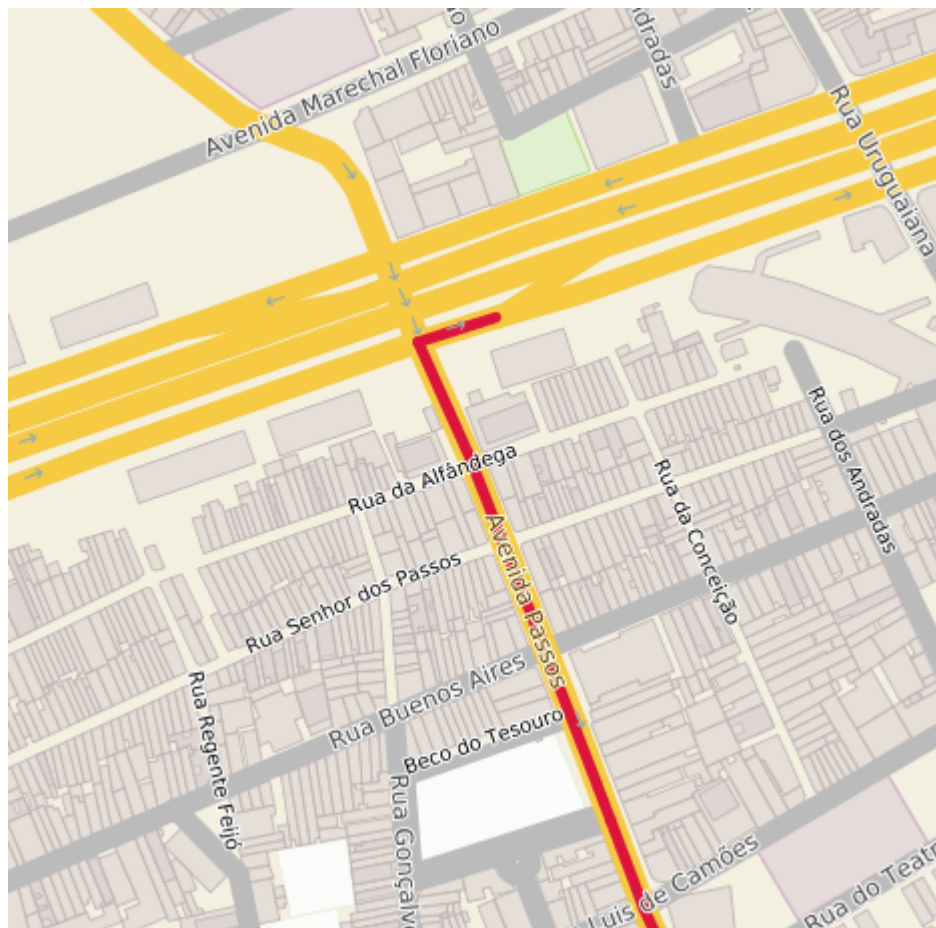
</FeatureTypeStyle>
</UserStyle>
</NamedLayer>
</StyledLayerDescriptor>

```

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

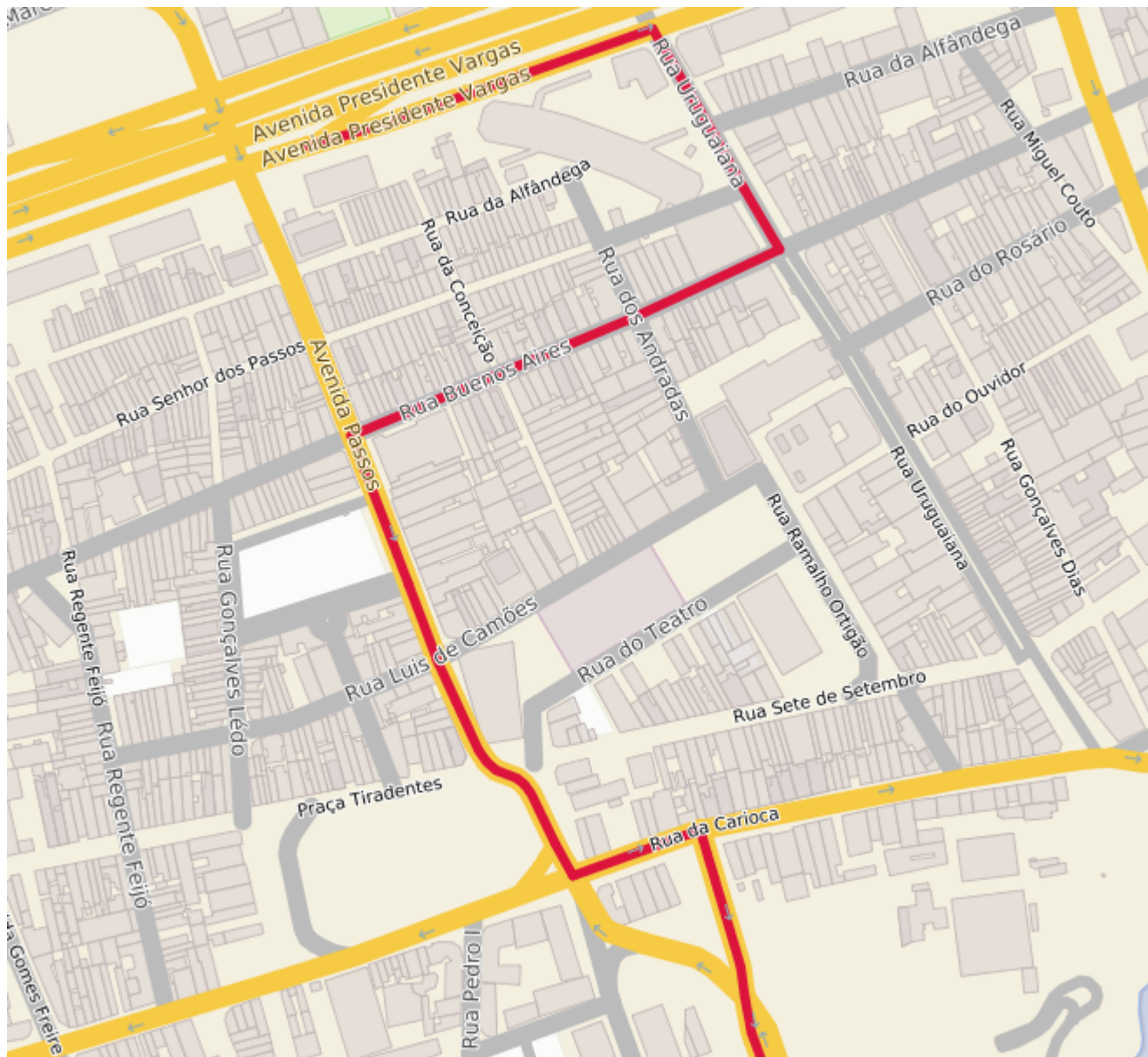
