Universidade do Minho
Escola de Engenharia

Projeto de Administração de Base de Dados
1º/4º ano MEI/MIEI

Título Projeto

Relatório de Desenvolvimento

João Paulo Oliveira de Andrade Marques
(a81826@alunos.uminho.pt)

José André Martins Pereira
(a82880@alunos.uminho.pt)

Ricardo André Gomes Petronilho
(a81744@alunos.uminho.pt)

4 de Janeiro de 2020

Conteúdo

Capítulo 1

Introdução

O presente relatório detalha o desenvolvimento do projeto da Unidade Curricular de Administração de Base de Dados do Perfil de Engenharia de Aplicações do 1º/4º ano do Mestrado em Engenharia Informática / Mestrado Integrado em Engenharia Informática da Escola de Engenharia da Universidade do Minho, o qual consiste em obter uma configuração de referência (desempenho) da aplicação **EscadaTPC-C** de forma a tentar otimizar esta.

Ao longo deste relatório abordam-se as fases de configuração de referência, desempenho da mesma e alterações realizadas à configuração da Base de Dados para melhorias de desempenho.

Inicialmente, fez-se uma contextualização do problema, bem como uma análise da aplicação **EscadaTPC-C**, para se entender a arquitetura da aplicação, bem como as suas dependências.

De seguida, descreve-se as configurações implementadas e decisões realizadas para se atigirem os objetivos referidos acima.

Por fim, e não menos importante, apresenta-se uma avaliação do desempenho do sistema, com os prós e contras das decisões aplicadas.

Capítulo 2

Descrição/perceção do Problema

Neste capítulo faz-se uma descrição/contextualização do problema em análise, abordando-se os objetivos pretendidos.

2.1 Objetivos

Os principais objetivos deste projeto concentram-se na configuração do benchmark TPC-C e de uma máquina para correr o mesmo, sendo esta a configuração de referência. Depois de obtida a configuração de referência serão alteradas as configurações da Base de Dados PostgreSQL (base de dados utilizada para guardar toda a informação do benchmark TPC-C) de forma a tentar obter melhorias no desempenho da configuração de referência.

2.2 Instalação e Configuração

Para a instalação do EscadaTPC-C foi utilizada a documentação fornecida pelo docente da Unidade Curricular, tendo sido necessário a instalação de outras peças de software durante a instalação deste. Para um total funcionamento da aplicação é necessário instalar uma base de dados (MySQL, Derby ou PostgreSQL), sendo que para este projeto foi utilizada a base de dados PostgreSQL.

Capítulo 3

Desenvolvimento

Nesta secção será ilustrada a configuração de referência obtida juntamente com os seus resultados. De seguida serão ilustrados quais as alterações realizadas às configurações da base de dados PostegreSQL e quais as otimizações de desempenho obtidas com essas mesmas alterações.

3.1 Configuração de Referência

Para determinar a configuração de referência foram realizados vários testes. Na configuração da aplicação EscadaTPC-C variou-se tanto o número de **warehouses**, bem como **número de clientes por warehouse** no ficheiro worload-proprieties. Nas máquinas variou-se os seus recursos: número de **CPUs**, **RAM** e **ROM**.

De forma a **maximizar o tempo disponível** foram instaladas e configuradas todas as dependências (postgresql e TPC-C) para realizar os testes numa máquina, de seguida foi criado um **snapshot** e criadas mais 8 máquinas equivalentes. Desta forma foi possível efetuar 9 testes em **simultâneo**.

Com o mesmo propóstivo foi criado um bash script - **start_bd.sh** - que cria a base de dados, percorre os scripts forncecidos pela equipa docente para a criação de tabelas, indices, etc; faz o povoamento (load.sh) e corre o benchmark (run.sh). Foi também criado um **makefile** que move os ficheiros .dat para uma pasta de resultados e remove ficheiros que já não são relevantes. Tudo isto contribuiu para **automatizar** a realização de testes.

A **monotorização** da utilização de CPU e memória RAM foi obtida através do serviço stackdriver fornecido pela Google. Desta forma, para cada máquina, foi possível registar as respetivas **métricas** e analisar as mesmas através de gráficos, tal como a seguinte figura ilustra.

