

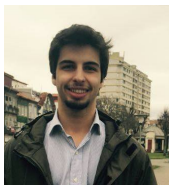
Universidade do Minho

Administração de Bases de Dados

MIEI - 4º ANO - 1º SEMESTRE

UNIVERSIDADE DO MINHO

TRABALHO PRÁTICO



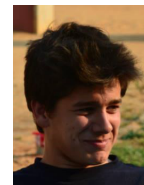
Dinis Peixoto
A75353



Ricardo Pereira
A74185



Marcelo Lima
A75210



Miguel Calafate
PG35405

29 de Dezembro de 2017

Conteúdo

1	Introdução	3
2	Contextualização	4
2.1	Especificações das máquinas	4
2.2	TPC-W Benchmark	4
2.3	Distribuição de proporções dos pedidos	5
3	Benchmark TPC-W	7
3.1	Configuração inicial	7
3.2	Falhas e problemas encontrados	9
3.2.1	Tamanho da base de dados fixo	10
3.2.2	Tempo de execução demasiado alto	10
3.2.3	Erros aleatórios na execução do benchmark	10
4	Análise e otimização de Queries	11
4.1	Otimização com <i>Índices</i>	11
4.2	Otimização com <i>Materialized Views</i>	12
4.2.1	<i>TopSellers</i>	12
4.2.2	<i>JoinItemAuthor</i>	14
4.2.3	<i>NewProducts</i>	17
4.2.4	<i>GetBooks</i>	18
4.2.5	<i>CustomerInfo</i>	20
4.2.6	<i>GetRelated</i>	22
4.3	Otimização individual de <i>Queries</i>	25
5	Análise e otimização dos parâmetros de configuração do PostgreSQL	27
5.1	Configurações <i>PostgreSQL</i>	27
5.1.1	Shared Buffers	27
5.1.2	Effective Cache Size	28
5.1.3	Work Memory	29
5.1.4	Planner Cost Constants	29
5.1.5	Autovacuum	30
5.1.6	Synchronous Commit	32
5.1.7	Write-Ahead Logging	32
5.1.8	Checkpoints	33
5.1.9	Default Statistics Target	34

5.1.10	Query Tuning	34
5.2	Níveis de isolamento	36
5.2.1	Read Committed	36
5.2.2	Read Uncommitted	36
5.2.3	Repeatable Read	37
5.2.4	Serializable	37
5.2.5	Análise	38
6	Replicação e <i>Sharding</i>	39
6.1	Replicação	39
6.2	<i>Sharding</i>	40
7	Resultados finais	41
8	Conclusão	43

1. *Introdução*

O presente relatório descreve todo o processo de realização e posterior análise no âmbito do trabalho prático da unidade curricular Administração de Base de Dados, do perfil Engenharia de Aplicações presente no Mestrado Integrado em Engenharia Informática da Universidade do Minho.

O trabalho prático consiste essencialmente na análise da aplicação de diferentes metodologias cujo objetivo passa pela melhoria do desempenho de uma determinada base de dados.

Recorreu-se ao **benchmark TPC-W** (www.tpc.org/tpcw/) e os objetivos passaram por: 1) instalar devidamente o *benchmark* e configurar este com escala adequada ao *hardware* utilizado; 2) otimizar ou justificar o desempenho das interrogações à base de dados; e por fim, 3) otimizar ou justificar o desempenho conforme os parâmetros de configuração do *PostgreSQL*.

A realização do trabalho passou pela análise de planos de execução e de *logs* produzidos, através do ficheiro *HTML* gerado pela ferramenta *PgBadger*, e ainda da comparação das diferentes métricas recolhidas/calculadas dos diversos testes efetuados.

2. Contextualização

A realização de uma investigação *a priori* sobre o contexto no qual a análise da base de dados se insere foi inevitável. Foi necessário reunir um conjunto de informações para que os alunos estivessem consideravelmente mais ambientados com o problema em questão e alcançarem mais facilmente as metodologias corretas para uma análise mais correta e completa.

2.1 Especificações das máquinas

Os resultados que serão demonstrados em capítulos posteriores foram calculados através de duas máquinas com especificações que, embora idênticas, são distintas. Seguem-se as suas especificações:

Xaomi Mi Notebook Air:

- Processador: 2.8GHz Intel Core I5-6200U
- Memória: 8GB 2133 MHz DDR4
- Disco: Samsung SSD PM951 NVMe 256GB

MacBook Pro:

- Processador: 2 GHz Intel Core I5
- Memória: 8GB 1867 MHz LPDDR3
- Disco: Apple SSD 256GB

É de notar que estes dados são relevantes para uma análise o mais correta possível, visto que fatores como o processador, a tecnologia do disco e a memória *RAM* influenciam bastante o desempenho e as métricas calculadas para a mesma configuração da base de dados e respetivas queries aplicadas a esta.

2.2 TPC-W Benchmark

O TPC Benchmark TM W (*TPC-W*) é um *benchmark* web transacional. A carga de trabalho é realizada num ambiente de comércio online, simulando as atividades de um servidor web transacional. Neste ambiente podemos imaginar um conjunto amplo de componentes que visam ser simuladas pelo sistema:

- Múltiplas sessões de navegador on-line
- Geração de página dinâmica com acesso e atualização de base de dados

- Objetos web consistentes
- A execução simultânea de vários tipos de transações que abrangem uma amplitude de complexidade
- Modos de execução de transação on-line
- Bases de dados constituídas por muitas tabelas e com uma grande variedade de tamanhos, atributos e relacionamentos
- Integridade da transação (propriedades *ACID*)
- Contenção no acesso e atualização de dados

A métrica de desempenho relatada por TPC-W é o número de interações na web processadas por segundo.

2.3 Distribuição de proporções dos pedidos

Um dos primeiros passos do grupo foi, inevitavelmente, uma pequena investigação sobre quais os tipos de *queries* utilizados mais frequentemente, considerando esta uma informação que pode moldar significativamente a aplicação das metodologias que se apresentarão durante o presente relatório.

As figuras que se seguem representam, para uma configuração completamente arbitrária (uma que vez que neste ponto não teria, ainda, sido alcançada a configuração ideal) a distribuição das queries conforme o número de vezes que estas são utilizadas aquando execução do *benchmark*.

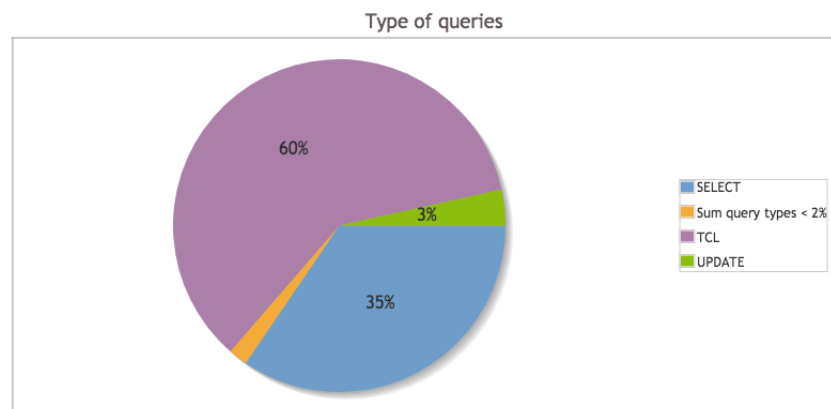


Figura 2.1: Gráfico representativo da distribuição de proporções das queries

Podemos facilmente observar uma forte predominância de queries do tipo **SELECT** e **TCL** sendo este último relativo às declarações de blocos transacionais, i.e, instruções que controlam as transações efetuadas durante a execução do *benchmark*, de modo a manter a integridade dos dados contidos nestas. Este tipo é composto por instruções **BEGIN Transaction**, **COMMIT Transaction** e ainda, quando necessário, **ROLLBACK Transaction**. Segundo responsáveis por, respetivamente,

iniciar uma transação, confirmar uma transação e retroceder ao último estado de coerência.

Para uma observação mais promenorizada sobre a distribuição de proporção para os diferentes tipos de queries efetuadas à base de dados segue-se a tabela abaixo.

Type	Count	Percentage
SELECT	150,884	34.52%
INSERT	7,224	1.65%
UPDATE	15,253	3.49%
DELETE	823	0.19%
COPY FROM	0	0.00%
COPY TO	0	0.00%
CTE	0	0.00%
DDL	0	0.00%
TCL	262,868	60.15%

Figura 2.2: Tabela representativa da distribuição de proporções das queries.

3. *Benchmark TPC-W*

3.1 Configuração inicial

A primeira etapa do trabalho corresponde a descobrir o ponto ótimo de execução do *benchmark*, de modo a que a máquina não se encontre sob o efeito de pouca carga, nem completamente saturada. Deste modo realizamos vários testes, variando dois fatores, o número de EBs, equivalente ao número de utilizadores a simular, e o TT (thinking time), correspondendo à simulação do tempo em que um utilizador fica a olhar para o ecrã do computador sem efetuar qualquer tipo de ação, enquanto isto fomos verificando, através da ferramenta *top* a utilização do CPU, que não deveria ser inferior a 60%.

O aumento do número de EBs origina um aumento proporcional no tamanho da base de dados, assim como o número de transações efetuadas. No entanto, tendo um número de EBs consideravelmente alto não era suficiente, uma vez que o processamento continuava exageradamente baixo (a rondar os 5%), isto devido ao TT inicial dos utilizadores. Para tal, diminuimos o TT para que o tempo de espera fosse cada vez menor, aumentando assim o processamento, conseguindo os valores pretendidos de utilização, i.e. acima dos 60%. No fim, e ao fim de muitas iterações, seguem-se os resultados conseguidos pelas diferentes máquinas.

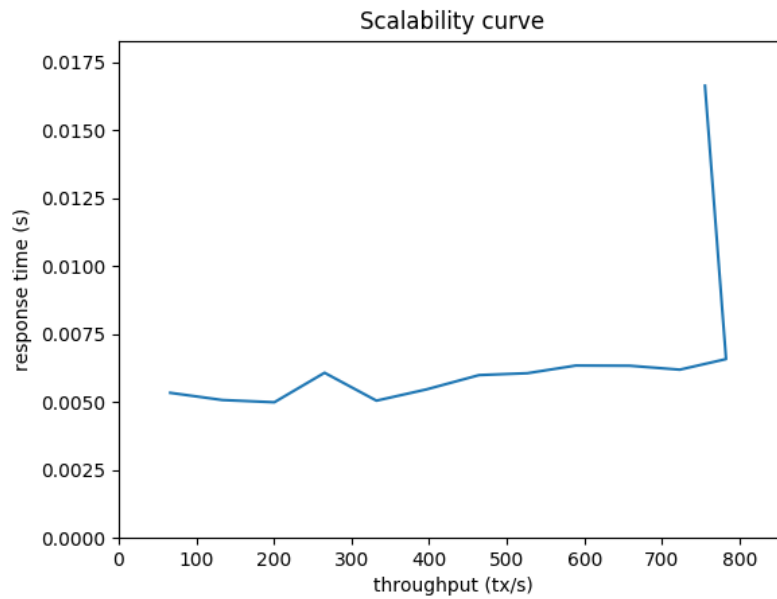


Figura 3.1: Gráfico representativo dos testes de configurações na máquina *Xaomi Mi*.

