UNIVERSIDADE DO MINHO

MESTRADO EM ENGENHARIA INFORMÁTICA

ADMINISTRAÇÃO DE BASES DE DADOS

Benchmark TPC-C

BERNARDO MOTA	(A77607)
CLÁUDIA CORREIA	(A77431)
JOANA PEREIRA	(A78565)

16 de Fevereiro de 2021

Conteúdo

Resumo

Este documento diz respeito ao trabalho prático proposto na unidade curricular de Administração de Bases de Dados da Universidade do Minho. O objetivo deste projeto consiste na configuração, otimização e avaliação do *benchmark* TPC-C.

1 Introdução

O TPC-C consiste num *benchmark* de processamento de transações online (OLTP) que lida com transações de diversos tipos e complexidades. Ao contrário do que o nome indica, o TPC-C tanto executa transações online como as armazena numa *queue* para serem processadas mais tarde. Este *benchmark* retrata qualquer tipo de negócio que envolva a gestão, venda ou distribuição de um produto/serviço, suportando as operações envolvidas neste setor.

No âmbito deste projeto será utilizado o *benchmark* Escada TPC-C. Este *benchmark* é semelhante ao TPC-C, sendo fácil de utilizar. Embora o Escada TPC-C suporte diferentes sistemas de gestão de bases de dados, ao longo deste projeto será alvo de estudo, única e exclusivamente, o PostgreSQL.

Com o intuito de analisar o *benchmark* será utilizado o *script* Python *showtpc.py*, que permite obter resultados em formato de texto ou em gráfico. Ao longo do presente relatório serão apresentados e devidamente explicitados os resultados provenientes da utilização deste *script*, na tomada de decisões cruciais para o desenvolvimento do projeto.

2 Configuração de referência

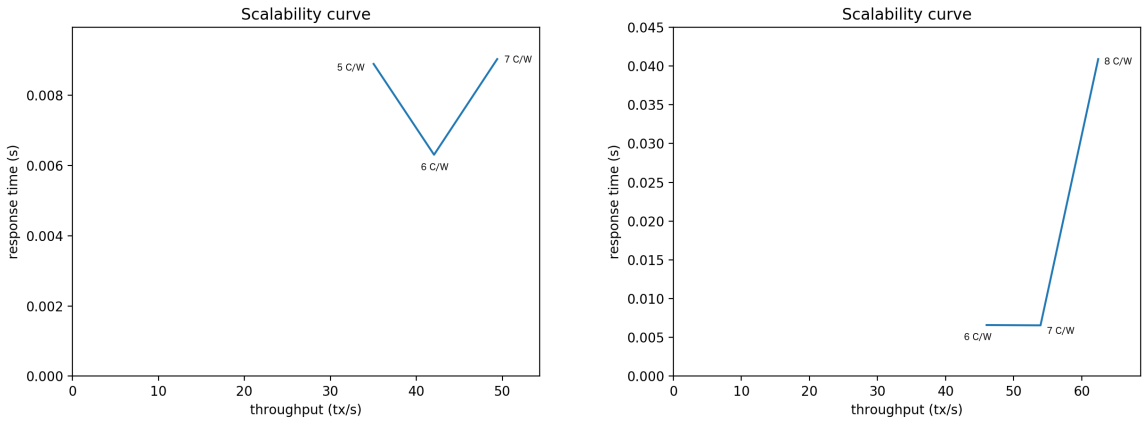
Para a seleção da configuração de referência foram avaliadas características de ***hardware*** (CPU e memória) e de ***workload*** (número de *warehouses* e número de clientes).

2.1 Hardware

No que toca a *hardware*, os alvos de estudo foram a **memória** e o **CPU** da máquina. Para cada uma das análises foram efetuados 6 testes, sendo que para isto utilizaram-se 80 e 90 *warehouses*, variando o número de clientes.

2.1.1 Memória

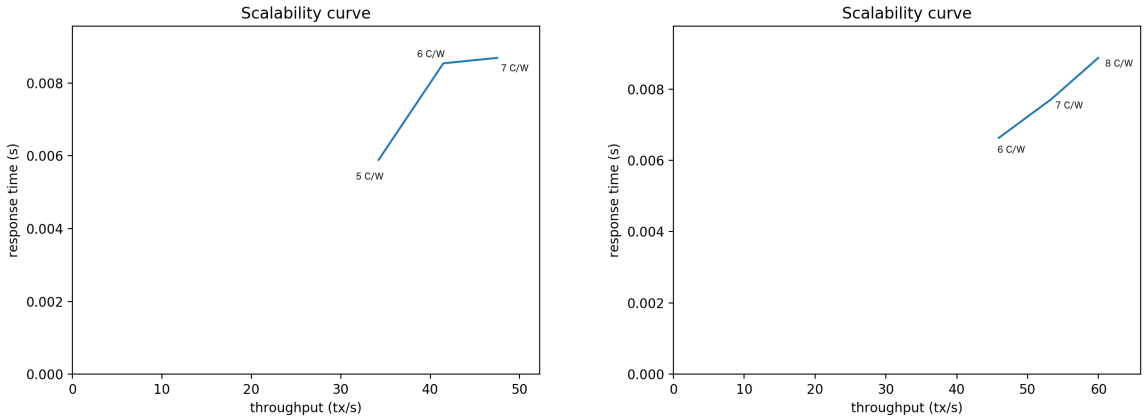
Ao escolher a memória para a configuração de referência, a indecisão rondou entre os 8GB e os 15GB. Os gráficos apresentados abaixo descrevem as curvas de escalabilidade para 80 e 90 *warehouses*, em ambientes de teste com 8GB de memória e 15GB de memória, respetivamente.



(a) 80 *warehouses*

(b) 90 *warehouses*

Figura 1: Curvas de escalabilidade para 8GB de memória



(a) 80 *warehouses* (b) 90 *warehouses*

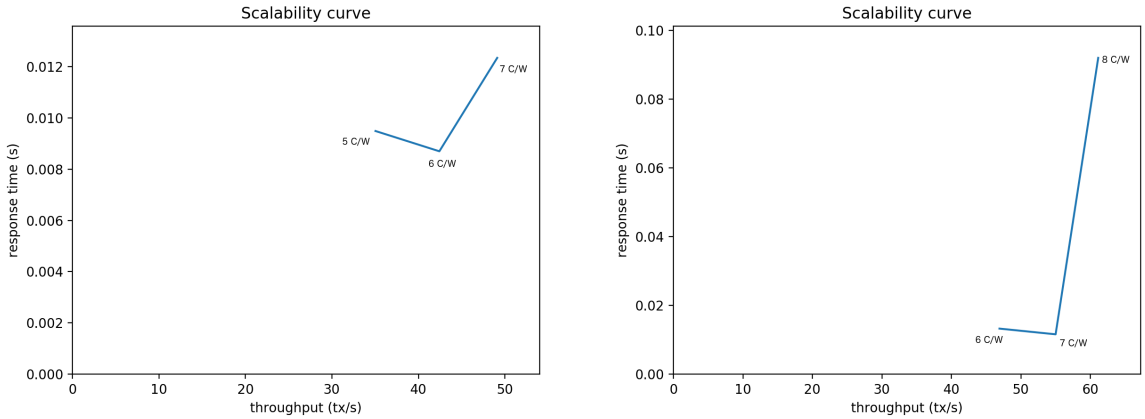
Figura 2: Curvas de escalabilidade para 15GB de memória

Tal como podemos observar através dos gráficos, existem casos nos quais a utilização de 15GB é vantajosa (p.e., com 80 *warehouses* e 5 c/w), no entanto, esta vantagem não é tão frequente quanto seria de esperar com uma quase duplicação da memória. Para além disto, nos casos em que os 15GB proporcionam um tempo de resposta menor, a divergência entre este e o tempo de resposta com 8GB não é grande suficiente para justificar a utilização dos 15GB.

Desta forma, a configuração de referência, no que toca a *hardware*, possuirá **8GB de memória**.

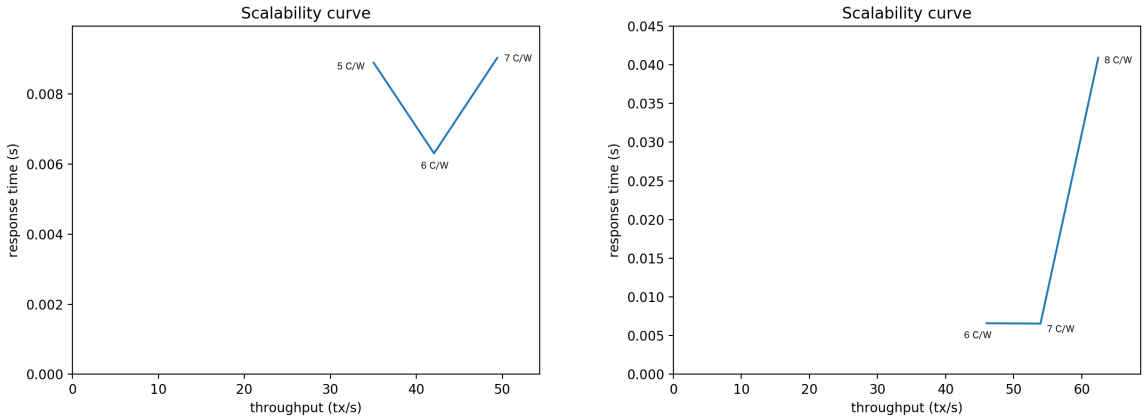
2.1.2 CPU

Para analisar o CPU que será utilizado na configuração de referência, foram utilizados dois casos de teste, nomeadamente uma máquina com 2 CPUs e outra com 4 CPUs. Tal como foi efetuado para a memória, também neste caso foram utilizados 80 e 90 *warehouses*, em ambientes de teste com 2 CPUs e 4CPUs, respetivamente, tal como podemos observar nos gráficos abaixo.