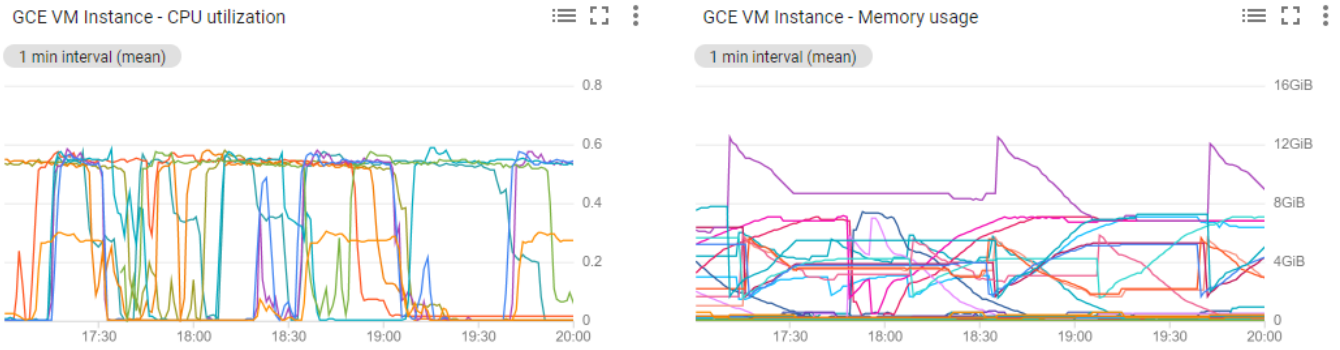

Figura 3.1: Monotorização do CPU e RAM.

A figura anterior apresenta informação relativa a todas as máquinas, no entanto é possível salientar os recursos de cada máquina **individualmente** e mesmo obter dados **temporais**, tal como se segue.

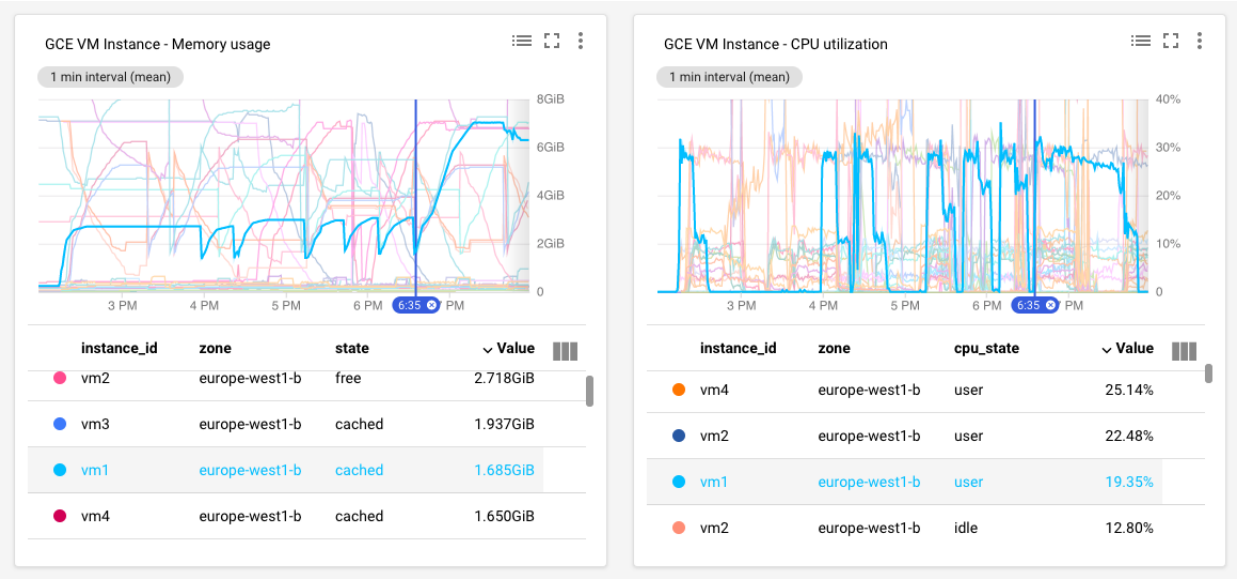

Figura 3.2: Métricas de CPU e memória RAM (linha azul realçada) da configuração de 60 warehouses e 150 clientes por warehouse na **vm1** com hora inicial do teste ás 06h35m PM.

Através da interpretação das métricas de **31 testes**, conclui-se que a máquina com resultados mais **eficientes** para os testes realizados contém os seguintes recursos: **2 cores**, **8 GB de RAM** e **30 GB de SSD** persistente.

Com o hardware anterior determinou-se que a configuração ideal do benchmark TPC-C consiste em em **70 warehouses** e **150 clientes**. A utilização de CPU foi cerca de 60% e foi utilizada cerca de 85% da memória RAM.

Note-se que era possível aumentar o número de warehouses e os respetivos clientes, no entanto, face à **limitação de recursos** e **tempo disponível para os testes**, conclui-se que a configuração anterior é a ideal, sendo suficiente para apresentar possíveis melhorias de desempenho.

