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, 21 de Dezembro de 2014

Conteúdo

1 Introdução

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

2.1 *Serializable* como Método de Isolamento

# 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

2.2 *Repeatable Read* como Método de Isolamento

# 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

2.3 *Read Uncommitted* como Método de Isolamento

# 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

2.4 *Read Committed* como Método de Isolamento

# 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

2.5 Comparação dos Métodos de Isolamento e Conclusões Finais

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 maior 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 do CHBenchmark e assim sucessivamente.

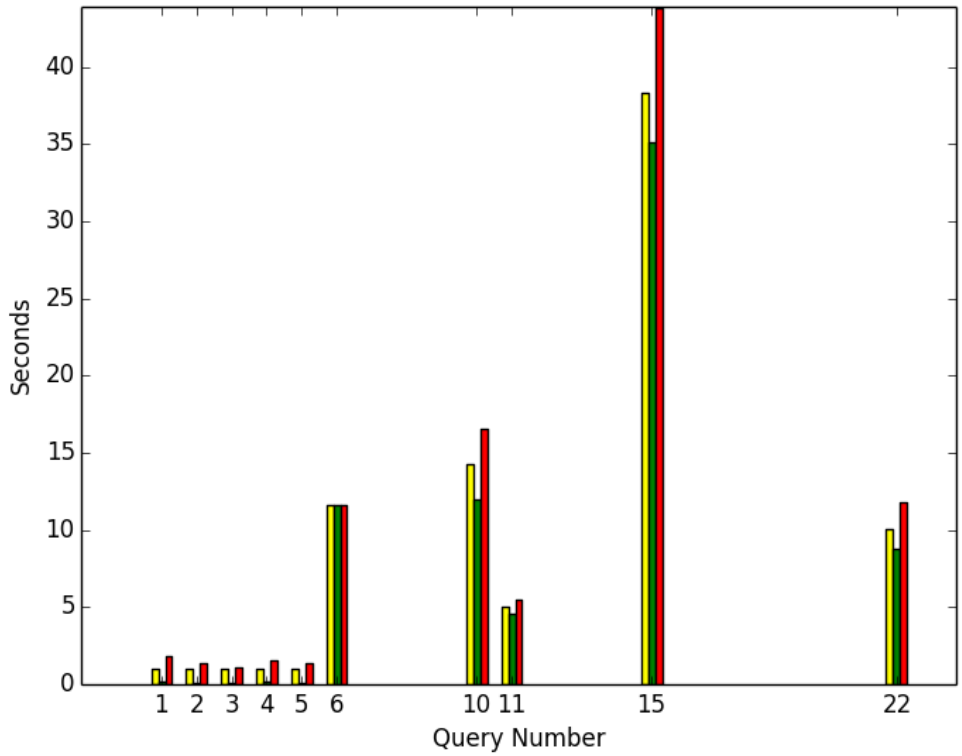

Figura 1: Tempos de execução para cada *query*

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 e quantidade mais a contagem total de todos esses *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
     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 uma 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 listagem o seu país, alguns detalhes deles e a quantidade de dinheiro que eles têm usado para tomar 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 fosse tomadas 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 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.

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 2: Plano de execução da *query 1* depois 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 3: Plano de execução da *query 1* antes da otimização

ADICIONAR AQUI UMA CONCLUSÃO SOBRE OS RESULTADOS OBTIDOS

3.2 Desempenho *Query 17*

Na *query 17* foi possível 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 4: Plano de execução da *query 17* antes da otimização

ADICIONAR AQUI UMA CONCLUSÃO SOBRE OS RESULTADOS OBTIDOS

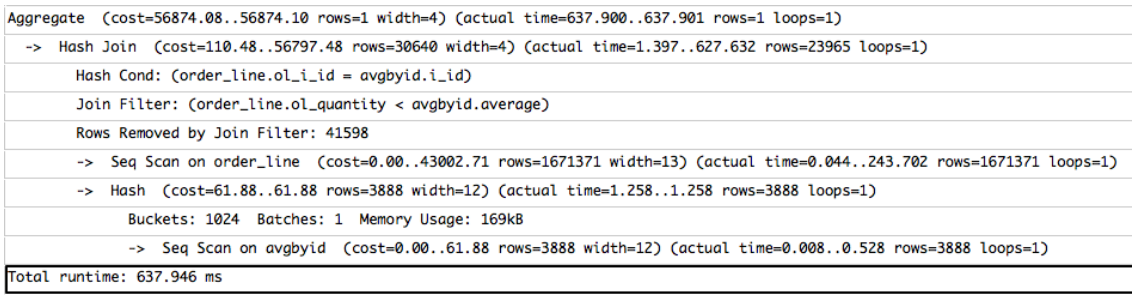

Figura 5: Plano de execução da *query 17* depois da otimização

3.3 Desempenho das restantes *queries*

Analisando a *query 5* pode-se ver 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