Como podemos ver na figura 3.1, existe um ponto em que o teste realizado sai fora dos valores comuns do parâmetro *response time*. Nesse momento encontramos o ponto de saturação da máquina, correspondendo a:

```
EB's da BD gerada = 100
EB's ao executar o script = 60
TT = 0.01
```

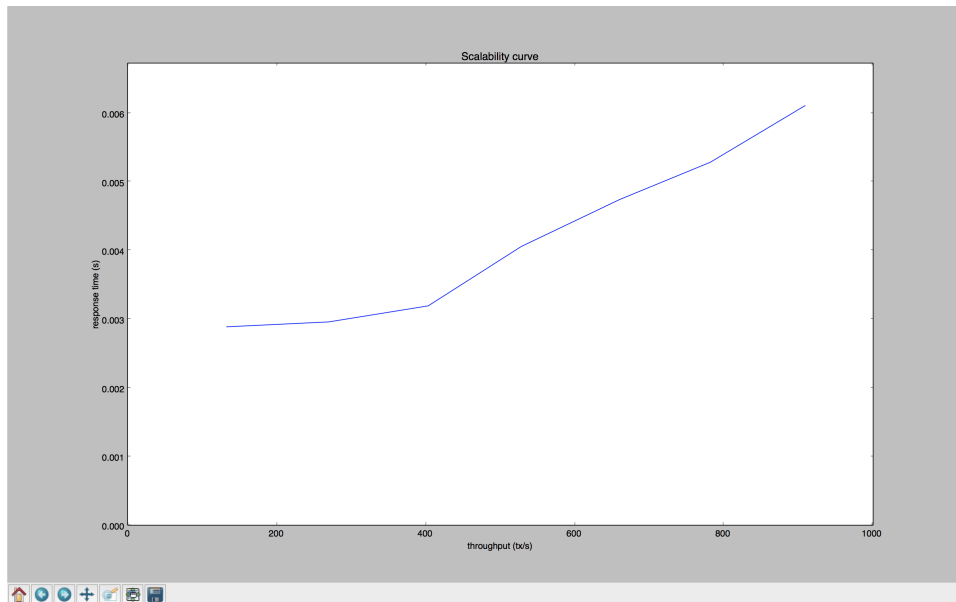


Figura 3.2: Gráfico representativo dos testes de configurações na máquina *MacBook Pro*.

Pela figura 3.2, podemos concluir que o ponto de saturação da máquina *MacBook Pro* ocorre com um número de EBs inferior ao da anterior, correspondendo a:

```
EB's da BD gerada = 100
EB's ao executar o script = 30
TT = 0.01
```

É de notar que o número máximo de EBs ao executar o script, foi 70 e 60, respectivamente para as máquinas *Xiaomi Air* e *MacBook Pro*, o que corresponde ao limite máximo de execução do *benchmark* sem problemas, isto é, sem que este começasse a lançar diferentes tipos de exceções, das quais falaremos mais adiante.

Para ambas as máquinas, os tempos de execução foram:

```
Ramp-up time = 60 sec
Measurement interval = 240 sec
Ramp-down time = 40 sec
```

3.2 Falhas e problemas encontrados

Nesta secção abordaremos as diferentes falhas e problemas encontrados aquando execução do benchmark TPCW. Estes representaram um contratempo considerável na realização do presente trabalho prático, em particular na fase inicial, da procura da configuração adequada conforme o hardware utilizado, correspondente à secção anterior.

3.2.1 Tamanho da base de dados fixo

Inicialmente ao gerar a base de dados para um determinado número de EBs reparamos que o tamanho desta se mantinha fixo, isto é, o tamanho gerado era exatamente igual, mesmo alterando o parâmetro `num.eb` em `tpcw.properties`, responsável pelo número de EBs para o qual a base de dados deve ser gerada. Verificávamos, no entanto, que ao alterar o parâmetro `num.item`, responsável pelo número de itens a ser gerado na base de dados, o tamanho desta aumentava/-diminuía efetivamente. Eventualmente o professor acabou por corrigir esta falha, ficando esta funcionalidade a funcionar conforme pretendido inicialmente.

3.2.2 Tempo de execução demasiado alto

Os tempos de execução iniciais eram demasiado altos, perdendo desde 5 a 10 minutos por cada teste realizado. Após algumas iterações o grupo verificou que não existiam diferenças significativas ao utilizar intervalos de tempo mais pequenos, uma vez que o TT já teria sido reduzido. Desta forma, a máquina "aqueceria" mais rapidamente, isto é, a cache seria mais rapidamente preenchida, por exemplo, e como tal não haveria necessidade de ter um *up time* de 2 minutos (valor inicialmente utilizado pelo grupo). Este princípio pode aplicar-se para os restantes tempos, *interval* e *down time*.

3.2.3 Erros aleatórios na execução do benchmark

Durante a execução do benchmark fomos surpreendidos com diversos erros e exceções lançadas pelo TPCW, onde para grande parte delas não conseguimos identificar a razão.

```
EB Error: Unable to find First name in HTML for buy request page.
```

```
EB Error: Unable to open URL.  
http://localhost:8080/tpcw/TPCW_product_detail_servlet;jsessionId=
```

```
Exception in thread "TPC-W Emulated Browser 25"  
java.lang.ArrayIndexOutOfBoundsException: 100000
```

Sendo este último o único que o grupo tem conhecimento da razão: ultrapassar o número máximo de transações possíveis, valor que pode efetivamente ser aumentado, mas que o grupo optou por não o fazer, adaptando-se assim a este. Isto é, garantindo um número de transações elevado mas sem nunca chegar ao máximo, variando para tal o TT, o número de EBs e o tempo de execução como vimos na secção anterior.

4. *Análise e otimização de Queries*

Após encontrado o ponto ótimo de execução dos testes, passamos para a análise e otimização dos resultados obtidos pelo benchmark, através das queries realizadas à base de dados. Para tal, foi necessário inferir quais os tipos de queries mais frequentes e em que proporções são realizadas face às demais. Tal como apresentado anteriormente na figura 2.1, gerada pelo *PgBadger*, verificou-se que queries do tipo **SELECT** são as que mais se realizam. Este é um parâmetro relevante para a análise que se segue.

Após a inspeção de outras componentes geradas pelo *PgBadger*, foi possível inferir o seguinte:

- **Materialized Views** - A construção de vistas materializadas poderá ser uma boa alternativa, visto que as proporções de queries realizadas não são muito próximas, isto é, o facto de existir um número de queries **UPDATE** muito inferior ao de **SELECT**'s, revela que para manter essas vistas atualizadas, poderá não ser muito penoso, mas que será abordado com mais atenção num capítulo vindouro.
- **Índices** - Apesar de analisados os resultados obtidos, conseguimos perceber que o modelo de dados já prevê quais as queries que poderão ser feitas, tendo sido então criados os diferentes índices nas diferentes tabelas de forma a garantir a maior performance possível. No entanto, estes mesmo índices deverão ter-se em conta após a migração da utilização de tabelas para *Materialized Views*, uma vez que a sua presença será fulcral para garantir uma performance ótima.

É de notar que os valores apresentados adiante correspondem aos valores diretamente obtidos com os testes realizados na máquina *MacBook Pro*.

4.1 *Otimização com Índices*

Após a análise de todas as queries realizadas à base de dados, verificamos as suspeitas iniciais, todos os possíveis índices a serem utilizados no modelo de dados já se encontravam aplicados. Sendo assim, a inserção de novos índices no modelo não aumentará a probabilidade de convergência do nível de performance da base de dados para um estado ótimo.

No entanto, e apesar de não ser vantajoso adicionar novos índices, estes já criados e aplicados são de grande importância, garantindo que inicialmente, o

conhecimento sobre os índices necessários de forma a assegurar a performance da BD, não seja totalmente desconhecido. Assim, todos os índices utilizados de origem deverão ser tidos em conta após a criação das *Materialized Views*.

4.2 Otimização com *Materialized Views*

Ao longo desta secção, é analisada a implementação de *Materialized Views* (MVs) e qual o seu impacto no desempenho das diferentes queries. Além destas, é também feita uma análise do impacto da adição de índices nos casos que estes sejam pertinentes. É ainda realizada uma avaliação do seu impacto de forma isolada, simples interrogação à BD utilizando a funcionalidade *explain analyze*, e feita a avaliação em contexto real de utilização com utilizadores simulados.

Nas próximas subsecções são apresentadas, de forma individual, as diferentes implementações em diferentes fases avaliando o seu impacto: (1) performance original; (2) performance aplicando MV; (3) performance adicionando índices à respetiva MV. Realça-se apenas que não se criou nenhum mecanismo de refrescamento das MVs, como por exemplo, triggers, apenas será abordado na próxima subsecção.

Várias MVs foram estudadas e implementadas, no entanto, algumas destas, que primeiramente apresentavam ser soluções para um aumento de performance, acabaram por ser removidas, sendo apresentado o seu estudo. Assim, ao longo das próximas subsecções, as seguintes MVs são apresentadas:

1. TopSellers
2. JoinItemAuthor
3. NewProducts
4. GetBooks
5. CustomerInfo
6. GetRelated

4.2.1 *TopSellers*

Performance Original

Uma das muitas funcionalidades da aplicação a ser avaliada é a listagem das obras mais vendidas. Como podemos ver na **Figura 4.1**, a operação em causa foi executada um elevado número de vezes, para além deste facto, podemos verificar que existe um tempo total gasto muito elevado, **24s**, e uma grande discrepância entre os tempos médios e máximos gastos.

11	4,660	23s673ms	3ms	68ms	5ms	<div> <div>SELECT i_id, i_title, a_fname, a_lname FROM item, author, order_line WHERE item.i_id = order_line.ol_i_id AND item.i_a_id = author.a_id AND order_line.ol_o_id > (SELECT max (o_id) 0 FROM orders) AND item.i_subject = '' GROUP BY i_id, i_title, a_fname, a_lname ORDER BY sum (ol_qty) DESC LIMIT 0;</div> <div>Examples</div> </div>
	Details					

Figura 4.1: Análise da operação de listagem das obras mais vendidas

Já na **Figura 4.2**, podemos observar que a operação, mesmo sendo executada de forma isolada, tem um tempo de execução elevadíssimo de **7.676 ms**. Assim, existe uma grande necessidade de melhorar a performance desta operação.