(a) 80 *warehouses* (b) 90 *warehouses*

Figura 3: Curvas de escalabilidade para 2 CPUs



(a) 80 *warehouses* (b) 90 *warehouses*

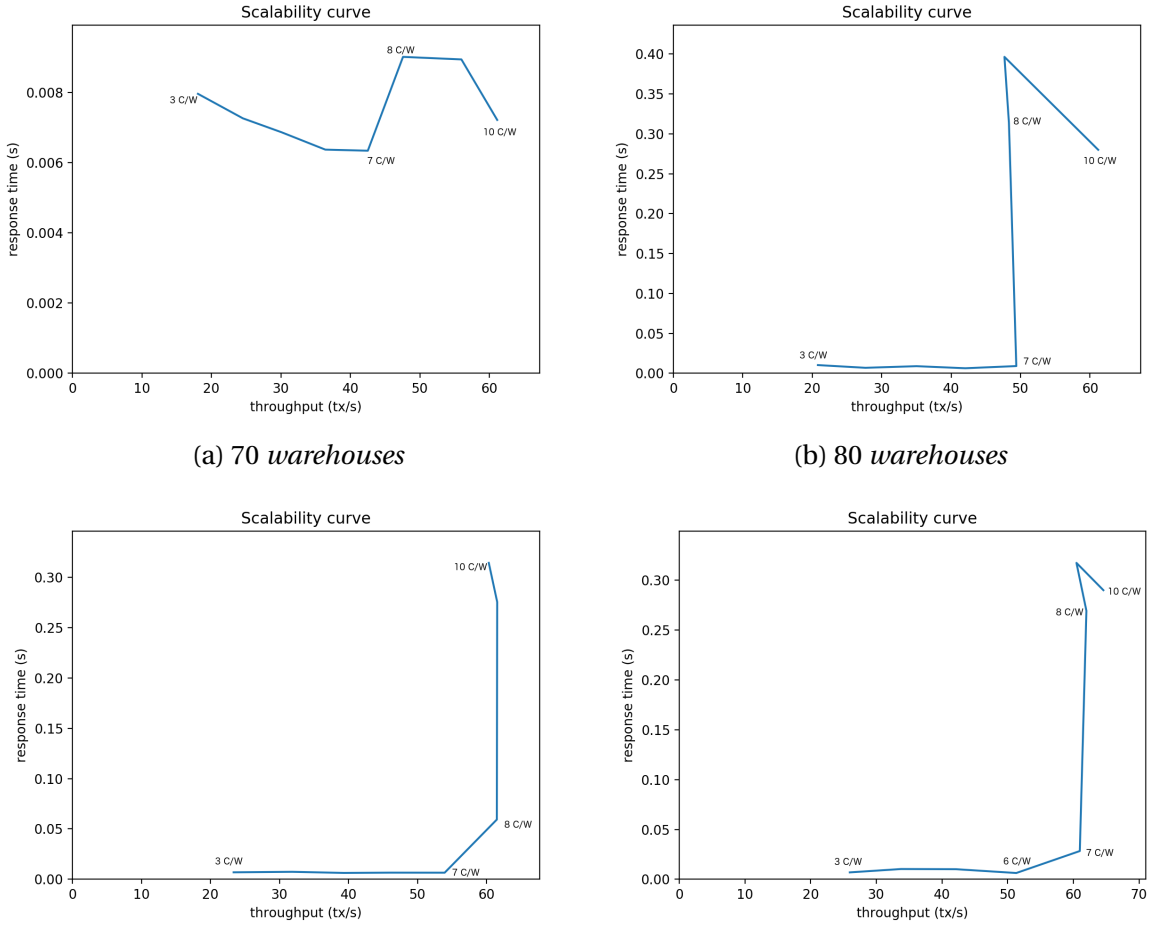
Figura 4: Curvas de escalabilidade para 4 CPUs

Ao contrário do que acontecia com a memória, no caso do CPU, o aumento para o dobro provoca uma melhoria bastante significativa do tempo de resposta. O facto de a utilização de 4 CPUs diminuir para menos de metade o tempo de resposta, faz com que valha a pena investir neste recurso. Sendo assim, a configuração de referência, em termos de *hardware*, irá possuir **4 CPUs**.

2.2 Workload

No que toca à análise do *workload*, os testes efetuados resultaram da variação do **número de *warehouses*** e do **número de *clientes***. Foram avaliados casos com 70, 80, 90 e 100 *warehouses*, sendo que, para cada uma destas configurações, o número de clientes por *warehouse* (c/w) variou entre 3 e 10, resultando num total de $4 \times 8 = 32$ testes.

As curvas de escalabilidade apresentadas abaixo surgiram a partir do agrupamento dos testes efetuados, para cada número de *warehouses*.



(a) 70 *warehouses*

(b) 80 *warehouses*

(c) 90 *warehouses*

(d) 100 *warehouses*

Figura 5: Curvas de escalabilidade para diferentes números de *warehouses*

Ao analisar os gráficos da figura 5, conseguimos observar que o ponto de saturação das curvas se localiza geralmente próximo dos 7 c/w, concluindo então que deverá ser este o número ótimo de clientes por *warehouse*.

Comparando as diferentes curvas obtidas, verifica-se que os testes efetuados para 90 *warehouses* alcançaram os melhores resultados no que toca à relação entre o débito e o tempo de resposta.

Posto isto, a configuração de referência deverá dispor de um *workload* com **90 *warehouses*** e 7 clientes por *warehouse*, perfazendo um total de **630 clientes**.

Em suma, a configuração de referência que será adotada nas secções subsequentes será a seguinte:

Recurso	**Quantidade**
Memória	8GB
CPU	4
Warehouses	90
Clientes	630

Tabela 1: Configuração de referência

3 Otimização

Utilizando a configuração de referência adotada, prossegiu-se para a otimização e justificação do desempenho da mesma. Para isso, serão analisadas as interrogações SQL, os mecanismos de redundância, como índices e vistas materializadas, e os parâmetros de configuração do PostgreSQL.

3.1 Interrogações

O *benchmark* TPC-C possui diversos tipos de interrogações em diferentes transações. Para analisar as interrogações, primeiramente foram analisadas algumas considerações sobre as transações do *benchmark* TPC-C, tais como:

- este envolve 5 transações de diferentes tipos que incluem: fazer e receber encomendas, registar pagamentos, verificar o estado das encomendas e verificar o *stock* nos armazéns;
- a transação mais frequente é a inserção de uma encomenda que, em média, é composta por 10 itens diferentes;
- é produzida uma transação do tipo pagamento por cada transação de nova encomenda;
- são produzidas uma transação do tipo entrega, uma do tipo estado da encomenda e outro do tipo nível de *stock* por cada 10 transações de nova encomenda.

Podemos então verificar que a frequência de cada transação é diferente consoante o seu tipo, ou seja, as interrogações de cada transação vão ser executadas consoante a execução da transação e, dentro da execução da transação, podem ser executadas mais do que uma vez, como acontece, por exemplo, no caso da inserção de uma encomenda que processa, em média, 10 itens em cada transação.

De seguida, analisamos o código SQL de cada uma das transações para analisar todas as interrogações. Posto isto, foi possível constatar que a maior parte de interrogações feitas são:

- *SELECT*, de todas as colunas, de um elemento de uma tabela;
- *UPDATE* de um campo de um elemento de uma tabela;
- *INSERT* de um elemento numa tabela;
- *DELETE* de um elemento numa tabela;
- *ORDER BY*, segundo um determinado critério, de um conjunto de resultados.

Concluímos, então, que a baixa complexidade das interrogações as torna bastante eficazes, e consequentemente, que as transações já têm um nível de otimização muito elevado.

Para além disso, os logs da base de dados foram também analisados através da ferramenta **pgBadger** para retirar informação sobre as interrogações (tipos de interrogação mais frequentes, interrogações mais lentas) para depois facilitar a otimização posterior. Apresentam-se em baixo os resultados obtidos.

