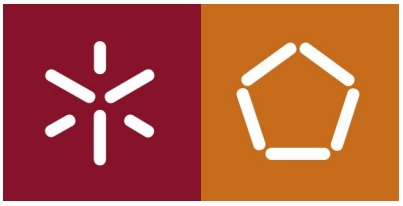


Universidade do Minho
mestrado integrado em engenharia informática

Engenharia de Aplicações

(1º Semestre / 4º Ano)

TP - Administração de Bases de Dados

Diogo Braga (a82547)
João Silva (a82005)
Nelson Sousa (a82053)

Conteúdo

1 Introdução

O presente relatório é referente ao trabalho prático da unidade curricular de Administração de Bases de Dados, integrada no perfil de especialização em Engenharia de Aplicações.

Este projeto tem como principal objetivo a conceção e desenvolvimento de práticas para otimizar o desempenho do *benchmark* TPC-C. Este *benchmark* simula um sistema de bases de dados de uma cadeia de lojas que suporta a operação diária de gestão de vendas e stocks.

As métricas escolhidas para desenvolvimento de projeto são, especificamente, a maximização do débito e a minimização do tempo de resposta das operações analíticas.

No final, é apresentada uma proposta de configuração, fundamentada nas conclusões retiradas dos passos intermédios.

1.1 Estrutura do documento

O documento encontra-se organizado da forma descrita de seguida. Após a introdução, na secção **??** será feita uma contextualização sobre o projeto. De seguida, na secção 2, será detalhada uma configuração de referência, em termos do número de warehouses, clientes e hardware escolhidos. Na secção 3 são apresentadas as otimizações de desempenho estabelecidas, tanto a nível da carga transacional como das interrogações analíticas. Na secção **??** é apresentada a configuração final estabelecida pelo grupo. Por fim, conclui-se apresentando a perspetiva de trabalho futuro.

2 Configuração de referência

Antes que existisse qualquer otimização foi necessário que se encontrasse uma configuração de referência. Esta configuração compõe o número de warehouses, assim como o número de clientes e ainda o hardware a usar nas fases seguintes.

2.1 Hardware

A escolha do tipo da máquina a utilizar foi relativamente simples uma vez que estávamos limitados por um *plafond* e não deveríamos usar uma máquina com características demasiado boas (para não gastar o *plafond* rapidamente), assim como não deveríamos utilizar uma máquina com as piores características possíveis (para que os resultados dos testes não ficassem aquém do que seria espectável).

Assim sendo, optámos por escolher uma máquina com:

- 2 CPUs, para que fosse possível existir concorrência nos testes desenvolvidos;
- 8 GB de RAM, para que existisse memória capaz de albergar uma quantidade de dados razoável;
- Disco de arranque com 100 GB de SSD, para que não fosse necessário anexar discos externos à máquina.

Para que fossem realizados mais testes em paralelo foram criadas 6 instâncias de máquinas virtuais com estas características.

2.2 Warehouses

Para escolha do número de warehouses, a análise centrou-se nos resultados obtidos para o **tempo de resposta** e para o **throughput**. O processo de recolha destes dados baseou-se em registar as duas métricas acima mencionadas, enquanto era incrementado o número de warehouses.

Desta forma, conseguimos estabelecer e visualizar uma linha de evolução para o tempo de resposta assim como para o throughput, ambos em função do crescimento do número de warehouses.

O tempo de teste estabelecido foi de 10 minutos sendo que, nesta fase, o número de clientes por warehouse se manteve o pré-estabelecido, ou seja, 10.

A decisão para o **número de warehouses** estabelecido teve em conta a conjugação de dois parâmetros:

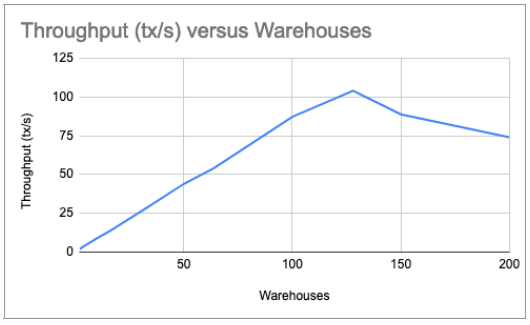
- maiores valores atingidos no throughput;
- aumentos consideráveis existentes no tempo de resposta;

De seguida, é apresentada uma tabela com os valores obtidos para as métricas estabelecidas, em função do crescimento do número de warehouses.

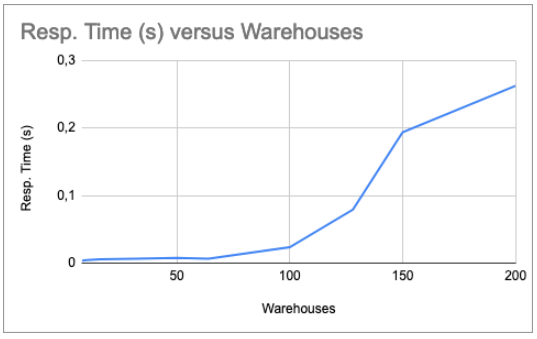
Warehouses	Resp. Time (s)	Throughput (tx/s)	VM
1	0,006	0,86	1
2	0,005	1,72	2
4	0,004	3,63	3
8	0,003	7,04	4
10	0,004	8,68	3
16	0,005	13,52	1
32	0,006	27,68	2
50	0,007	43,89	3
64	0,006	54,22	4
100	0,023	87,28	4
128	0,079	104,19	3
150	0,194	88,88	2
200	0,263	74	1

Figura 1: Resultados para análise do número de warehouses

De modo a facilitar a escolha do número de warehouses estabelecido, foram criados dois gráficos com as linhas de evolução das duas métricas definidas.



