Universidade do Minho

Mestrado em Engenharia Informática

Administração de Base de Dados

Ano lectivo 2014/2015

Trabalho Prático

Relatório Final

Autores:
Luís Miguel Silva (pg24165)
Luís Miguel Pinto (pg27756)
Pedro Carneiro (pg25324)

Braga, 22 de Dezembro de 2014

Conteúdo

1 Introdução **2**
1.1 Ambiente de Testes 2
1.1.1 Especificações do Computador 2

2 Resultados de Desempenho na configuração híbrida *TPC-C + CH-benCHmark* **3**
2.1 *Serializable* como Método de Isolamento 3
2.2 *Repeatable Read* como Método de Isolamento 5
2.3 *Read Uncommitted* como Método de Isolamento 7
2.4 *Read Committed* como Método de Isolamento 9
2.5 Comparação dos Métodos de Isolamento e Conclusões Finais 11

3 Otimização e/ou justificação do desempenho tendo em conta as *queries* analisadas **15**
3.1 Desempenho *Query 1* 18
3.2 Desempenho *Query 17* 19
3.3 Desempenho das restantes *queries* 20

4 Otimização e/ou justificação do desempenho tendo em conta os parâmetros de configuração do PostgreSQL **21**
4.1 Configuração *shared_buffers* 21
4.2 Configuração *effective_cache_size* 22
4.3 Configuração *checkpoint_segments* 23
4.4 Configuração *autovacuum_naptime* 24
4.5 Configuração *work_mem* 25
4.6 Configuração do *random_page_cost* 26
4.7 Configuração do *shared_buffers* com o *effective_cache_size* 27
4.8 Configuração anterior com o *checkpoint_segments* 28
4.9 Configuração anterior com o *autovacuum_naptime* 29
4.10 Configuração anterior com o *random_page_cost* 30
4.11 Configuração anterior com o *work_mem* 31
4.12 Configuração final 32

5 Conclusão **33**

6 Anexos **34**
6.1 Plano de execução da *query* 5 34
6.2 Plano de execução da *query* 6 34
6.3 Plano de execução da *query* 10 35

1 Introdução

Este trabalho tem como objetivo submeter uma base de dados em *PostgreSQL* a vários testes de perfomance, usando uma configuração híbrida *TPC-C + CH-benCHmark* com escala adequada ao hardware.

Neste sentido pretende-se testar várias configurações, analisar os resultados obtidos e alcançar uma configuração que seja considerada eficiente para um determinado número de clientes.

Desta forma, procedeu-se a uma descrição de todo o processo, incluindo não só os testes efetuados até se alcançar a configuração ideal ou não, como também os melhoramentos que poderiam ser feitos, nomeadamente nas *queries* efetuadas à base de dados.

1.1 Ambiente de Testes

De forma a que os testes corram com a menor interferência possível, todos os programas não essenciais à execução dos *benchmark* foram terminados. Durante a execução dos testes teve-se o cuidado de não fazer nenhuma ação que perturbasse o desempenho e os resultados dos *benchmarks*.

1.1.1 Especificações do Computador

O computador utilizado na execução deste trabalho foi um *Macbook Pro Early 2011* com as seguintes especificações:

Processor: *2,3 GHz Intel Core i5*
Memory: *8 GB 1333 MHz DDR3*
Graphics: *Intel HD Graphics 3000 512MB*
Disk: *Samsung SSD 840 EVO 250GB*

2 Resultados de Desempenho na configuração híbrida *TPC-C + CH-benCHmark*

O objetivo desta primeira questão passa por analisar, recorrendo à configuração por defeito do PostgreSQL instalado no sistema, a *performance* da execução híbrida dos *benchmarks* `TPC-C` e `CH-Benchmark`.

O teste consiste na execução de *queries* SQL disponibilizadas pelos *benchmarks*, e no qual se faz variar o **método de isolamento**, o **número de clientes** e o **número de armazéns**.

2.1 *Serializable* como Método de Isolamento

São apresentados em seguida, sobre a forma de uma tabela, os resultados obtidos na execução do *benchmark* usando como Método de Isolamento o *Serializable*.

# clientes	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
2	5068	84.4667	1.0291	1.0702
4	3226	53.7653	1.1079	2.0538
8	1377	22.9500	1.3489	14.7304
16	703	11.7166	1.8898	24.8298

Tabela 1: Resultados obtidos para dois (2) armazéns

# clientes	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
2	2503	41.7166	1.0631	1.0909
4	434	7.2333	0.9899	1.2105
8	516	8.5999	2.1197	57.2194
16	474	7.9000	1.6451	25.3641

Tabela 2: Resultados obtidos para quatro (4) armazéns

A partir dos dados exibidos nas tabelas anteriores, foram desenhados os seguintes gráficos representativos do **débito** (número de pedidos por segundo), da **latência média** e do valor do **percentil 99 da latência**, interpretada, aqui, como latência máxima.

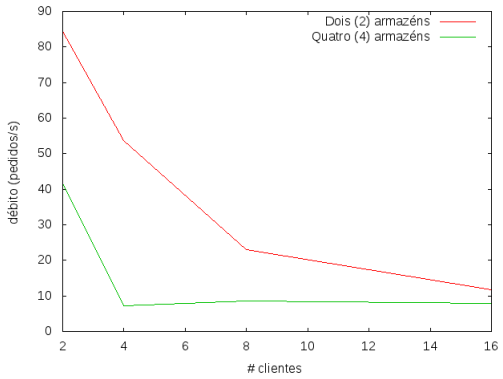

Figura 1: Débito obtido de acordo com o número de clientes para dois e quatro armazéns

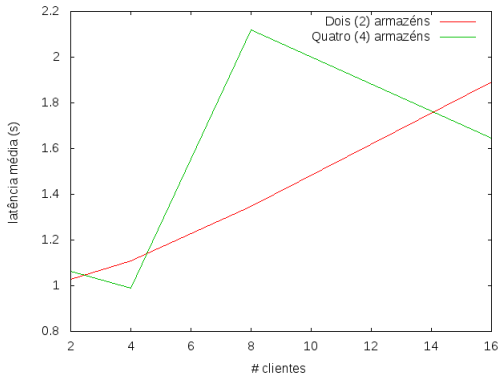

Figura 2: Latência média obtida de acordo com o número de clientes para dois e quatro armazéns

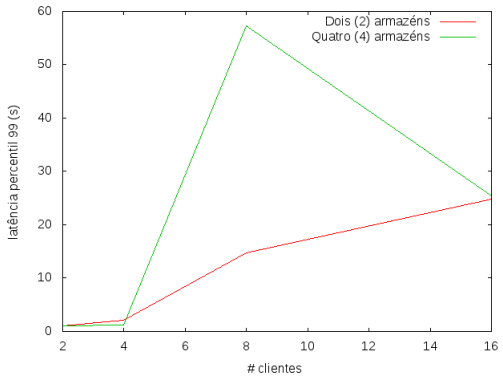

Figura 3: Percentil 99 da latência obtida de acordo com o número de clientes para dois e quatro armazéns

Relativamente aos valores obtidos para o débito, observa-se uma tendêcia clara de diminuição deste à medida que o número de clientes vai aumentando, quer para dois, quer para quatro armazéns. Verifica-se também, como seria de esperar tendo em conta o método "sequencial"de isolamento, que o débito apresenta melhores resultados para bases de dados com menos informação (2 armazéns).

Relativamente à análise aos valores da latência, e como seria de esperar, quanto maior é a base de dados, maior será a latência média e máxima. Para a configuração com quatro armazéns, é notório um crescimento exponencial de 4 para 8 clientes. Já para dois armazéns, o crescimento desta é proporcional. Curiosamente, assiste-se, para 16 clientes, a uma igualdade de valores entre as duas configurações.

2.2 *Repeatable Read* como Método de Isolamento

São apresentados em seguida, sobre a forma de uma tabela, os resultados obtidos na execução do *benchmark* usando como Método de Isolamento o *Repeatable Read*.

# clientes	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
2	17138	285.6333	1.0078	1.0305
4	18435	307.2458	1.0196	1.1275
8	15426	257.0989	1.0439	1.2903
16	13315	221.9153	1.0917	1.6116

Tabela 3: Resultados obtidos para dois (2) armazéns

# clientes	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
2	16818	280.2991	1.0088	1.0297
4	20372	339.5289	1.0113	1.1042
8	17256	287.5948	1.0304	1.2829
16	8744	145.7319	1.1217	2.4365

Tabela 4: Resultados obtidos para quatro (4) armazéns

A partir dos dados exibidos nas tabelas anteriores, foram desenhados os seguintes gráficos representativos do **débito** (número de pedidos por segundo), da **latência média** e do valor do **percentil 99 da latência**, interpretada, aqui, como latência máxima.

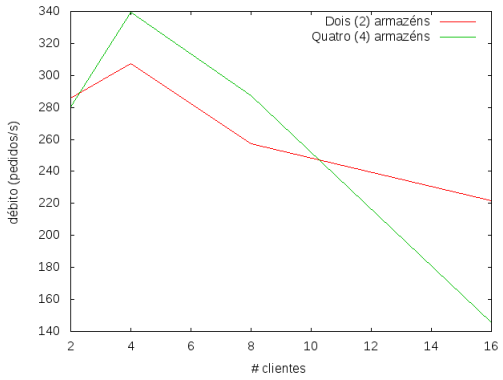

Figura 4: Débito obtido de acordo com o número de clientes para dois e quatro armazéns

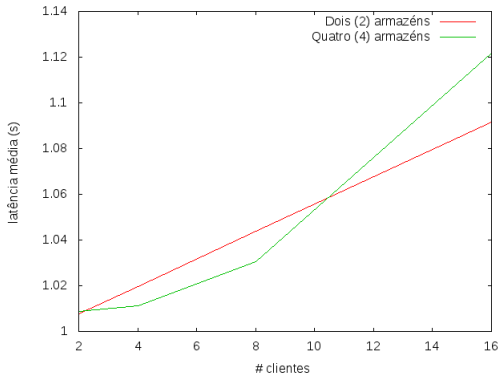

Figura 5: Latência média obtida de acordo com o número de clientes para dois e quatro armazéns

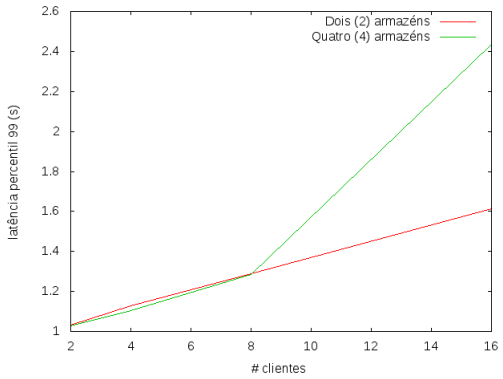

Figura 6: Percentil 99 da latência obtida de acordo com o número de clientes para dois e quatro armazéns

No que ao débito se refere, a configuração com a base de dados maior (4 armazéns) apresenta melhores valores para 4 (o valor máximo, com 340 pedidos/s) e 8 clientes, sofrendo uma queda abrupta quando o número de clientes passa para 16. Com dois armazéns, a diferença de 8 para 16 clientes é pouco significativa.

Já quanto à latência, assiste-se a um crescimento proporcional desta para dois armazéns. Com quatro armazéns, assim como se passou com o débito, assiste-se a uma degradação abrupta com 16 clientes. No entanto, até 8 clientes, uma configuração com quatro armazéns apresenta melhores resultados. Esta tendência é também seguida no valor do percentil 99 da latência já que, como se pode ver, este valor dispara para a configuração com quatro armazéns e 16 clientes.

2.3 *Read Uncommitted* como Método de Isolamento

São apresentados em seguida, sobre a forma de uma tabela, os resultados obtidos na execução do *benchmark* usando como Método de Isolamento o *Read Uncommitted*.

# clientes	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
2	17797	296.6167	1.0090	1.0291
4	20594	343.2303	1.0171	1.1176
8	20852	347.5300	1.0278	1.2382
16	17933	298.8668	1.0770	1.4699

Tabela 5: Resultados obtidos para dois (2) armazéns

# clientes	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
2	17253	287.5477	1.0080	1.0308
4	18406	306.7542	1.0194	1.1353
8	16434	273.8942	1.0306	1.2895
16	18418	306.9657	1.0667	1.5075

Tabela 6: Resultados obtidos para quatro (4) armazéns

A partir dos dados exibidos nas tabelas anteriores, foram desenhados os seguintes gráficos representativos do **débito** (número de pedidos por segundo), da **latência média** e do valor do **percentil 99 da latência**, interpretada, aqui, como latência máxima.