Type	Count	Percentage
SELECT	190,159	28.35%
INSERT	0	0.00%
UPDATE	0	0.00%
DELETE	0	0.00%
COPY FROM	0	0.00%
COPY TO	0	0.00%
CTE	0	0.00%
DDL	0	0.00%
TCL	290,506	43.30%
OTHERS	190,195	28.35%

Figura 6: Tipos de interrogação e número de ocorrências

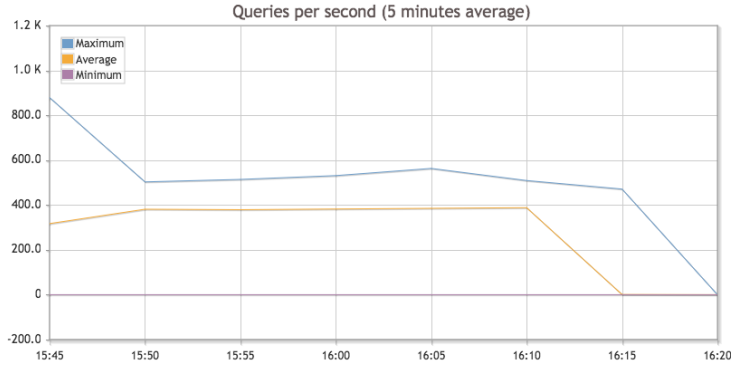


Figura 7: Distribuição temporal das interrogações

Rank	Total duration	Times executed	Min duration	Max duration	Avg duration	Query
1	4m34s	42,656 Details	1ms	3s443ms	6ms	SELECT tpcc_neworder (?, ?, ?, ?, ?, ?, ?, ?, ?); Examples User(s) involved App(s) involved
2	2m38s	92,648 Details	0ms	504ms	1ms	COMMIT TRANSACTION; Examples User(s) involved App(s) involved
3	2m12s	93,507 Details	0ms	5s359ms	1ms	FETCH ALL IN "..."; Examples User(s) involved App(s) involved
4	1m11s	3,955 Details	2ms	4s527ms	18ms	SELECT tpcc_delivery (?, ?); Examples User(s) involved App(s) involved
5	43s245ms	39,163 Details	0ms	807ms	1ms	SELECT tpcc_payment (?, ?, CAST(? AS numeric(?, ?)), ?, ?, ?, CAST(? AS char(?))); Examples User(s) involved App(s) involved

Figura 8: Interrogações mais lentas

3.2 Mecanismos de Redundância

A análise do desempenho dos mecanismos de redundância vai basear-se em índices e vistas materializadas e vai ter por base de comparação os resultados da configuração de referência da tabela 2.

Débito (tx/s)	54.235550097222685
Tempo de resposta (s)	0.004844544554793352
Abort Rate	0.0239669559135

Tabela 2: Resultado da configuração de referência

3.2.1 Índices

Para fazer a otimização dos índices da base de dados, primeiramente foram analisados os índices já existentes na configuração do *benchmark* TPC-C. De seguida foi feita a análise da utilização destes e foram realizados testes para verificar a sua utilidade.

Posto isto, constatámos a existência dos seguintes índices nas diferentes tabelas da configuração predefinida do *benchmark* TPC-C:

- **warehouse**

  `"pk_warehouse"` PRIMARY KEY, btree (w_id)

  `"keywarehouse"` UNIQUE CONSTRAINT, btree (key)

- **customer**

  `"pk_customer"` PRIMARY KEY, btree (c_w_id, c_d_id, c_id)

  `"keycustomer"` UNIQUE CONSTRAINT, btree (key)

  `"ix_customer"` btree (c_w_id, c_d_id, c_last)

- **district**

  `"pk_district"` PRIMARY KEY, btree (d_w_id, d_id)

  `"keydisctrict"` UNIQUE CONSTRAINT, btree (key)

- **history**

  `"keyhistory"` UNIQUE CONSTRAINT, btree (key)

- **item**

  `"pk_item"` PRIMARY KEY, btree (i_id)

"keyitem" UNIQUE CONSTRAINT, btree (key)

- **new_order**

  "keyneworder" UNIQUE CONSTRAINT, btree (key)

  "ix_new_order" btree (no_w_id, no_d_id, no_o_id)

- **order_line**

  "pk_order_line" PRIMARY KEY, btree (ol_w_id, ol_d_id, ol_o_id, ol_number)

  "keyorderline" UNIQUE CONSTRAINT, btree (key)

  "ix_order_line" btree (ol_i_id)

- **orders**

  "keyorders" UNIQUE CONSTRAINT, btree (key)

  "ix_orders" btree (o_w_id, o_d_id, o_c_id)

  "pk_orders" btree (o_w_id, o_d_id, o_id)

- **stock**

  "pk_stock" PRIMARY KEY, btree (s_w_id, s_i_id)

  "keystock" UNIQUE CONSTRAINT, btree (key)

  "ix_stock" btree (s_i_id)

Noutro sentido, e tendo em conta os índices e as tabelas existentes na configuração, vamos abordar diferentes considerações dos índices do PostreSQL, tais como:

- utilizar índices para pesquisas com poucos resultados e com pouca frequência na tabela;
- utilizar índices para acelerar *queries* do tipo *SELECT*, cláusulas *WHERE* e operações de *ORDER BY*;
- não utilizar índices que prejudiquem *queries* de escrita, tais como *INSERT*, *UPDATE* e *DELETE*;
- verificar a utilidade de índices das chaves primárias definidos aquando a criação da tabela;
- não utilizar índices em tabelas pequenas;
- utilizar o tipo de índice adequado às *queries*;
- utilizar duas ou mais colunas nos índices (*multicolumn index*) quando estas são frequentemente usadas em conjunto nos filtros das cláusulas *WHERE*.

Analisando as diferentes transações deste *benchmark*, foi possível verificar onde estava a ser utilizado cada índice e se a sua existência afetava ou melhorava o desempenho.

Verificamos, então, na figura 9, quais as colunas das tabelas que estão a ser utilizadas como cláusulas de condições *WHERE* em interrogações do tipo *UPDATE* e *SELECT*, quais estão a ser utilizadas para operações de *ORDER BY* e quais afetam operações de *INSERT* e *DELETE*.

Table	Column	Primary Key	Index	SQL Statement				
				INSERT	UPDATE	DELETE	SELECT	ORDER BY
customer	c_id	X		0	1	0	1	0
	c_d_id	X	X	0	1	0	1	1
	c_w_id	X	X	0	1	0	1	1
	c_last	X		0	0	0	1	1
	c_first			0	0	0	0	1
district	d_id	X		0	1	0	1	0
	d_w_id	X		0	1	0	1	0
history				1	0	0	0	0
item	i_id	X		0	0	0	1	0
new_order	no_o_id		X	1	0	1	0	1
	no_d_id		X	1	0	1	1	0
	no_w_id		X	1	0	1	1	0
order_line	ol_o_id	X		1	1	0	1	0
	ol_d_id	X		1	1	0	1	0
	ol_w_id	X		1	1	0	1	0
	ol_number	X		1	0	0	0	0
	ol_i_id		X	1	0	0	0	0
orders	o_id	X		1	1	0	1	1
	o_d_id	X	X	1	1	0	1	0
	o_w_id	X	X	1	1	0	1	0
	o_c_id	X		1	0	0	1	0
stock	s_i_id	X	X	0	1	0	1	0
	s_w_id	X		0	1	0	1	0
	s_quantity			0	0	0	1	0
warehouse	w_id	X		0	1	0	1	0

Figura 9: Impacto causado pela existência de índices no *benchmark*

A frequência da ocorrência de cada uma destas interrogações não está representada na figura, utilizando a notação 1 para representar se existe nas transações e 0 caso não exista nas transações. A cor vermelha representa, caso esse índice exista, um impacto negativo no desempenho, enquanto a cor verde representa um impacto positivo. Está também evidenciado na figura qual o tipo de cada coluna (chave primária e/ou índice) na configuração do *benchmark* TPC-C.

Para além disso, analisámos também a ocorrência de conjuntos de colunas como cláusulas de condições *WHERE*. Assim, tendo em conta cada tabela, constatámos a utilização dos seguintes conjuntos:

- **costumer**
  - *c_id, c_d_id, c_w_id*
  - *c_d_id, c_w_id, c_last*
- **district**
  - *d_id, d_w_id*
- **item**
  - *i_id*
- **new_order**
  - *no_o_id, no_d_id, no_w_id*
  - *no_d_id, no_w_id*
- **order_line**
  - *ol_o_id, ol_d_id, ol_w_id*
- **orders**
  - *o_id, o_d_id, o_w_id*
  - *o_d_id, o_w_id, o_c_id*
- **stock**
  - *s_i_id, s_w_id*
  - *s_i_id, s_w_id, s_quantity*
- **warehouse**
  - *w_id*

Posto isto, e tendo em conta os valores da configuração de referência da tabela 2, realizamos diversos testes, quer no que toca a otimizações, quer para provar o já ótimo desempenho da mesma:

- **Adicionar índice na tabela *customer* na coluna *c_first***

débito(tx/s) = 53.85630931718304
tempo de resposta(s) = 0.005530556269820864
abort rate = 0.0263706918134

Observando a figura 9, vemos que a coluna *c_first* é utilizada nas interrogações do tipo *ORDER BY*, pensando que um índice iria acelerar estas operações, melhorando o desempenho final. No entanto, isto não foi verificado.