(a) Resultados para análise do throughput



(b) Resultados para análise do tempo de resposta

Figura 2: Métricas

Após análise dos resultados e gráficos obtidos, o grupo decidiu escolher **128** como o **número de warehouses** indicado para a configuração de referência. Tal escolha é fundamentada na conjugação dos seguintes fatores:

- o throughput para 128 warehouses está entre os valores mais elevados, neste caso, é mesmo o maior com cerca de 104 transações/segundo;

- o tempo de resposta, também nesse valor (128), possui um considerável aumento na linha do gráfico apresentado na figura **??**;

2.3 Clientes por Warehouse

Para escolha do número de clientes por warehouse, a análise centrou-se igualmente nos resultados obtidos para o **tempo de resposta** e para o **throughput**, no entanto, foram também tidos em conta os resultados obtidos para a **percentagem de CPU** usada em cada teste, assim como o **abort rate** correspondente. O processo de recolha destes dados baseou-se em registar as quatro métricas acima mencionadas, enquanto era incrementado o número de clientes por warehouse.

Desta forma, conseguimos estabelecer e visualizar linhas de evolução para as quatro métricas referidas, todas em função do crescimento do número de clientes por warehouse.

O tempo de teste estabelecido manteve-se o de 10 minutos, sendo que o número de warehouses se manteve o atingido anteriormente, ou seja, 128.

A decisão para o **número de clientes por warehouse** estabelecido teve em conta a conjugação de quatro parâmetros:

- maiores valores atingidos no throughput;
- alterações existentes no tempo de resposta;
- valores medianos de utilização de CPU (que rondem os 40% e 60%);
- menores valores de abort rate;

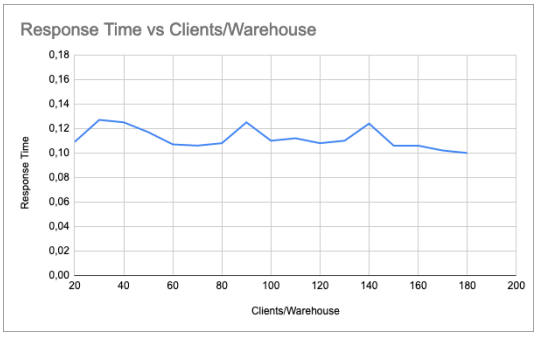
De seguida, é apresentada uma tabela com os valores obtidos para as métricas estabelecidas, em função do crescimento do número de clientes por warehouse.

Clients/Warehouse	Resp. Time (s)	Throughput (tx/s)	CPU%	VM	Abort Rate
10	0,079	104,19	29,29	3, 1	0,018
20	0,109	146,66	34,05	1, 2	0,024
30	0,127	153,3	36,61	2, 3	0,031
40	0,125	156,75	41,15	3, 4	0,029
50	0,117	167,74	38,72	4, 5	0,029
60	0,107	183,53	37,41	1	0,028
70	0,106	183,12	37,47	1	0,026
80	0,108	183,42	39,30	2	0,026
90	0,125	157,47	35,06	3	0,025
100	0,110	177,88	39,04	4	0,026
110	0,112	175,3	37,72	1	0,025
120	0,108	181.92	38,61	2	0,024
130	0,110	178,07	42,70	3	0,024
140	0,124	158,81	38,64	4	0,024
150	0,106	185,81	40,48	1	0,025
160	0,106	181,9	42,13	2	0,026
170	0,102	192,09	38,36	3	0,024
180	0,100	191,97	39,04	4	0,025
500	0,102	189,07	41,97	2	0,042

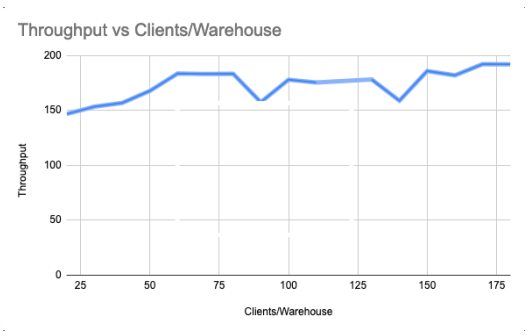
Figura 3: Resultados para análise do número de clientes por warehouse

Os valores obtidos para a média de utilização de CPU foram obtidos com ajuda da ferramenta *iostat* que gera relatórios do sistema. Foi apenas necessário criar uma script para ler os valores do CPU durante a execução dum *run* e fazer a média dos resultados obtidos.

De modo a facilitar a escolha do número de clientes por warehouse estabelecido, para cada uma das métricas definidas foi criado um gráfico com a linha de evolução em função dos clientes por warehouse.



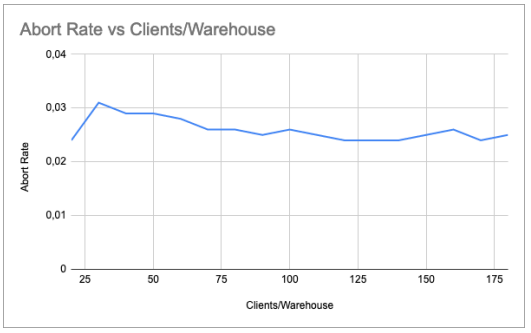
(a) Resultados para análise do tempo de resposta



(b) Resultados para análise do throughput

Figura 4: Métricas

(a) Resultados para análise da percentagem de CPU utilizado



(b) Resultados para análise do abort rate

Figura 5: Métricas

Após análise dos resultados e gráficos obtidos, o grupo decidiu escolher **60** como o **número de clientes por warehouse** indicado para a configuração de referência. Tal escolha é fundamentada na conjugação dos seguintes fatores:

- a percentagem de CPU estabiliza por volta do número escolhido, com um valor de 38% de utilização;
- essa percentagem de CPU já é razoável para efeitos de otimização da performance da base de dados;
- o throughput, também nesse valor, inicia uma estabilização por volta das 183 tx/s;
- a estabilização destes parâmetros pode ser explicada pelo facto do CPU estar à espera de operações em disco, sendo este o gargalo da performance, que implica diretamente que o aumento do número de clientes não altere os parâmetros avaliados;

Além dos fatores referidos, faz sentido fazer referência à monotonia dos valores de abort rate, sendo esta a razão para o parâmetro não influenciar a escolha do valor final.