Com a ajuda do script **showtpc.py** fornecido pela equipa docente foi possível obter dados de desempenho como o débito, tempo de resposta e taxa de cancelamento de transações.

3.2 Otimização do Desempenho da Carga Transacional

A otimização no desempenho da carga transacional, usando a configuração de referência, realizou-se com alterações às configurações da base de dados PostgreSQL. Foram alterados alguns paramêtros de configuração, complementados com testes, para verificação da vantagem dessa alteração.

A configuração de referência apenas utiliza cerca de 85% da RAM como foi referido anteriormente, esta decisão foi propositada uma vez que era de esperar aumentar a quantidade de RAM disponível para os diferentes buffers dos processos do servidor postgresql, desta forma **tem de existir uma margem diponível**.

3.2.1 Parâmetro shared_buffer

Em qualquer circustância acessos (leitura e escrita) em memória RAM são **extremamente mais rápidos** que em disco, desta forma o servidor postgresql usufrui de uma cache denominada de **shared_buffer** onde armazena parte de tabelas ou indices temporariamente e realiza operações sobre as mesmas, de seguida, se necessário, volta a escrever os dados em memória presistente.

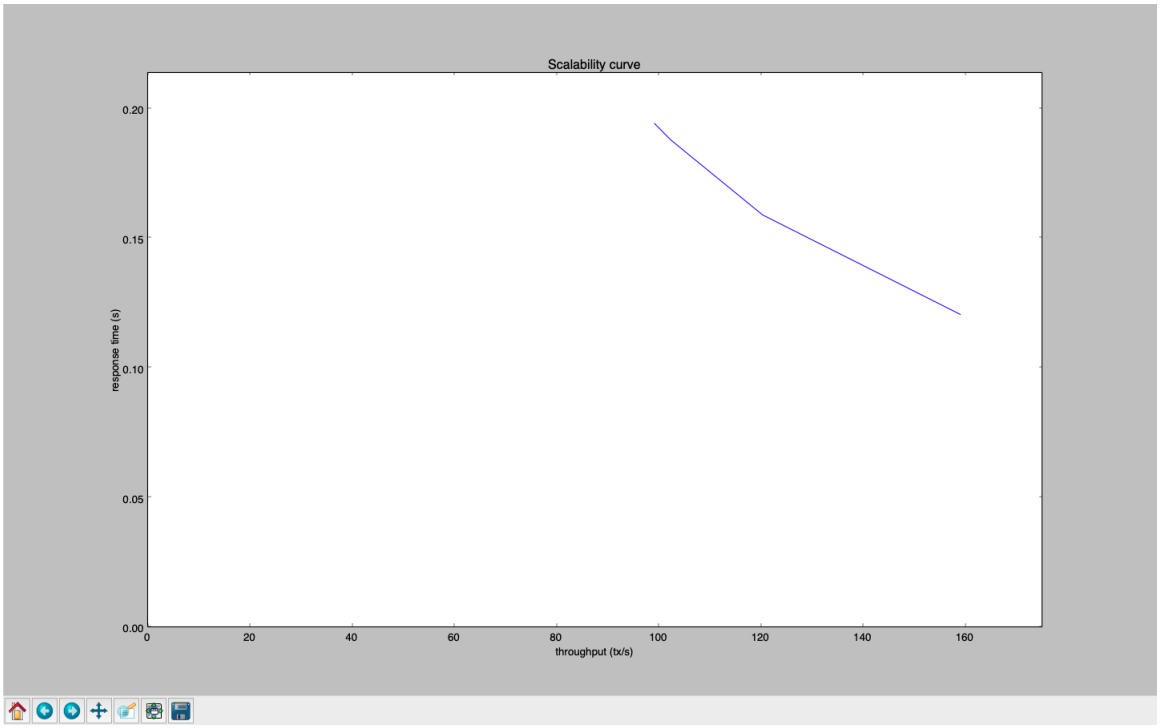

Figura 3.3: Escalabilidade do *troughput* para valores do shared_buffer de 128MB, 256MB, 512MB e 1024MB.

Apenas com a alteração do parâmetro **shared_buffer** de 128MB (padrão) para **1024MB** foi possível **aumentar o débito em cerca de 60%** e o **tempo de resposta diminuiu em cerca de 40%**. Também foram realizados testes com 256MB e 512MB como pode ser visualizado na figura 3.3, sendo que o melhor valor obtido foi o de 1024MB.

Esta **melhoria tão acentuada** justifica-se uma vez que com mais memória RAM disponível para este parâmetro, os processos do servidor postgresql não necessitam de aceder ao disco (memória lenta) tantas vezes.