- **Adicionar *multicolumn index* na tabela *customer* nas colunas *c_w_id, c_d_id, c_last* e *c_first***

  débito(tx/s) = 55.21531860210925
  tempo de resposta(s) = 0.004690006854926353
  abort rate = 0.000300857443715

  Em vez de adicionar um índice apenas na coluna *c_first* da tabela *customer*, adicionamos um *multicolumn index* semelhante ao já existente, que foi removido, com a adição da nova coluna. Como o índice em *c_first* não afeta o desempenho geral, esta mostrou ser uma opção viável que otimiza os resultados da configuração.

- **Remover índice na tabela *customer* na coluna *c_last***

  débito(tx/s) = 54.6817874727884
  tempo de resposta(s) = 0.004891702257474069
  abort rate = 0.0239528882419

  Analisando a coluna *c_last* da tabela *customer*, verificamos que este, apesar de poder não ser considerado índice porque o número de valores repetidos é elevado, melhora o desempenho das interrogações de *SELECT* e *ORDER BY*. Para além disso, este é um *multicolumn index* visto ser utilizado com frequência nas cláusulas *WHERE* com as colunas *c_d_id* e *c_w_id*. Desta forma, a remoção deste índice pirou o desempenho geral.

- **Substituir o *multicolumn index* da tabela *costumer* com as colunas *c_d_id, c_w_id* e *c_last* pelo *multicolumn index* das colunas *c_id, c_d_id* e *c_w_id***

  débito(tx/s) = 53.826153520053836
  tempo de resposta(s) = 0.005162195625257945
  abort rate = 0.0252112109427

  O conjunto das colunas formado por *c_id*, *c_d_id* e *c_w_id*, apesar ser utilizado com maior frequência do que o índice *c_d_id*, *c_w_id* e *c_last*, tem um impacto bastante negativo no desempenho das *queries* de escrita, logo, mostra não ser uma opção viável. Desta forma, a utilização deste índice piora o desempenho geral.

- **Adicionar *multicolumn index* na tabela *district* nas colunas *d_id* e *d_w_id***

  débito(tx/s) = 54.302101116950645
  tempo de resposta(s) = 0.004906246789273606
  abort rate = 0.0245377252911

Após a análise dos *multicolumn indexes* vimos que estas duas colunas são bastante usadas em conjunto nas cláusulas *WHERE*. Desta forma, decidimos testar como funcionaria aqui um índice. No entanto, como podemos verificar nos resultados obtidos, este índice não melhora o desempenho, porque prejudica as interrogações de *UPDATE*, prejudicando o desempenho geral.

- **Adicionar índice na tabela *item* na coluna *i_id***

débito(tx/s) = 54.0346339009844
tempo de resposta(s) = 0.004839182094501244
abort rate = 0.0230965194945

Um índice na coluna *i_id* não afeta o desempenho de nenhuma *querie* de escrita. Desta forma, criando aqui um índice pudemos comprovar uma melhoria no desempenho apesar de não ter sido significativa.

- **Remover índice *ix_order_line* na tabela *order_line* na coluna *ol_i_id***

débito(tx/s) = 54.13835326047159
tempo de resposta(s) = 0.004192733747667232
abort rate = 0.0219695574273

Optamos por testar remover este índice, visto que este não estava a ser praticamente utilizado e estava a prejudicar o desempenho das interrogações de *INSERT*. Constatámos que melhorou o tempo de resposta, mostrando ser uma otimização possível.

- **Adicionar *multicolumn index* na tabela *order_line* nas colunas *ol_o_id, ol_d_id* e *ol_w_id***

débito(tx/s) = 54.13206988341312
tempo de resposta(s) = 0.004608013309778228
abort rate = 0.0242175732218

Após a análise das interrogações constatámos que estas três colunas são bastante usadas em conjunto nas cláusulas *WHERE*. Desta forma, decidimos testar como funcionaria um índice nestas colunas. No entanto, como podemos verificar nos resultados obtidos, este índice não melhora o desempenho, porque prejudica as interrogações de *INSERT* e *UPDATE*, prejudicando o desempenho geral.

- **Remover índice na tabela *orders* nas colunas *o_w_id, o_d_id* e *o_c_id***

débito(tx/s) = 54.13201270575716
tempo de resposta(s) = 0.005020821204753263
abort rate = 0.0239818172703

Visto que estes índices afetam as operações de escrita, decidimos experimentar removê-lo. No entanto, isto mostrou ser uma má opção pois prejudicou o desempenho geral.

- **Adicionar índice na tabela *orders* nas colunas *o_w_id*, *o_d_id* e *o_id*, removendo o índice existente nas colunas *o_w_id*, *o_d_id* e *o_c_id***

  débito(tx/s) = 54.063669388348806
  tempo de resposta(s) = 0.004969290879802585
  abort rate = 0.0241153254398

  Visto que a frequência de utilização das colunas *o_w_id*, *o_d_id* e *o_id* em conjunto em cláusulas *WHERE* é superior, esta poderia ser uma opção viável. No entanto, estes índices afetam negativamente o desempenho e, como tal, não devem ser utilizados.

- **Adicionar índice na tabela *stock* nas colunas *s_i_id* e *s_w_id***

  débito(tx/s) = 54.15578800480237
  tempo de resposta(s) = 0.004977642241600851
  abort rate = 0.0250058463903

  Embora as colunas *s_i_id* e *s_w_id* sejam frequentemente utilizadas em conjunto, estas provocam um impacto negativo no desempenho, logo, não representam uma boa opção de otimização.

- **Alterar o tipo do índice *ix_stock* da tabela *stock* na coluna *s_i_id* de *B-tree* para *Hash***

  débito(tx/s) = 53.946157248016604
  tempo de resposta(s) = 0.005182971076501798
  abort rate = 0.0251268019221

  Tendo em conta que todas as operações utilizadas por este índice são do tipo '=', uma possibilidade a considerar foi alterar o tipo de índice de *B-tree* para *Hash*. No entanto, observando os resultados obtidos, podemos concluir que esta não é uma boa opção visto que piora o desempenho geral.

Analisando todos os resultados obtidos, podemos concluir que, relativamente a índices, as opções viáveis de otimização são:

- Adicionar um *multicolumn index* na tabela *customer* com as colunas *c_w_id*, *c_d_id*, *c_last* e *c_first*, removendo o índice já existente;
- Remover o índice *ix_order_line* na tabela *order_line*, que quase não é utilizado;
- Adicionar um índice na coluna *i_id* da tabela *item*, apesar da sua presença não ser muito significativa.

3.2.2 Vistas Materializadas

Ainda no processo de otimização baseado em mecanismos de redundância, analisamos a possibilidade de criar vistas materializadas. Para isso, vamos analisar o tipo de interrogações feitas a cada tabela e, aplicando determinados critérios, testar diversas vistas e verificar a possível otimização do desempenho.

Para fazer esta análise vamos abordar diferentes considerações das vistas materializadas do PostreSQL, tais como:

- considerar o espaço em disco utilizado, tendo em conta que este é mais *'barato'* do que as operações de leitura e escrita de dados;
- definir vistas materializadas baseadas em tabelas pequenas;
- não utilizar vistas materializadas em tabelas sujeitas a muitas operações de *INSERT*, *UPDATE* e *DELETE*.

Assim, a primeira análise a ser feita, representada na figura 10, foi ao tipo de interrogações a que as tabelas são sujeitas na execução das transações.

Table	SQL Statement			
	INSERT	UPDATE	DELETE	SELECT
customer		■		■
district		■		■
history	■			
item				■
new_order	■		■	■
order_line	■	■		■
orders	■	■		■
stock		■		■
warehouse		■		■

Figura 10: Tipos de interrogações efetuadas a cada tabela

A frequência da ocorrência de cada interrogação não está representada na figura, sendo que a coloração verde representa apenas a existência, nas transações, do tipo de interrogações executadas na tabela.

Atendendo ao facto de a grande maioria das transações executar as interrogações coletando todas as colunas da tabela referentes a uma entrada apenas, no exemplo específico deste *benchmark*, na tentativa de otimizar uma interrogação individualmente, não nos foi possível definir vistas materializadas que envolvessem *queries* de elevada complexidade.