QUERY PLAN	
Limit (cost=13346.58..13346.70 rows=50 width=71) (actual time=7.190..7.200 rows=44 loops=1)	
InitPlan 2 (returns \$1)	
→ Result (cost=0.46..0.48 rows=1 width=4) (actual time=0.075..0.075 rows=1 loops=1)	
InitPlan 1 (returns \$0)	
→ Limit (cost=0.42..0.46 rows=1 width=4) (actual time=0.073..0.073 rows=1 loops=1)	
→ Index Only Scan Backward using orders_pkey on orders (cost=0.42..11091.90 rows=276827 width=4) (actual time=0.073..0.073 rows=1 loops=1)	
Index Cond: (o_id IS NOT NULL)	
Heap Fetches: 1	
→ Sort (cost=13346.10..13373.60 rows=11000 width=71) (actual time=7.190..7.193 rows=44 loops=1)	
Sort Key: (sum(order_line.ol_qty)) DESC	
Sort Method: quicksort Memory: 31kB	
→ HashAggregate (cost=12370.69..12980.69 rows=11000 width=71) (actual time=7.083..7.138 rows=44 loops=1)	
Group Key: item.i_id, author.a_fname, author.a_lname	
→ Hash Join (cost=163.45..12752.62 rows=11807 width=67) (actual time=0.665..6.856 rows=389 loops=1)	
Hash Cond: (order_line.ol_i_id = item.i_id)	
→ Index Scan using order_line.ol_o_id on order_line (cost=0.42..11465.27 rows=268334 width=8) (actual time=0.117..4.886 rows=9393 loops=1)	
Index Cond: (ol_o_id > \$1)	
→ Hash (cost=162.48..162.48 rows=44 width=63) (actual time=0.531..0.531 rows=44 loops=1)	
Buckets: 1024 Batches: 1 Memory Usage: 13kB	
→ Hash Join (cost=27.24..162.48 rows=44 width=63) (actual time=0.255..0.507 rows=44 loops=1)	
Hash Cond: (item.i_a_id = author.a_id)	
→ Bitmap Heap Scan on item (cost=8.62..143.30 rows=44 width=45) (actual time=0.049..0.281 rows=44 loops=1)	
Recheck Cond: ((i_subject)::text = 'CHILDREN'::text)	
Heap Blocks: exact=40	
→ Bitmap Index Scan on item_i_subject (cost=0.00..8.61 rows=44 width=0) (actual time=0.037..0.037 rows=44 loops=1)	
Index Cond: ((i_subject)::text = 'CHILDREN'::text)	
→ Hash (cost=15.50..15.50 rows=250 width=26) (actual time=0.187..0.187 rows=250 loops=1)	
Buckets: 1024 Batches: 1 Memory Usage: 23kB	
→ Seq Scan on author (cost=0.00..15.50 rows=250 width=26) (actual time=0.009..0.114 rows=250 loops=1)	
Planning time: 1.588 ms	
Execution time: 7.676 ms	
(51 rows)	

Figura 4.2: Explain Analyze da operação de listagem das obras mais vendidas

Criação de MV

De forma a melhorar a performance da operação em causa, foi criada a MV representada na **Figura 4.3**, tendo repercussões na query `getBestSellers` utilizada pela aplicação.

```
CREATE MATERIALIZED
VIEW top_sellers AS
SELECT i_id, i_title, a_fname, a_lname, item.i_subject
FROM item, author, order_line
WHERE item.i_id = order_line.ol_i_id AND item.i_a_id = author.a_id AND
      order_line.ol_o_id > ( SELECT MAX (o_id) - 3333
      FROM orders)
GROUP BY i_id, i_title, a_fname, a_lname
ORDER BY SUM (ol_qty) DESC;
```

Figura 4.3: *Materialized View - TopSellers*

Como podemos ver na **Figura 4.4** apenas fazendo uso desta MV, conseguimos diminuir significativamente o tempo gasto de **24s** para apenas **1s**, para além disto, vemos também que os tempos máximo e médio foram reduzidos a valores aceitáveis.

12	4,469	1s118ms	0ms	5ms	0ms	Details	Examples
<code>SELECT i_id, i_title, a_fname, a_lname FROM top_sellers WHERE i_subject = '' LIMIT 0;</code>							

Figura 4.4: Análise da operação de listagem das obras mais vendidas após MV

Criação de Índices

Apesar da evolução da performance com o uso de uma MV, foi ainda adicionado um índice à mesma para o atributo `i_subject`, sendo apresentado na **Figura 4.5** o impacto deste. Tal como podemos ver, todos os valores anteriormente melhorados foram novamente reduzidos para a casa dos milissegundos, tendo agora valores quase nulos, havendo um melhoramento de quase 6000%.

12	4,466	420ms	0ms	40ms	0ms	Details	Examples
<code>SELECT i_id, i_title, a_fname, a_lname FROM top_sellers WHERE i_subject = '' LIMIT 0;</code>							

Figura 4.5: Análise da operação de listagem das obras mais vendidas após índice

4.2.2 JoinItemAuthor

Performance Original

A funcionalidade de pesquisa nas diferentes aplicações é muito comum, sendo assim, a aplicação em causa não foge à regra, tendo também esta opção. Ao longo das diferentes análises, esta operação foi aparecendo na lista de operações mais morosas e mais frequentes. Como podemos ver na **Figura 4.6**, existe alguma discrepância entre os tempos médios e máximos gastos, sendo esta uma operação regular.

5	3s383ms	5,220	0ms	12ms	0ms	Details	Examples
<code>SELECT * FROM item, author WHERE item.i_a_id = author.a_id AND item.i_subject = '' ORDER BY item.i_title LIMIT 0;</code>							

Figura 4.6: Análise da operação de pesquisa por assunto

Já na **Figura 4.7**, podemos observar que a operação, sendo executada de forma isolada, tem um tempo de execução de **0.110 ms**. Assim, existe algum espaço para melhorar a performance desta operação.

```

QUERY PLAN
-----
Limit (cost=35.49..35.50 rows=4 width=892) (actual time=0.060..0.060 rows=0 loops=1)
-> Sort (cost=35.49..35.50 rows=4 width=892) (actual time=0.059..0.059 rows=0 loops=1)
    Sort Key: item.i_title
    Sort Method: quicksort  Memory: 25kB
-> Nested Loop (cost=4.31..35.45 rows=4 width=892) (actual time=0.053..0.053 rows=0 loops=1)
    -> Seq Scan on author (cost=0.00..16.12 rows=1 width=362) (actual time=0.052..0.052 rows=0 loops=1)
        Filter: ((a_lname)::text ~ 'BABABABABASENG%')::text
        Rows Removed by Filter: 250
    -> Bitmap Heap Scan on item (cost=4.31..19.28 rows=4 width=530) (never executed)
        Recheck Cond: (i_a_id = author.a_id)
        -> Bitmap Index Scan on item_i_a_id (cost=0.00..4.31 rows=4 width=0) (never executed)
            Index Cond: (i_a_id = author.a_id)

Planning time: 0.275 ms
Execution time: 0.110 ms
(14 rows)

```

Figura 4.7: Explain Analyze da operação de pesquisa por autor

Criação de MV

Sendo esta operação utilizada em várias queries diferentes, a MV a criar deve conseguir de forma eficaz servir as necessidades de cada uma delas, tendo este aspeto em conta, a MV criada é representada na **Figure 4.8**. A utilização desta MV teve implicações nas seguintes queries: `doSubjectSearch`, `doTitleSearch`, `doAuthorSearch`.

```

CREATE MATERIALIZED
VIEW join_item_author AS
SELECT *
FROM item, author
WHERE item.i_a_id = author.a_id
ORDER BY item.i_title;

```

Figura 4.8: *Materialized View - JoinItemAuthor*

Como podemos observar pelos resultados nas **Figuras 4.9, 4.11, 4.10**, os valores de tempo total gasto e médio aumentaram, pelo que se pode concluir que a utilização desta MV por si só não é suficiente para aumentar a performance.

7	3s910ms	4,973	0ms	87ms	0ms	SELECT * FROM join_item_author WHERE i_subject = '' LIMIT 0;
		Details				Examples

Figura 4.9: Análise da operação de pesquisa por assunto após MV

10	4,825	1s569ms	0ms	8ms	0ms	SELECT * FROM join_item_author WHERE i_title LIKE '' LIMIT 0;
	Details					Examples

Figura 4.10: Análise da operação de pesquisa por título após MV

9	2s919ms	4,962	0ms	12ms	0ms	SELECT * FROM join_item_author WHERE a_lname LIKE '' LIMIT 0;
	Details					Examples

Figura 4.11: Análise da operação de pesquisa por autor após MV

Criação de Índices

Como já foi afirmado na secção anterior, o uso da MV não é o suficiente para aumentar a performance das operações, com isto, foram adicionados diferentes índices de forma a ir de encontro às necessidades das diferentes pesquisas. Assim, foram adicionados vários índices nos seguintes atributos: (1) `i_a_id`, (2) `i_subject`, (3) `i_title`, (4) `a_id`, (5) `a_lname`. Após a implementação destes índices, podemos ver nas **Figuras 4.12, 4.14, 4.13** que conseguimos ter alguns aumentos de performance, sendo que nuns casos estes são mais ou menos significativos.

7	2s558ms	5,086	0ms	63ms	0ms	SELECT * FROM join_item_author WHERE i_subject = '' LIMIT 0;
		Details				Examples

Figura 4.12: Análise da operação de pesquisa por assunto após índices

8	5,238	77ms	0ms	1ms	0ms	SELECT * FROM join_item_author WHERE i_title LIKE '' LIMIT 0;
	Details					Examples

Figura 4.13: Análise da operação de pesquisa por titulo após índices

20	72ms	5,121	0ms	1ms	0ms	SELECT * FROM join_item_author WHERE a_lname LIKE '' LIMIT 0;
	Details					Examples

Figura 4.14: Análise da operação de pesquisa por autor após índices

Para além de vermos um aumento de performance, no geral, em ambiente simulado, na **Figura 4.15**, onde foi feita uma análise da operação utilizando a funcionalidade *Explain Analyze* do *postgres*, verifica-se uma redução significativa do tempo de execução da query em mais de 50% quando comparado com a **Figura 4.7**, que representa a performance inicial.

```
QUERY PLAN
-----
Limit (cost=27.53..27.54 rows=4 width=892) (actual time=0.008..0.008 rows=0 loops=1)
-> Sort (cost=27.53..27.54 rows=4 width=892) (actual time=0.008..0.008 rows=0 loops=1)
    Sort Key: item.i_title
    Sort Method: quicksort  Memory: 25kB
-> Nested Loop (cost=4.45..27.49 rows=4 width=892) (actual time=0.004..0.004 rows=0 loops=1)
    -> Index Scan using author_a_lname on author (cost=0.14..8.17 rows=1 width=362) (actual time=0.004..0.004 rows=0 loops=1)
        Index Cond: (((a_lname)::text >= 'BABABABABASENG'::text) AND ((a_lname)::text < 'BABABABABASENH'::text))
        Filter: ((a_lname)::text ~ 'BABABABABASENG%'::text)
    -> Bitmap Heap Scan on item (cost=4.31..19.28 rows=4 width=530) (never executed)
        Recheck Cond: (i_a_id = author.a_id)
        -> Bitmap Index Scan on item_i_a_id (cost=0.00..4.31 rows=4 width=0) (never executed)
            Index Cond: (i_a_id = author.a_id)

Planning time: 0.241 ms
Execution time: 0.043 ms
(14 rows)
```

Figura 4.15: Explain Analyze da operação de pesquisa por autor após índices

4.2.3 NewProducts

Performance Original

Entre as operações mais habituais está a listagem de novas obras. Como podemos ver na **Figura 4.16**, é gasto cerca de **1,4 s** nesta operação, apresentando uma grande discrepância entre os valores de tempos médios e máximos.