Actualizar la definición de la vista SQL y sus metadatos

**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

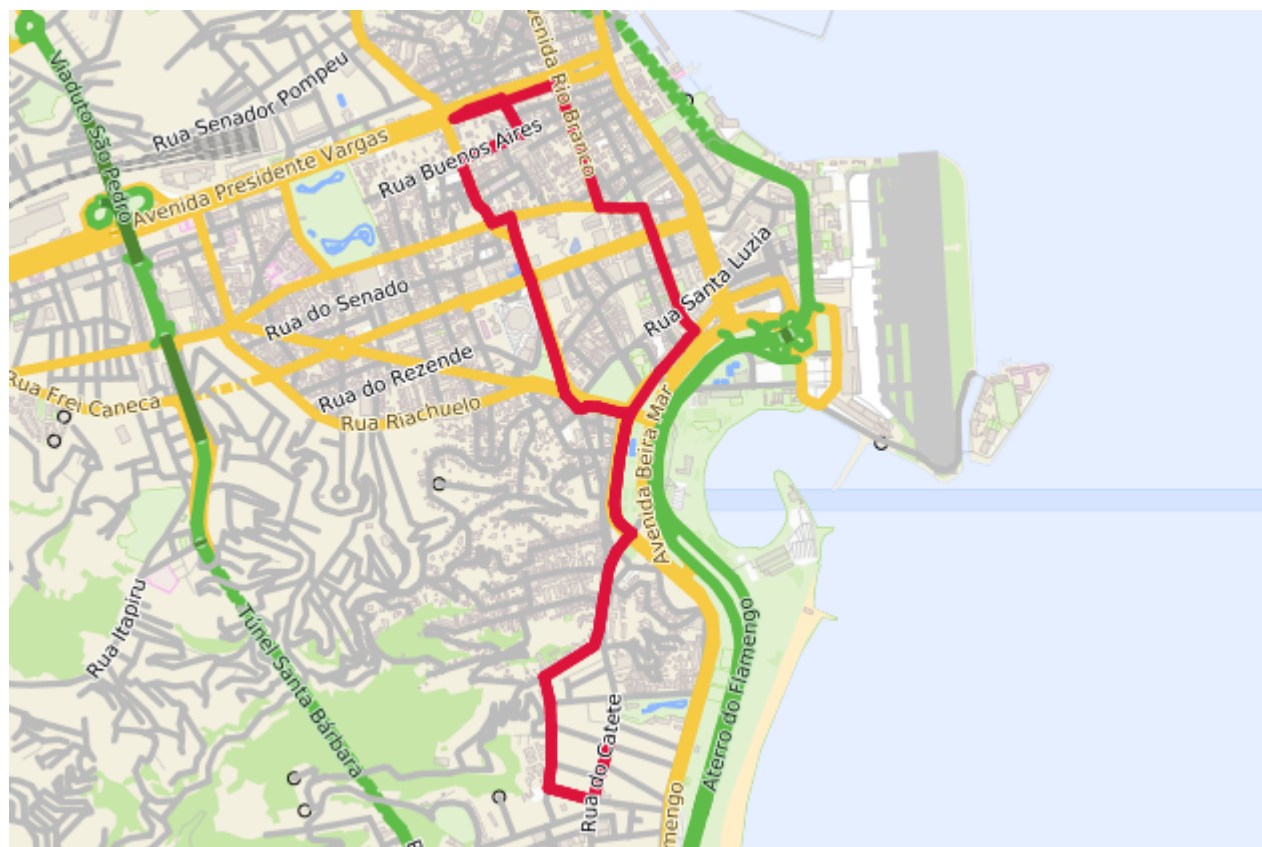
**Nome da View de Dados**

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.