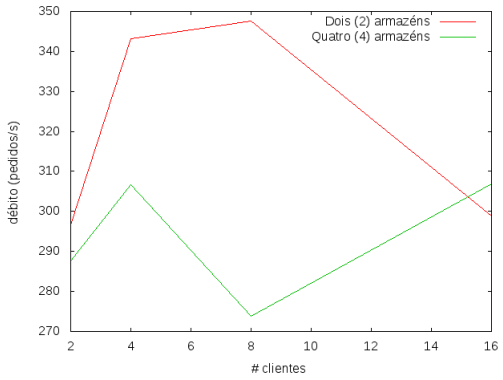

Figura 7: Débito obtido de acordo com o número de clientes para dois e quatro armazéns

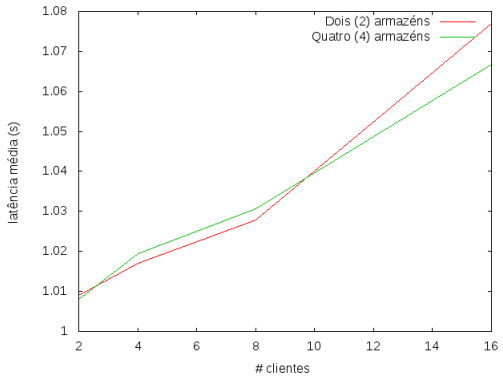

Figura 8: Latência média obtida de acordo com o número de clientes para dois e quatro armazéns

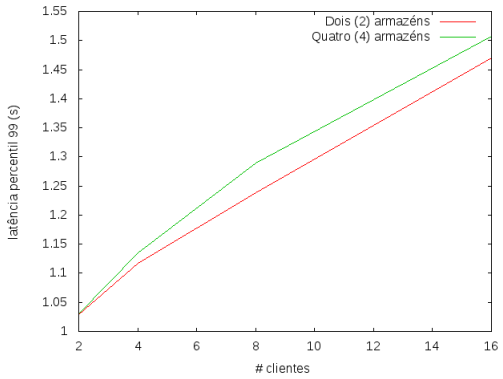

Figura 9: Percentil 99 da latência obtida de acordo com o número de clientes para dois e quatro armazéns

Os resultados obtidos para o débito da aplicação usando como método de isolamento o `Read Uncommited` sugere que uma configuração com quatro armazéns é bastante penalizadora em relação a dois. O melhor valor ocorre, para dois armazéns, com oito clientes. Curiosamente, com 16 clientes, o débito obtido é melhor para 4 armazéns. Neste caso, seria interessante aumentar o número de clientes (por exemplo, 24) para se estudar se, de facto, se trata de uma tendência.

No que à latência média e ao valor do percentil 99 diz respeito, os resultados para ambas as configurações são bastantes semelhantes, o que indícia, de facto, que para este método de isolamento o tamanho da base de dados não é um fator determinante na latência obtida.

2.4 *Read Committed* como Método de Isolamento

São apresentados em seguida, sobre a forma de uma tabela, os resultados obtidos na execução do *benchmark* usando como Método de Isolamento o *Read Comitted*.

# clientes	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
2	14810	246.8311	1.0114	1.0446
4	20792	346.5310	1.0170	1.1070
8	21509	358.4819	1.0310	1.2144
16	12534	208.8956	1.0901	1.9881

Tabela 7: Resultados obtidos para dois (2) armazéns

# clientes	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
2	17495	291.5808	1.0051	1.0264
4	19219	320.3136	1.0165	1.1110
8	18722	312.0306	1.0276	1.2623
16	18302	305.0299	1.0609	1.5071

Tabela 8: Resultados obtidos para quatro (4) armazéns

A partir dos dados exibidos nas tabelas anteriores, foram desenhados os seguintes gráficos representativos do **débito** (número de pedidos por segundo), da **latência média** e do valor do **percentil 99 da latência**, interpretada, aqui, como latência máxima.

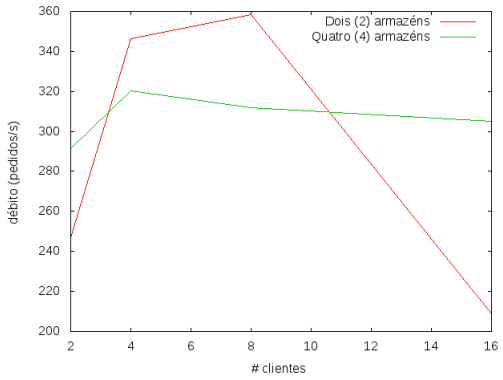

Figura 10: Débito obtido de acordo com o número de clientes para dois e quatro armazéns

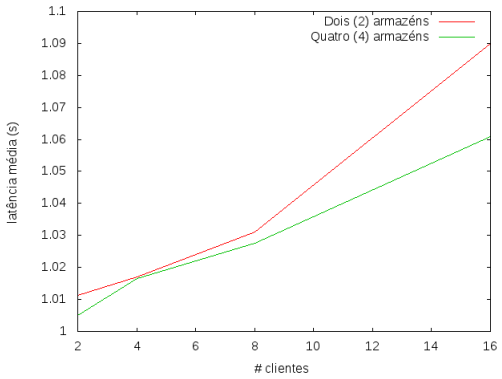

Figura 11: Latência média obtida de acordo com o número de clientes para dois e quatro armazéns

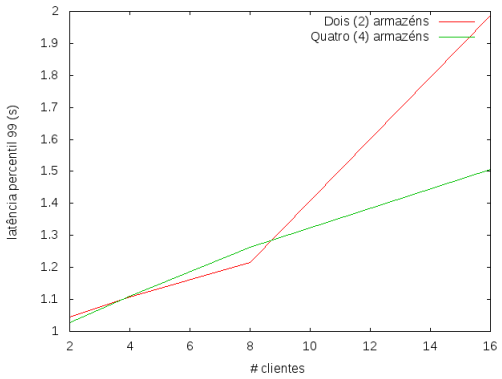

Figura 12: Percentil 99 da latência obtida de acordo com o número de clientes para dois e quatro armazéns

Como se pode verificar através do gráfico do débito da aplicação, os valores máximos são obtidos para uma configuração de dois armazéns, para 4 e 8 clientes, assistindo-se, depois, a uma perda substancial para 16 clientes. Já para uma base de dados maior (4 armazéns), os valores obtidos parecem manter-se constantes à medida que o número de clientes vai aumentando (cerca de 310 pedidos/s), mas inferior ao máximo registado com dois armazéns (360 pedidos/s). No entanto, para 16 clientes, o valor obtido é bastante melhor quando comparado com uma configuração com dois armazéns.

No que se refere à latência, duas tendências são facilmente observáveis, tanto para a latência média como para o valor do percentil 99 desta. A primeira é que, em ambas as configurações, até 8 clientes, os resultados obtidos em ambas são bastante semelhantes. No entanto, para 16 clientes verifica-se um crescimento quase exponencial para dois armazéns, ao passo que para quatro, existe um crescimento, mas proporcional.

2.5 Comparação dos Métodos de Isolamento e Conclusões Finais

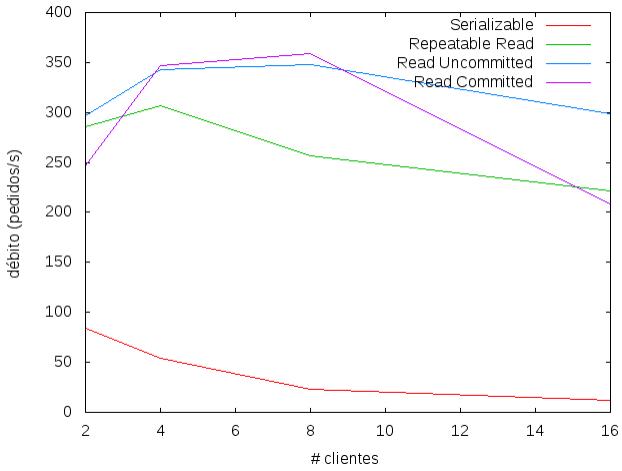

Figura 13: Comparação do débito obtido para cada método de isolamento em função do número de clientes para uma configuração com dois (2) armazéns

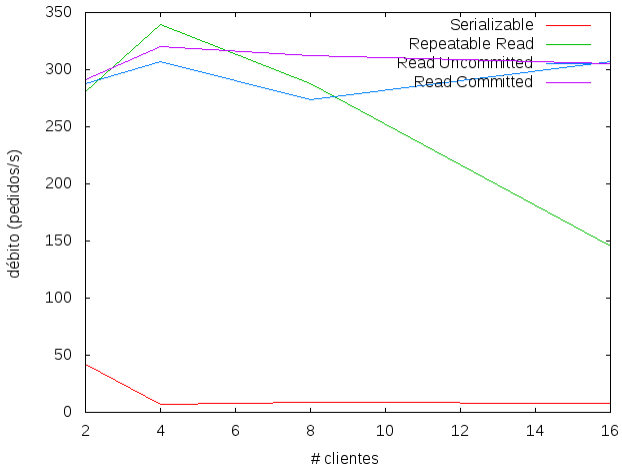

Figura 14: Comparação do débito obtido para cada método de isolamento em função do número de clientes para uma configuração com quatro (4) armazéns

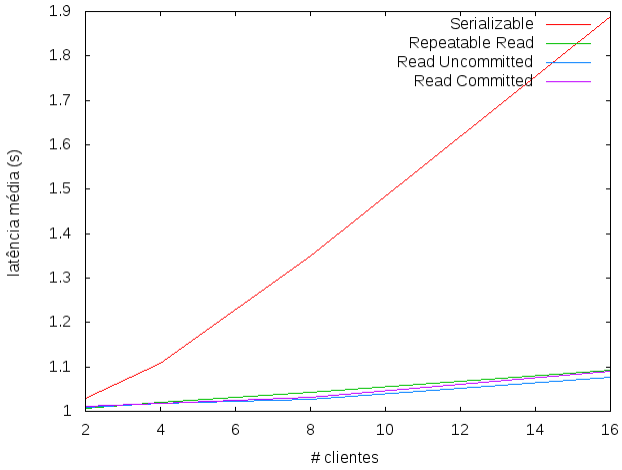

Figura 15: Comparação da latência média obtida para cada método de isolamento em função do número de clientes para uma configuração com dois (2) armazéns

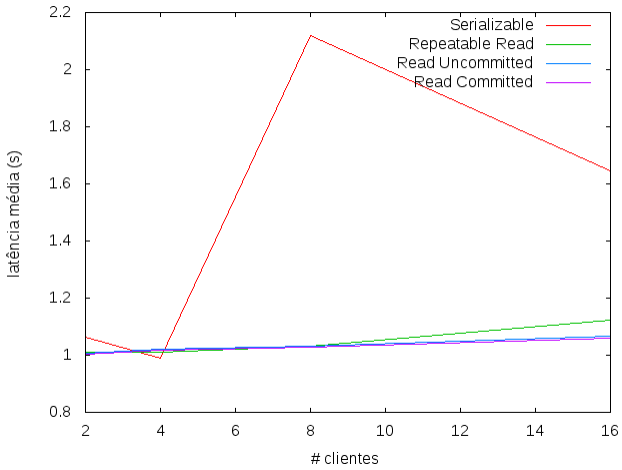

Figura 16: Comparação da latência média obtida para cada método de isolamento em função do número de clientes para uma configuração com quatro (4) armazéns

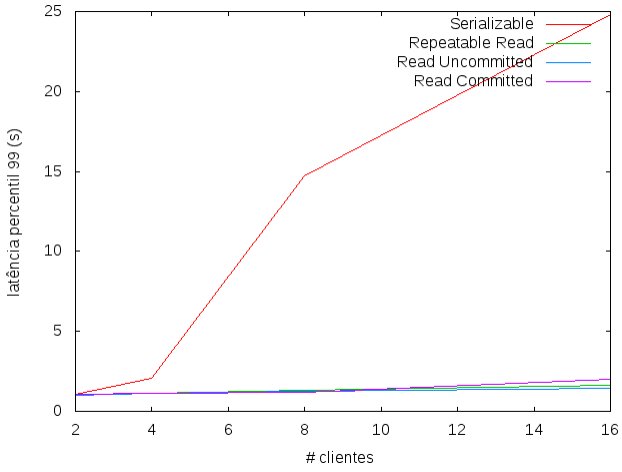

Figura 17: Comparação do valor do percentil 99 da latência obtido para cada método de isolamento em função do número de clientes para uma configuração com dois (2) armazéns