Na mesma lógica, o tempo de resposta possui também pequenas variações, no entanto, não são significativas para a escolha do valor final.

2.4 Tamanho da Base de Dados

Após encontrar a configuração de referência para a base de dados foi necessário ver o quanto ela ocupava em disco assim como todas as suas tabelas. Esta informação será útil para passos posteriores na otimização do desempenho da base de dados.

Os tamanhos encontram-se na tabela seguinte:

Tabela	Tamanho(GB)
customer	2,380
district	0,000312
history	0,423
item	0,016
new_order	0,108
order_line	5,605
orders	0,594
stock	5,369
warehouse	0,00008
Total	14,495

Figura 6: Tamanhos das tabelas da Base de Dados

3 Otimização do desempenho

3.1 Carga transacional

Nesta secção vamos responder ao requisito 1 do enunciado cujo objetivo é otimizar o desempenho da carga transacional tendo em conta, principalmente, os parâmetros de configuração do PostgreSQL, usando a configuração de referência obtida no capítulo anterior.

3.1.1 Análise dos parâmetros a avaliar

Tendo em conta o ficheiro de configuração do PostgreSQL (*postgresql.conf*), que está dividido em secções pelo tipo de parâmetros e a interferência que cada um tem no funcionamento do SGBD, reparamos que para completar este requisito iríamos ter que alterar parâmetros que residem nas secções dos *Resouce Usage (except WAL)* e *Write-Ahead Log*, sendo que as restantes secções não fariam interferência direta com o desempenho da carga transacional.

Na secção ***Resouce Usage (except WAL)*** seria necessário configurar parâmetros que interferissem com a memória (*shared_buffers*, *work_mem*), até porque foram os estudados nas aulas desta disciplina, sendo desta forma os únicos em que podemos sustentar uma explicação.

Na secção ***Write-Ahead Log*** seria necessário configurar parâmetros que interferissem com os *checkpoints* (*checkpoint_timeout*, *max_wal_size*, *min_wal_size*, etc). Esta subsecção tem o objetivo de garantir integridade de dados, mas acarreta custos (escritas para disco) que interferem com o desempenho da base de dados sendo por isso, alvo de configuração.

3.1.2 Sequência de ações de melhoria de desempenho

Após a análise demos início à configuração dos parâmetros acima falados. A sequência de parâmetros configurados, está feita pela ordem real que o grupo alterou os mesmos.

1º- Shared Buffers

Inicialmente começamos por analisar os *shared_buffers*. Este é um parâmetro que é gerido pelo servidor do PostgreSQL e determina a quantidade de memória dedicada a *caching*.

Devido ao facto da nossa base de dados ocupar cerca de 14.5GB e de termos 128 *warehouses* com 60 clientes por *warehouse*, pareceu-nos que o valor de predefinição do PostgreSQL (128MB) para os Shared Buffers poderia ser baixo, uma vez

que iríamos ter bastante informação de muitos *warehouses* a ser acedida por vários clientes.

Usando a ferramenta *postgresqltuner* percebemos que realmente o *hit rate* poderia ser aumentado, aumentando o valor deste parâmetro.

Após alguns testes obtivemos os seguintes resultados.

Shared Buffers (MB)	Resp. Time (s)	Throughput (tx/s)	Abort Rate	Hit Rate	VM
128	0,118	164,81	0,028	Quite Good	1
1024	0,108	181,26	0,028	Quite Good	2
2048	0,105	184,47	0,027	Quite Good	3
2560	0,109	180,42	0,026	Very Good	1
3072	0,106	184,92	0,026	Very Good	2
3584	0,102	191,13	0,027	Very Good	3

Figura 7: Resultados da variação dos Shared Buffers

Dados os resultados da figura acima reparamos que para um valor de **3584MB** obteve-se um aumento substancial no débito para **191,13 transações/segundo** e um melhoramento no tempo de reposta para **0,102ms**.

Para **3.5GB** de memória nos *Shared Buffers* obtivemos resultados para *hit rates* que podemos considerar muito bons comparando àqueles obtidos inicialmente com a memória de predefinição. Tais resultados podem ser vistos na figura 8.

```
-----  Shared buffer hit rate  -----
[INFO]    shared_buffer_heap_hit_rate: 63.69%
[INFO]    shared_buffer_toast_hit_rate: 93.57%
[INFO]    shared_buffer_tidx_hit_rate: 95.20%
[INFO]    shared_buffer_idx_hit_rate: 98.61%
```

Figura 8: Hit Rate nos Shared Buffers

2º- Checkpoint Timeout

Após mexer num parâmetro do *Resource Usage* percebemos que poderia ser mais interessante configurar os parâmetros relativos o *Write-Ahead Log* mais especificamente nos parâmetros relativos aos *checkpoints*.