12	4,651	1s408ms	0ms	14ms	0ms	 <code>SELECT i_id, i_title, a_fname, a_lname FROM item, author WHERE item.i_a_id = author.a_id AND item.i_subject = '' ORDER BY item.i_pub_date DESC, item.i_title LIMIT 0;</code>
	Details					Examples

Figura 4.16: Análise da operação de listagem de novas obras

Já na **Figura 4.17**, podemos observar que a operação, sendo executada de forma isolada, tem um tempo de execução de **0.514 ms**. Assim, existe espaço para melhorar a sua performance.

```
QUERY PLAN
-----
Limit (cost=96.96..97.09 rows=50 width=69) (actual time=0.424..0.443 rows=50 loops=1)
-> Sort (cost=96.96..97.10 rows=56 width=69) (actual time=0.423..0.433 rows=50 loops=1)
    Sort Key: item.i_pub_date DESC, item.i_title
    Sort Method: quicksort  Memory: 32kB
-> Hash Join (cost=23.33..95.33 rows=56 width=69) (actual time=0.277..0.371 rows=56 loops=1)
    Hash Cond: (item.i_a_id = author.a_id)
-> Bitmap Heap Scan on item (cost=4.71..75.94 rows=56 width=50) (actual time=0.040..0.088 rows=56 loops=1)
    Recheck Cond: ((i_subject)::text = 'CHILDREN'::text)
    Heap Blocks: exact=37
-> Bitmap Index Scan on item_i_subject (cost=0.00..4.70 rows=56 width=0) (actual time=0.027..0.027 rows=56 loops=1)
    Index Cond: ((i_subject)::text = 'CHILDREN'::text)
-> Hash (cost=15.50..15.50 rows=250 width=27) (actual time=0.227..0.227 rows=250 loops=1)
    Buckets: 1024  Batches: 1  Memory Usage: 24kB
-> Seq Scan on author (cost=0.00..15.50 rows=250 width=27) (actual time=0.007..0.096 rows=250 loops=1)

Planning time: 0.972 ms
Execution time: 0.514 ms
(16 rows)
```

Figura 4.17: Explain Analyze da operação de listagem de novas obras

Criação de MV

De forma a melhorar a performance da operação em causa, foi criada a MV representada na **Figura 4.18**, tendo repercussões na querie `getNewProducts` utilizada pela aplicação.

```
CREATE MATERIALIZED
VIEW new_products AS
SELECT i_id, i_title, i_subject, a_fname, a_lname
FROM item, author
WHERE item.i_a_id = author.a_id
ORDER BY item.i_pub_date DESC,item.i_title;
```

Figura 4.18: *Materialized View - NewProducts*

Como podemos observar pelos resultados na **Figura 4.19**, o valor máximo gasto foi reduzido significativamente de **14 ms** para **4 ms**. No entanto, vemos que

o valor total permaneceu praticamente idêntico, assim sendo, podemos concluir que a utilização desta MV por si só não é suficiente para aumentar a performance.

12	4,488	1s46ms	0ms	4ms	0ms	Details	Examples
<code>SELECT i_id, i_title, a_fname, a_lname FROM new_products WHERE i_subject = '' LIMIT 0;</code>							

Figura 4.19: Análise da operação de listagem de novas obras após MV

Criação de Índices

Como já foi afirmado na secção anterior, o uso da MV não é o suficiente para aumentar a performance da operação, com isto, foi adicionado um índice no atributo `i_subject`. Após a implementação deste índice, podemos ver na **Figura 4.20** que para além de diminuir novamente o tempo máximo da operação, conseguimos reduzir em quase 400% o tempo gasto pela operação, vindo que a implementação da MV juntamente com o índice aumentou a performance da operação.

12	4,384	387ms	0ms	2ms	0ms	Details	Examples
<code>SELECT i_id, i_title, a_fname, a_lname FROM new_products WHERE i_subject = '' LIMIT 0;</code>							

Figura 4.20: Análise da operação de listagem de novas obras após índice

Para além de vermos um aumento de performance em ambiente simulado, na **Figura 4.21**, onde foi feita uma análise da operação utilizando a funcionalidade *Explain Analyze* do *postgres*, verifica-se uma redução significativa do tempo de execução da *querie* em mais de 700% quando comparado com a **Figura 4.17**, que representa a performance inicial.

```

QUERY PLAN
-----
Limit  (cost=4.71..17.83 rows=50 width=65) (actual time=0.022..0.051 rows=50 loops=1)
-> Bitmap Heap Scan on new_products  (cost=4.71..19.41 rows=56 width=65) (actual time=0.021..0.045 rows=50 loops=1)
    Recheck Cond: ((i_subject)::text = 'CHILDREN'::text)
    Heap Blocks: exact=11
-> Bitmap Index Scan on new_products_i_subject_idx  (cost=0.00..4.70 rows=56 width=0) (actual time=0.015..0.015 rows=56 loops=1)
    Index Cond: ((i_subject)::text = 'CHILDREN'::text)
Planning time: 0.256 ms
Execution time: 0.072 ms
(8 rows)

```

Figura 4.21: Explain Analyze da operação de listagem de novas obras após índices

4.2.4 GetBooks

Performance Original

Entre as operações mais habituais está ainda a listagem de uma obra em específico. Como podemos ver na **Figura 4.22**, apesar de não ser gasto muito com a operação,

esta é executada cerca de 16.000 vezes, exigindo que seja feita uma análise da sua performance.

4	15,881	661ms	0ms	1ms	0ms	 <code>SELECT * FROM item, author WHERE item.i_a_id = author.a_id AND i_id = '';</code>
	Details					Examples

Figura 4.22: Análise da operação de listagem de uma obra

Já na **Figura 4.23**, podemos observar que a operação, sendo executada de forma isolada, tem um tempo de execução de **0.043 ms**, sendo um tempo aceitável.

```
QUERY PLAN
-----
Nested Loop (cost=0.42..16.52 rows=1 width=892) (actual time=0.012..0.013 rows=1 loops=1)
-> Index Scan using item_pkey on item (cost=0.28..8.29 rows=1 width=530) (actual time=0.006..0.007 rows=1 loops=1)
    Index Cond: (i_id = 954)
-> Index Scan using author_pkey on author (cost=0.14..8.16 rows=1 width=362) (actual time=0.003..0.003 rows=1 loops=1)
    Index Cond: (a_id = item.i_a_id)
Planning time: 0.244 ms
Execution time: 0.043 ms
(7 rows)
```

Figura 4.23: Explain Analyze da operação de listagem de uma obra

Criação de MV

De forma a tentar melhorar a performance da operação em causa, foi criada a MV representada na **Figura 4.24**, tendo repercussões na query `getBook` utilizada pela aplicação.

```
CREATE MATERIALIZED
VIEW get_books AS
SELECT *
FROM item, author
WHERE item.i_a_id = author.a_id;
```

Figura 4.24: *Materialized View - GetBooks*

Como podemos observar pelos resultados na **Figura 4.25**, apesar de ser esperada uma melhoria, o valor máximo gasto aumentou de forma inesperada para quase **17 ms**. Para além deste valor, vemos que também o valor total aumentou para quase **5 s**. Assim sendo, podemos concluir que a utilização desta MV teve um impacto negativo na performance da operação.

4	15,136	4s743ms	0ms	17ms	0ms	 <code>SELECT * FROM get_books WHERE i_id = '';</code>
	Details					Examples

Figura 4.25: Análise da operação de listagem de uma obra após MV

Criação de Índices

Como foi verificado na secção anterior, o uso da MV teve resultados negativos na performance da operação, com isto, foi adicionado um índice no atributo `i_id`. Após a implementação deste índice, podemos ver na **Figura 4.26** que o tempo total gasto com a tarefa diminuiu, no entanto, verificamos também que o número de execuções também reduziu, existindo na verdade a mesma proporção que a análise apresentada na **Figure 4.22**. Assim sendo, conclui-se que a implementação de uma MV com índice tem um impacto positivo, embora pouco significativo, na operação.



Figura 4.26: Análise da operação de listagem de uma obra após índice

4.2.5 *CustomerInfo*

Inicialmente, foi percebido que toda a informação sobre um dado cliente poderia ser previamente calculada, sendo esta constituída por um cálculo entre as tabelas *Customer*, *Address* e *Country*. Assim, toda esta informação, que aparentemente se manteria intacta durante a utilização normal da aplicação, estaria previamente calculada e de mais fácil acesso. Para além deste aspeto, esta operação é efetuada frequentemente, havendo espaço para melhorar a sua performance.

Criação de MV

No processo de *tunning* da operação em causa, foi criada a MV representada na **Figura 4.27**, sendo assim necessário alterar a query `getCustomer`.

```
CREATE MATERIALIZED
VIEW customer_info AS
SELECT *
FROM customer, address, country
WHERE customer.c_addr_id = address.addr_id AND address.addr_co_id =
country.co_id;
```

Figura 4.27: *Materialized View - CustomerInfo*

No entanto, apesar de se pensar que a criação desta MV traria um grande impacto na operação em causa, não foi possível constatar um aumento significativo de performance.

Refrescamento da MV

Após um periodo de testes, foi constatado que na verdade existem alterações de registos nas tabelas que constituem a MV adicionada. Para isso, foi necessário programar o refrescamento da MV fazendo uso de *triggers*. Como resultado, foi criada a função representada na **Figura 4.28**, tendo esta como resultado o refrescamento da MV em causa.

```
CREATE OR REPLACE FUNCTION trig_refresh_customer_info  
( ) RETURNS trigger AS  
BEGIN  
    REFRESH MATERIALIZED VIEW customer_info;  
RETURN NULL;  
END;
```

Figura 4.28: *Function - Refrescamento da MV CustomerInfo*

A função da **Figura 4.28**, é depois utilizada em 3 *triggers* distintos de TRUNCATE, INSERT, UPDATE, DELETE nas tabelas Customer (**Figura 4.29**), Address (**Figura 4.30**) e Country (**Figura 4.31**).

```
CREATE TRIGGER trig_01_refresh_customer_info AFTER  
TRUNCATE OR  
INSERT OR  
UPDATE OR DELETE ON customer FOR EACH STATEMENT  
EXECUTE PROCEDURE trig_refresh_customer_info();
```

Figura 4.29: *Trigger - Customer*

```
CREATE TRIGGER trig_02_refresh_customer_info AFTER  
TRUNCATE OR  
INSERT OR  
UPDATE OR DELETE ON address FOR EACH STATEMENT  
EXECUTE PROCEDURE trig_refresh_customer_info();
```

Figura 4.30: *Trigger - Address*

```
CREATE TRIGGER trig_03_refresh_customer_info AFTER  
TRUNCATE OR  
INSERT OR  
UPDATE OR DELETE ON country FOR EACH STATEMENT  
EXECUTE PROCEDURE trig_refresh_customer_info();
```

Figura 4.31: *Trigger - Country*

Após a criação dos 3 triggers e da função, sendo estes obrigatórios de forma a garantir a congruência dos dados, foi possível verificar que o tempo necessário para o refrescamento da MV consoante a despoletamento dos triggers é superior ao tempo de utilização úteis da MV. Na verdade, torna-se impossível a sua utilização pois esta é constantemente refrescada, acabando por entupir o servidor com o acumular de pedidos que são colocados em espera.

Assim, podemos concluir que a tentativa de aumento de performance da operação em causa não tem quaisquer resultados positivos, ficando assim as alterações propostas sem efeito.

4.2.6 *GetRelated*

Performance Original

Entre as operações mais executadas está a listagem de obras relacionadas com uma obra em específico. Como podemos ver na **Figura 4.32**, é gasto um total de 4s com a operação, no entanto, e ainda mais importante, esta é executada cerca de 70.000 vezes, exigindo que seja feita uma análise da sua performance.

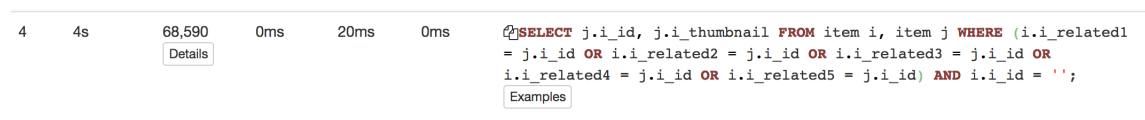


Figura 4.32: Análise da operação de listagem relações entre obras

Criação de MV

De forma a tentar melhorar a performance da operação em causa, foi criada a MV representada na **Figura 4.33**, tendo repercussões na querie *getRelated* utilizada pela aplicação.

```
CREATE MATERIALIZED
VIEW get_related AS
SELECT I.i_id AS i_i_id, J.i_id AS j_i_id, I.i_thumbnail AS
i_i_thumbnail, J.i_thumbnail AS j_i_thumbnail
from item I, item J
where (I.i_related1 = J.i_id or I.i_related2 = J.i_id or I.i_related3 =
J.i_id or I.i_related4 = J.i_id or I.i_related5 = J.i_id);
```

Figura 4.33: *Materialized View - GetRelated*

Como podemos observar pelos resultados na **Figura 4.34**, apesar de ser esperada uma melhoria, tanto o valor total como o valor máximo gasto aumentou

de forma inesperada em **1.000%** e **250%** respetivamente. Assim sendo, podemos concluir que a utilização desta MV teve um impacto negativo na performance da operação.