4.1 Configuração do *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 queries.

Inicialmente, podia-se prever que quanto maior fosse a quantidade de memória dedicada melhor seriam os resultados de desempenho esperados. 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 segue 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 9: Resultados obtidos para X (X) armazéns

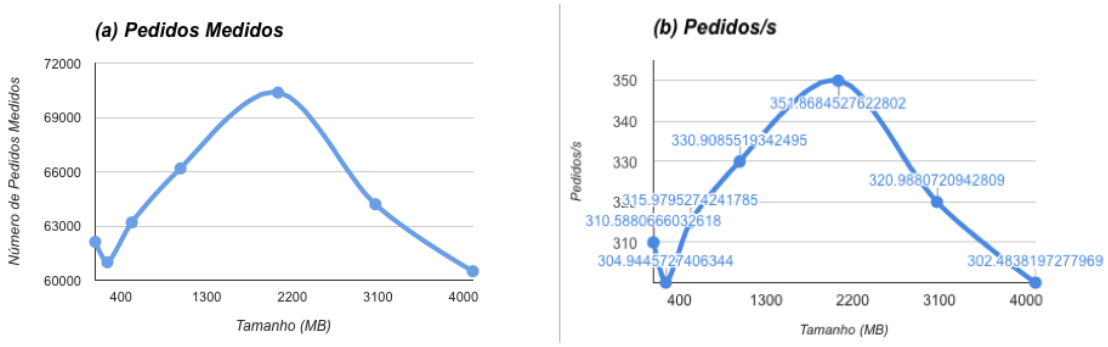

Figura 6: Gráfico correspondente aos valores da tabela anterior

Ao analisar-mos 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 de 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 10: Resultados obtidos para X (X) armazéns

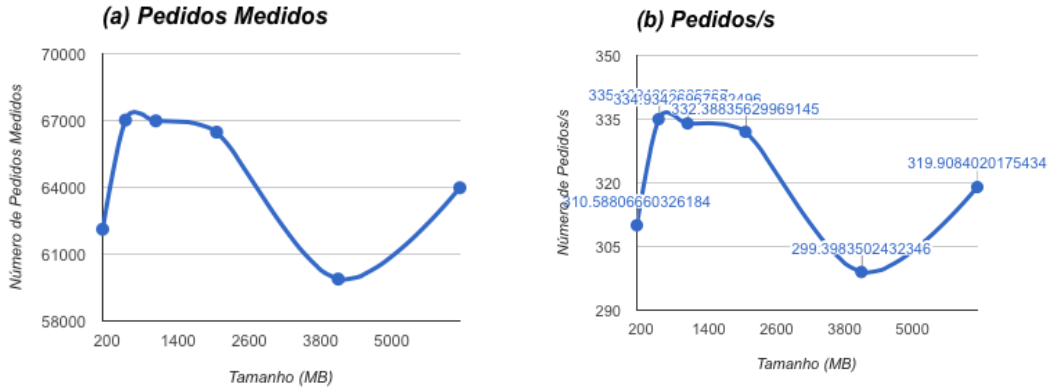

Figura 7: Gráfico correspondente aos valores da tabela anterior

Ao analisar-se os valores obtidos identificasse que na configuração mais conservadora (**4096MB**) e agressiva (**6144MB**) não se obteve os melhores resultados. O número de pedidos medidos foi muito inferior a outros valores assim como a latência a 99%, a latência média praticamente se encontra 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 e pedidos por segundo e um ligeiro aumento nas latências do teste, que influência negativamente o resultado. Com o parâmetro a **2048MB** mantém-se 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 praticamente 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.

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 11: Resultados obtidos para o *checkpoint_segments*.

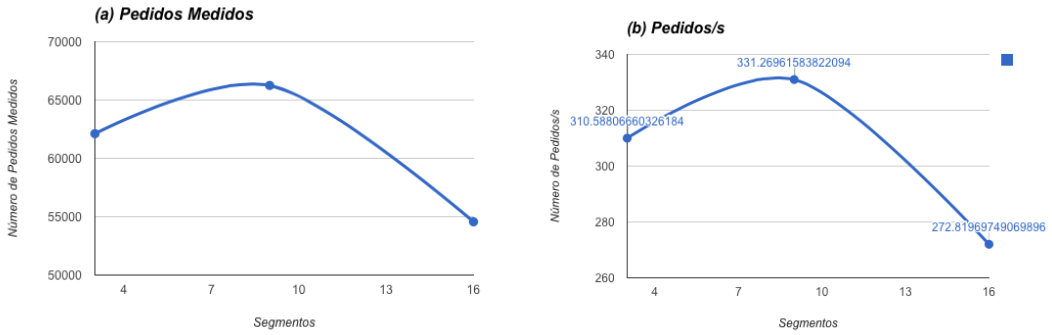

Figura 8: Gráfico correspondente aos valores da tabela anterior

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 dimininuiram abruptamente mas na latência a 99% obteve-se o melhor resultado, ainda que na latência média manteve-se perto dos resultados anteriores. Estes resultados podem-se dever à recuperação mais lenta da base de dados.