O PostgreSQL tem um processo que está encarregue de fazer *checkpoints* de **checkpoint_timeout** segundos em **checkpoint_timeout**. Isto acontece apenas se o **max_wal_size** não estiver prestes a ser excedido. Basicamente um *checkpoint* pode ocorrer nesse intervalo de tempo ou se o **max_wal_size** estiver prestes a ser

excedido, pelo que os valores destas duas componentes podem ser importantes para a avaliação do desempenho da base de dados.

Diminuindo o valor de qualquer um destes destes parâmetros é fácil de perceber que vão ocorrer mais *checkpoints* num dado período de tempo, pelo que irão ser necessárias mais operações de disco. E sendo o acesso a disco um normal gargalo, é normal que quantas mais escritas ocorrerem menos desempenho a base de dados terá.

Inicialmente, fixando o valor do **max_wal_size**, variamos o parâmetro **checkpoint_timeout** para perceber se estava ou não a ter influencia no normal funcionamento da base de dados. Obtivemos os seguintes valores.

checkpoint_timeout (min)	Resp. Time (s)	Throughput (tx/s)	Abort Rate	VM
1	0,102	190,86	0,026	2
2	0,107	183,47	0,029	3
3	0,101	192,07	0,028	3
5 (default)	0,102	191,13	0,027	3
6	0,102	191.2	0,027	2
7	0,103	190.1	0,028	3
8	0,105	186.86	0,028	2
9	0,101	193.80	0,028	3
10 (max)	0,101	193.81	0,028	2

Figura 9: Resultados da variação do Checkpoint Timeout

Por defeito o PostgreSQL dá o valor de 5 minutos a este parâmetro, e neste caso como apenas fazemos testes de 10 minutos, este é o valor máximo que podemos avaliar.

Diminuímos o valor de predefinição na tentativa de ver se existiria alguma degradação no desempenho mas tal não se verificou. Da mesma forma aumentamos o valor de predefinição na tentativa de registar melhorias de desempenho mas tal também não se verificou.

Todos os valores se mantiveram estáveis durante este teste, o que nos leva a concluir que este parâmetro não está a influenciar o desempenho da base de dados. Se o que está a despoletar *checkpoints* não é este parâmetro então resta observar o comportamento da base de dados na modificação dos parâmetros **min_wal_size** e **max_wal_size**.

Concluindo esta fase, mantivemos o valor deste parâmetro inalterado, com o valor de predefinição.

3º- Minimum WAL Size

Antes de passarmos diretamente para a configuração do parâmetro **max_wal_size**, decidimos testar, sabendo *à priori* o resultado, o parâmetro **min_wal_size** para confirmar que este também não estava a ter qualquer influência no desempenho da base de dados.

As nossas previsões estavam corretas como se pode ver na figura abaixo.

min_wal_size	Resp. Time (s)	Throughput (tx/s)	Abort Rate	VM	max_wal_size
40MB	0,102	190,72	0,027	2	1GB
60MB	0,103	190,78	0,027	3	
80MB (default)	0,102	191,13	0,027	3	
100MB	0,104	187,56	0,027	2	
160MB	0,106	184,96	0,026	3	
250MB	0,103	189,95	0,026	3	
500MB	0,103	189,31	0,029	2	
750MB	0,105	187,08	0,027	2	
950MB	0,105	185,77	0,027	3	

Figura 10: Resultados da variação do Min Wal Size

Não houve melhorias de desempenho que fossem significativas pelo que mantivemos o valor de predefinição.

Mais tarde, após a configuração do parâmetro **max_wal_size**, este vai ser novamente analisado de forma a verificar possíveis melhorias de desempenho.

4º- Maximum WAL Size

Como já tínhamos afirmado duas subsecções acima, o valor deste parâmetro é o que está a afetar o desempenho da atual base de dados. Deve-se ao facto do valor ser tão baixo que obriga a escritas mais frequentes para disco, afetanto desta forma negativamente o desempenho da base de dados.

O aumento deste parâmetro acarreta alguns inconvenientes. Uma vez que se se aumenta este valor, em caso de recuperação esta será mais custosa pois existirá uma maior quantidade de logs a terem que ser recuperados. Neste caso meramente académico, estamos a correr testes de 10 minutos pelo que não há nem grande probabilidade de falha, nem a quantidade de dados gerada será tão grande para que seja sentido o impacto negativo do aumento deste parâmetro.

O PostgreSQL mantém ficheiros de log com tamanho de **16MB**, por isso o aumento/dimuição deste parâmetro determina a quantidade de ficheiros de log a serem mantidos antes de toda a informação que reside neles ser passada para disco completando um *checkpoint*.

max_wal_size	Resp. Time (s)	Throughput (tx/s)	Abort Rate	VM	min_wal_size
1GB (default)	0,102	191,13	0,027	3	80MB
2GB	0,065	302,05	0,027	3	
2.5GB	0,057	343,36	0,025	2	
3GB	0,05	387,14	0,039	2	
3.5GB	0,055	351	0,046	3	
4GB	0,058	295,57	0,027	2	

Figura 11: Resultados da variação do Max Wal Size

Tal como esperado, o aumento do valor deste parâmetro melhorou imenso o desempenho, traduzindo um aumento de quase **200 transações/segundo** em relação ao valor tido como referência.

Desta forma a configuração passa a ter um tempo médio de resposta de **0,05ms** e um débito de **387,14 transações/segundo**.

5º- Minimum WAL Size

Após configurar o parâmetro anterior, poderia ser interessante voltar a mudar os valores deste parâmetro uma vez que existe um maior alcance de valores possíveis. Seria de esperar que, novamente, os valores deste parâmetro não interferissem com o normal desempenho da base de dados uma vez que o parâmetro que tem influência direta no desempenho é o **max_wal_size**.