Mesmo assim, foram efetuados testes criando vistas materializadas *'mais simples'*, tais como:

- **Criar uma vista materializada da tabela *warehouse***

```
CREATE MATERIALIZED VIEW matview_warehouse AS
    SELECT * FROM warehouse;
CREATE OR REPLACE FUNCTION refresh_matview_warehouse()
    RETURNS trigger LANGUAGE plpgsql AS
    $$ BEGIN REFRESH MATERIALIZED VIEW matview_warehouse;
    RETURN NULL;
    END;
    $$;
CREATE TRIGGER tg_refresh_matview_warehouse
    AFTER INSERT OR UPDATE OR DELETE ON warehouse
    FOR EACH STATEMENT EXECUTE PROCEDURE
    refresh_matview_warehouse();
```

Neste caso, devido ao facto de a tabela *warehouse* ser relativamente pequena, foi definida uma vista materializada de toda a tabela suportando as interrogações de *SELECT*. No entanto, esta tabela está sujeita a interrogações do tipo *UPDATE*, ou seja, a vista materializada tem de ser atualizada.

débito(tx/s) = 53.370064551807054
tempo de resposta(s) = 0.005779759731566961
abort rate = 0.036710153908

Analisando os resultados do teste podemos concluir que esta opção piora o desempenho geral.

- **Criar uma vista materializada da tabela *item***

```
CREATE MATERIALIZED VIEW matview_item AS SELECT * FROM item;
```

Neste caso, como esta tabela apenas suporta operações de *SELECT*, não tendo de ser atualizada, foi definida uma vista materializada baseada em toda a tabela *item*. No entanto, como a tabela é relativamente maior, podemos ter em conta o espaço por ela ocupado, que é de ... .

débito(tx/s) = 55.277130277130276
tempo de resposta(s) = 0.0048669154436988575
abort rate = 0.000568628434767

Analisando os resultados obtidos, podemos concluir que, devido à grande melhoria do débito e da *abort rate*, o aumento do tempo de resposta não é significativo o suficiente para esta otimização ser descartada.

Tendo em conta os testes realizados, podemos concluir que existem 4 otimizações, individuais, possíveis referentes a mecanismos de redundância.

Id	Teste	Débito	Tempo de resposta	Abort Rate
0	**Configuração de Referência**	54.235550097222685	0.004844544554793352	0.0239669559135
1	***multicolumn index* com *c_w_id*, *c_d_id*, *c_last* e *c_first***	55.21531860210925	0.004690006854926353	0.000300857443715
2	**remover índice *ix_order_line***	54.13835326047159	0.004192733747667232	0.0219695574273
3	**índice em *i_id***	54.0346339009844	0.004839182094501244	0.0230965194945
4	***materialized view* em *item***	55.277130277130276	0.0048669154436988575	0.000568628434767

Tabela 3: Resultado das otimizações individuais dos mecanismos de redundância em relação à configuração de referência

No entanto, estas otimizações podem ser combinadas, podendo não obter apenas resultados melhores. Desta forma, vamos combinar diferentes otimizações, tentando encontrar a que represente o melhor desempenho possível.

Teste	Débito	Tempo de resposta	Abort Rate
1+2	54.15312915618551	0.004582572259278263	0.0238405288991
1+2+3	54.397563973028106	0.004370288399332629	0.0223368841545
2+3	53.62958400489794	0.004460688066296444	0.0225691526905
2+4	55.285268730876936	0.004402111356487989	0.000601926163724
2+3[1]+4	53.76604773652195	0.004519017623571528	0.0236754994707
1+2+4	54.096759711377786	0.004635666740815524	0.02300349399
1+4	54.33548708863398	0.005382909495676001	0.0252879318978

Tabela 4: Resultado do conjunto das otimizações dos mecanismos de redundância em relação à configuração de referência

[1] Índice aplicado na *materialized view* em vez de na tabela

3.3 Parâmetros de Configuração do PostgreSQL

Outras vertentes de otimização de desempenho são encontradas nos parâmetros de configuração do PostgreSQL. Através da utilização de diferentes parâmetros conseguimos controlar a utilização de memória ou a utilização dos Write-Ahead Logs, por exemplo.

3.3.1 Utilização de Memória

Começando pela utilização de memória, as configurações do PostgreSQL oferecem controlo sobre as diferentes maneiras como a memória é alocada e utilizada, sendo que para este trabalho nos iremos focar nos três seguintes parâmetros: *shared_buffers*, *effective_cache_size* e *work_mem*.

O parâmetro *shared_buffers* define a quantidade de memória utilizada para a cache dedicada do PostgreSQL. Para além desta memória dedicada, também é utilizada a cache do OS para guardar informação, apesar da cache dedicada dar mais garantias sobre os dados armazenados. Por outro lado, a cache dedicada é mais lenta do que a do OS. A utilização de mecanismos de caching permite reduzir os acessos ao disco, melhorando a performance.

O *effective_cache_size* configura o tamanho de cache (dedicada ou de OS) disponível para ser utilizada pelo postgreSQL. No entanto, esta variável não reserva a cache do OS ou altera o espaço alocado para a cache dedicada em *shared_buffers*, apenas é utilizado pelo planeador para estimar o espaço de cache com que pode contar. O aumento do valor deste parâmetro leva a uma maior utilização dos índices.

O *work_mem* define o tamanho da memória utilizada para operações de ordenação (*ORDER BY*, *DISTINCT* e *MERGE JOIN*) e para operações sobre tabelas de Hash (*Hash Joins* e *Hash-Based Aggregation*). O tamanho definido é para cada operação de ordenação ou Hash a decorrer, podendo levar à utilização de múltiplos do tamanho definido visto que estas operações podem ocorrer em paralelo.

Através da análise de cada parâmetro, foram estabelecidos casos para teste, de forma a melhorar a utilização de memória e consequentemente, a performance do *benchmark*. A seguir são apresentados os testes para cada parâmetro, assim como a justificação do valor ótimo e conclusões.

- ***shared_buffers***

Tamanho de shared buffers	Débito	Tempo de resposta	Abort Rate
128MB	54.35687513066331	0.005774214041654505	0.0271277040007
256MB	53.74454902718243	0.005446817333609787	0.0263286283878
512MB	53.47686495298685	0.005990996128853865	0.0286222699726
1GB	53.551518878093404	0.006398018295318959	0.0297636025032
2GB	53.80717243679416	0.006448639807162535	0.0290710309433
4GB	53.390336600668704	0.006347384047088879	0.0291914537076

Tabela 5: Resultado das otimizações do tamanho da cache dedicada do PostgreSQL

Como podemos verificar, a configuração com melhores resultados foi os 256MB, valor acima do *default* para esta *cache*, mas menor do que o valor esperado pelo grupo depois da análise do *shared_buffers*.

Uma razão porque os tamanhos de *cache* dedicada maiores aumentem o tempo de resposta pode ser o facto de a *cache* dedicada ser de mais lento acesso do que a de OS, levando a um melhor desempenho quando uma parte maior dos acessos à *cache* são feitos à *cache* do sistema.

- ***effective_cache_size***

Estimativa do tamanho da cache	Débito	Tempo de resposta	Abort Rate
1GB	53.9233159929153	0.005651724078569093	0.0277340742476
2GB	54.27778083502699	0.005598044438334508	0.0264496269281
4GB	55.46190810131689	0.005666894642023086	0.00046685340803
6GB	53.57349345652779	0.006461571667846599	0.0291035893309

Tabela 6: Resultado das otimizações da estimativa do tamanho da cache

Como podemos verificar, obtivemos os melhores resultados entre 2GB e 4GB, com uma diminuição do desempenho para valores acima de 4GB. Posto isto, o valor ótimo escolhido é 4GB.

Ao aumentar a estimativa do tamanho da cache, o planeador pode fazer planos que envolvam menos acesso a disco, melhorando assim o desempenho do *benchmark*.

- ***work_mem***

Tamanho da Work Memory	Débito	Tempo de resposta	Abort Rate
16MB	53.82185089902693	0.005780842475677913	0.0298872117541
64MB	53.95745437768551	0.005648483911953574	0.0270297360508
128MB	53.51723425412937	0.005226971869829012	0.0256616955529
256MB	54.16018017232825	0.005276102922294695	0.0266284514506

Tabela 7: Resultado das otimizações do tamanho da Work Memory

O tamanho ótimo para *work_mem* encontrado foi 128MB. Apesar de 256MB apresentar uma performance parecida, o facto desta memória ser dedicada a apenas uma operação e de várias operações poderem correr em paralelo leva à escolha do valor mais pequeno, que apresenta uma performance semelhante.

No entanto, a análise das transações feita anteriormente mostra que as operações que envolvem *work_mem* não devem causar este tipo de problemas, no contexto do TPC-C.

Visto que não foram escolhidos tamanhos de memória demasiado grandes para os 8GB de RAM disponíveis, as otimizações escolhidas não devem levar ao uso excessivo da memória. As otimizações irão levar a um uso maior da memória, levando a menos acessos a disco e melhor performance.

No fim dos testes de utilização de memória, chegou-se à seguinte configuração de parâmetros.