1	49s540ms	1,964 Details	16ms	316ms	25ms	🔗 <code>UPDATE item SET i_stock = '' WHERE i_id = '';</code> Examples
2	37s243ms	54,962 Details	0ms	50ms	0ms	🔗 <code>SELECT j_i_id, j_i_thumbnail FROM get_related WHERE i_i_id = '';</code> Examples

Figura 4.34: Análise das operações: (1) atualização do stock; (2) listagem das relações entre obras (após MV)

Criação de Índices

Como foi verificado na secção anterior, o uso da MV teve resultados negativos na performance da operação, com isto, foi adicionado um índice no atributo `i_id`. Após a implementação deste índice, podemos ver na **Figura 4.35** que o tempo total gasto com a tarefa diminuiu em **50%** relativamente à análise inicial da **Figura 4.32**. No entanto, verificou-se que o tempo máximo gasto se manteve idêntico à análise em comparação. Assim sendo, verifica-se que a implementação de uma MV com índice tem impacto positivos na operação.

7	2s314ms	53,981 Details	0ms	21ms	0ms	🔗 <code>SELECT j_i_id, j_i_thumbnail FROM get_related WHERE i_i_id = '';</code> Examples
---	---------	-----------------------------------	-----	------	-----	---

Figura 4.35: Análise da operação de listagem relações entre obras após índice

Apesar de se ter verificado que a migração da utilização de uma tabela para uma MV tem resultados positivos, deverá de se frizado que estas alterações poderão ter impactos negativos nas restantes operações.

Uma das operações que é afetada pela MV, é a atualização do *Stock* de uma obra. Como podemos ver, a **Figura 4.36** representa a análise da operação de atualização de stock antes de qualquer alteração, quer da migração para MV quer do uso de índices extra.

15	207ms	2,665 Details	0ms	45ms	0ms	🔗 <code>UPDATE item SET i_stock = '' WHERE i_id = '';</code> Examples
----	-------	----------------------------------	-----	------	-----	--

Figura 4.36: Análise da operação de atualização de stock de uma obra

Já na **Figura 4.37**, podemos observar os resultados da análise desta mesma operação após a criação do índice.

1	56s696ms	1,974 Details	19ms	144ms	28ms	🔗 <code>UPDATE item SET i_stock = '' WHERE i_id = '';</code> Examples
---	----------	----------------------------------	------	-------	------	--

Figura 4.37: Análise da operação de atualização de stock de uma obra após índice

Como podemos observar ao longo do processo, enquanto que por uma lado a performance da operação de obter obras relacionadas foi sendo melhorada, o mesmo não acontece com a operação de atualização do stock, havendo um deterioramento de performance de quase **28.000%**. Neste caso, a pertinência da utilização ou não da MV e do índice acaba por ser questionada, sendo que tendo em conta os ganhos globais a sua utilização não será a melhor opção.

Refrescamento da MV

Caso os resultados anteriormente apresentados não tenham sido suficientes para optar pelo uso ou não da "otimização" conseguida, os impactos do refrescamento da MV poderão ser o fator decisivo. De forma a forçar o refrescamento da MV da **Figura 4.33**, foi utilizado um *trigger* e uma função. Como resultado, foi criada a função representada na **Figura 4.38**, tendo esta como resultado o refrescamento da MV em causa.

```
CREATE OR REPLACE FUNCTION trig_refresh_get_related  
( ) RETURNS trigger AS  
BEGIN  
    REFRESH MATERIALIZED VIEW get_related;  
RETURN NULL;  
END;
```

Figura 4.38: *Function - Refrescamento da MV GetRelated*

A função da **Figura 4.38**, é depois utilizada no *trigger* de TRUNCATE, INSERT, UPDATE, DELETE na tabela Item (**Figura 4.39**)

```
CREATE TRIGGER trig_01_refresh_get_related AFTER  
TRUNCATE OR  
INSERT OR  
UPDATE OR DELETE ON item FOR EACH STATEMENT  
EXECUTE PROCEDURE trig_refresh_get_related();
```

Figura 4.39: *Trigger - Item*

Após a criação do trigger e da função, frisando novamente a obrigatoriedade destes de forma a garantir a congruência dos dados, foi possível verificar que existindo o grande número de atualizações de stock, há também uma grande frequência de refrescamento da MV, causando assim uma enorme redução de débito derivado do congestionamento de pedidos.

Assim, podemos concluir que a tentativa de aumento de performance da operação em causa, para além da enorme perda de performance global derivado à MV utilizada no processo de atualização de stock, temos também o facto de haver uma perda de performance com o refrescamento obrigatório da MV, assim as alterações propostas não deverão ser implementadas.

4.3 Otimização individual de *Queries*

Para além da otimização feita ao modelo através da criação de índices, *Materialized Views* e triggers, foi também feita uma revisão do plano de queries executadas pela aplicação. Após esta análise, e feito um estudo do funcionamento da aplicação, conseguiu-se perceber que uma das queries, query esta que é executada com elevada frequência, podia ser reformulada, aumentando assim a performance desta operação.

A operação em causa é a query *enterOrder.maxId*, em que é utilizada a operação *count* de forma a calcular o identificador da próxima encomenda (Order) a ser criada. Esta operação, que tem um resultado aparentemente simples, acaba por ter um processamento complexo, como podemos ver na **Figura 4.40**, esta, quando executada de forma isolada, ou seja, sem quaisquer interferências de quando é executada em ambiente real, tem um tempo de execução elevadíssimo de **151.603 ms**.

```
QUERY PLAN
-----
Finalize Aggregate (cost=5724.90..5724.91 rows=1 width=8) (actual time=149.011..149.011 rows=1 loop
s=1)
  -> Gather (cost=5724.69..5724.90 rows=2 width=8) (actual time=148.881..149.002 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (cost=4724.69..4724.70 rows=1 width=8) (actual time=142.548..142.548
rows=1 loops=3)
              -> Parallel Seq Scan on orders (cost=0.00..4439.75 rows=113975 width=4) (actual time
=0.237..128.528 rows=88268 loops=3)
Planning time: 0.828 ms
Execution time: 151.603 ms
(8 rows)
```

Figura 4.40: Explain Analyze da operação original de cálculo do Id da próxima encomenda

De forma a otimizar esta simples tarefa, a operação *count* foi substituída pela operação *max*. Esta troca, apenas é exequível dado o facto de que, nunca, em nenhuma circunstância, uma encomenda é eliminada. Caso se queira "eliminar" uma destas, é utilizado o atributo estado, sendo assim desabilitada ou habilitada, concluindo assim, que o número máximo do identificador das encomendas, será sempre igual ao número total das encomendas, permitindo-nos trocar a operação *count* pela operação *max*.

Como podemos observar pelos resultados da **Figura 4.41**, após aplicadas as alterações apresentadas, podemos verificar que houve uma redução drástica dos anteriores **151.603 ms** para os atuais **0.155 ms**, sofrendo assim uma redução fantástica de cerca de **98.000%**.

```

QUERY PLAN
-----
Result (cost=0.46..0.47 rows=1 width=4) (actual time=0.097..0.097 rows=1 loops=1)
  InitPlan 1 (returns $0)
    -> Limit (cost=0.42..0.46 rows=1 width=4) (actual time=0.094..0.095 rows=1 loops=1)
          -> Index Only Scan Backward using orders_pkey on orders (cost=0.42..11006.37 rows=273540
width=4) (actual time=0.094..0.094 rows=1 loops=1)
                Index Cond: (o_id IS NOT NULL)
                Heap Fetches: 1
  Planning time: 0.235 ms
  Execution time: 0.155 ms
(8 rows)

```

Figura 4.41: Explain Analyze da operação otimizada de cálculo do Id da próxima encomenda

Será de notar que, a alteração feita nesta operação, apenas é possível dado as regras de lógica atualmente aplicadas na aplicação, sendo que, neste caso, se de facto uma encomenda pudesse ser eliminada definitivamente da aplicação, esta alteração iria provocar incongruência nos dados. Este tipo de otimizações apenas podem ser feitas quando tanto as regras de lógica e de negócio estão bem definidas e compreendidas, garantindo assim que não existem implicações no bom funcionamento da aplicação.

5. *Análise e otimização dos parâmetros de configuração do PostgreSQL*

No presente capítulo abordaremos a exploração dos diversos parâmetros de configuração do PostgreSQL, desde a parâmetros relativos ao tamanho disponível em memória/cache, aos diferentes níveis de isolamento oferecidos. Tal como seria de esperar alguns dos parâmetros produzem mudanças realmente significativas no produto final, conseguindo melhorar consideravelmente o desempenho da base de dados.

Devemos ter em conta que as configurações obtidas estão diretamente relacionadas com os recursos disponibilizados pela máquina física utilizada, como tal estariam sujeitos a modificações numa outra máquina. No entanto, foram realizados testes para ambas as máquinas e, como não houveram diferenças significativas entre ambas, os valores apresentados, por questões de precisão, correspondem aos valores obtidos diretamente pela máquina *Xiaomi Air*.

5.1 Configurações *PostgreSQL*

Esta secção é focada na análise do impacto que a alteração de certos parâmetros produzem no desempenho durante a execução do *benchmark*. Como tal, optamos por avaliar individualmente quais as melhores configurações de cada parâmetro, produzindo a melhor solução composta por estes.

5.1.1 Shared Buffers

Este parâmetro pode ser visto no ficheiro `postgresql.conf`, com o nome de `shared_buffers` e diz respeito à quantidade de memória (*RAM*) disponível para armazenar dados em cache durante a realização de queries. Por defeito o valor do parâmetro é de 128MB. O aumento do valor deste parâmetro permitirá um aumento no desempenho global, no entanto, para um valor muito elevado o desempenho pode diminuir por outros fatores.

Realizaram-se testes para o seguinte conjunto: 128MB, 256MB, 512MB, 1024MB, 2048MB e 4096MB. Desta forma, podemos determinar o impacto que estes valores causam no desempenho global do *benchmark*, figura 5.1, mais concretamente no débito e no tempo de resposta.

Tamanho	Débito	Tempo de Resposta
128MB	782.175	0.00699963243519
256MB	779.193020174	0.00705248889103
512MB	781.71522856	0.00711020499536
1024MB	781.830454239	0.00714022830466
2048MB	780.502743716	0.00723620697939
4096MB	787.59281018	0.00669269469441

Figura 5.1: Tabela com os resultados obtidos, variando o valor do parâmetro *Shared Buffers*.

Após analisar os dados dos testes, conclui-se que o melhor resultado, para o parâmetro estudado, foi obtido com o valor **4096MB**. Este permitiu um débito maior, e, portanto, um tempo de resposta menor que os demais. Isto deve-se ao facto de, ao aumentar o tamanho de uso da memória *RAM* se evitar escritas em disco que é consideravelmente mais custoso.

5.1.2 Effective Cache Size

Este parâmetro é usado pelo *query planner* de modo a determinar se o plano de execução que delineou irá caber todo em memória. Caso este valor seja demasiado baixo a utilização de índices poderá não acontecer da forma esperada. Ou seja, é uma diretriz que indica quanta memória disponível é expectável ter na cache do Sistema Operativo e do *PostgreSQL*, e não uma alocação de memória.