Após alguns testes verificamos o anteriormente previsto. Os valores mantém-se sem variações significativas como mostra a figura abaixo.

min_wal_size	Resp. Time (s)	Throughput (tx/s)	Abort Rate	VM	max_wal_size
80MB (default)	0,05	387,14	0,039	2	3GB
1GB	0,049	383,3	0,037	2	
2GB	0,05	385,35	0,038	3	
3GB	0,05	385,56	0,04	3	

Figura 12: Resultados da variação do Min Wal Size

Pelo que nos cabe tomar uma decisão sobre este valor pois ele terá impacto quando a base de dados estiver em períodos mínimos de carga. Isto é, quando há poucas transações a ocorrerem na base de dados, o **max_wal_size** não será atingido pelo que o parâmetro que despoletará os *checkpoints* será o **checkpoint_timeout**.

Quando este *timeout* é atingido sem que seja excedido o **max_wal_size**, o PostgreSQL avalia se o parâmetro **min_wal_size** foi ou não excedido. Em caso afirmativo dá-se início a um *checkpoint*, mas em caso contrário esse *checkpoint* é passado à frente reutilizando os logs até então criados e escrevendo para memória apenas quando for estritamente necessário.

Analisando o âmbito deste projeto e tendo em conta o programa de *Benchmarking* utilizado, sabemos que não existirão estes períodos de carga baixos, pelo que este parâmetro não pode ser devidamente avaliado. Mas numa situação hipotética onde existissem baixas na carga, o mesmo se verificaria no tempo de resposta da base de dados que seria melhorada. Neste caso o facto de se escrever mais frequentemente para disco não seria um gargalo, uma vez que a atividade dos utilizadores não exigiria ao sistema demasiadas operações de disco.

Desta forma, achamos que é legítimo e algo lógico manter o valor do parâmetro **min_wal_size**, deixá-lo predefinido (80MB). Em períodos de picos de carga o que influenciará o desempenho será o **max_wal_size**, e pelo contrário, em períodos de baixa de carga o que influenciará a ocorrência de *checkpoints* serão o **min_wal_size** e o **checkpoint_timeout**.

6º- Checkpoint Completion Target

O próximo parâmetro avaliado foi o **checkpoint_completion_target**. Este parâmetro lida com o balanceamento de escritas para disco no período entre duas ocorrências de *checkpoints*. A taxa de I/O é ajustada para que o *checkpoint* termine quando o tempo especificado de **checkpoint_timeout** minutos tiver decorrido ou antes que o **max_wal_size** seja excedido.

Exemplificando, o valor de predefinição deste parâmetro é **0.5**, o que significa que o PostgreSQL pode esperar concluir cada *checkpoint* em cerca de metade do tempo antes do início do próximo *checkpoint*.

Desta forma, é expectável que o valor deste parâmetro seja aumentado, para que a carga de I/O seja balanceada de forma mais suave durante o tempo entre *checkpoints*.

Por outro lado é preciso ter algum cuidado a aumentar este valor, pois o restante tempo que sobra entre *checkpoints* é usado para realizar outras operações relacionadas com o atual *checkpoint*, e se não houver tempo suficiente pode resultar numa conclusão do mesmo fora de tempo, degradando assim o desempenho. Desta forma é totalmente desaconselhado usar o valor **1** para este parâmetro.

Outro aspeto negativo do aumento do valor deste parâmetro reside no facto de afetar o tempo de recuperação, pois serão necessários manter mais ficheiros de log para serem usados nessa recuperação. Mas tal como discutido na subsecção anterior, neste projeto muito provavelmente não existirão situações de recuperação pelo que este problema pode ser parcialmente ignorado.

As medições feitas encontram-se na seguinte figura.

checkpoint_completion_target	Resp. Time (s)	Throughput (tx/s)	Abort Rate	VM
0.1	0,052	378,97	0,026	2
0.2	0,055	351,45	0,028	3
0.3	0,049	395,34	0,049	2
0.4	0,05	390,97	0,049	3
0.5 (default)	0,05	387,14	0,039	2
0.6	0,051	385,69	0,025	2
0.7	0,056	351,26	0,026	3
0.8	0,059	332,16	0,034	2
0.9	0,06	323,63	0,034	3

Figura 13: Resultados da variação do Checkpoint Completion Target

Para este parâmetro a modificação de valores em nada melhorou o desempenho da base de dados. Entre os valores **0.3** e **0.6** obtiveram-se os valores maiores que na sua generalidade pouco oscilaram. Desta forma podemos assumir que a configuração predefinida pode continuar a ser usada uma vez que neste caso é um dos valores ótimos.

7º- Análise de Checkpoint Flush After

Sendo este um parâmetro relacionado com a gestão de *checkpoints* achamos por bem compreender a sua possível implicação no desempenho da base de dados. Rapidamente percebemos que é um parâmetro que se relaciona com o Sistema Operativo e com as páginas que este guarda na sua cache antes de as persistir em disco.

Desta forma percebemos que poderíamos forçar o Sistema Operativo a persistir as suas ao fim dum dado número de bytes. Achamos por bem deixar este valor com a sua predefinição, pois o seu ajuste não seria de todo interessante analisar neste projeto.

8º- Análise de Checkpoint Warning

Tal como o parâmetro anterior, este está relacionado com a gestão de *checkpoints* e por isso merece também uma menção. Este é um parâmetro que em nada altera o desempenho da base de dados, apenas serve para informar se é preciso ou não aumentar o **max_wal_size** consoante o tempo entre *checkpoints* tenha sido baixo ou alto.