Shared Buffers	256MB
Effective Cache Size	4GB
Work Mem	128MB

Tabela 8: Resultado das configuração de utilização de memória

3.3.2 Write-Ahead Log

Para garantir a integridade dos dados o PostgreSQL utiliza a técnica de *Write-Ahead Logging* (WAL). Isto consiste em guardar informação sobre as transações de uma base de dados num *log* antes de efetuar as mudanças nos ficheiros de dados (tabelas e índices), para recuperar as alterações no caso de uma falha no sistema e tornar as escritas assíncronas.

De modo a não ter de recuperar o WAL todo em caso de falha, são utilizados *checkpoints* para marcar pontos na sequência de transações (presentes no *log*) onde os ficheiros de dados persistentes têm toda a informação escrita até esse ponto. Deste modo, os dados sobre as transações no *log* antes de um *checkpoint* podem ser removidos. Em situação de falha, apenas é preciso recuperar os dados desde o último *checkpoint*.

A frequência de checkpoints e a sua distribuição ao longo do tempo são fatores que influenciam a performance da base de dados, visto que a escrita das páginas com alterações (*dirty pages*) dos *shared buffers* para disco é uma operação pesada.

Dado isto, os parâmetros de configuração de *checkpoint* foram analisados e modificados para verificar qual a configuração com melhor performance, balançando o número de *checkpoints* e a sua duração para obter bons resultados tanto durante a execução normal como em recuperação de falhas.

Para isto, foram alterados três parâmetros nas configurações do PostgreSQL: *max_wal_size*, *checkpoint_timeout* e *checkpoint_completion_target*.

Os parâmetros *max_wal_size* e *checkpoint_timeout* definem a frequência dos *checkpoints*, pois configuram o tamanho máximo que o WAL pode atingir antes do *checkpoint* e o tempo entre *checkpoints*, respetivamente. O parâmetro *checkpoint_completion_target* define a distribuição temporal das operações do *checkpoint*, em termos de fração de tempo entre dois *checkpoints*. Ou seja, define a fração de tempo entre dois *checkpoints* onde as escritas do *checkpoint* são feitos, distribuindo a carga de IO ao longo do tempo.

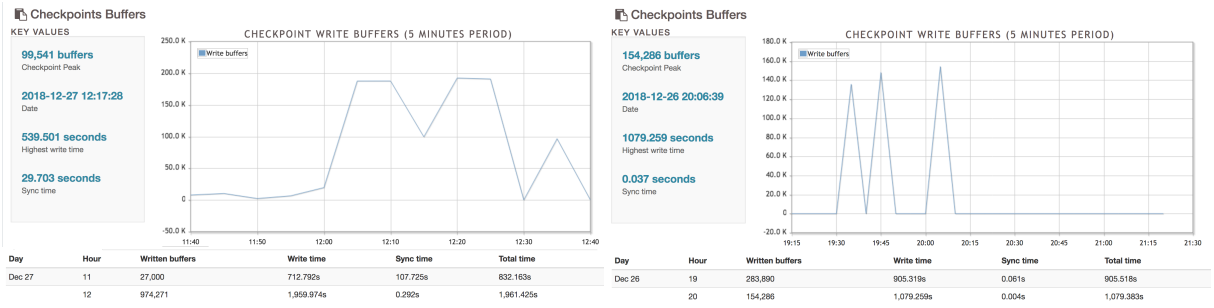
O objetivo das otimizações de *checkpoints* incluem: redução de escritas de *checkpoints* (para diminuir o impacto destas e reduzir escritas de dados "obsoletos"), otimização do espaçamento temporal das escritas (espalhando as escritas ao longo do tempo) e diminuição do tempo de sincronização para disco. Com estas otimizações, a quantidade de escritas e o impacto destas devem reduzir, melhorando o desempenho.

Para a análise do desempenho, para além das medidas utilizadas até aqui (tempo de resposta e débito) foi também utilizada a análise do PGBadger aos *logs* da base de dados. Com o objetivo de minimizar o impacto dos *checkpoints*, foram testadas várias configurações dos parâmetros, apresentadas abaixo.

Checkpoint Timeout (mins)	Max WAL Size	Completion Target	Débito	Tempo de resposta
Default (5)	**Default (1GB)**	**Default (0.5)**	53.361545593205136	0.00591072929802438
10	**2**	**0.5**	55.517265656554414	0.006948489904889039
10	**4**	**0.9**	55.54419373376505	0.005251457798807106
15	**4**	**0.9**	55.50998745537005	0.005806238090462342
20	**6**	**0.9**	55.30204740941165	0.006746311304202525

Tabela 9: Resultado das otimizações dos parâmetros dos checkpoints

Comparando os resultados e os gráficos produzidos pelo *pgBadger*, podemos concluir que o *checkpoint_completion_target* deve ser aumentado para 0.9, ou seja, 90% do tempo do checkpoint deve ser utilizado para a escrita das *dirty pages* e 10% para o *fsync* destas para disco. Isto deve-se à escrita das páginas com alterações ser uma operação custosa enquanto o *fsync* é mais rápido caso todas as mudanças já estejam escritas, diminuindo o tempo de sincronização para disco.

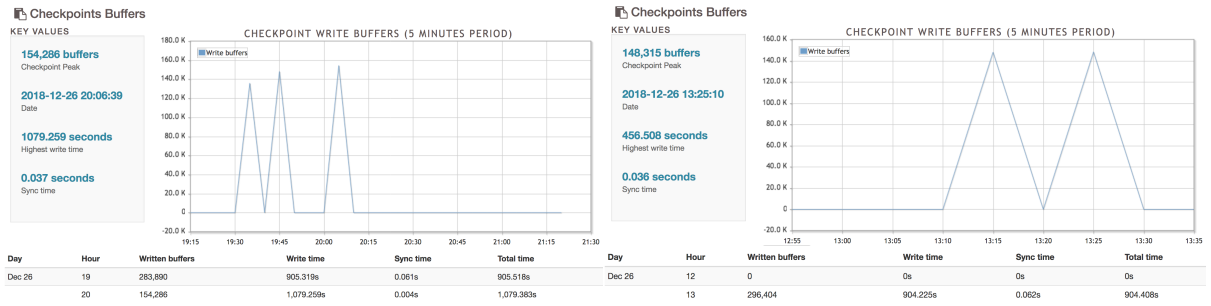


(a) *checkpoint_completion_target = 0.5* (b) *checkpoint_completion_target = 0.9*

Figura 11: Análise das escritas com diferentes *checkpoint_completion_target*

Como podemos verificar, o tempo de sincronização diminiu, assim como o tempo de escrita total.

Outra otimização possível é aumentar os parâmetros *checkpoint_timeout* e *max_wal_size* para diminuir o número de *checkpoints* que ocorrem, diminuindo a carga que cada *checkpoint* tem para o desempenho. No entanto, diminuir o número de *checkpoints* pode piorar o desempenho do *crash recovery*.



(a) *checkpoint_timeout = 10 mins* (b) *checkpoint_timeout = 15 mins*

Figura 12: Análise das escritas com diferentes *checkpoint_timeout*

Comparando estas duas configurações, conseguimos perceber que espaçar os *checkpoints* resulta num tempo de escrita total menor com o mesmo tempo de sincronização, resultando numa carga menor de IO. O número de dados escritos também é menor, pois são escritos menos dados "obsoletos".

Dado este resultado, foi escolhida a configuração com *checkpoint_timeout* de **15 minutos**, *max_wal_size* de **6GB** e *checkpoint_completion_target* de **0.9**.

3.3.3 Isolamento

Relativamente ao nível de isolamento do *benchmark*, foram analisados os três tipos de isolamento suportados pelo PostgreSQL, nomeadamente, *Read Committed*, o nível de isolamento padrão, *Repeatable Read* e *Serializable*.

Para isso vamos analisar os problemas que podem ocorrer no *benchmark*, tais como *lost update*, *dirty read*, *non-repeatable read*, *phantoms* e *write skews*, e os problemas resolvidos ou não por cada nível de isolamento do PostgreSQL.

Primeiramente verificamos quais os problemas existentes no *benchmark* TPC-C, constatando a ocorrência de todos. No entanto não foi verificada a frequência do acontecimento destes. Na tabela 10 podemos verificar um exemplo de transações, para cada tipo de problema, onde podem ocorrer.

Lost Update	New Order -> New Order
Dirty Read	Payment -> Delivery
Non-repeatable Read	New Order -> Order Status
Phantoms	Delivery -> New Order
Write Skews	New Order -> New Order

Tabela 10: Transações onde pode ocorrer cada tipo de problema de isolamento

No que diz respeito aos níveis de isolamento, o PostgreSQL permite apenas três tipos, visto que o *read uncommitted* se comporta como o *read committed*. Para além disso, o *repeatable read* não permite *phantoms*. Está representado na tabela 11 quais os problemas permitidos, ou não, por cada tipo de isolamento.