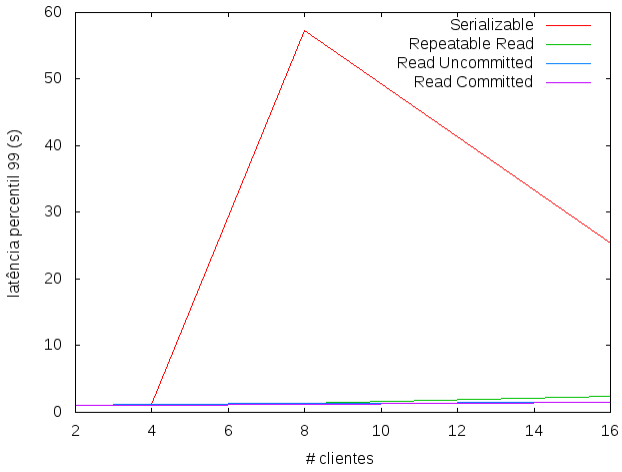

Figura 18: Comparação do valor do percentil 99 da latência obtido para cada método de isolamento em função do número de clientes para uma configuração com quatro (4) armazéns

Dos tipos de isolamento anteriores percebemos que o *Repeatable Read* é que o que acaba por o ter o isolamento mais equilibrado, pois as falhas que podem acontecer durante a execução dos testes são muitos menos prováveis. Os isolamentos *Read Committed* e *Read Uncommitted* têm maior probabilidade de falhas e maior número de falhas possíveis. Já o *Serializable* é um tipo de isolamento muito rígido em que só permite executar uma *query* depois de terminar a anterior. Isto impede que haja falhas na base de dados mas faz com que não seja um isolamento praticavél.

Depois de definido o melhor tipo de isolamento, *Repeatable Read*, é essencial perceber quantos armazéns e número de clientes se deve usar como configuração base deste *benchmark*. Assim, depois de analisarmos os gráficos anteriores percebe-se que para um número de **4** clientes e **4** se obteve o melhor resultado, tanto a nível de débitos como de latências.

De notar que tivemos que alterar na configuração do *XML* o parâmetro do débito pois este estava a causa impacto no *benchmarking* o que não era pretendido. Foi então colocado um valor de 10000 de débito de forma que o número de transações por segundo não tivesse qualquer tipo de limitação.

Na configuração anterior referimos que o *time* é de 200 segundos mas nos testes executados

Parâmetro	Valor
isolation	*Repeatable Read*
scalefactor	4
terminals	4
time	200
rate	10000

Tabela 9: Configuração inicial do *Benchmark*

inicialmente o *time* foi de 60 segundos. Esta alteração ocorreu porque percebemos que 60 segundos era insuficiente para avaliar os resultados da configuração, futuramente, do *postgresql.conf*. Como tal, executamos mais um teste para a melhor configuração, definida anteriormente mas com vários clientes, e obteve-se os seguintes resultados:

# clientes	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
2	50985	254.924654589839	1.01284884652349	1.040724
4	62118	310.588066603262	1.02196725140861	1.124054
8	46867	234.333927170071	1.05701739208825	1.360126
16	45012	225.058994290125	1.12131417364258	1.622081

Tabela 10: Resultados para 200 segundos com vários clientes

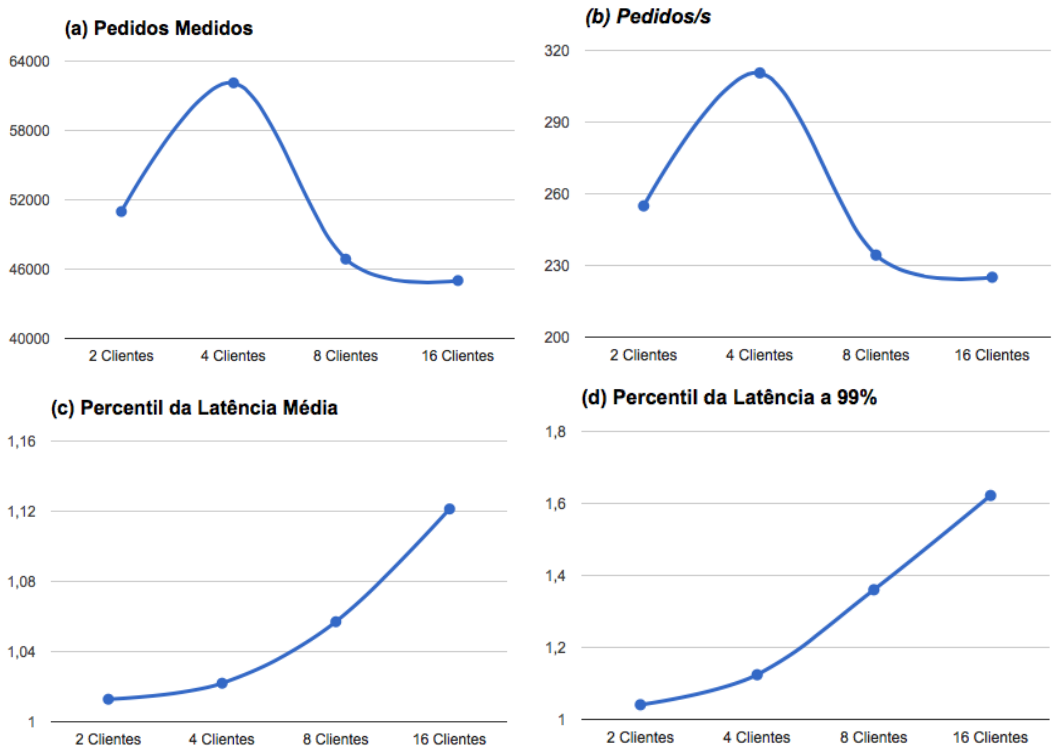

Figura 19: Gráficos resultantes da tabela anterior

Estes resultados serão a partir deste momento usados como referência para os próximos *benchmarks* efectuados.

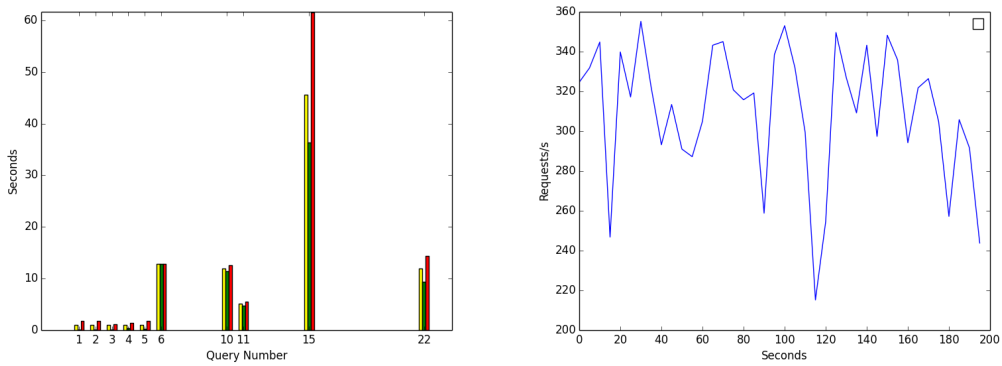

Figura 20: Valores de tempo de execução por *query* e débito de transações

3 Otimização e/ou justificação do desempenho tendo em conta as *queries* analisadas

Neste ponto deu-se a análise de queries para perceber quais as que tem um tempo de execução superior e desta forma analisar e perceber o porquê de isso acontecer.

O gráfico seguinte mostra a execução das *queries*, primeiramente do *TPC-C* e seguidamente do *CHBenchmark*, onde o *query number* de 1 a 5 representa as queries do TPC-C e as restantes representam as queries do CHBenchmark. De notar que, por exemplo, o *query number* 6 corresponde à *query* 1 do *CHBenchmark*, a query 10 corresponde à *query* 5, e assim sucessivamente.

Para uma identificação mais fácil das *queries* do *CHBenchmark* que foram executadas no gráfico acima, segue-se o que cada uma faz:

Query 1 - Esta consulta informa a quantidade total e quantidade de todas as *OrderLines* enviados, dadas por um período de tempo específico. Além disso, informa sobre a quantidade média mais a contagem total de todos essas *OrderLines* ordenados pelo número *OrderLine* individual.

```
select
  ol_number, sum(ol_quantity) as sum_qty, sum(ol_amount) as sum_amount, avg(ol_quantity) as avg_qty,
  avg(ol_amount) as avg_amount, count(*) as count_order
from
  order_line
where
  ol_delivery_d > '2007-01-02 00:00:00.000000'
group by
  ol_number
order by
  ol_number
```

Query 5 – Esta consulta serve para obter informações sobre as receitas conseguidas das nações dentro de uma determinada região. Todas as nações são classificadas segundo o valor total da receita adquirida desde a data indicada.

```
select n_name,sum(ol_amount) as revenue
from customer, oorder, order_line, stock, supplier, nation, region
where c_id = o_c_id
     and c_w_id = o_w_id
     and c_d_id = o_d_id
     and ol_o_id = o_id
```

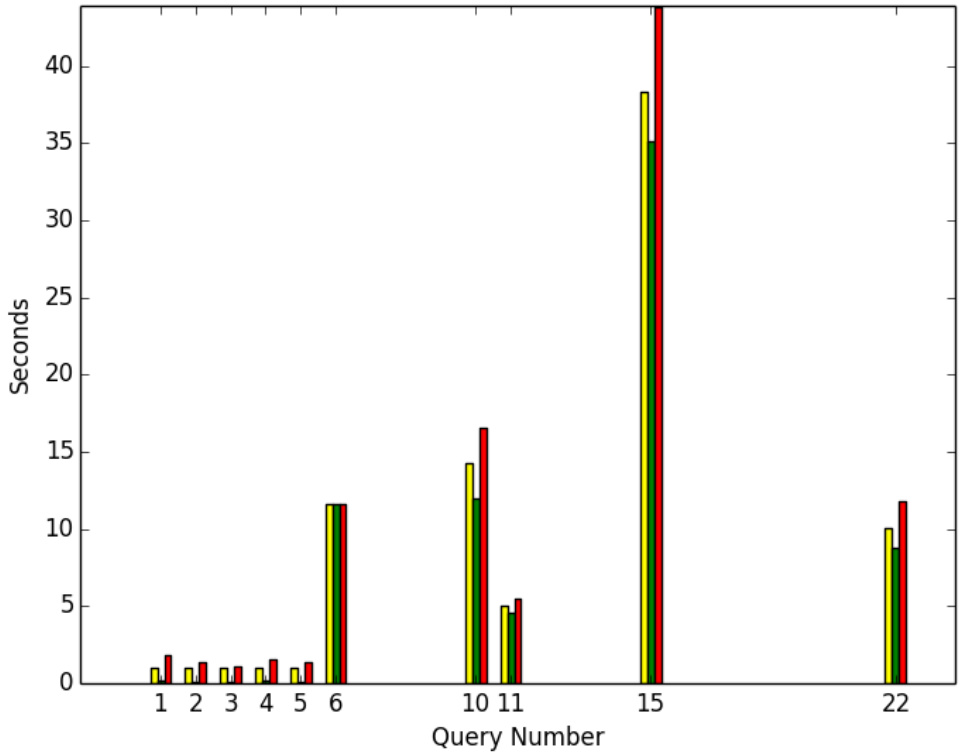

Figura 21: Tempos de execução para cada *query*

```
    and ol_w_id = o_w_id
    and ol_d_id=o_d_id
    and ol_w_id = s_w_id
    and ol_i_id = s_i_id
    and mod((s_w_id * s_i_id),10000) = su_suppkey
    and ascii(substr(c_state,1,1)) = su_nationkey
    and su_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'Europe'
    and o_entry_d >= '2007-01-02 00:00:00.000000'
group by n_name
order by revenue desc;
```