Percebemos que numa fase inicial poderíamos ter usado este valor para obter informações nos logs sobre o que seria necessário fazer.

9º- Análise da Work Memory

Por último voltando à secção de *Resouce Usage* investigámos o que poderíamos alterar nos valores da **work_mem**, mas com a análise de algum código do *Benchmark* e associando-o às operações de possíveis clientes, percebemos que não iriam ser colocadas *queries* pelos clientes que necessitassem de *Joins* muito complexos, nem de imensos dados para satisfazer uma interrogação.

Desta forma percebemos que mexer neste valor não seria apropriado e provavelmente não traria qualquer melhoramento no desempenho da base de dados. Assim o valor da **work_mem** manteve-se em **4MB**.

Por último comparando os valores obtidos após a conclusão do requisito 1 com os valores obtidos na configuração de referência conseguimos concluir que a alteração dos parâmetros do PostgreSQL melhoraram demasiado o desempenho da base de dados. Tal pode ser observado na figura abaixo.

Configuração	Resp. Time (s)	Throughput (tx/s)	Abort Rate
Referência	0,107	183,53	0,028
Requisito 1	0,05	387,14	0,039

Figura 14: Comparação de resultados

O **tempo de resposta** baixou para um pouco mais de **metade** do valor de referência, e aumentamos o **débito** em cerca de **200 transações por segundo**.

3.2 Interrogações analíticas

Neste capitulo vamos dar resposta ao requisito número 2 do enunciado. O objetivo é otimizar quatro *queries* analíticas tendo em conta os seus planos de execução e os mecanismos de redundância usados.

3.2.1 Query 1

```
1  select su_name, su_address
2  from supplier, nation
3  where su_suppkey in
4      (select mod(s_i_id * s_w_id, 10000)
5          from stock, order_line
6          where s_i_id in
7          (select i_id
8              from item
9              where i_data like 'c\%')
10         and ol_i_id=s_i_id
11         and extract(second from ol_delivery_d) > 50
12         group by s_i_id, s_w_id, s_quantity
13         having 2*s_quantity > sum(ol_quantity))
14     and su_nationkey = n_nationkey
15     and n_name = 'GERMANY'
16 order by su_name;
```

A primeira análise a fazer é executar o *explain analyze* da *query* no *postgres* para observar que operações estão a ser feitas, e quais delas estão a demorar mais tempo, de maneira a melhorar o seu desempenho.

Pelo plano de execução da *query*, concluímos que a maior parte do seu tempo de execução é passado nas duas `inner queries` existentes. Para otimizar este tempo foi ponderado criar uma vista materializada.

Contudo, após melhor análise, decidimos não envergar por esse caminho. Considerando que a *query* é para ser executada ocasionalmente, por exemplo uma vez ao final do dia, não achamos compensatório ter uma *materialized view* que terá que ser atualizada sempre que for inserido, alterado ou apagado um novo registo nas tabelas `order_line`, `stock`, e `item`. Tendo em conta que estas tabelas sofrem um

grande número de alterações diárias, pelo menos as primeiras duas, é melhor fazer uma única vez esta operação quando realmente se quiser executar a *query 1*.

Prosseguindo então, conseguimos reparar que a operação de ordenação está a ser feita em disco, usando cerca de **88 MB**. Como indica a seguinte parte do plano de execução:

```
Sort  (cost=1611819.13..1626012.05 rows=5677168 width=16)
       (actual time=11927.493..13039.957 rows=3568171 loops=3)
  Sort Key: stock.s_i_id, stock.s_w_id
  Sort Method: external merge  Disk: 85208kB
  Worker 0:  Sort Method: external merge  Disk: 98872kB
  Worker 1:  Sort Method: external merge  Disk: 88416kB
```

Para aumentar a performance deste sort podemos aumentar a `work_mem` nas definições do *postgres*. Vamos considerar um aumento para os **100 MB** que achamos ser suficiente para fazer toda a operação de ordenação em memória.

Desta forma, todas as restantes *queries* irão sofrer subidas de desempenho, pois utilizam *joins* e *sorts* complexos, que são executados utilizando a **work_mem**.

Quanto a esta *query*, com o aumento do `work_mem` verificamos uma melhoria no tempo de execução para os 10ms, e no plano já não consta que é necessário recorrer a disco para executar qualquer operação.

```
Tempo de execução inicial: 15.12 s
Tempo de execução otimizado: 10.04 s
```

3.2.2 Query 2

```
1  select i_name,
2      substr(i_data, 1, 3) as brand,
3      i_price,
4      count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
5  from stock, item
6  where i_id = s_i_id
7      and i_data not like 'z\%'
8      and (mod((s_w_id * s_i_id),10000) not in
9      (select su_suppkey
10     from supplier
11     where su_comment like '\%bean\%'))
12 group by i_name, substr(i_data, 1, 3), i_price
13 order by supplier_cnt desc;
```

Pelo plano de execução da *query*, concluímos que a maior parte do seu tempo de execução é passado no `seq scan` no stock, mais precisamente na linha 6 (condição `i_id = s_i_id`). Estes dois atributos são *primary keys*, especificamente, das relações `item` e `stock`. Devido a tal, estas chaves já são índices, não conseguindo o grupo melhorar este passo mais crítico.

```
Hash Join (cost=5,729.14..911,382.04 rows=6,335,369 width=71)
(actual time=47.183..44,709.945 rows=11,302,716 loops=1)
Hash Cond: (stock.s_i_id = item.i_id)

   Seq Scan on stock (cost=348.77..806,168.08 rows=6,400,009
   width=8) (actual time=4.110..35,608.463 rows=11,474,190 loops=1)
   Filter: (NOT (hashed SubPlan 1))
   Rows Removed by Filter: 1325810
```