Importante referir, que provavelmente para valores de **shared_buffer** maiores do que 1024MB **(até a valores próximos do limite de RAM)**, poderia se obter ainda melhores resultados. No entanto, devido à falta de tempo para realização de mais testes, e como já se obteve melhoria aceutado em relação à configuração padrão, decidiu-se utilizar este valor.

3.2.2 Parâmetro work_mem

Este parâmetro controla a quantidade de memória RAM alocada, tipicamente, **por cada** operação de ordenação (ORDER BY, DISTINCT, etc). Note-se que podem existir muitas operações deste tipo a correr em simultâneo no servidor, desta forma a quantide de RAM disponível para cada operação **não pode ser muito grande, caso contrário a RAM é esgotada rapidamente**, tornando o sistema operativo e todos os processos mais lentos.

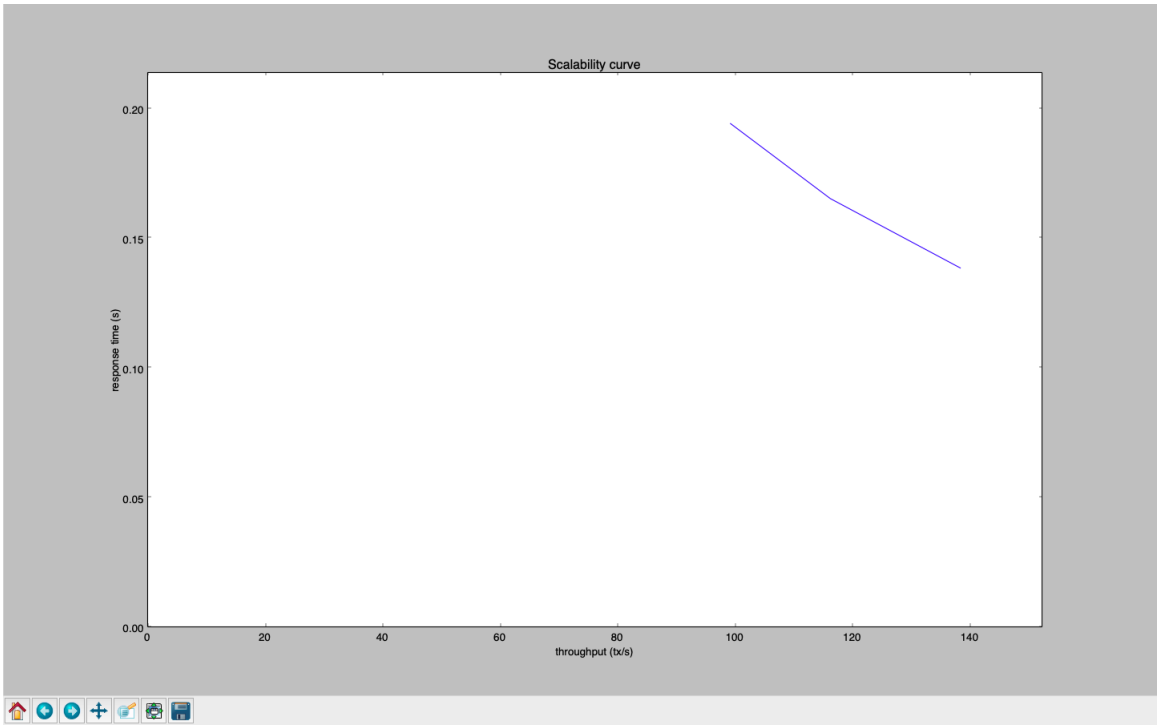

Figura 3.4: Escalabilidade do *troughput* para valores do work_mem de 4MB, 8MB, e 16MB.

Com a alteração do parâmetro **work_mem** também foram obtidos ganhos quando alterado de 4MB para 8MB, 16MB, 32MB e 64MB, sendo que destes o que teve a melhor otimização de desempenho foi o valor de **16MB**.

Note-se que para valores de 32MB de 64MB os resultados obtidos foram piores do que com 16MB, isto justifica-se uma vez que esta quantidade de RAM é alocada **por cada** operação, tal como referido anteriomente, desta forma, provavelmente valores de 32MB e 64MB são suficientes para existir **escassez de RAM tornando os processos mais lentos**.

Os resultados que demonstram este **declínio de desempenho** apresentam-se de seguida na figura 3.14. O débito (throughput) diminui e o tempo de resposta aumentou em relação aos teste realizados com 16MB.