Para tal foi necessário efetuar alguns testes com diferentes valores neste parâmetro, nomeadamente com 128MB, 256MB, 512MB, 1024MB, 2048MB, 4096MB e 6144MB, o que se pode ver na figura 5.2, onde está apresentado o impacto destes no desempenho, medido pelo benchmark, nomeadamente no débito e no tempo de resposta.

Tamanho	Débito	Tempo de Resposta
128MB	783.0	0.00731521392081
256MB	742.212110605	0.0104384747533
512MB	737.615160568	0.0115965902351
1024MB	772.7625	0.0079883858235
2048MB	718.221477768	0.0129563325617
4096MB	781.28359627	0.00692111747576
6144MB	780.692982675	0.00705912868053

Figura 5.2: Tabela com os resultados obtidos, variando o valor do parâmetro *Effective Cache Size*.

Conclui-se que os melhores resultados são obtidos com os valores de 128MB e 4096MB. O valor de 128MB vai contra a hipótese colocada acima, em que quanto maior o valor, maior seria a probabilidade da utilização de índices. Por esta razão obtamos por escolher o valor de **4096MB**.

Como podemos verificar, os valores obtidos para os diferentes parâmetros variam pouco. Estes resultados sugerem que as leituras de disco não são assim tão más, devido à tecnologia do disco ser Solid State Disk (*SSD*) em ambas as

máquinas utilizadas para teste, se estivessemos perante uma máquina com os tradicionais Hard Drive Disk (*HDD*) os valores seriam consideravelmente diferentes.

5.1.3 Work Memory

Este parâmetro indica a memória (*RAM*) que as operações *bitmapscan*, *hashagg*, *hashjoin*, *indexscan*, *indexonlyscan*, *material*, *mergejoin*, *nestloop*, *seqscan*, *sort* e *tidscan* possuem antes de serem escritas em disco.

O valor deste parâmetro irá permitir melhorar o desempenho visto que quando se realiza uma query a base de dados irá determinar, através de uma estimativa da quantidade de dados envolvida nessa query, se será ou não possível executar a(s) operação(ões) em memória sem auxílio do disco, o que normalmente piora o desempenho. Como tal realizamos testes com os valores 128MB, 256MB e 512MB, cujos resultados, do desempenho obtido, podem ser consultados na figura 5.3.

Tamanho	Débito	Tempo de Resposta
128MB	697.609880494	0.016111099165
256MB	780.1	0.00736977631073
512MB	754.402849822	0.00952995559679

Figura 5.3: Tabela com os resultados obtidos, variando o valor do parâmetro *work_memory*.

Conclui-se que o melhor valor para este parâmetro é **256MB**, visto que produziu o melhor débito e o menor tempo de resposta entre os três valores testados.

5.1.4 Planner Cost Constants

Neste parâmetro iremos abordar a influência dos custos de acesso ao disco na performance do *benchmark*. Para tal iremos variar 2 valores presentes no ficheiro *postgresql.conf* da base de dados: *random_page_cost* e *seq_page_cost*. Como os testes foram efetuados em Solid State Disks (*SSD*) decidimos usar valores baixos para ambas as variáveis, sendo que definimos duas combinações para testes:

```
---- Configuração 1 ----
random_page_cost=0.2
seq_page_cost=0.1

---- Configuração 2 ----
random_page_cost=0.1
seq_page_cost=0.05
```

Configuração	Débito	Tempo de Resposta
Conf1	713.375	0.014121272122
Conf2	780.78201955	0.00702404700533

Figura 5.4: Tabela com os resultados obtidos, comparando as duas configurações apresentadas.

Os resultados obtidos, e disponíveis na figura 5.4, mostram uma melhoria óbvia com a redução dos custos de acesso sequencial ou aleatório na configuração 2, que contém os custos mais reduzidos. Estes dois testes servem para averiguarmos a influência dos acessos ao disco na performance do benchmark, mas comparando a influência destas variáveis com outras como `shared_buffers`, `work_memory` ou o processo de `vacuuming`, verifica-se que a influência dos acessos a disco não é de tão elevada importância.

5.1.5 Autovacuum

Quando são efetuadas operações de `delete` ou `update`, existem vestígios de registos que o *PostgreSQL* não elimina. Esses vestígios afetam negativamente a performance do benchmark, uma vez que ocupam espaço nas tabelas da base de dados podendo até conduzir a um problema de sobrelotação da base de dados. Durante a execução das queries o *PostgreSQL* pode efetuar uma limpeza das tabelas se assim for pedido. Para tal no ficheiro de configurações é necessário alterar os parâmetros relativos ao `autovacuum`. Desses parâmetros decidimos fazer variar apenas dois: `autovacuum_naptime` e `autovacuum_vacuum_threshold`.

Naptime

Este processo de limpeza efetua vários tipos de manutenção na base de dados, sendo que um dos quais é a libertação de espaço de armazenamento ocupado por registos obsoletos. Estes registos resultam das operações de `update` e `delete`, porque apesar dos registos serem alterados, o *PostgreSQL* não os elimina das tabelas, apenas os marca como `updated` ou `deleted` de modo a que deixem de estar disponíveis para o utilizador.

Quantas mais queries forem aplicadas às tabelas mais registos obsoletos estas irão conter, pelo que a limpeza destas mesmas tabelas trará mais eficiência à execução do nosso benchmark. À partida, quanto menos vezes for efetuado este processo de limpeza, maior será a performance, pois o nosso benchmark irá dedicar todo o seu tempo à execução de queries sem perder tempo com a limpeza. Com estas conclusões decidimos variar o parâmetro que indica o período de limpeza, o `autovacuum_naptime`, atribuindo valores de `off`, 5, 10, 20 e 40 e 80 segundos, sendo que podemos observar os resultados obtidos na figura 5.5.

Valor	Débito	Tempo de Resposta
off	782.175	0.00699963243519
5s	785.009812623	0.00678361464968
10s	781.05	0.0073047660201
20s	774.055229833	0.007059141782
40s	783.409792622	0.00680801633903
80s	781.745684537	0.00716221099421

Figura 5.5: Tabela com os resultados obtidos, variando o valor do parâmetro *autovacuum_naptime*.

Como podemos concluir com a figura anterior, a execução do benchmark com os valores de 5 e 40 segundos (intervalo) possuem os melhores resultados de débito e de tempo de resposta. Com estes resultados podemos concluir que o intervalo estipulado não influencia muito a performance global do benchmark. Isto porque os resultados com o parâmetro a **off** e com o valor de 80 segundos, não variam significativamente. Posto isto, o processo de limpeza não tem um grande impacto no benchmark. Como tal, optamos pelo valor de **40** segundos, tendo em conta que teoricamente implica menor processamento para efectuar as limpezas, o que permite maior processamento para o benchmark.

Threshold

O parâmetro *autovacuum_vacuum_threshold* corresponde ao número mínimo de atualizações de registos entre execuções de limpezas. De forma a explorar a influência deste parâmetro, no ficheiro de configuração do *PostgreSQL* foi variado o valor deste parâmetro entre 50, 300, 600 e 1200 registos. Apresentámos na figura 5.6 com os resultados do débito e do tempo de resposta.

Threshold	Débito	Tempo de Resposta
50	782.9375	0.00699485910433
300	778.519462987	0.00692182080925
600	779.340258247	0.00709807852697
1200	780.425	0.00681564532146

Figura 5.6: Tabela com os resultados obtidos, variando o valor do parâmetro *autovacuum_vacuum_threshold*.

Concluindo, podemos verificar com os resultados acima que o parâmetro não influencia muito a performance global do benchmark, como já tínhamos concluído com o parâmetro anterior. No entanto, podemos observar que os melhores resultados obtidos são com os valores 50 e 1200. Como, teoricamente, quanto maior o número de actualizações de registos menor é o número de limpezas, escolhemos o valor de **1200** registos.

5.1.6 Synchronous Commit

O parâmetro `synchronous_commit` existente nas configurações do PostgreSQL permite controlar a forma como os commits são efectuados. Quando esta opção se encontra ligada, sempre que um commit é efectuado pelo cliente, este tem de esperar obrigatoriamente que esse commit seja escrito para o disco, causando assim um bottleneck que vai limitar o número de transações que esse cliente pode efectuar por segundo. No entanto, esta espera pode ser evitada se permitirmos escrever no disco de forma assíncrona, isto é, sem ser necessário esperar por uma mensagem de resposta vinda do servidor. Porém, tornando os commits assíncronos, há possibilidade de perda de dados, no entanto, isto não significa que a base de dados fique inconsistente, o estado desta continuará como se as transações tivessem abortado corretamente. De seguida encontra-se a tabela com o resultado das duas medições no qual se fez variar o parâmetro `synchronous_commit`.

Valor	Débito	Tempo de Resposta
Off	780.5875	0.00724503979374
On	768.7375	0.00783768841769

Figura 5.7: Tabela com os resultados obtidos, variando o valor do parâmetro `synchronous_commit`.

Como podemos observar na tabela, a medição feita com commits assíncronos é aquela que apresenta melhor performance, com maior débito e menor tempo de resposta. Isto deve-se ao facto anteriormente explicado, de não haver espera entre os commits efectuados por cada cliente, e de provavelmente haver uma tabela que esteja sujeita a imensas inserções. No entanto, esta opção só é válida caso a perda momentânea de dados nessa ou nessas tabelas não seja um problema. Desta forma, apesar de não apresentar a melhor performance, vamos utilizar **commits síncronos**, por razões de segurança.

5.1.7 Write-Ahead Logging

Depois de analisarmos as queries verificou-se que grande parte do trabalho efectuado por estas é em inserções e portanto faz sentido tentar otimizar o acesso ao disco durante o logging. Para isso, fizemos variar um parâmetro designado `wal_buffers`, o qual serve para definir a quantidade de memória que se vai partilhar para dados de *Write-Ahead Logging* a serem escritos para disco. O valor pré-definido pelo PostgreSQL é de -1, ou seja, considera um tamanho de sensivelmente 3% do valor atribuído ao `shared_buffers`, mas nunca menos de 74kB. Visto que estamos a lidar com um servidor em que bastantes clientes tentam fazer commits em simultâneo, vale a pena tentar aumentar este valor, pois sempre que ocorre um commit, o conteúdo presente no WAL buffers é escrito para disco, podendo assim haver um ganho de performance. Decidimos variar o valor deste parâmetro com os valores 4MB, 16MB, 32MB, 64MB e 128MB. De seguida encontra-se a tabela com os resultados obtidos.

Valor	Débito	Tempo de Resposta
4MB	764.625	0.00887515121791
16MB	772.968175796	0.008181584437
32MB	772.051102555	0.00791973899386
64MB	774.106852671	0.008125179647
128MB	760.5125	0.00854221659736

Figura 5.8: Tabela com os resultados obtidos, variando o valor do parâmetro *wal_buffers*.

Como podemos observar nos resultados obtidos, à medida que o valor do *wal_buffers* aumenta, maior é o débito e menor o tempo de espera. Isto deve-se ao facto anteriormente explicado: uma vez que existem bastantes clientes a realizar commits em simultâneo, e como o valor do *wal_buffers* é consideravelmente grande, os dados desses mesmos commits ficam assim partilhados, sendo preciso apenas um acesso ao disco em vez de um por cada commit.

No entanto, verificou-se que para o valor do *wal_buffers* de 128MB houve uma queda significativa de performance, isto deve-se ao facto de este valor ser demasiado grande e consequentemente não trazer qualquer melhoria de performance. Assim, conclui-se que o valor que deverá ser utilizado é o de **64MB** pois é aquele que apresenta maior performance individual.