Query 6 – Esta consulta lista o montante total das receitas arquivadas do OrderLines, que foram entregues num período específico e numa certa quantidade.

```
select
  sum(ol_amount) as revenue
from
  order_line
where
  ol_delivery_d >= '1999-01-01 00:00:00.000000'
  and ol_delivery_d < '2020-01-01 00:00:00.000000'
  and ol_quantity between 1 and 100000
```

Query 10 – Esta consulta serve para analisar as despesas de todos os clientes numa listagem do seu país, alguns detalhes deles e a quantidade de dinheiro que eles têm usado para comprar as suas ordens desde uma data específica. A lista inteira é ordenada pela quantidade de encomendas dos clientes.

```
select
  c_id, c_last, sum(ol_amount) as revenue, c_city, c_phone, n_name
from
  customer, oorder, order_line, nation
where
  c_id = o_c_id
  and c_w_id = o_w_id
  and c_d_id = o_d_id
  and ol_w_id = o_w_id
  and ol_d_id = o_d_id
  and ol_o_id = o_id
  and o_entry_d >= '2007-01-02 00:00:00.000000'
  and o_entry_d <= ol_delivery_d
  and n_nationkey = ascii(substr(c_state,1,1))
group by
  c_id, c_last, c_city, c_phone, n_name
order by
  revenue desc
```

Query 17 – Esta consulta determina a perda anual de receita, se as encomendas apenas com uma quantidade de mais do que a quantidade média de todas as ordens no sistema fossem compradas e enviadas aos clientes.

```
select
  sum(ol_amount) / 2.0 as avg_yearly
from
  order_line, (
  select
    i_id, avg(ol_quantity) as a
  from
    item, order_line
  where
    i_data like '%b' and ol_i_id = i_id group by i_id) t
where
  ol_i_id = t.i_id and ol_quantity < t.a
```

Posto isto chegou-se à conclusão que é necessário avaliar as 5 *queries* do *CHBenchmark* que foram executadas, e perceber as que tiveram um tempo de execução muito elevado, ou seja, verificar o que se passa nas mesmas e tentar melhor o seu desempenho. Um dos pontos que foi possível concluir é que estas *queries* apresentadas têm um tempo de execução elevado porque possuem várias condições para as mesmas serem realizadas com sucesso. Após análise às mesmas foi possível concluir que em apenas duas conseguimos melhorar o desempenho alterando o código *SQL* da mesma sem afetar a base de dados e os seus resultados.

3.1 Desempenho *Query 1*

Na *query 1* com o código seguinte, conseguimos melhorar a performance levando menos tempo a executar a query e obtendo o mesmo resultado, simplesmente dividindo a *query* em dois *select* de forma a melhorar a performance como é demonstrado a seguir:

```
select ol_number,
  sum_qty,
  sum_amount,
  sum_qty/count_order as avg_qty,
  sum amount/count_order as avg_amount,
  count_order
from
  (select
    ol_number, sum(ol_quantity)as sum_qty, sum(ol_amount) as sum_amount, count(*) as count_order
  from
    order_line
  where
    ol_delivery_d > '2007-01-02 00:00:00.000000'
    group by ol_number
    order by ol_number) as t
```

```
Sort  (cost=66194.11..66194.14 rows=12 width=13) (actual time=2740.484..2740.486 rows=15 loops=1)
  Sort Key: ol_number
  Sort Method: quicksort  Memory: 27kB
  ->  HashAggregate  (cost=66193.71..66193.89 rows=12 width=13) (actual time=2740.456..2740.473 rows=15 loops=1)
        ->  Seq Scan on order_line  (cost=0.00..47181.14 rows=1267505 width=13) (actual time=0.048..411.462 rows=1266486 loops=1)
              Filter: (ol_delivery_d > '2007-01-02 00:00:00'::timestamp without time zone)
              Rows Removed by Filter: 404885
Total runtime: 2740.541 ms
```

Figura 22: Plano de execução da *query 1* antes da otimização

```
Subquery Scan on t  (cost=59856.52..59856.79 rows=12 width=76) (actual time=1344.861..1344.881 rows=15 loops=1)
  ->  Sort  (cost=59856.52..59856.55 rows=12 width=13) (actual time=1344.851..1344.854 rows=15 loops=1)
        Sort Key: order_line.ol_number
        Sort Method: quicksort  Memory: 26kB
        ->  HashAggregate  (cost=59856.19..59856.31 rows=12 width=13) (actual time=1344.833..1344.835 rows=15 loops=1)
              ->  Seq Scan on order_line  (cost=0.00..47181.14 rows=1267505 width=13) (actual time=0.045..384.742 rows=1266486 loops=1)
                    Filter: (ol_delivery_d > '2007-01-02 00:00:00'::timestamp without time zone)
                    Rows Removed by Filter: 404885
Total runtime: 1344.975 ms
```

Figura 23: Plano de execução da *query 1* depois da otimização

Como é possivel visualizar nas figuras acima obtivemos uma melhoria de desempenho, onde conseguimos reduzir para menos de metade o tempo total de execução da *query*. Como podemos confirmar a otimização foi efetuada corretamente isto porque os resultados obtidos são exatamente os mesmos. Como é possivel verificar, dividindo a *query* em duas *queries* ou efetuado uma *sub-query* conseguimos reduzir para metade o tempo de execução, isto porque primeiro os dados são selecionados de acordo com as condições pedidas e só depois na *query* principal é que são mostrados os dados que interessam.

3.2 Desempenho *Query 17*

Na *query 17* foi possível, também, fazer algumas melhorias, isto porque a mesma fica mais eficiente se houver uma *materialized view* para calcular a média para cada índice. Segue-se abaixo o código *SQL* que permite obter os mesmos resultados mas com mais eficiência e os respetivos planos de execução para comparação:

```
CREATE MATERIALIZED VIEW AvgByID AS
  select i_id, avg(ol_quantity) as average
  from item, order_line
  where i_data like '%b' and ol_i_id = i_id
  group by i_id

select
  sum(ol_amount) / 2.0 as avg_yearly
from
  order_line,
  (select
    i_id, average
  from
    AvgByID) t
where
  ol_i_id = t.i_id
  and ol_quantity < t.average
```

```
Aggregate  (cost=138724.29..138724.30 rows=1 width=4) (actual time=2383.389..2383.389 rows=1 loops=1)
  ->  Hash Join  (cost=126468.09..138684.49 rows=15919 width=4) (actual time=1897.717..2373.617 rows=23965 loops=1)
        Hash Cond: (item.i_id = order_line.ol_i_id)
        Join Filter: (order_line.ol_quantity < (avg(order_line_1.ol_quantity)))
        Rows Removed by Join Filter: 41598
        ->  HashAggregate  (cost=54412.24..54437.49 rows=2020 width=9) (actual time=798.340..802.836 rows=3888 loops=1)
              ->  Hash Join  (cost=2546.25..54243.43 rows=33762 width=9) (actual time=31.810..719.339 rows=65563 loops=1)
                    Hash Cond: (order_line_1.ol_i_id = item.i_id)
                    ->  Seq Scan on order_line order_line_1  (cost=0.00..43002.71 rows=1671371 width=9) (actual time=0.027..273.132 rows=1671371 loops=1)
                    ->  Hash  (cost=2521.00..2521.00 rows=2020 width=4) (actual time=31.755..31.755 rows=3888 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 137kB
                          ->  Seq Scan on item  (cost=0.00..2521.00 rows=2020 width=4) (actual time=0.011..30.896 rows=3888 loops=1)
                                Filter: ((i_data)::text ~~ '%b'::text)
                                Rows Removed by Filter: 96112
        ->  Hash  (cost=43002.71..43002.71 rows=1671371 width=13) (actual time=1088.434..1088.434 rows=1671371 loops=1)
              Buckets: 4096  Batches: 128  Memory Usage: 692kB
              ->  Seq Scan on order_line  (cost=0.00..43002.71 rows=1671371 width=13) (actual time=0.034..524.538 rows=1671371 loops=1)
Total runtime: 2383.482 ms
```

Figura 24: Plano de execução da *query 17* antes da otimização

No melhoramento desta *query* é possivel visualizar que através duma *materialized view*, conseguimos reduzir drasticamente o tempo de execução da *query*. Neste caso, como o pretendido é consultar os dados e não altera-lós a *materialized view* encaixa como um possivel método de visualização, isto porque apesar de ocupar mais espaço na nossa base de dados, ficamos com os dados que pretendemos ver pré-compilados tendo assim uma melhoria elevada de perfomance aquando duma consulta da *materialized view*. A *materialized view* permite que os cálculos sejam feitos previamente, e atualizados por o método *refresh* e assim ao contrário das *views* normais que consultam os dados em tempo de execução perdendo assim desempenho.

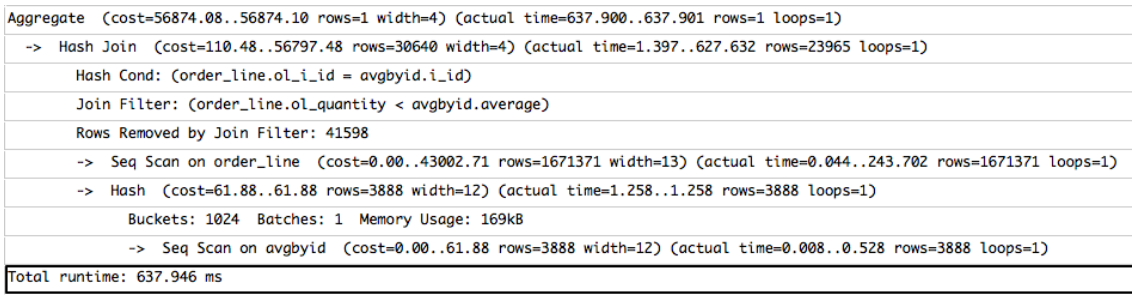

Figura 25: Plano de execução da *query 17* depois da otimização

3.3 Desempenho das restantes *queries*

Analisando a *query 5* pode ver-se que o tempo de execução da mesma é elevado porque a condição *where* é bastante extensa, tendo várias condições que são feitas uma a uma, diminuindo assim o desempenho da mesma.

Analisando a *query 6* é possível notar que a mesma tem uma longa duração de execução devido às condições impostas na condição *where*. A gama de valores é bastante elevada tendo de percorrer cerca de 20 anos de registos bem como a quantidade presente que pode variar entre valores muito elevados (1 a 100000).

Analisando a *query 10* é possível notar que o que esta a fazê-la perder performance é o facto de estar a fazer um *group by* muito grande, o que leva a que o custo da query seja maior e a mesma demore mais tempo a ser executada.

Os planos de execução destas queries encontram-se em anexo.

4 Otimização e/ou justificação do desempenho tendo em conta os parâmetros de configuração do PostgreSQL

O objetivo desta questão passa por alterar certos parâmetros, recorrendo à configuração do *PostgreSQL* para tentar aumentar a *performance* da execução híbrida dos *benchmarks* `TPC-C` e `CH-Benchmark`.

4.1 Configuração *shared_buffers*

O parâmetro de configuração *shared_buffers* determina a quantidade de memória RAM dedicada ao *PostgreSQL* que vai ser usada para armazenar dados enquanto executa as queries.

Inicialmente, podia-se prever que quanto maior fosse a quantidade de memória dedicada melhor seriam os resultados de desempenho. É uma conclusão precipitada, pois o *PostgreSQL* utiliza esta memória para efetuar operações e não para *cache* de disco, por exemplo.

Um valor razoável para para o *shared_buffers* é de 25% da memória RAM do sistema (2GB) mas decidiu-se extender os testes até 50% da memória do sistema (4GB). Apesar de valores acima de 40% serem, provavelmente, menos eficazes, pois o *PostgreSQL* conta também com a *cache* do sistema operativo.

Para o *benchamark* foram usados valores de memória RAM entre os 128MB, valor por defeito na configuração, e os 4096MB. Os resultados seguem-se na próxima tabela.