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

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*, isto é, só ocorre um novo depois de ter passado o tempo de execução do anterior.

Para 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 12: Resultados obtidos para a configuração do *autovacuum_naptime*

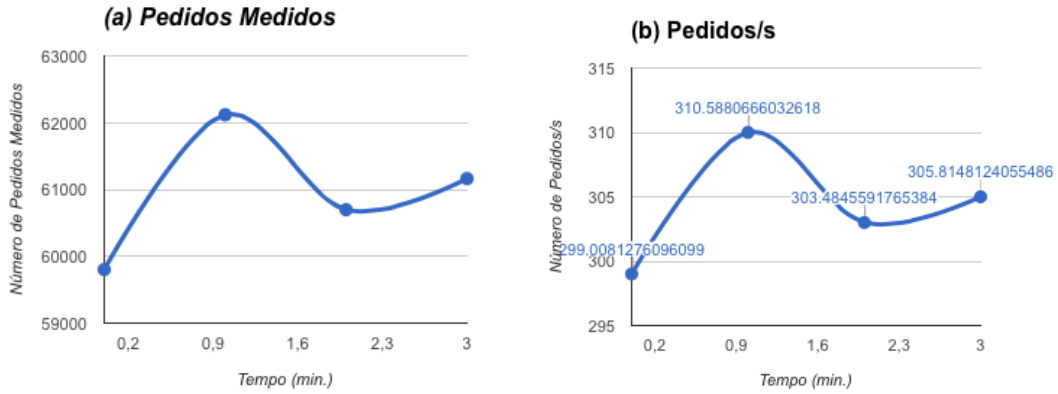

Figura 9: Gráficos correspondente aos valores da tabela

Dos resultados apresentados, verifica-se que, os 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 fez que os resultados obtidos fossem mais razoáveis com os que se pode comparar. De facto, ao haver com mais frequência o ***autovacuum*** faz com que em cada operação haja menos dados para limpar.

Assim, decidiu-se escolhar para esta configuração o melhor resultado o 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 *queries* 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 13: Resultados obtidos para X (X) armazéns

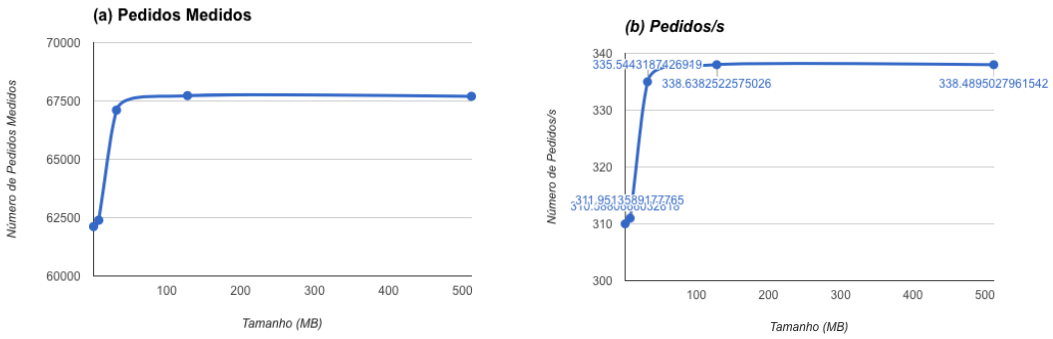

Figura 10: Gráficos correspondente aos valores da tabela

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

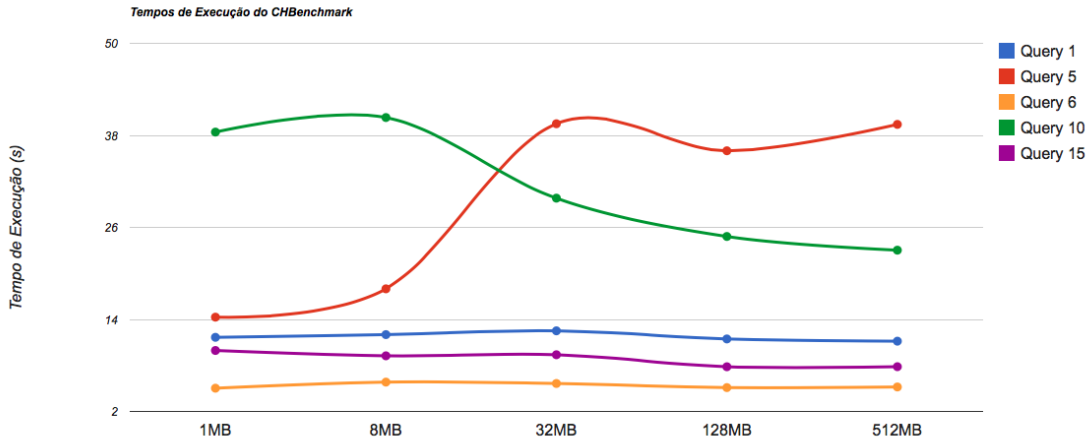

Figura 11: Gráfico com os valores de execução para cada query com diferentes valores de memória

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 17* 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.