	Read Committed	***Repeatable Read***	***Serializable***
Lost Update	Possível	Impossível	Impossível
Dirty Read	Impossível	Impossível	Impossível
Non-repeatable Read	Possível	Impossível	Impossível
Phantoms	Possível	Impossível	Impossível
Write Skews	Possível	Possível	Impossível

Tabela 11: Níveis de isolamento do PostgreSQL e problemas relacionados

Após realizar esta análise, foram executados testes para cada tipo de isolamento.

Nível de Isolamento	**Débito (tx/s)**	**Tempo de Resposta (s)**	***Abort rate***
Read Committed	54.235550097222685	0.004844544554793352	0.0239669559135
Repeatable Read	54.16828235189347	0.005047557884641733	0.0242630840184
Serializable	53.45835938042648	0.005813869687992655	0.0283468278149

Tabela 12: Resultados da execução do *benchmark* para os diferentes tipos de isolamento

Tendo em conta os resultados obtidos, podemos concluir que tanto o nível *Read Committed* como *Repeatable Read* são bastante favoráveis em termos de desempenho. Isto deve-se ao facto de, em comparação com o nível *Serializable*, ambos possuirem débito superior para um tempo de resposta e uma *abort rate* inferiores. Posto isto, a dúvida recai sobre o *Read Committed* e o *Repeatable Read*, cuja decisão vai depender do objetivo pretendido. Caso o desempenho seja mais relevante, então deverá optar-se pelo *Read Committed*, visto que este proporciona tempos de resposta inferiores. Em contrapartida, se o objetivo principal for a resolução de problemas, isto é, se se pretender sobretudo efetuar as transações desejadas independentemente do tempo de resposta atingido, então o *Repeatable Read* deverá ser o nível de isolamento adotado.

No caso específico do *benchmark* TPC-C, como não temos um conhecimento preciso acerca da frequência com que ocorre cada problema gerado por concorrência, não é possível decidir entre o *Repeatable Read* e o *Read Committed*, que são as opções mais viáveis de escolha.

4 Mecanismos de escalabilidade

4.1 Processamento distribuído

Com o intuito de, mais uma vez, melhorar o desempenho obtido, optou-se por separar o *benchmark* da base de dados segundo uma arquitetura distribuída. Para isso utilizaram-se duas instâncias da Google Cloud Platform diferentes, sendo que cada uma possui a seguinte configuração:

Máquina	CPU	Memória
Benchmark	1 vCPU	3.75GB
Base de dados	4 vCPUs	8GB

Tabela 13: Especificações de *hardware* do processamento distribuído

De acordo com esta arquitetura, o *benchmark* está a ser executado numa máquina, enquanto as transações relativas à base de dados estão a correr noutra. Tendo isto em conta, seria de esperar que os resultados de performance obtidos fossem melhores, dado que a carga imposta pela execução do *benchmark* e pela base de dados está distribuída por duas máquinas distintas. Embora a sobrecarga de CPU e memória em cada instância seja menor, os resultados atingidos demonstraram o contrário.

Arquitetura	Débito (tx/s)	Tempo de resposta (s)	Abort rate
Monolítica	54.235550097222685	0.004844544554793352	0.0239669559135
Distribuída	52.659719134550485	0.007595512153848867	0.0459926680792

Tabela 14: Comparação entre resultados da configuração monolítica e da configuração distribuída

Os resultados acima apresentados foram realizados com a *workload* da configuração de referência, nomeadamente 90 *warehouses* e 630 clientes. No entanto, os resultados obtidos para outras configurações demonstraram a mesma discrepância entre os resultados. Tal como podemos observar através da tabela 14, os resultados provenientes do processamento distribuído foram piores, em todos os aspetos, que os obtidos anteriormente com uma arquitetura monolítica. A explicação mais viável para esta situação é a possibilidade de a latência existente entre as duas máquinas se sobrepor ao expectável ganho de desempenho originado pela separação dos dois componentes. Dado isto, o processamento distribuído do *benchmark* não deve ser tido como uma opção viável, visto que a carga imposta

no sistema pela execução do *benchmark* é muito reduzida, logo, os recursos despendidos acabam por ser desperdiçados.

4.2 Sharding

Dado que o *benchmark* TPC-C lida com grandes quantidades de dados e aplica sobre estes diversas interrogações, considerou-se a utilização de *sharding* para particionar os dados. Esta estratégia foi adotada considerando que, ao separar horizontalmente os dados por diferentes servidores, a quantidade de informação presente em cada tabela e em cada base de dados seria menor, logo, a performance de pesquisa seria superior.

No contexto deste projeto, a forma mais razoável de dividir os dados seria por região, isto é, *warehouses* dentro da mesma região deveriam estão contidos no mesmo *shard*. No entanto, dado que neste caso específico de estudo não é possível identificar regiões evidentes nas quais os *warehouses* estão inseridos, assumiu-se a existência de três regiões fictícias. Uma vez que a configuração de referência contempla 90 *warehouses*, decidiu-se que cada região é então composta por 30 *warehouses*, e que esses *warehouses* possuem chaves adjacentes, isto é:

- **região A -** contém os *warehouses* cujo $w\_id \in [1,31[$
- **região B -** contém os *warehouses* cujo $w\_id \in [31,61[$
- **região C -** contém os *warehouses* cujo $w\_id \in [61,91[$

Embora no caso anterior (processamento distribuído) a latência proveniente da separação dos componentes tenha prejudicado o desempenho geral do *benchmark*, no caso do *sharding* consideramos que a opção mais produtiva seria distribuir os *shards* por diferentes servidores. Sendo assim, para além da máquina utilizada para correr o *benchmark* fora criadas mais três instâncias. Já que o processamento distribuído não originou resultados positivos, neste cenário o *benchmark* e a base de dados principal estarão inseridos na mesma máquina. Posto isto, ao todo serão utilizadas 4 máquinas, cada uma com 4 vCPUs e 8GB de memória.

4.2.1 Particionamento de tabelas

Para ser possível a realização do *sharding* foi necessário particionar as tabelas de acordo com o *warehouse* associado. Para este efeito, o *script* responsável pela criação de tabelas do Escada TPC-C (*createtable.sql*) foi alterado para interrogações do género da seguinte:

```
CREATE TABLE table
(
    ...
    t_w_id int NOT NULL,
    ...
) PARTITION BY RANGE (t_w_id);
```

No excerto acima, `t_w_id` representa o atributo que associa a tabela a um determinado *warehouse* ou, no caso da tabela *warehouse*, representa o próprio identificador.

Ao criar as tabelas particionadas segundo um atributo, isto permite que mais tarde possam ser criadas partições dessas mesmas tabelas, com intervalos específicos de valores desse atributo, que é exatamente o que é pretendido. Tomando como exemplo o excerto acima, ao criar a tabela `table` com particionamento segundo o atributo `t_w_id`, mais tarde pode-se originar uma partição de `table` onde $\texttt{t\_w\_id} \geq 1$ e $\texttt{t\_w\_id} < 31$.

4.2.2 Criação dos *shards*

Em cada uma das máquinas relativas a um *shard* foi necessário criar uma base de dados com as tabelas que nela serão armazenadas. Como a única tabela que não está diretamente relacionada com um *warehouse* específico é a `item`, esta foi a única que não foi criada nas bases de dados dos *shards*.

A criação dos *shards* foi efetuada com recurso ao módulo `postgres_fdw` (*foreign-data wrapper*) do PostgreSQL. Este módulo permite o acesso a dados armazenados em servidores PostgreSQL externos, que neste caso serão os três servidores que conterão os *shards*. Com isto, foram então criadas três partições de cada tabela do *benchmark* TPC-C, cada uma com um dos intervalos de valores mencionados anteriormente. Para que os dados sejam armazenados nos servidores de cada *shard*, as partições criadas foram como *foreign tables* pertencentes a diferentes servidores. Posto isto, para cada tabela do *benchmark* e para cada intervalo de valores, foi executada uma interrogação deste género:

```
CREATE FOREIGN TABLE table_01
PARTITION OF table (t_w_id)
FOR VALUES FROM (1) TO (31)
SERVER shard_01;
```

Após este processo, as interrogações que chegarem à base de dados principal serão encaminhadas para os respetivos *shards*, mediante o *warehouse* associado.

Uma vez concluída a execução do *benchmark* para a configuração de referência (90 *warehouses* e 630 clientes), os resultados obtidos foram os seguintes:

Configuração	**Débito (tx/s)**	**Tempo de resposta (s)**	**Abort rate**
Referência	54.235550097222685	0.004844544554793352	0.0239669559135
Com *sharding*	2.4038186957189436	10.886987856388597	0.079241614001

Tabela 15: Comparação entre resultados da configuração de referência e da configuração com *sharding*

Tal como podemos observar na tabela acima, a discrepância entre os resultados obtidos foi enorme. Ao analisar os valores absurdos resultantes desta abordagem concluímos que

esta configuração de *sharding* não é a mais correta. Ao distribuir os dados por diferentes servidores, o objetivo seria que cada servidor consegui-se lidar com um domínio de clientes, isto é, que cada cliente acabasse por necessitar de comunicar apenas com o *shard* existente na sua região. Isto faria com que o desempenho aumentasse significativamente, dado que cada *shard* possuiria muito menos informação para processar.