O grupo reparou também que a relação `supplier` não possuía qualquer tipo de chave, não tendo por isso também qualquer tipo de índice. Desta forma, foi implementado um índice no atributo `sup_supkey_ind`.

```
create index sup_supkey_ind on supplier(su_suppkey);
```

Além disso, o grupo decidiu construir uma *materialized view* para os passos demonstrados de seguida, pois a relação que está em causa é a `supplier`. Analisando esta mesma, achamos possível aplicar tal *materialized view* pois concluímos que o normal funcionamento (*run*) da base de dados não altera os valores desta relação. Desta forma, adquirimos as vantagens de possuir neste passo uma *materialized view*.

```
create materialized view su_suppkey_bean_mv as (select su_suppkey
    from supplier
    where su_comment like '%bean%');
```

Em jeito de conclusão da *query*, apresentamos a nova constituição desta mesma, assim como os tempos de execução, tanto o inicial como o final com otimização.

```
1  select i_name,
2      substr(i_data, 1, 3) as brand,
3      i_price,
4      count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
5  from stock, item
6  where i_id = s_i_id
7      and i_data not like 'z\%'
8      and (mod((s_w_id * s_i_id),10000) not in
9      (select * from su_suppkey_bean_mv))
10 group by i_name, substr(i_data, 1, 3), i_price
11 order by supplier_cnt desc;
```

```
Tempo de execução inicial: 68.06 s
Tempo de execução otimizado: 48.67 s
```

3.2.3 Query 3

```
1  select  n_name,
2      sum(ol_amount) as revenue
3  from  customer, orders, order_line, stock, supplier, nation, region
4  where  c_id = o_c_id
5      and c_w_id = o_w_id
6      and c_d_id = o_d_id
7      and ol_o_id = o_id
8      and ol_w_id = o_w_id
9      and ol_d_id = o_d_id
10     and ol_w_id = s_w_id
11     and ol_i_id = s_i_id
12     and mod((s_w_id * s_i_id),10000) = su_suppkey
13     and ascii(substr(c_state,1,1))-ascii('a') = su_nationkey
14     and su_nationkey = n_nationkey
15     and n_regionkey = r_regionkey
16     and r_name = 'EUROPE'
17 group by n_name
18 order by revenue desc;
```

Pelo plano de execução da *query*, concluímos que a maior parte do seu tempo de execução é passado no `parallel seq scan` no `stock`, e consequentemente no seu interior, no `parallel seq scan` no `customer`.

```
Parallel Seq Scan on stock (cost=0.00..635,152.41 rows=5,333,341
width=8) (actual time=0.045..42,836.766 rows=4,266,667 loops=3)
   ...
   Parallel Seq Scan on customer (cost=0.00..271,999.00
   rows=1,600,000 width=15) (actual time=0.016..28,116.988
   rows=1,280,000 loops=3)
```

Tendo em conta alguns dos atributos utilizados nas linhas 13, 14 e 15, o grupo concluiu que era útil criar índices nestes mesmos de forma a tornar mais célere o acesso a estes atributos.

Tal foi também fundamentado no facto destas relações não possuírem qualquer índice, tal como aconteceu no atributo `sup_supkey_ind` na *query 2*. Esse índice influencia também a execução desta *query*, neste caso a linha 13. Os índices estabelecidos são agora demonstrados.

```
1 create index sup_natkey_ind on supplier(su_nationkey);
2 create index nat_regkey_ind on nation(n_regionkey);
3 create index nat_natkey_ind on nation(n_nationkey);
4 create index reg_regkey_ind on region(r_regionkey);
```

Com estas alterações, os pontos do plano de execução mostrados anteriormente melhoram consideravelmente, sendo alteradas também as operações realizadas.

```
Parallel Index Only Scan using pk_stock on stock
(cost=0.43..257,729.93 rows=5,333,341 width=8) (actual
time=2.185..979.403 rows=4,266,667 loops=3)
   ...
   Parallel Seq Scan on orders (cost=0.00..51,887.99
   rows=1,599,999 width=16) (actual time=0.007..214.713
   rows=1,280,000 loops=3)
```

De seguida são apresentados os tempos de execução, tanto o inicial como o final com otimização.

```
Tempo de execução inicial: 82.55 s
Tempo de execução otimizado: 10.61 s
```

3.2.4 Query 4

```
1  select  c_last, c_id o_id, o_entry_d, o_ol_cnt, sum(ol_amount)
2  from  customer, orders, order_line
3  where  c_id = o_c_id
4      and c_w_id = o_w_id
5      and c_d_id = o_d_id
6      and ol_w_id = o_w_id
7      and ol_d_id = o_d_id
8      and ol_o_id = o_id
9  group by o_id, o_w_id, o_d_id, c_id, c_last, o_entry_d, o_ol_cnt
10 having  sum(ol_amount) > 200
11 order by sum(ol_amount) desc, o_entry_d
```

Executando a *query* apenas tendo em conta as alterações anteriores realizadas, neste caso com maior influência da alteração do `work_mem` para 100MB, é possível visualizar um tempo de execução de 121.68 s, quando inicialmente, sem qualquer mudança, era de 213.64 s.

Pelo plano de execução da *query*, concluímos que a maior parte do seu tempo de execução é passado no `sort` do `group by`, executado na linha 9.