Os resultados das *queries 1, 6 e 17* os resultados praticamente não surtiram grande variação nos resultados.

Caso fosse necessário escolher a melhor configuração para este parâmetro, este ficaria 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 deste valor.

Finalmente, se analisar-mos 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 14: Resultados obtidos para 3 tipos diferentes de custos

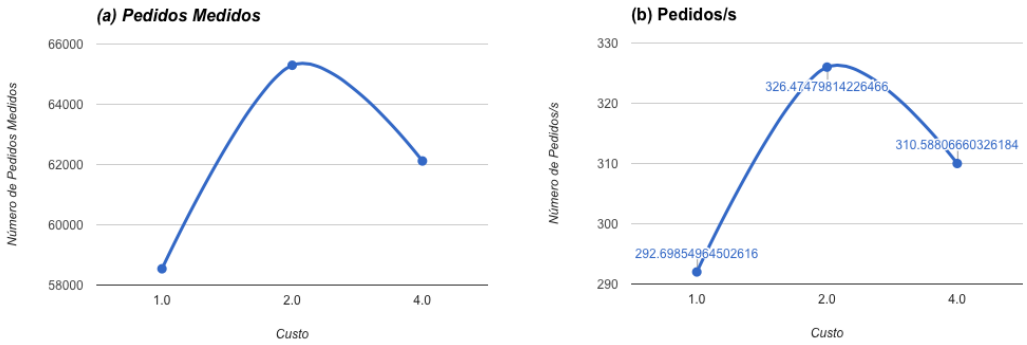

Figura 12: Gráficos correspondente aos valores da tabela

Da análise dos resultados obtidos nas tabelas e gráficos verifica-se 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 medios por tempo é superior ao definido por defeito na configuração inicial.

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 do algoritmo 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*

5 Anexos

5.1 Plano de execução da *query* 5

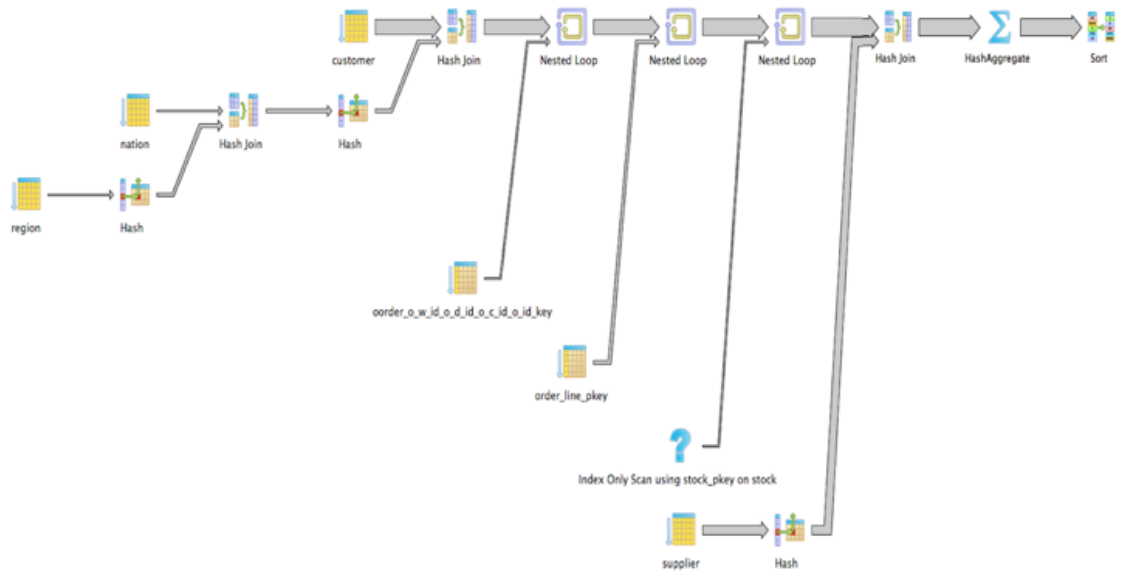

Figura 13: Plano de execução da *query 5*

5.2 Plano de execução da *query* 5

```
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 14: Plano de execução da *query 6*

5.3 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 15: Plano de execução da *query 10*