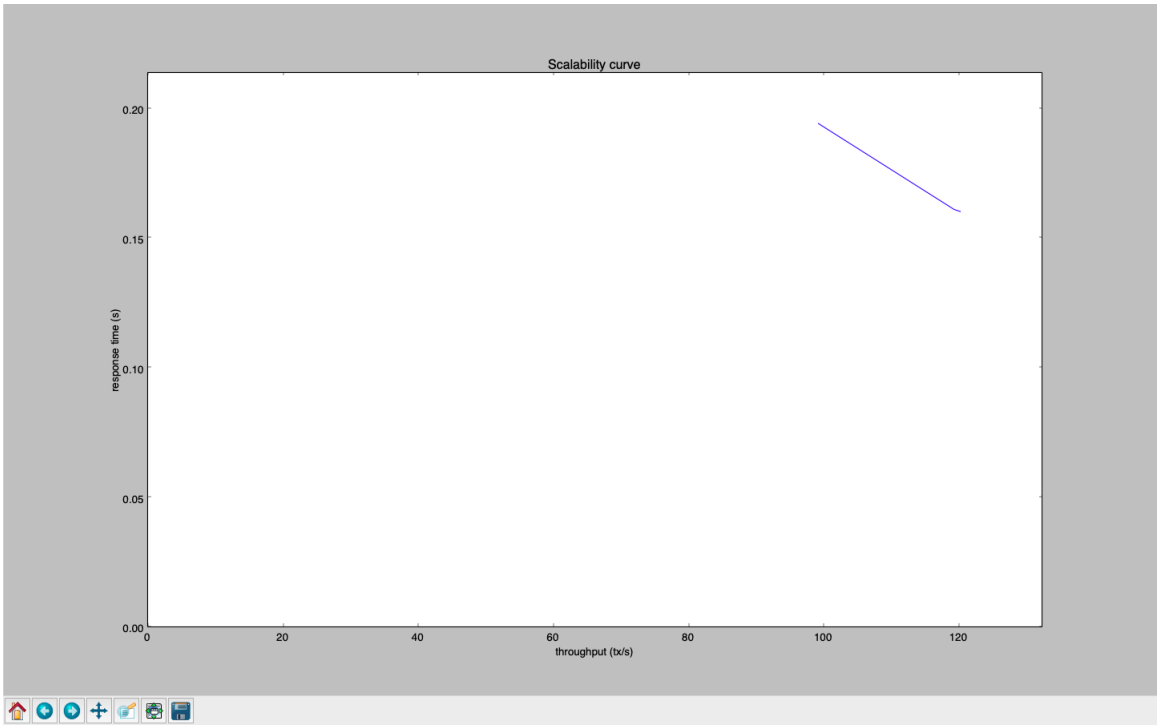

Figura 3.5: Escalabilidade do *troughput* para valores do work_mem de: 4MB, 32MB e 64MB.

3.2.3 Parâmetro wal_buffer

Este parâmetro controla a quantidade de RAM disponível para armazenar os WAL - **W**rite **A**head **L**ogs. Todas as alterações aos dados são registadas nestes logs para que, caso o sevidor falhe, seja possível reconstruir os dados totalmente. Desta forma esta porção de memória RAM está constantemente a ser preenchida e quando cheia é copiada para memória presistente. Enquanto está a ser copiada, todos os processos de escrita ficam em espera (para registar os seus logs).