# tamanho (MB)	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
128 (Default)	62118	310.5880666032618	1.0220994499090980	1.130365
256	60989	304.9445727406344	1.0224922605223894	1.133323
512	63196	315.9795274241785	1.0210114359611369	1.125195
1024	66182	330.9085519342495	1.0217862895198089	1.110705
2048	70374	351.8684527622802	1.0194107100775853	1.098318
3072	64198	320.9880720942809	1.0198997528427676	1.129971
4096	60497	302.4838197277969	1.0220000570937402	1.134960

Tabela 11: Resultados obtidos no *shared_buffers* para diferentes tamanhos de memória

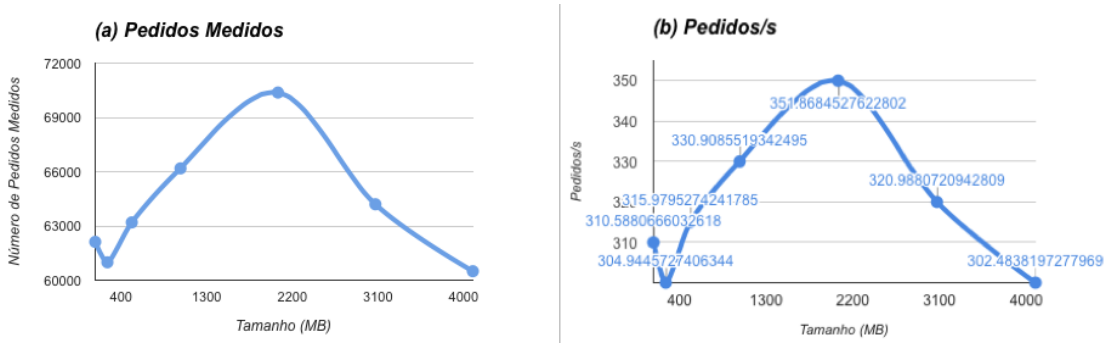

Figura 26: Gráfico correspondente aos valores da tabela anterior

Ao analisarmos os resultados obtidos, verifica-se que o percentil da latência média e a 99% estão constantemente a diminuir à medida que a quantidade de memória disponível aumenta. Isto acontece devido à ausência de escrita dos resultados para o disco, o que provoca a redução na quantidade de *I/O* para a obtenção de dados. Conclui-se ainda que o melhor resultado aconteceu quando a memória disponível era de **2048MB**, tendo ocorrido mais pedidos por segundo e as latências (média e 99%) apresentarem os resultados mais inferiores. A segunda melhor configuração para esta métrica aconteceu quando a memória disponível era de **1024MB**.

Assim sendo, para esta configuração o valor escolhido será o de **2048MB** por apresentar os melhores resultados tanto a nível de pedidos como de latência.

4.2 Configuração *effective_cache_size*

O *effective_cache_size* é definido como uma estimativa da quantidade de memória disponível para a *cache* do disco. Este parâmetro é utilizado *PostgreSQL Query Planner* para perceber que planos pode utilizar tendo em conta o espaço disponível em memória.

A documentação do *PostgreSQL* afirma que definir 50% da memória total do dispositivo será uma configuração conservadora e que 75% uma configuração agressiva, mas aceitável.

Então para a configuração deste *benchmark* utilizou-se os seguintes parâmetros, 128MB, 512MB, 1024MB, 2048MB, 4096MB e 6144MB.

# tamanho (MB)	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
128 (Default)	62118	310.58806660326184	1.022099449909098	1.130365
512	67022	335.1094363325237	1.0208270027602877	1.112753
1024	66987	334.93426967582496	1.0204846512009793	1.109516
2048	66478	332.38835629969145	1.020474478669635	1.112206
4096	59880	299.3983502432346	1.0226386898630595	1.131897
6144	63982	319.9084020175434	1.0210112850958082	1.130562

Tabela 12: Resultados obtidos no *effective_cache_size* para diferentes tamanhos de memória

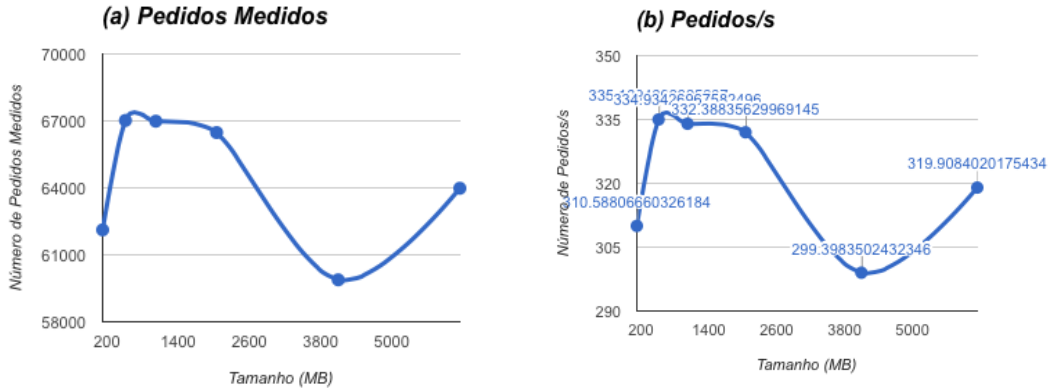

Figura 27: Gráfico correspondente aos valores da tabela anterior

Ao analisar-se os valores obtidos identificasse que na configuração mais conservadora (**4096MB**) e agressiva (**6144MB**) não se obteve os melhores resultados. Pensámos que se deve ao facto de a base de dados do benchmark não ter um tamanho, cerca de 650MB, que faça a diferença neste aspecto. O número de pedidos medidos foi muito inferior aos outros valores assim como o percentil da latência a 99%, a latência média, praticamente, encontra-se entre a média dos resultados.

Podemos, então, definir o intervalo de [**512-2048**]MB como o que se obteve melhor resultados para esta configuração. O valor **1024MB** obteve melhor resultados no percentil da lantência a 99%. Já o valor **512MB** obteve mais pedidos por segundo e um ligeiro aumento nas latências no teste, este último influência negativamente o resultado. Com o parâmetro a **2048MB** consegue-se manter na média dos resultados anteriores, possuíndo o resultado com menor latência média.

Dentro do intervalo definido anteriormente se fosse necessário escolher um resultado como melhor valor, escolhíamos a configuração com **1024MB** de *effective_cache_size* onde se obteve a menor latência a 99% e valor similares na latência média para o mesmo número de pedidos.

4.3 Configuração *checkpoint_segments*

O *PostgreSQL* escreve novas transações na base de dados em ficheiros chamados de segmentos *Write-Ahead Logging (WAL)* que tem um tamanho de 16MB. Nas configurações do *postgresql.conf* o valor atualmente por defeito é de 3 *checkpoint_segments*.

Aumentar o número de *checkpoint_segments* melhora o desempenho do benchmark pois faz com que os *checkpoints* ocorram com menor frequência, obrigando assim a base de dados a uma recuperação mais lenta após uma falha. No entanto o *I/O* ao disco diminui provocando melhorias, eventuais, no teste.

Para este *benchmarking* foram utilizados os seguintes valores de *checkpoint_segments*: 3, 9 e 16.

# segmentos	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
3 (Default)	62118	310.58806660326184	1.022099449909098	1.130365
9	66254	331.26961583822094	1.0211749849065717	1.113339
16	54564	272.81969749069896	1.0238594005571438	1.103626

Tabela 13: Resultados obtidos para o *checkpoint_segments*

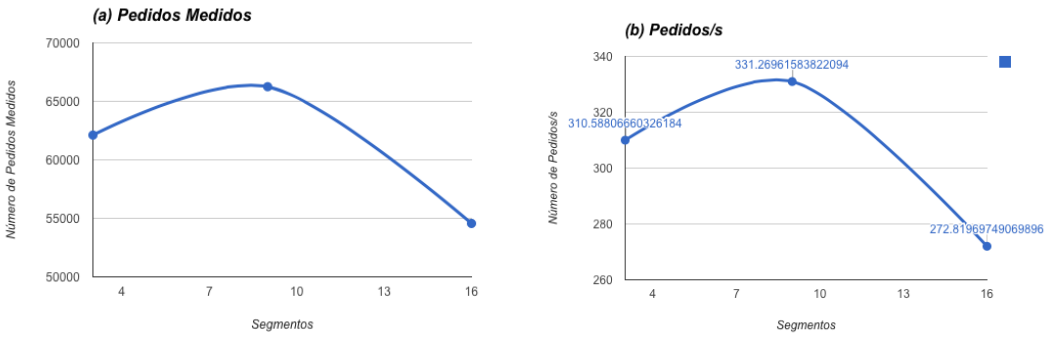

Figura 28: Gráfico correspondente aos valores da tabela anterior

Como foi de esperar, ao aumentar o valor de *checkpoint_segments* o valores de transações aumentaram, mas somente quando o valor foi de **9** segmentos. Quando o teste foi efetuado com **16** segmentos o número de transações diminuiu abruptamente, mas na latência a 99% obteve-se o melhor resultado, ainda que a latência média se tenha mantido. Estes resultados devem-se à recuperação mais lenta da base de dados.

Nesta situação, a configuração mais indicada seria de **9** segmentos. O *benchmark* obteve o melhor resultado ao nível das transações e de latência média, que foi inferior.

Normalmente para valores elevados de *checkpoint_segments* é recomendado aumentar o *checkpoint_timeout* por causa dos intervalos de controlo.

4.4 Configuração *autovacuum_naptime*

Este processo corresponde ao tempo mínimo entre a execução do *autovacuum* (limpar resíduos deixados na base de dados), isto é, só ocorre um novo depois de ter passado o tempo de execução do anterior.

Para se perceber o que acontece nesta configuração foram feitos vários *benchmarks* para os seguintes tempos de *autovacuum_naptime*: desligado, 1 minuto, 2 minutos e 3 minutos entre cada execução. Obteve-se, assim, os resultados da próxima tabela.

# tempo (min)	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
0 (off)	59802	299.0081276096099	1.021858928530818	1.136963
1 (Default)	62118	310.5880666032618	1.022099449909098	1.130365
2	53296	266.4799488385146	1.025329163370609	1.138289
3	61163	305.8148124055486	1.022736959518009	1.124206

Tabela 14: Resultados obtidos para a configuração do *autovacuum_naptime*

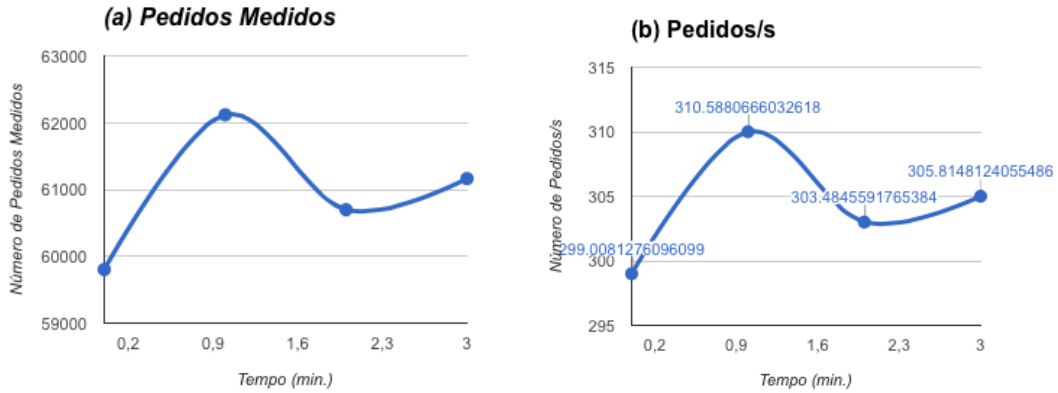

Figura 29: Gráficos correspondente aos valores da tabela

Dos resultados apresentados verifica-se que aqueles que apresentam melhores tempos de execução foram o de **1** minuto e **3** minutos. Este último só ocorreu no máximo uma vez, assim como o teste a **2** minutos, o que leva a querer que não teve muita influência nos valores da latência. Com o parâmetro desligado não se obteve os resultados mais corretos, isto porque não ocorreu nenhuma execução do ***autovacuum*** o que contribui para estatísticas erradas e consequentemente para escolha errada dos algoritmos usados pelo *PostgreSQL*.

Ao contrário do esperado ter um ***autovacuum*** mais regular faz com que os resultados obtidos fossem mais razoáveis. De facto, ao haver com mais frequência o ***autovacuum*** faz com que em cada operação haja menos dados para limpar.

Assim, decidiu-se escolher para o *autovacuum_naptime* como melhor resultado o tempo de **1** minuto, por defeito na configuração do *postgresql.conf*.