No âmbito deste projeto, esta estratégia de *sharding* é praticamente impossível de concretizar visto que todos os clientes partem do mesmo ponto, que neste caso é a localização do *benchmark*. Tendo isto em conta, o simples facto de não ser possível fazer esta separação de clientes por região, e consequentemente por *warehouse*, deixa de ser possível.

Para além deste aspeto, ao realizar o *sharding* a tabela *item* deveria ter sido replicada em cada *shard*, de forma a que não fosse necessário cada *shard* aceder à base de dados principal sempre que necessitasse de consultar um item.

Posto isto, a forma como se executou o *sharding* não foi, de todo, a forma mais correta, visto que os *shards* em vez de estarem a melhorar o desempenho ao dividirem o processamento das transações por vários pontos, estão simplesmente a servir de "armazéns"remotos de informação. Isto fará com que o processamento seja todo realizado na base de dados principal e esta aceda apenas aos *shards* para consultar e inserir dados. Apesar de a consulta de dados ser aparentemente mais eficiente, dado que se tratam de bases de dados mais pequenas, o custo e a latência inerentes ao armazenamento remoto acabam por ser muitíssimo mais dispendiosos que executar todo o processo localmente. Sendo assim, o *sharding* feito através do particionamento de tabelas e da utilização de *foreign-data wrapper* não deve ser utilizado com o *benchmark* TPC-C.

4.2.3 Abordagem alternativa

Uma alternativa ao *sharding* feito através do particionamento de tabelas é a alteração da forma como o *benchmark* Escada TPC-C realiza as transações. Isto é, uma abordagem muito mais simplista do *sharding* seria simplesmente criar três bases de dados distintas, uma em cada servidor, sendo que cada uma seria responsável por uma das regiões mencionadas no início desta secção. Neste cenário, a única alteração que necessitaria de ser realizada era a filtragem das transações por *warehouse*, e o seu posterior encaminhamento para o *shard* respetivo. Para isto o *benchmark* necessitaria de lidar não com uma conexão à base de dados mas com três conexões diferentes, uma para cada *shard*. Ao iniciar uma transação o *benchmark* deveria ser capaz de reconhecer em qual das regiões a transação deveria ser processada, mediante o *warehouse* a ela associado, e encaminhá-la de acordo com o mesmo.

4.3 Replicação

Mais uma vez com o intuito de melhorar o desempenho do *benchmark* decidiu-se utilizar replicação da base de dados. Para além de alta disponibilidade, esta estratégia permite também maior eficiência em termos de leitura, o que, consequentemente vai deverá melhorar a performance.

Para este caso optou-se por criar apenas uma réplica, situada na mesma localização da base de dados (Carolina do Sul). Como a base de dados *master* e o *benchmark* correm na mesma máquina, foi apenas necessário criar duas instâncias da Google Cloud Platform, ambas com 4 vCPUs e 8GB de memória. Após configurar devidamente as bases de dados *master* e *slave*, procedeu-se à execução do *benchmark*.

Configuração	**Débito (tx/s)**	**Tempo de resposta (s)**	**Abort rate**
Referência	54.235550097222685	0.004844544554793352	0.0239669559135
Com replicação	52.91683691486234	0.008762538651014099	0.0434789873841

Tabela 16: Comparação entre resultados da configuração de referência e da configuração com replicação

Uma vez que de entre as cinco transações possíveis, duas delas são de consulta (*order status* e *stock level*), previa-se que o desempenho melhorasse ligeiramente ao utilizar replicação dos dados. No entanto, os resultados apresentados em 17 demonstram o contrário, o que significa que o *overhead* causado pela replicação dos dados acaba por ser mais custoso que os benefícios de desempenho provocados pela existência de uma réplica. Desta forma, esta abordagem aparenta não ser viável no âmbito do *benchmark* TPC-C.

5 Automatização do processo

Com o objetivo de simplificar os processos de instalação e execução do *benchmark* TPC-C na Google Cloud Platform, decidiu-se utilizar a ferramenta *open source* Ansible. Esta ferramenta permite automatizar o provisionamento e *deployment* de *software*, podendo lidar com múltiplas máquinas em simultâneo. O Ansible utiliza *playbooks* como uma forma de enviar comandos para as máquinas remotas, à semelhança de um *script*.

5.1 Instalação e execução

Foram criados quatro *playbooks* diferentes responsáveis por fazer as instalações e execuções mais relevantes, nomeadamente:

- `create-instances.yml`: cria intâncias na Google Cloud Platform com as características de *hardware* que pretendermos, sendo também capaz de criar um *bucket* caso seja necessário;
- `install-benchmark.yml`: descarrega do *bucket* os ficheiros relacionados com o *benchmark* TPC-C e instala-os nas instâncias criadas;
- `install-db.yml`: instala e configura o PostgreSQL;
- `load.yml`: efetua o *load* dos dados para a base de dados, com um determinado número de *warehouses*. Para além disto faz também *dump* da base de dados e *upload* do *backup* para o *bucket*;
- `run.yml`: faz *restore* da base de dados a partir de um ficheiro de *dump*, configura as propriedades de *workload* de acordo com o pretendido, corre o *benchmark* e, por último, faz *upload* do resultado para o *bucket*;
- `teardown-instances.yml`: remove as instâncias da Google Cloud Platform.

5.2 Criação dos *shards*

Para facilitar a criação dos *shards* descritos na secção 4.2, foi utilizado um *script* SQL responsável por criar os servidores, fazer o *mapping* dos utilizadores de cada base de dados e criar as *foreign keys*. As alterações feitas aos *scripts createtable.sql* e *createindex.sql* foram também cruciais para o correto funcionamento do *sharding*.

Todos os ficheiros utilizados no processo de automatização do projeto encontram-se em `https://github.com/claudiacorreia60/abd-1819`.

6 Conclusão

Embora não tenham sido analisados planos de execução, *materialized views* complexas, nem *vacuums*, os resultados obtidos revelaram que não existia grande margem de otimização no que toca ao desempenho do *benchmark* Escada TPC-C.

Relativamente a mecanismos de redundância, foram abordados os fatores considerados mais relevantes para o melhoramento da performance, nomeadamente a aplicação de índices, a otimização de *checkpoints* e o controlo da utilização de memória. Ao analisar os tipos de isolamento possíveis, embora se tivesse chegado à conclusão de que o *Read Committed* e o *Repeatable Read* são os níveis mais favorecedores, para determinar o desempenho máximo possível alcançado pelo *benchmark* tendo em conta todas as otimizações consideradas, optou-se por utilizar o nível de isolamento predefinido pelo PostgreSQL, o *Read Committed*.

Outra das estratégias, que embora se considerasse viável não foi estudada, foi o controlo de concorrência (*phantoms*, *write skews*, etc) através da alteração do código das interrogações. Como optamos por utilizar o nível de isolamento *Read Committed* uma boa opção seria corrigir problemas de concorrência para ser possível manter o correto funcionamento das transações sem abdicar do bom desempenho. No entanto, como não foi possível abranger todas as alternativas possíveis, acabou por não se analisar esta possibilidade, no entanto, esta poderia ser uma importante fonte de melhoria para o funcionamento do *benchmark*.

Em termos de mecanismos de escalabilidade, nenhum se mostrou capaz de beneficiar o desempenho do *benchmark*, logo, não se considera viável a utilização de nenhum. Apesar de o código do *benchmark* ter sido analisado, não foi possível desenvolver um conhecimento profundo o suficiente que permitisse realizar *sharding* dos dados ao nível do *benchmark* (secção 4.2.3). Apesar disto, considera-se que a aplicação deste tipo de *sharding* seria o mais indicado no âmbito deste projeto.

Concluído o estudo efetuado relativamente ao *benchmark* TPC-C, foram aplicadas todas as otimizações consideradas viáveis, nomeadamente:

- remoção do índice *ix_order_line*;
- *timeout* de 15 minutos para o *checkpoint*;
- *max_wal_size* de 6GB;
- *checkpoint_completion_target* de 0.9;
- *shared_buffers* de 256MB;
- *effective_cache_size* de 4GB;
- *work_mem* de 128MB.

Os resultados da configuração de referência, obtidos inicialmente, e da configuração otimizada, obtidos no final do projeto, são os apresentados de seguida.

Configuração	Débito (tx/s)	Tempo de resposta (s)	Abort rate
Referência	54.235550097222685	0.004844544554793352	0.0239669559135
Otimizada	54.22686129247934	0.004730081912546814	0.0236910645118

Tabela 17: Comparação entre resultados da configuração de referência e da configuração otimizada

Como a análise de cada otimização foi feita individualmente, ao juntar todas as possibilidades o resultado não se revelou tão positivo como seria de esperar. Uma oportunidade de estudo futura seria a análise de vários conjuntos de otimizações para determinar qual a melhor combinação.