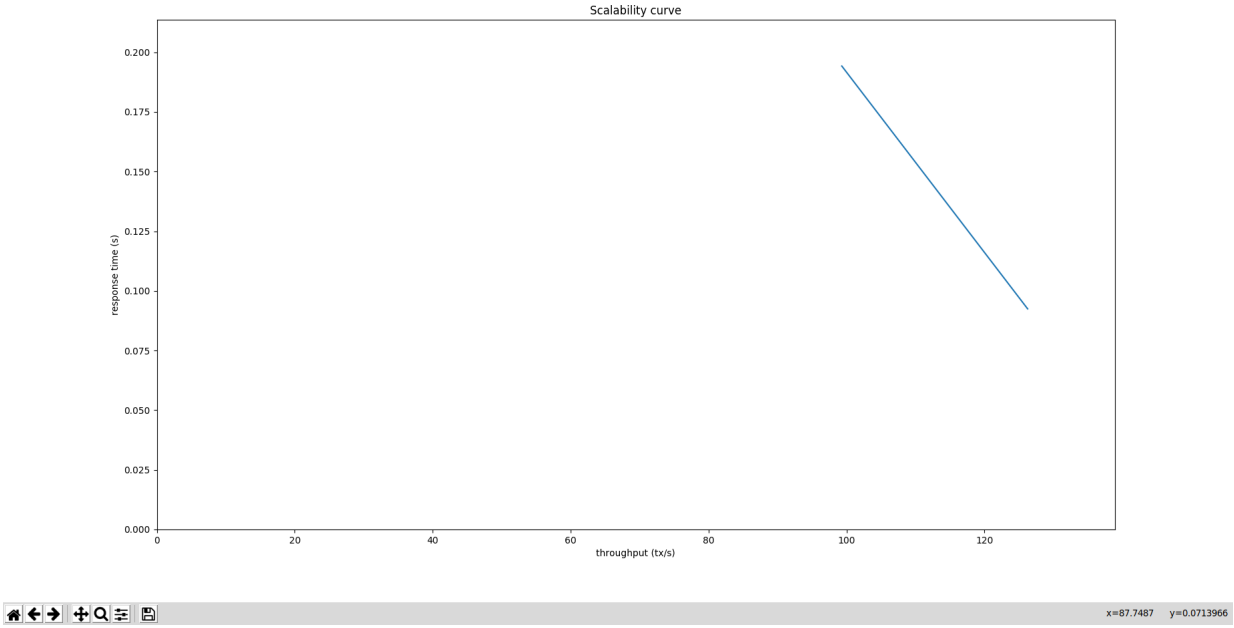

Figura 3.6: Escalabilidade do *troughput* para valores do wal_buffers: padrão e 32MB.

Como se pode observar no gráfico anterior, com **32MB** de wal_buffer o débito aumentou em cerca de 27.13% e o tempo de resposta diminui muito. Esta otimização justifica-se uma vez que o buffer **demora mais tempo a ficar totalmente preenchido, desta forma o número de vezes que os processos ficam à espera (da cópia da RAM para o disco) é menor**.

3.2.4 Configuração ideal

Após alterar cada parâmetro individualmente e obter melhorias em todos os testes, decidiu-se realizar um teste final com os 3 parêmetros modificados. Desta forma a configuração ideal consiste em:

- shared_buffer = 1024 MB
- work_mem = 16 MB
- wall_buffer = 32 MB

Consequentemente o **débito aumentou em cerca de 70.83%** e o **tempo de resposta diminui**, tal como se pode observar no seguite gráfico.

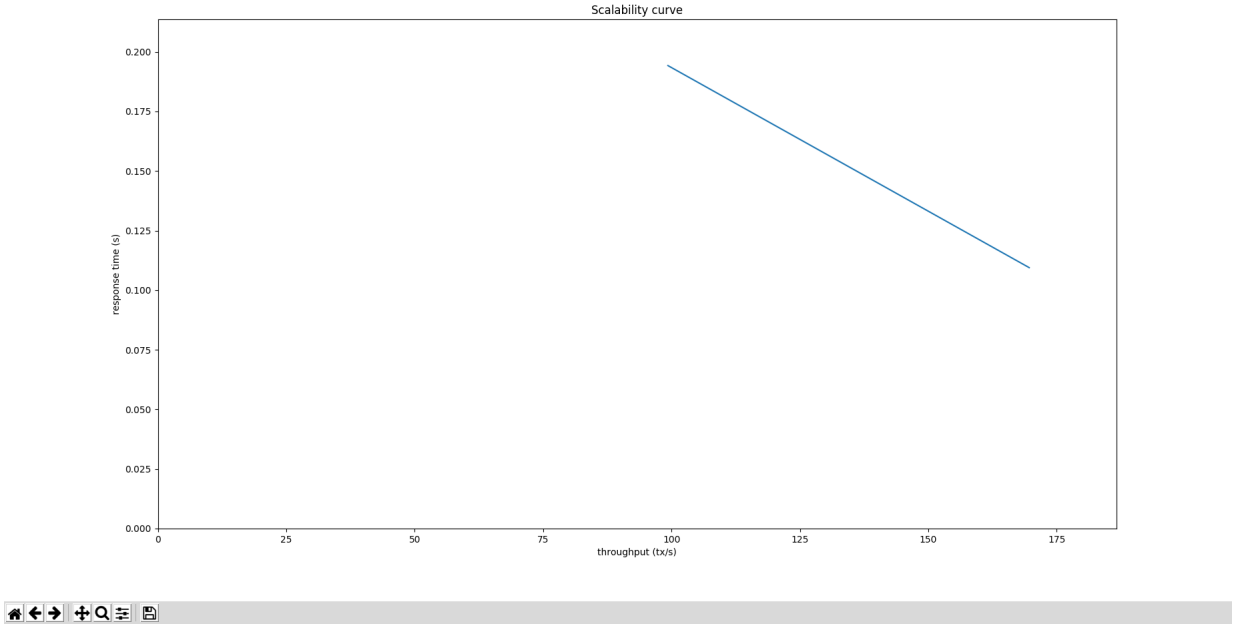

Figura 3.7: Escalabilidade do *troughput* com configuração final otimizada.