4.5 Configuração *work_mem*

O *work_mem* especifica a quantidade de memória RAM que pode ser usada para operações internas (*Sort e Hash Tables*) antes de serem gravadas para ficheiros temporários em disco. O valor predefinido na configuração do *PostgreSQL* é de 1MB.

Como existem algumas *queries* complexas no teste do *CHBenchmark* decidimos testar como vão variar o tempo de execução das mesmas e perceber se é possível obter mais transações com a menor latência possível. Para isto foram feitos testes para 1MB, 8MB, 32MB, 128MB e 512MB.

# tamanho (MB)	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
1 (Default)	62118	310.5880666032618	1.022099449909098	1.130365
8	62391	311.9513589177765	1.021819045423218	1.130889
32	67109	335.5443187426919	1.019860019833405	1.117067
128	67728	338.6382522575026	1.018854204671627	1.118497
512	67698	338.4895027961542	1.020378764261869	1.118507

Tabela 15: Resultados obtidos no *work_mem* para diferentes tamanhos de memória

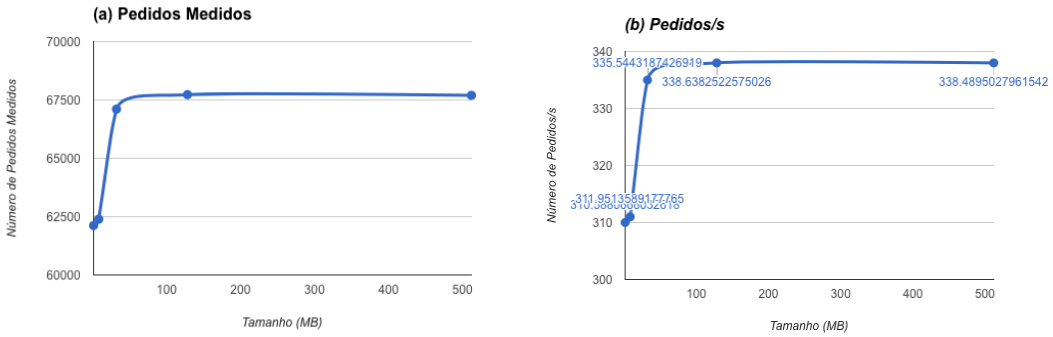

Figura 30: Gráficos correspondente aos valores da tabela

Através das ferramentas que acompanham o *oltpbench*, nomeadamente o *plot_latencies.py* obtivemos o seguinte gráfico, para os valores médios de execução:

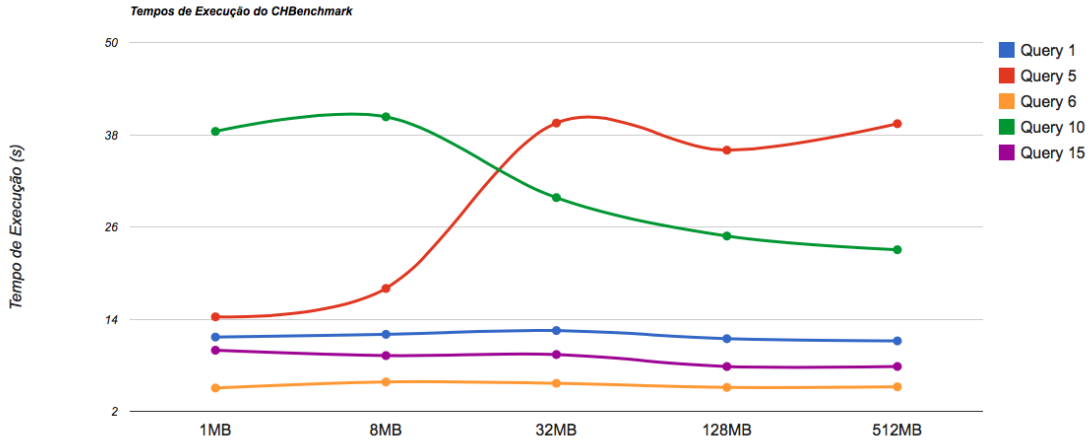

Figura 31: Gráfico com os valores de execução para cada query com diferentes valores de memória

Como verificámos no gráfico anterior o tempo de execução variou significativamente em duas

queries, na *Query 10* que tinha um tempo de execução superior na configuração inicial e que passou a obter um tempo de execução inferior, proporcionalmente à memória disponível. Já na *Query 5* que inicialmente tinha um tempo de execução inferior passou a tempos muitos superiores ao esperado. Isto deve-se ao facto dos algoritmos escolhidos terem efeito na quantidade de memória definida.

Nos resultados das *queries 1, 6 e 17* não se surtiu grandes variações.

Caso fosse necessário escolher a melhor configuração para este parâmetro, este situava-se entre os **8MB e 32MB**. Será necessário efetuar mais um teste, por exemplo, com **16MB** para perceber se houve alguma alteração na execução das *queries* que tire partido da memória disponível.

Finalmente, se analisarmos o valor das latências junto com as transações medidas, percebe-se que **32MB** será provavelmente uma boa configuração para este parâmetro.

4.6 Configuração do *random_page_cost*

Como o dispositivo utilizado para correr os testes possui um *Solid State Drive (SSD)* decidou-se alterar na configuração do *PostgreSQL* o custo dos acessos aleatórios. Neste teste vamos mostrar os resultados obtidos para a configuração inicial que tem definido um custo igual a 4, e alterações de custo entre 1 e 2.

# custo	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
4.0 (Default)	62118	310.58806660326184	1.022099449909098	1.130365
2.0	65295	326.47479814226466	1.0193406799754958	1.116984
1.0	58540	292.69854964502616	1.0225425792791254	1.133753

Tabela 16: Resultados obtidos para 3 tipos diferentes de custos

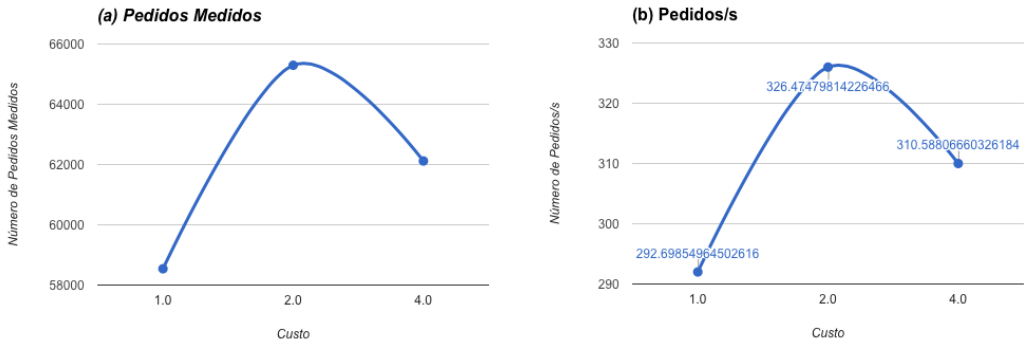

Figura 32: Gráficos correspondente aos valores da tabela

Na análise dos resultados obtidos nas tabelas e gráficos anteriores verificasse que se o custo dos acessos aleatórios for de **2** encontram-se melhores valores de latência média e a 99%. Também o número de pedidos médios por segundo é superior.

De notar, que quando o custo é de **1** os valores são muito inferiores aos desejados.

Conclui-se então que se diminuir o custo dos acessos aleatórios para **2** irá beneficiar a execução do *benchmark*. Isto acontece porque a escolha dos algoritmos vai passar a optar em algumas partes da *query* por usar acessos aleatórios em vez dos, preteridos, acessos sequenciais.

4.7 Configuração do *shared_buffers* com o *effective_cache_size*

Escolhida a melhor configuração no *shared_buffers*, o intervalo [1024-2048]MB de memória. No *effective_cache_size* decidiu-se usar o melhor resultado da configuração anterior, 1024MB, e testar, também, os valores de memória conservadora e agressiva definidos anteriormente, 4096MB e 6144MB, respectivamente.

Os resultados obtidos para esta configuração são os seguintes:

# tamanho	# pedidos	pedidos/s	lat. perct. média	lat. perct. 99
1024MB	69015	345.073418193803	1.02032663525321	1.094924
4096MB	67377	336.882924828134	1.02042478753877	1.095611
6144MB	65443	327.214804387718	1.02116393962685	1.092516

Tabela 17: Resultados obtidos para o valor de 1024MB de *shared_buffers*

# tamanho	# pedidos	pedidos/s	lat. perct. média	lat. perct. 99
1024MB	74724	373.619008187245	1.01803675496494	1.088794
4096MB	66985	334.923957423586	1.01998354016571	1.112503
6144MB	69708	348.538465057512	1.01924645791014	1.095748

Tabela 18: Resultados obtidos para o valor de 2048MB de *shared_buffers*

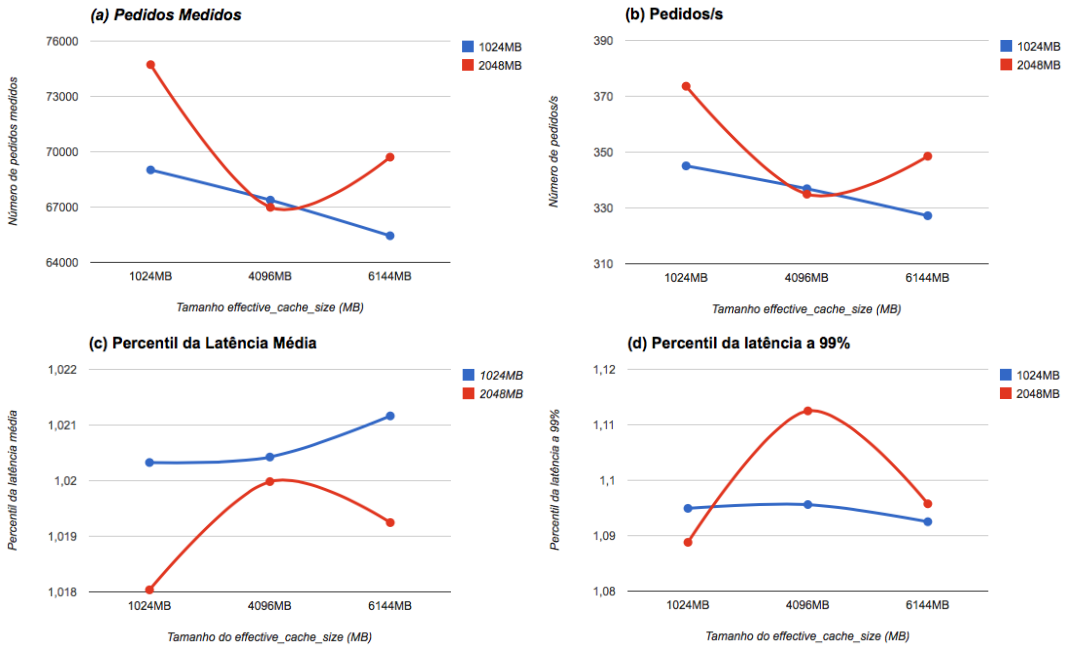

Figura 33: Gráficos corresponde às variações do *shared_buffers* para cada *effective_cache_size*

Desta vez, conseguiu-se perceber que a configuração mais agressiva do *effective_cache_size* (**6144MB**) teve melhores resultados dos que nos resultados indivuais do parâmetro.

Conclui-se facilmente que a melhor configuração obtida foi de **2048MB** de *shared_buffers* e de **1024MB** de *effective_cache_size*. Através da análise dos gráficos confirmasse que se obteve mais pedidos por segundo e o percentil das latências (média e a 99%) são os mais baisxos. Esta configuração será mantida para a próxima configuração.

Desta forma, percebe-se que ter muita quantidade de memória RAM disponível não significa que se vai obter resultados com qualidade superior, antes pelo contrário, como provado neste

benchmark.

4.8 Configuração anterior com o *checkpoint_segments*

Parâmetro	Valor
shared_buffers	2048MB
effective_cache_size	1024MB

Para os melhores resultados obtidos na configuração individual do *checkpoint_segments*, 3 e 9 segmentos, vamos testar estes valores com a configuração definida na tabela anterior.