5.1.8 Checkpoints

Os checkpoints dizem respeito ao instante durante a transação em que todos os dados são escritos em disco de modo a reflectir a informação existente no ficheiro de log. O parâmetro *checkpoint_completion_target* diz respeito à fracção de tempo entre checkpoints, ou seja, quando este valor se encontra a 0.5 (pré-definido) significa que as escritas para disco acabam quando o próximo checkpoint estiver 50% completo. Ora, isto pode levar a quedas de performance significativas (*bottlenecks*). Desta forma, fizemos medições com este valor a 0.5, 0.7, 0.8, 0.9 e 0.95 e cujos resultados se encontram abaixo.

Valor	Débito	Tempo de Resposta
0.5	765.273472653	0.00854574704383
0.7	768.725	0.00814675273992
0.8	769.959624495	0.00785606208195
0.9	765.675	0.00819714630881
0.95	766.653749516	0.00823807272134

Figura 5.9: Tabela com os resultados obtidos, variando o valor do parâmetro *checkpoint_completion_target*.

Como podemos analisar pelos resultados, a performance do sistema vai aumentando até ao valor 0.8, diminuindo a partir daí. Concluimos assim, que o valor por defeito do PostgreSQL não é o mais indicado para o caso em estudo, sendo aquele

que apresenta menor performance. O valor **0.8** é aquele que apresenta o valor de débito maior e tempo de resposta menor, sendo assim o mais indicado.

5.1.9 Default Statistics Target

Os query plans dizem respeito à forma como a base de dados decide executar as queries, tendo em conta os dados estatísticos que vai recolhendo sobre as suas tabelas. Uma das várias formas que temos para influenciar esse processo de decisão é alterando um parâmetro designado `default_statistics_target`. Este parâmetro controla a forma como as estatísticas são recolhidas, o seu valor pode variar entre 1 e 10000. O valor padrão é 100, sendo que, quanto maior o valor deste parâmetro, maior o tempo necessário para fazer *analyze*, no entanto, pode melhorar a qualidade das estimativas feitas. Assim sendo realizamos três testes, com valores deste parâmetro em 100, 1000 e 10000. De seguida apresentam-se os resultados.

Valor	Débito	Tempo de Resposta
100	782.9375	0.00699485910433
1000	777.7125	0.0074152080621
10000	595.54761131	0.0301459334663

Figura 5.10: Tabela com os resultados obtidos, variando o valor do parâmetro `default_statistics_target`.

Ao analisar os resultados, podemos verificar que à medida que o valor deste parâmetro aumenta, a performance vai diminuindo, isto é, o debito diminui e o tempo de resposta aumenta. Isto pode-se dever ao facto de as queries já estarem parcialmente otimizadas e não serem muito complexas nem apresentarem tamanhos fora do normal. Fazendo assim com que este processo de recolha de estatísticas e escolha de um plano ótimo seja relativamente fácil e rápida. Assim sendo, o valor padrão **100** é aquele que será escolhido por apresentar melhor performance.

5.1.10 Query Tuning

O PostgreSQL sempre que executa uma query faz uso de determinada estratégia ou plano, com o intuito de tornar essa query o mais rápida possível. No entanto, nós podemos limitar os recursos que o PostgreSQL tem ao seu dispor, bloqueando certas estratégias e forçando-o assim a optar pelas restantes. Desta forma, com o intuito de avaliar a importância de certas estratégias e de que forma estas afetam a performance do sistema, decidimos criar quatro diferentes configurações.

Configuração 1

```
enable_bitmapscan = on
enable_hashagg = on
enable_hashjoin = on
enable_indexscan = on
enable_indexonlyscan = on
```

```
enable_material = on
enable_mergejoin = on
enable_nestloop = on
enable_seqscan = on
enable_sort = on
enable_tidscan = on
```

Configuração 2

```
enable_bitmapscan = on
enable_hashagg = off
enable_hashjoin = off
enable_indexscan = on
enable_indexonlyscan = on
enable_material = on
enable_mergejoin = on
enable_nestloop = on
enable_seqscan = on
enable_sort = on
enable_tidscan = on
```

Configuração 3

```
enable_bitmapscan = on
enable_hashagg = off
enable_hashjoin = off
enable_indexscan = on
enable_indexonlyscan = on
enable_material = off
enable_mergejoin = off
enable_nestloop = on
enable_seqscan = on
enable_sort = off
enable_tidscan = on
```

Configuração 4

```
enable_bitmapscan = off
enable_hashagg = off
enable_hashjoin = off
enable_indexscan = off
enable_indexonlyscan = off
enable_material = off
enable_mergejoin = off
enable_nestloop = on
enable_seqscan = off
enable_sort = off
enable_tidscan = off
```

Configuração	Débito	Tempo de Resposta
Conf1	782.222277778	0.0072338079486
Conf2	773.527830902	0.00785963834979
Conf3	775.244381109	0.00793313231642
Conf4	755.615835594	0.00959870630469

Figura 5.11: Tabela com os resultados obtidos, com as várias configurações.

Como podemos observar na tabela, a configuração que obteve melhor resultado foi a primeira, onde todas as estratégias se encontravam disponíveis, enquanto que nas restantes configurações se verificou uma significativa perda de débito. Isto pode-se dever ao facto que nem sempre é bom forçar a base de dados a utilizar certas estratégias que não sejam as mais indicadas para determinados casos.

5.2 Níveis de isolamento

Aqui iremos expor e comparar o funcionamento do benchmark tendo em conta os níveis de isolamento. Para cada um iremos explicar em que é que esse tipo de métrica consiste e apresentar as métricas débito e tempo de execução resultantes dos três testes efectuados. É de salientar que o nível de isolamento é a única variável em estudo, sendo que todas as outras configurações se mantêm constantes e consoante os valores predefinidos dos PostgreSQL. No entanto, dentro de cada métrica decidimos realizar três testes onde fizemos variar apenas o número de clientes (EB's).

5.2.1 Read Committed

O *Read Committed* é o nível de isolamento padrão do PostgreSQL. Quando uma transação utiliza este nível de isolamento, uma operação **SELECT** apenas vê os dados que foram escritos (**committed**) antes desta operação ter começado, nunca é capaz de ter acesso a dados que não foram escritos (**uncommitted**) ou que foram escritos durante a execução da query por transações concorrentes. De seguida apresentam-se os resultados dos testes efectuados utilizando este nível de isolamento.

Valor	Débito	Tempo de Resposta
10EB	132.980850957	0.0061648651189
30EB	390.4	0.00751623335041
60EB	771.8375	0.00800889112021

Figura 5.12: Tabela com os resultados obtidos, com o nível de isolamento *Read Committed*.

5.2.2 Read Uncommitted

O *Read Uncommitted*, ao contrário do *Read Committed*, consegue ver os dados que são escritos durante uma determinada operação (query) numa dada transação,

podendo levar ao surgimento de dirty reads, tornando assim a base de dados inconsistente. Segundo a documentação, o PostgreSQL não suporta dirty reads e, como tal, trata o *Read Uncommitted* como *Read Committed*, no entanto, decidimos realizar na mesma os testes para este nível de isolamento.

Valor	Débito	Tempo de Resposta
10EB	131.848351896	0.00619548729617
30EB	384.10480131	0.00839459125228
60EB	763.215459807	0.00849136886239

Figura 5.13: Tabela com os resultados obtidos, com o nível de isolamento *Read Uncommitted*.

5.2.3 Repeatable Read

O nível de isolamento *Repeatable Read* apenas vê os dados escritos antes da própria transação ter começado, nunca é capaz de ter acesso a dados que não foram escritos (*uncommitted*) ou que foram escritos durante a execução da transação por transações concorrentes. De seguida apresentam-se os resultados dos testes efectuados utilizando este nível de isolamento.

Valor	Débito	Tempo de Resposta
10EB	127.956693504	0.00805559355154
30EB	375.907801152	0.00991520633126
60EB	600.0125	0.0295611133101

Figura 5.14: Tabela com os resultados obtidos, com o nível de isolamento *Repeatable Read*.

5.2.4 Serializable

O nível de isolamento *Serializable* providencia o isolamento mais restrito da transação. Este nível emula a execução em série de todas as transações, como que se as transações tivessem sido executados umas após as outras, sequencialmente, em vez de concorrentemente. Na prática, este nível de isolamento funciona exatamente como o Repeatable Read à exceção que este está atento a condições que podem fazer com que a execução de um conjunto concorrente de transações se comporte de uma maneira inconsistente tendo em conta todas as possíveis execuções sequenciais dessas transações. De seguida apresentam-se os resultados dos testes efectuados utilizando este nível de isolamento.

Valor	Débito	Tempo de Resposta
10EB	127.692288461	0.00746989720999
30EB	375.85	0.00990824131968
60EB	765.652929338	0.0084209916249

Figura 5.15: Tabela com os resultados obtidos, com o nível de isolamento *Serializable*.

5.2.5 Análise

Como podemos observar nos resultados obtidos, é bem visível a dominância do nível de isolamento *Read Committed* sobre os restantes níveis, sendo que é este que apresenta o maior débito e o menor tempo de resposta para os três grupos de clientes. Isto deve-se exatamente ao que foi dito atrás, este é o nível de isolamento que faz melhor uso da concorrência de transações e, como tal, é aquele que consegue fornecer melhor performance. Logo de seguida vem o *Read Uncommitted*, que, tal como dito atrás, se comporta como *Read Committed* e, desta forma, apresenta um resultado bastante semelhante a este. Por ultimo, surge o *Repeatable Read* e o *Serializable* com as piores performances, devido ao facto de estes tornarem a execução das transações mais sequencial e menos concorrente.

6. *Replicação e Sharding*

Neste capítulo iremos abordar as estratégias de Replicação e de *Sharding* para tentar potenciar mais a performance do *benchmark TPC-W*, dado que já esgotamos todas as estratégias e mecanismos de otimização deste nos capítulos anteriores.

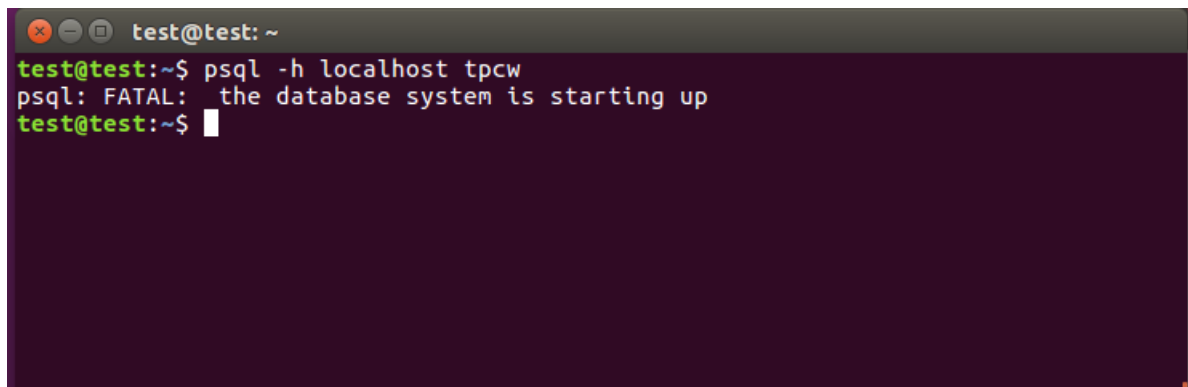
6.1 Replicação

A Replicação da Base de Dados é uma estratégia para resolver os problemas de disponibilidade e elevado número de interrogações de leitura realizadas a mesma. É possível criar várias réplicas da Base de Dados original (primária), e usar o mecanismo de *heartbeat*, para verificar se as réplicas se encontram ativas. Com isto, podemos obter uma melhoria na performance, pois todas as interrogações de leitura eram balanceadas (com *load balancing*) entre a Base de Dados primária e as réplicas criadas.