Mais especificamente, a *query* executa um `Partial GroupAggregate` que necessita de um `sort`, que neste caso é realizado com 7 parâmetros, sendo por isso muito custoso em termos de memória. Este custo deve-se ao facto de, para se realizar o `sort`, a base de dados necessitar de ter todos os registos destes parâmetros em memória, mais especificamente na **work_mem**. As principais operações realizadas são apresentadas de seguida.

```
Partial GroupAggregate (cost=5,829,864.85..6,305,126.70
rows=13,578,910 width=77) (actual time=94,974.186..110,015.672
rows=1,337,555 loops=3)

Group Key: orders.o_id, orders.o_w_id, orders.o_d_id,
customer.c_id, customer.c_last, orders.o_entry_d,
orders.o_ol_cnt

   Sort (cost=5,829,864.85..5,863,812.12 rows=13,578,910 width=48)
   (actual time=94,974.143..102,629.223 rows=10,879,862 loops=3)
```

```
Sort Key: orders.o_id, orders.o_w_id, orders.o_d_id,
customer.c_id, customer.c_last, orders.o_entry_d,
orders.o_ol_cnt

Sort Method: external merge Disk: 711928kB

Worker 0: Sort Method: external merge Disk: 596152kB

Worker 1: Sort Method: external merge Disk: 659432kB
```

Todos os campos usados nesta interrogação são índices das tabelas **customer**, **orders** e **order_line**, não sendo possível, por isso, adicionar mais nenhum índice. Como estas 3 tabelas são mudadas frequentemente, uma *materialized view* também não é solução. Tentamos ainda utilizar um **CLUSTER** no campo **ol_o_id** uma vez que poderia fazer algum sentido ter as linhas duma encomenda agrupadas pelo identificador dessa mesma encomenda. Tal não gerou melhorias no desempenho.

Desta forma, concluímos que não é possível melhorar o desempenho desta *query*.

Em jeito de conclusão da *query*, apresentamos os tempos de execução, tanto o inicial como o final com otimização.

```
Tempo de execução inicial: 213.64 s
Tempo de execução otimizado:  121.68 s
```

4 Replicação da Base de Dados

Para satisfazer o requisito 3, decidimos fazer uma replicação master/slave da base de dados. As réplicas irão estar instaladas em máquinas diferentes na cloud.

4.1 Replicação Lógica

Alterações às configurações iniciais

Para proceder à implementação de uma replicação master/slave, foi necessário alterar alguns parâmetros na base de dados já instalada.

Alterar o ficheiro pg_hba.conf para permitir conexões, alterar o parâmetro *wal_level* para o valor *logical*, e criar um publicação com todas as tabelas da base de dados executando:

```
 create publication tpccpub for table customer, district, history,
item, new\_order, order\_line, orders, stock, warehouse;
```

Instalação da base de dados slave

Para alojar a base de dados slave criamos outra máquina na cloud com um container postgres. Foi então necessário criar uma base de dados com o nome tpcc, criar as tabelas e executar o comando:

```
 create subscription tpccsub connection 'dbname=tpcc host=10.154.0.8
port=5432 user=root password= root' publication tpccpub;
```

Testes

Para testar a performance desta solução íamos recorrer novamente ao *run.sh* do TPC-C, para visualizar as medidas de response time e throghput.

Infelizmente não foi possível correr o script, porque para a criação da subscrição à base de dados master foi necessário definir uma password para o utilizador root, ou seja o script de execução deixou de conseguir aceder à base de dados, não sendo assim possível testar a mesma.

Fica então para trabalho futuro alterar o código do TPC-C para fazer com que este se consiga conectar a base de dados usando o utilizador root e a sua password.

5 Conclusão e perspetiva de trabalho futuro

Neste trabalho conseguimos achar a uma configuração de referência de qualidade, que nos permitiu também executar o requisito 1 na totalidade. Quanto ao requisito 2, apesar de termos melhorado as *queries* sentimos que com mais tempo estas poderiam ser ainda mais otimizadas, com a criação de colunas extra e/ou mais índices, apesar de que usamos aqueles que achamos que teriam maior impacto no desempenho das *queries*.

Neste requisito fomos aumentando a **work_mem** para melhorar o desempenho das *queries* sabendo que esta modificação iria ter um custo na execução do *run*. Exemplificando, aumentando este valor para **100MB** e tendo 1000 clientes a fazerem interrogações, iria ser preciso 100MB*1000 para satisfazer as *queries* dos utilizadores. Iria ser gasta mais memória e possivelmente o desempenho degradar-se-ia, no entanto não calculamos tais resultados, não podendo dizer qual seria o custo desta decisão.

Quanto ao requisito 3 achamos que o que fizemos não foi suficiente para analisar o possível processamento distribuído, uma vez que essa componente requeria que usássemos uma possível ferramenta de balanceamento das queries pelas bases de dados existentes saindo assim fora do âmbito do projeto mas com resultados que poderiam ser prometedores.

Desta forma, percebemos que o processo de otimização de bases de dados não é simples nem automático pois exige uma análise profunda dos parâmetros do SGBD e uma análise lógica e fundamentada para escolher mecanismos de otimização de queries. A otimização de uma *query* pode afetar a de outra e até o possível desempenho de toda a base de dados. É, por isso, um trabalho de ponderação sobre aquelas que serão as melhores combinações para um melhor desempenho da base de dados, e é compreensível que o melhoramento do desempenho seja um processo extenso e de contínuo melhoramento.

Concluindo, o grupo achou que o âmbito do trabalho foi de extrema importância para perceber certos mecanismos que as bases de dados possuem e podem ser explorados pelos administrados. Desta forma, achamos que os objetivos iniciais do trabalho foram concluídos dando por terminado este projeto.

Referências

PostgreSQL Documentation. https://www.postgresql.org/docs/11/index.html . Accessed Dez. 2019 - Jan. 2020.