# segmentos	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
3Segmentos	74724	373.619008187245	1.01803675496494	1.088794
9Segmentos	75410	377.048437684716	1.01657169486805	1.085938

Tabela 19: Resultados com variações do número de segmentos

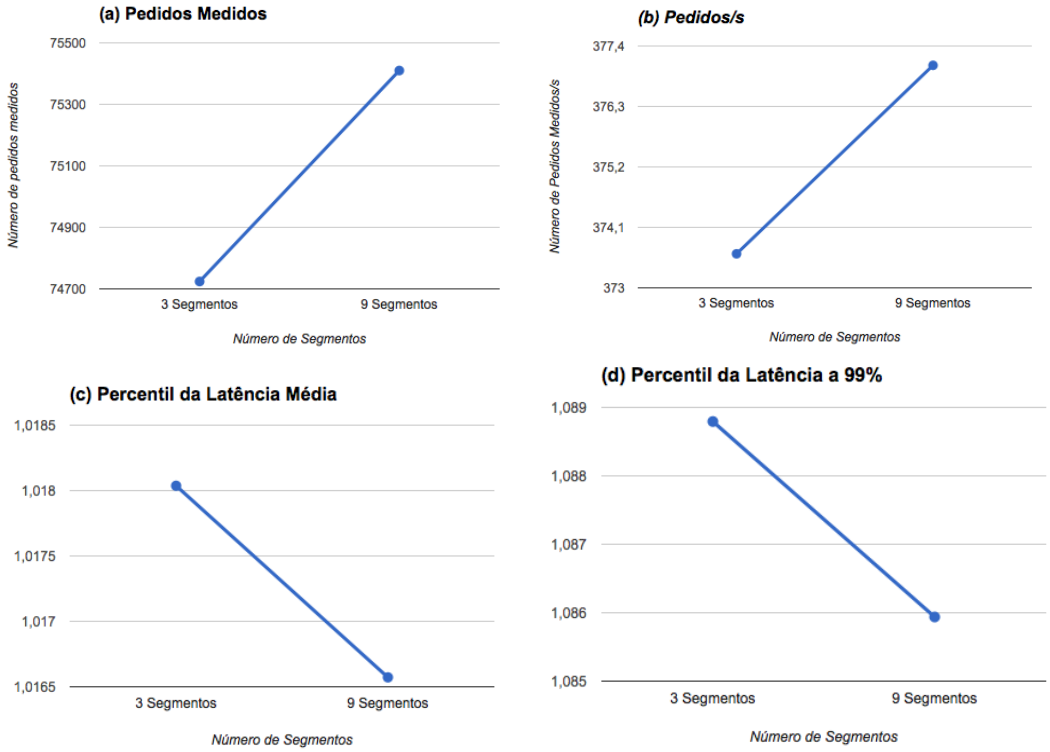

Figura 34: Gráficos corresponde às variações das configurações anteriores com *checkpoint_segments*

Como esperado o aumento do número de *checkpoint_segments* permitiu aumentar o número de transações medidas e diminuir as latências (média e a 99%).

Assim para a próxima configuração será usado um *checkpoint_segments* de 9 segmentos. De notar que apesar deste aumento significativo dos resultados o tempo de recuperação da Base de Dados será mais lento e que ainda que o número de segmentos tenha aumentado ainda é possível notar que em determinados momentos os *checkpoints* são ainda muito frequentes. Como tal, poderia-se aumentar o número de segmentos, mas poderia causar alguns problemas de desempenho no *benchmark*.

4.9 Configuração anterior com o *autovacuum_naptime*

Parâmetro	Valor
shared_buffers	2048MB
effective_cache_size	1024MB
checkpoint_segments	9 Segmentos

Como na configuração individual do *autovacuum_naptime* não se obteve resultados razoáveis para 2 e 3 minutos, decidiu-se para este teste efetuar novos tempos de execução do *autovacuum*. Assim vamos usar 30, 60 (predefinido) e 90 segundos para a configuração deste *benchmark*.

Com os valores definidos na tabela anterior mais os novos parâmetros obteve-se os seguintes resultados, representados na seguinte tabela e nos próximos gráficos.

# tempos	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
30Segundos	76469	382.342824656676	1.01569113987367	1.088295
60Segundos	75410	377.048437684716	1.01657169486805	1.085938
90Segundos	74100	370.498798459429	1.0184869048448	1.089411

Tabela 20: Resultados obtidos para diferentes tempos de *autovacuum_naptime*

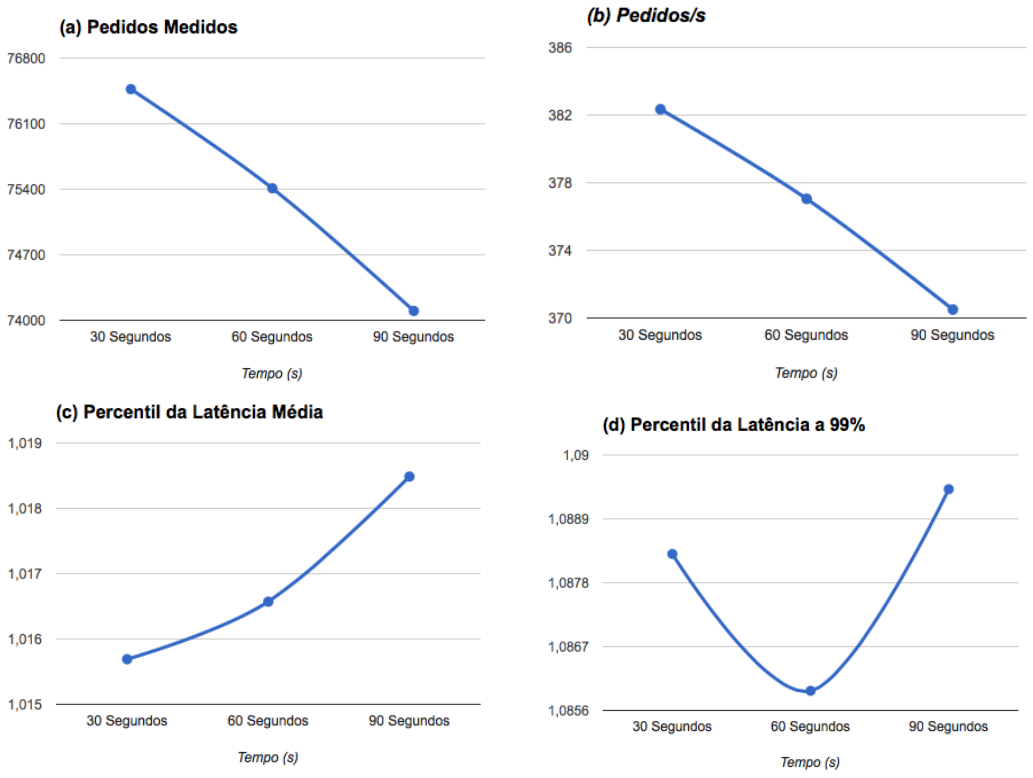

Figura 35: Corresponde às variações das configurações anteriores com *autovacuum_naptime*

A partir dos resultados acima mostrados somos capazes de verificar que o melhor resultado aconteceu quando o tempo de **30** segundos estava definido. Com o aumento do valor do *autovacuum_naptime* nota-se que os resultados pioraram de forma significativa.

Como tal na próxima configuração o valor do *autovacuum_naptime* será de **30** segundos.

4.10 Configuração anterior com o *random_page_cost*

Parâmetro	Valor
shared_buffers	2048MB
effective_cache_size	1024MB
checkpoint_segments	9 Segmentos
autovacuum_naptime	30 Segundos

Como na configuração individual deste parâmetro o melhor resultado obtido foi um custo de 2, decidiu-se para este novo *benchmark* fazer usar este valor e também tornar novamente iguais os valores das acessos sequenciais e das acessos aleatórias, de forma a perceber como iriam variar os resultados.

# custo	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
Custo1	69864	349.318453834434	1.01832794880053	1.091778
Custo2	77165	385.822548996668	1.01697276913108	1.085727
Custo4	76469	382.342824656676	1.01569113987367	1.088295

Tabela 21: Resultados obtidos para diferentes custos do *random_page_cost*

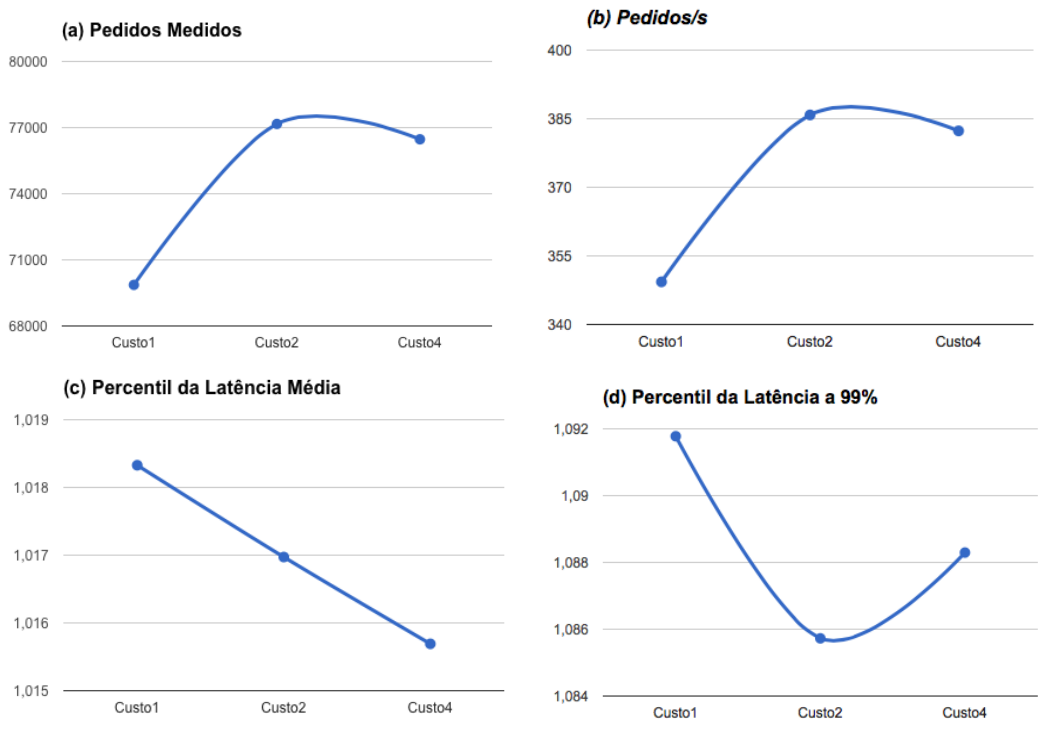

Figura 36: Corresponde às variações das configurações anteriores com *random_page_cost*

Da análise dos valores resultantes da tabela e dos gráficos anteriores, verifica-se, novamente, que o custo **2** apresenta os melhores resultados, tanto ao nível dos pedidos como das latências. Já o custo **1** reproduz o pior resultado. Com este valor mostra que o *PostgreSQL* altera os tipos de algoritmos escolhidos, passando de preterir alguns índices sequenciais a aleatórios fazendo atingir valores inviavéis para a configuração.

Com o custo igual a **2** no *random_page_cost*, significa que é uma configuração equilibrada na escolha de determinados algoritmos.

4.11 Configuração anterior com o *work_mem*

Parâmetro	Valor
shared_buffers	2048MB
effective_cache_size	1024MB
checkpoint_segments	9 Segmentos
autovacuum_naptime	30 Segundos
random_page_cost	2

Anteriormente, o melhor resultado para este parâmetro situava-se no intervalo [8-32]MB. Para este teste não seguímos diretamente esse intervalo. Foram realizados testes para 16MB, 32MB e 64MB, isto aconteceu, porque se percebeu que poderíamos atingir um resultado melhor para esta configuração.