No entanto, esta estratégia apresenta um aspecto negativo. Quando criamos as várias réplicas, é necessário garantir a consistência dos dados, isto é, quando se efectua uma actualização à Base de Dados primária, todas as réplicas tem que efectuar a mesma actualização, para manter a consistência da Base de Dados. Isto implica que as Bases de Dados se encontram sincronizadas para ocorrer a propagação dos dados da Base de Dados primária, para as respectivas réplicas. Como tal, este processo torna-se dispendioso, e consequentemente pode baixar a performance do *benchmark TPC-W*.

Como já foi analisado, existe uma percentagem significativa de queries do tipo *Update* e *Insert* no *benchmark TPC-W*. Posto isto, apesar da possível melhoria de performance ao balancear as interrogações de leitura, dados a percentagem de actualizações realizadas, a performance global irá baixar, o que não é pretendido.

Desta forma, tentamos implementar a estratégia no nosso *benchmark*, para ver se confirmava as nossas conclusões. No entanto, durante a realização da replicação, esta apresentou um erro que não nos permitiu testar o benchmark. Conseguimos realizar uma réplica na íntegra da Base de Dados primária, no entanto, ao iniciar o *PostgreSQL*, este não arrancava com a réplica da Base de Dados. Podemos ver na seguinte figura 6.1, o erro apresentado ao tentar ligar a Base de Dados replicada:

A terminal window with a dark background and light-colored text. The window title is 'test@test: ~'. The prompt is 'test@test:~\$'. The user has entered 'psql -h localhost tpcw'. The response is 'psql: FATAL: the database system is starting up'. The prompt is now 'test@test:~\$' with a cursor.

```
test@test:~$ psql -h localhost tpcw
psql: FATAL: the database system is starting up
test@test:~$
```

Figura 6.1: Erro apresentado ao iniciar a Base de Dados.

Este impasse, não nos possibilitou confirmar as nossas conclusões.

6.2 *Sharding*

A estratégia de Sharding não se encontra disponível no *PostgreSQL*, no entanto é de grande importância, e como tal iremos abordar teoricamente a sua relevância no nosso *benchmark*.

Sharding permite-nos dividir a Base de Dados original, em vários fragmentos (*Shards*), isto é, dividir num conjunto de partições horizontais. Ao realizar *Sharding* numa Base de Dados, criamos réplicas do esquema (tabelas e os seus respectivos relacionamentos), e repartimos os dados da Base de Dados original pelas várias partições, baseando-nos numa *shard-key*. Esta repartição é efectuada por vários servidores, por forma a distribuir os acessos à Base de Dados.

Como tal, só se deve utilizar este mecanismo como último recurso, depois de termos explorado todas as opções de optimização da Base de Dados e quando é provável que esta escale além dos limites dos recursos de armazenamento e processamento disponíveis. Tem como principal foco aumentar o desempenho através da redução da carga efectuada a uma Base de Dados única, replicando essa carga como descrito anteriormente, aumentar a escalabilidade e melhorar a disponibilidade do sistema.

Posto isto, apesar das inúmeras vantagens apresentadas, um aumento da escalabilidade implica um desempenho menor das queries, por causa da latência existente nos acessos às *shards*. Para além disso, nem sempre é possível gerar-se *shard-keys* e pode acontecer que estas sejam mal geradas, o que piora drasticamente o desempenho. A distribuição entre as várias *shards* tem que ser homogénea, o que implica o rebalanceamento das mesmas. Apesar da distribuição dos dados, a distribuição da carga pode não seguir o mesmo rumo, isto é, caso os dados mais acedidos permaneçam na mesma *shard*, a tentativa de distribuição de carga é falhada. Para tal é necessário realizar um balanceamento que tenha este aspecto em atenção.

7. Resultados finais

Neste capítulo abordaremos os resultados finais conseguidos, aplicando as metodologias de otimização mencionadas nos capítulos anteriores. É de relembrar que todas as metodologias foram testadas individualmente, uma vez que estamos num contexto académico e era necessário demonstrar, para cada uma, o aumento de performance conseguido, e seleccionadas conforme os resultados individuais que estas tiveram. Sendo assim, num contexto profissional diversos conjuntos de aplicações de metodologias diferentes deveriam ser aplicados, uma vez que a selecção das que produziram, individualmente, um resultado ótimo não implica necessariamente que o conjunto final resultante desta selecção seja também o melhor resultado, isto é, o maior aumento da performance do benchmark.

Seguem-se os gráficos representativos do aumento de performance conseguido, para cada uma das máquinas.

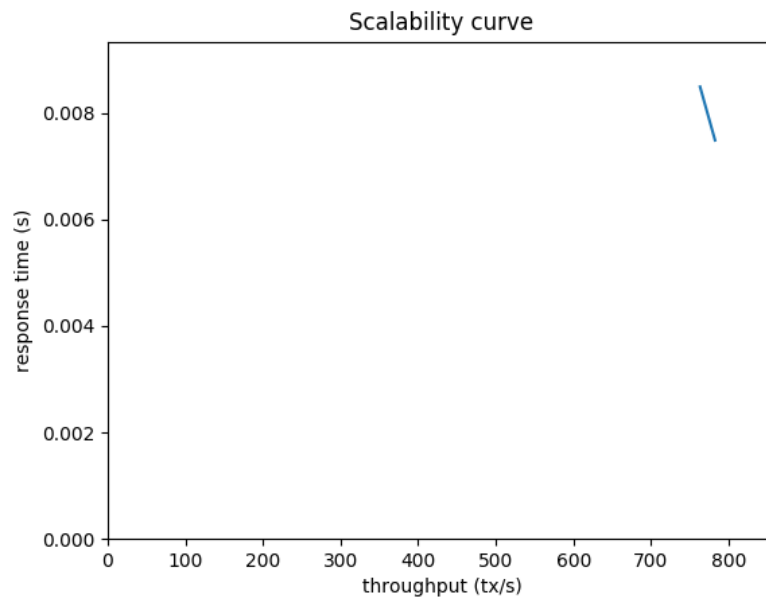


Figura 7.1: Aumento de performance conseguido na máquina *Xiaomi Air*

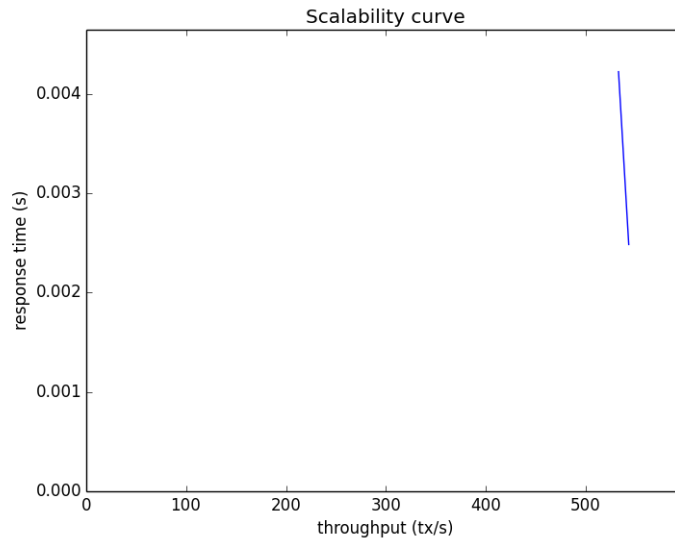


Figura 7.2: Aumento de performance conseguido na máquina *MacBook Pro*

Podemos verificar nas figuras 7.1 e 7.2 que, apesar de um aumento pouco significativo no débito, houve uma redução considerável no tempo de resposta médio aos pedidos efetuados.

Assim, tal como era um dos grandes objetivos deste trabalho, conseguimos melhorar a performance do acesso à base de dados depois de realizadas as alterações resultantes de toda a análise e trabalho efectuado e documentado ao longo de todo o projeto.

8. *Conclusão*

Após nos ter sido apresentado o trabalho prático desta unidade curricular, iniciamos a nossa jornada com um ar de convicção e dedicação, uma vez que estávamos perante um problema que nunca nos tinha anteriormente sido apresentado e, portanto, despertara em nós um enorme entusiasmo e curiosidade de como todo este processo se iria desenrolar.

Ora, o trabalho focou-se em três tópicos fundamentais, em que o primeiro passava por instalar e configurar o benchmark TPC-W de um modo que o funcionamento deste se adequasse ao hardware das nossas máquinas. Todo este processo, em que ao início nos pareceu bastante simples e intuitivo, tornou-se um bocado cansativo e consequentemente desmotivante devido aos vários erros e problemas que nos foram surgindo. No entanto, apesar de um pouco desmotivado, o grupo continuou sólido e sempre com os objetivos bem definidos. Assim, conseguimos chegar a uma escala adequada a cada uma das máquinas utilizadas, com uma utilização do CPU ao nível que era supostamente ideal.

Passando para o segundo tópico do trabalho, o qual passava por otimizar ou justificar o desempenho tendo em conta as interrogações, podemos dizer que este foi aquele ao qual o grupo gastou mais tempo e dedicação. No início, ao observarmos os resultados através da ferramenta PgBadger tivemos algumas dificuldades para seleccionar as interrogações que deveriam ser alvo da nossa maior atenção. No entanto, depois de vários testes efectuados, conseguimos seleccionar aquelas que inevitavelmente poderiam ser melhoradas de modo a diminuir os tempos tão elevados de execução. Assim sendo, pusemos mãos à obra e, fazendo uso de vistas materializadas e índices, conseguimos indiscutivelmente aumentar a performance de um grande conjunto de interrogações. Por outro lado, houve uma série de interrogações que, depois de serem alvo da nossa dedicação, não tiveram uma melhoria visível da sua performance, o qual foi parcialmente justificável.

Finalmente chegamos ao último objetivo do trabalho, o qual passava por otimizar ou justificar o desempenho tendo em conta os parâmetros de configuração. Esta foi a etapa mais mecânica e objectiva de todo o projeto, no sentido em que não tivemos de fazer um grande planeamento antes de iniciar as tarefas, ao contrário da tarefa anterior. Seleccionamos um conjunto de parâmetros do ficheiro de configuração do PostgreSQL que nos pareceram relevantes no sentido de assumirem um papel influenciador no desempenho do nosso benchmark. Depois de termos realizado os testes para cada um desses parâmetros, verificamos que os resultados obtidos não se afastaram muito daquilo que era suposto e, portanto, obtivemos um resultado bastante positivo.

Por fim, depois de toda esta jornada, onde muitos problemas foram surgindo e

também muitas soluções foram encontradas, achamos que o resultado final vem de encontro às expectativas do grupo e conseqüentemente do próprio objetivo do trabalho. Conseguimos atingir melhorias de performance bastante significativas e, ao mesmo tempo, justificar as tentativas falhadas de atingir essas mesmas melhorias. Assim, achamos que temos aqui o resultado de um projeto bastante satisfatório que conseguiu cumprir todos os objetivos inicialmente propostos.

Concluindo, numa nota mais pessoal, a realização deste trabalho permitiu a todos os elementos do grupo adquirir e aprofundar muitos conhecimentos que anteriormente não nos eram tão familiares, e que, desta forma, certamente nos serão muito úteis num futuro próximo.