3.3 Otimização de Desempenho de Queries Analíticas

Este capítulo aborda as téncias utilizadas para otimizar o desempenho das queries analíticas fornecidas pela equipa docente.

Tal como referido nas aulas a introdução de **redudância** através de **indices** e **vistas materializadas** é espectável que seja suficiente para otimizar com grande relância todas as queries.

As views materializadas devem ser criadas quando uma determinada queire é **demasiado lenta** a processar ao ponto de ser intolerável esperar esse tempo cada vez que a mesma é executada.

No entanto como o resultado das views materializas **é gerado no momento que a mesma é criada**, a informação contida nestas views pode não ser a mais atual quando é evocada. Por este motivo este tipo de views deve ser associado a informação que raramente é alterada ou a sua alteração não tem impacto crítico na validade dos resultados.

Posto isto, inicialmente, começou-se por gerar o plano da queire nº1 através do comando **explain** no postgres.

Verificamos que a querie utilizava o algoritmo NLJ - **N**ested **L**oop **J**oin - para unir a tabelas, no entanto tal como referido nas aulas em muitos casos este algoritmo **pode não ser o ótimo** uma vez que caso a primeira tabela tenha N linhas, o algoritmo vai percorrer N vezes a segunda tabela com complexidade $\mathbf{N^2}$, tendo a vantagem de necessitar menor memória RAM para a sua execução, uma vez que apenas necessita de uma linha de cada tabela em cada momento.

Desta forma decidiu-se desativar o algoritmo NLJ nos ficheiros de configuração do postgres. Foram realizados alguns testes medindo o tempo que cada querie demorava a responder utilizando o comando **timming** e verificou-se que, **ao contrário do espectável pelo grupo**, o tempo de processamento aumentou.

Analisando ainda a querie nº1 decidiu-se que no sublinhado a azul encontra-se a sub-querie transformada em view materializada e a verde encontra-se as colunas transformadas indices.

A.1

```
select  su_name, su_address
from  supplier, nation
where  su_suppkey in
          (select  mod(s_i_id * s_w_id, 10000)
           from     stock, order_line
           where    s_i_id in
                    (select i_id
                     from item
                     where i_data like 'c%')
             and ol_i_id=s_i_id
             and extract(second from ol_delivery_d) > 50
           group by s_i_id, s_w_id, s_quantity
           having   2*s_quantity > sum(ol_quantity))
  and su_nationkey = n_nationkey
  and n_name = 'GERMANY'
order by su_name;
```

Figura 3.8: Alterações feitas na queri 1 (Azul = View Materializada, Verde = index).

Note-se que é possível criar a view materializada a partir da sub-querie referida uma vez os campos **i_data** e **i_id** da tabela item, á partida, **não são alterados** e memso que fossem alterados, visto que esta querie é possivlemente apenas executada por membros de administração, não deverá ter grande impacto ter dados desatualizados.

Com a introdução desta view meterializada tivemos **ganhos significativos**, o tempo de processamento

da querie **reduziu cerca de 138 ms** tal como é possível observar na seguinte figura.

```
com@vm2:~/queries$ psql -h localhost -f a1.sql tpcc --username=com
Timing is on.
Time: 7928.076 ms (00:07.928)
com@vm2:~/queries$ psql -h localhost -f a1-with-materialized-view.sql tpcc --username=com
Timing is on.
Time: 7790.602 ms (00:07.791)
com@vm2:~/queries$
```

Figura 3.9: Otimização de desempenho com introdução da view materializada.

Em primeira análise pode-se pensar que 138 ms é um ganho sem grande impacto mas se se multiplicar este valor por inúmeros clientes simultâneamente a evocar esta querie, tal como nos testes do benchamark TPC-C, provavelmente os ganhos são significativos.