Seguem-se então os resultados obtidos:

# tamanho	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
1MB	77165	385.822548996668	1.01697276913108	1.085727
16MB	78158	390.788969075252	1.0171380589447	1.085266
32MB	79480	397.398960072492	1.01716957381731	1.087713
64MB	80278	401.388611052913	1.01766492785072	1.090232

Tabela 22: Resultados obtidos para diferentes tamanhos do *work_mem*

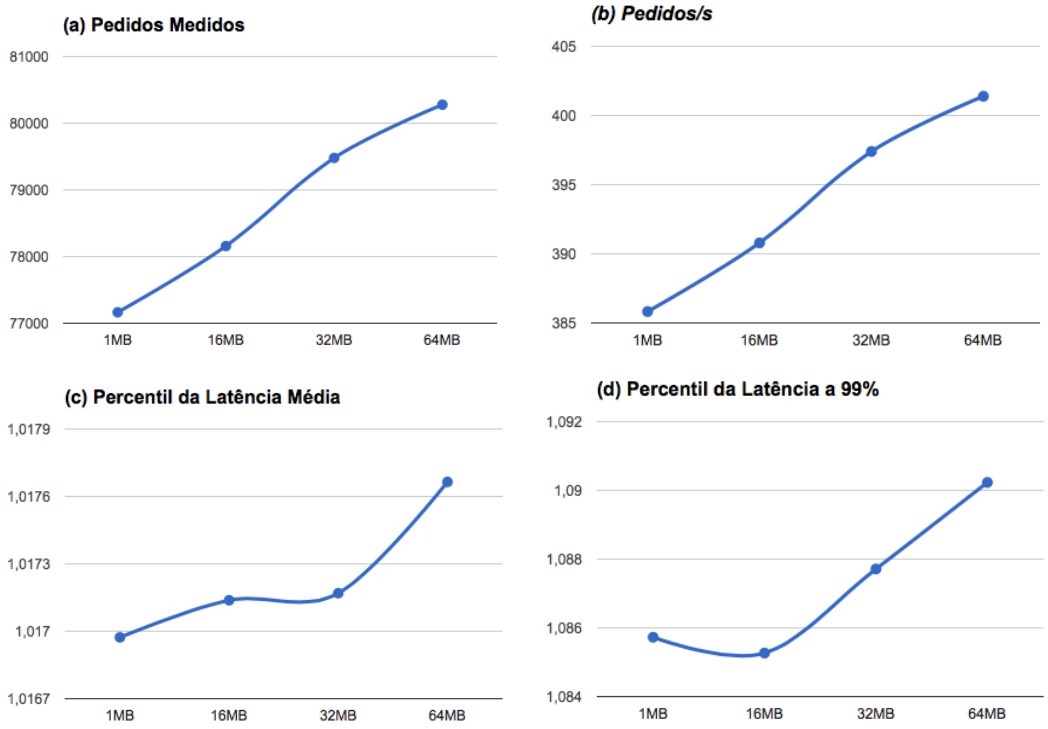

Figura 37: Corresponde às variações das configurações anteriores com *work_mem*

Nos resultados gerados verificámos que para **1MB** as latências têm valores inferiores às restantes no entanto o número de transações é inferior. Os tamanhos **16MB**, **32MB** e **64MB** apresentam um valor significativamente mais elevado de latência, mas conseguem processar mais transações por segundo.

Escolhemos o valor de **64MB** como o melhor resultado deste *benchmark* porque apesar da latência aumentar significativamente, o nível de pedidos por segundos é muito superior. De notar, na imagem mais abaixo, que o tempo de execução de cada *query*, indivualmente, diminuiu o seu tempo de execução.

4.12 Configuração final

Parâmetro	Valor
shared_buffers	2048MB
effective_cache_size	1024MB
checkpoint_segments	9 Segmentos
autovacuum_naptime	30 Segundos
random_page_cost	2
work_mem	64MB

Tabela 23: Valores recomendados para a configuração do *PostgreSQL*

Agora é possível comparar os resultados entre a configuração incialmente usada, a predefinida no *PostgrSQL*, e a que conseguímos obter por alteração de alguns critérios. Assim na próxima tabela vai ser possível verificar estes resultados.

# configuração	# pedidos	pedidos/s	lat. média (s)	lat. perct. 99 (s)
Default	62118	310.5880666032618	1.0220994499090980	1.130365
Final	80278	401.388611052913	1.01766492785072	1.090232

Tabela 24: Configuração inicial vs. Configuração final

Conclui-se que as evoluções que tem vindo a ocorrer com a alteração de certos parâmetros tem feito evoluir positivamente o desempenho do *benchmark*. Notasse que o número de transações por segundo aumentou e que a latência entre estas transações diminuiu significativamente.

Segue-se ainda um gráfico que mostra o tempo que cada *query* em média demorou a executar e o número de pedidos por segundo.

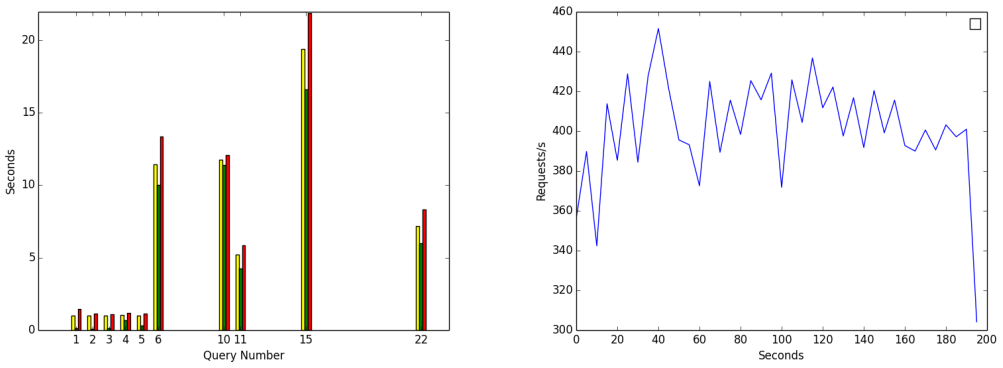

Figura 38: Tempo de execução de cada *query* e o número de pedidos/s

5 Conclusão

Neste trabalho foi essencial a análise e percepção dos parâmetros de configuração do *PostgreSQL*, para que se retirem desempenhos cada mais superiores e aceitáveis por parte da base de dados. Ainda assim, para o *benchmark* atual poderiam ainda ser otimizados mais alguns parâmetros, o *default_statistics_target*, *fsync* e *commit_delay*, por exemplo. De facto, deve-se ter consiência que a alteração dos parâmetros não se aplica a todo o tipo de situações, isto é, depende, também, do *hardware* utilizado e da quantidade de carga que é gerada em determindados momentos de tempo.

Para este *benchmarking* foram utilizados parâmetros de configuração relativamente bons em termos de desempenho. Apesar de nesta máquina o resultado ter sido positivo não implica que noutra máquina estes valores definidos façam com que o desempenho seja superior, daí ser sempre necessário configurar os parâmetros à medida do *hardware*.

6 Anexos

6.1 Plano de execução da *query* 5

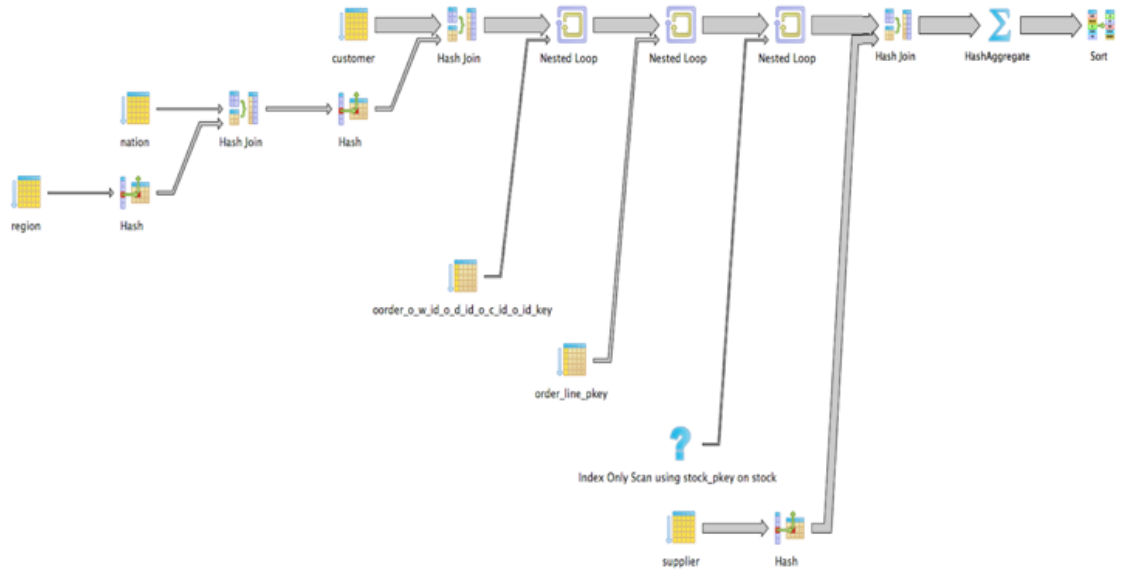

Figura 39: Plano de execução da *query 5*

6.2 Plano de execução da *query* 6

```
Aggregate  (cost=62884.88..62884.89 rows=1 width=4) (actual time=1389.263..1389.263 rows=1 loops=1)
  ->  Seq Scan on order_line  (cost=0.00..59716.42 rows=1267386 width=4) (actual time=0.068..928.630 rows=1266486 loops=1)
        Filter: ((ol_delivery_d >= '1999-01-01 00:00:00'::timestamp without time zone) AND (ol_delivery_d < '2020-01-01 00:
        Rows Removed by Filter: 404885
Total runtime: 1389.314 ms
```

Figura 40: Plano de execução da *query 6*

6.3 Plano de execução da *query* 10

```
Sort  (cost=362648.14..363074.76 rows=170651 width=78) (actual time=10085.241..10157.261 rows=120000 loops=1)
  Sort Key: (sum(order_line.ol_amount))
  Sort Method: external merge  Disk: 10536kB
  ->  GroupAggregate  (cost=327956.07..332648.98 rows=170651 width=78) (actual time=7597.692..9721.575 rows=120000 loops=1)
        ->  Sort  (cost=327956.07..328382.70 rows=170651 width=78) (actual time=7597.657..8770.758 rows=1266486 loops=1)
              Sort Key: customer.c_id, customer.c_last, customer.c_city, customer.c_phone, nation.n_name
              Sort Method: external merge  Disk: 110680kB
              ->  Hash Join  (cost=40128.54..297956.91 rows=170651 width=78) (actual time=822.893..3570.937 rows=1266486 loops=1)
                    Hash Cond: ((order_line.ol_w_id = customer.c_w_id) AND (order_line.ol_d_id = customer.c_d_id) AND (order_line.ol_o_id = oor
                    Join Filter: (oorder.o_entry_d <= order_line.ol_delivery_d)
                    Rows Removed by Join Filter: 404885
                    ->  Seq Scan on order_line  (cost=0.00..43002.71 rows=1671371 width=24) (actual time=0.066..540.657 rows=1671371 loops=1)
                    ->  Hash  (cost=38413.76..38413.76 rows=51759 width=102) (actual time=822.202..822.202 rows=167265 loops=1)
                          Buckets: 1024  Batches: 32 (originally 8)  Memory Usage: 1025kB
                          ->  Hash Join  (cost=14862.40..38413.76 rows=51759 width=102) (actual time=306.163..703.798 rows=167265 loops=1)
                                Hash Cond: ((oorder.o_c_id = customer.c_id) AND (oorder.o_w_id = customer.c_w_id) AND (oorder.o_d_id = customer
                                ->  Seq Scan on oorder  (cost=0.00..3630.81 rows=167249 width=24) (actual time=0.015..68.334 rows=167265 loops=
                                      Filter: (o_entry_d >= '2007-01-02 00:00:00'::timestamp without time zone)
                                ->  Hash  (cost=13702.40..13702.40 rows=37200 width=82) (actual time=305.678..305.678 rows=120000 loops=1)
                                      Buckets: 1024  Batches: 16 (originally 8)  Memory Usage: 1025kB
                                      ->  Hash Join  (cost=3.40..13702.40 rows=37200 width=82) (actual time=0.105..232.941 rows=120000 loops=1)
                                            Hash Cond: (ascii(substr((customer.c_state)::text, 1, 1)) = nation.n_nationkey)
                                            ->  Seq Scan on customer  (cost=0.00..12427.00 rows=120000 width=59) (actual time=0.027..64.661 row
                                            ->  Hash  (cost=2.62..2.62 rows=62 width=30) (actual time=0.045..0.045 rows=62 loops=1)
                                                  Buckets: 1024  Batches: 1  Memory Usage: 4kB
                                                  ->  Seq Scan on nation  (cost=0.00..2.62 rows=62 width=30) (actual time=0.012..0.025 rows=62
Total runtime: 10187.030 ms
```

Figura 41: Plano de execução da *query 10*