De seguida foi testada a introdução dos indices anteriormente referidos e medido o tempo de processamento com esta alteração. As seguintes figuras ilustram a melhoria em tempo de processamento ganho com a introdução destes indices.

```
com@vm2:~/queries$ psql -h localhost -f a1.sql tpcc --username=com
Timing is on.
Time: 8071.259 ms (00:08.071)
com@vm2:~/queries$ psql -h localhost tpcc com
psql (10.10 (Ubuntu 10.10-0ubuntu0.18.04.1))
Type "help" for help.

tpcc=# create index n_name_index on nation (n_name);
CREATE INDEX
tpcc=# \q
com@vm2:~/queries$ psql -h localhost -f a1.sql tpcc --username=com
Timing is on.
Time: 7851.899 ms (00:07.852)
com@vm2:~/queries$
```

Figura 3.10: Testes antes e depois da inserção do index $\mathbf{n\_name}_{index}$.

```
com@vm2:~/queries$ psql -h localhost -f a1.sql tpcc --username=com
Timing is on.
Time: 7930.781 ms (00:07.931)
com@vm2:~/queries$ psql -h localhost tpcc com
psql (10.10 (Ubuntu 10.10-0ubuntu0.18.04.1))
Type "help" for help.

tpcc=# create index su_nationkey_index on supplier (su_nationkey);
CREATE INDEX
tpcc=# \q
com@vm2:~/queries$ psql -h localhost -f a1.sql tpcc --username=com
Timing is on.
Time: 7811.670 ms (00:07.812)
com@vm2:~/queries$
```

Figura 3.11: Testes antes e depois da inserção do index $\mathbf{su\_nationkey}_{index}$.

A.2

```
select  i_name,
 substr(i_data, 1, 3) as brand,
 i_price,
 count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
from  stock, item
where  i_id = s_i_id
 and i_data not like 'z%'
 and (mod((s_w_id * s_i_id),10000) not in
(select su_suppkey
 from supplier
 where su_comment like '%bean%'))
group by i_name, substr(i_data, 1, 3), i_price
order by supplier_cnt desc;
```

Figura 3.12: Alterações feitas na queri 2 (Azul = View Materializada, Verde = index).

```
com@vm2:~/queries$ psql -h localhost -f a2.sql tpcc --username=com
Timing is on.
Time: 31468.573 ms (00:31.469)
com@vm2:~/queries$ psql -h localhost tpcc com
psql (10.10 (Ubuntu 10.10-0ubuntu0.18.04.1))
Type "help" for help.

tpcc=# create index i_data_index on item (i_data);
CREATE INDEX
tpcc=# \q
com@vm2:~/queries$ psql -h localhost -f a2.sql tpcc --username=com
Timing is on.
Time: 31447.690 ms (00:31.448)
com@vm2:~/queries$
```

Figura 3.13: Testes antes e depois da inserção do index **i_data_index**.

```
com@vm2:~/queries$ psql -h localhost -f a2.sql tpcc --username=com
Timing is on.
Time: 31234.069 ms (00:31.234)
com@vm2:~/queries$ psql -h localhost tpcc com
psql (10.10 (Ubuntu 10.10-0ubuntu0.18.04.1))
Type "help" for help.

tpcc=# create index i_name_index on item (i_name);
CREATE INDEX
tpcc=# \q
com@vm2:~/queries$ psql -h localhost -f a2.sql tpcc --username=com
Timing is on.
Time: 30949.433 ms (00:30.949)
com@vm2:~/queries$
```

Figura 3.14: Testes antes e depois da inserção do index **i_name_index**

Capítulo 4

Conclusão

Assim conclui-se que os objetivos inicialmente propostos não totalmente atingidos.

A fase inicial, teve como o objetivo encontrar uma configuração de referência, sendo que esta etapa foi bastante trabalhosa, pois foram necessários muitos testes até se chegar aos valores tanto das configurações da máquina, bem como do número de warehouses e clientes por warehouse.

A otimização do teste de carga baseou-se na análise dos parâmetros do **PostgreSQL**, para se conseguir melhor desempenho a partir da configuração de referência. Para tal, também foram realizados testes, para diferentes valores desses campos isoladamente, e de seguida, conjuntamente, para se chegar à configuração ideal. Algumas dificuldades encontradas nesta fase, foi a quantidade de testes necessários para se conseguir provar a otimização da configuração final.

De seguida, na fase de otimização das queries analíticas, começou-se por analisar o plano das queries, e tentar perceber possíveis otimizações, no entanto, onde se conseguiu melhores resultados, foi na colocação de views materializadas e indexs. No entanto, é de realçar que não houve grandes melhorias, devendo-se isto ao facto de a configuração de referência ainda ser "simples".

Em relação ao passo sobre a replicação, o grupo não conseguiu implementar, no entanto, percebe-se o conceito de replicação e processamento distribuído, e as consequências de processamento que o mesmo acarreta, para se garantir fiabilidade e consistência dos dados. Deste modo, também a equipa sabe, que o processo de replicação fica mais facilitado com base de dados não relacionais (NoSQL).

Por fim, e não menos importante, apesar de não se ter atingido todos os objtivos inicialemente apresentados, dos que foram concretizados, considera-se um bom resultado.