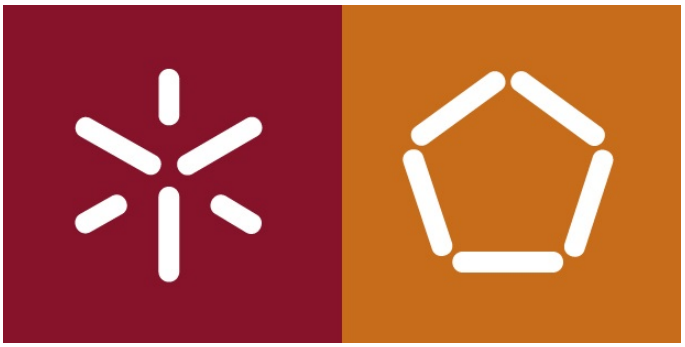


Universidade do Minho - Escola de Engenharia

Administração de Base de Dados

Benchmark TPC-C em PostgreSQL

Autores :
Diogo Costa(A78034)

Marco Silva(A79607)

Patrícia Barreira (A79007)

Versão 1.0
7 de Fevereiro de 2019

Resumo

Este documento explicita o processo de otimização de uma base de dados (BD) PostgreSQL. Nela foram realizados dois tipos de carga: o *benchmark* TPC-C, onde foi tido em conta o débito produzido, e algumas interrogações analíticas, onde o importante era otimizar o tempo de resposta. Primeiramente, foi encontrada uma configuração de referência com a qual as restantes iterações da BD foram comparadas. De seguida, foi realizada uma otimização vertical, isto é, apenas afinando os parâmetros do PostgreSQL, o plano de execução e mecanismos de redundância. Por fim, abordou-se algumas possibilidades de otimização horizontal como a replicação e o *sharding*. Ainda, realçar que todo o processo foi realizado recorrendo ao serviço `Google Cloud Platform`.

Conteúdo

1 Introdução

A base de dados PostgreSQL é uma das mais usadas atualmente e por isso está em constante evolução, nomeadamente no que toca às funcionalidades que permitem otimizar o sistema como um todo [1]. Este trabalho tem como objetivo explorar quais os mecanismos que permitem otimizar da melhor forma dois tipos de carga. Por um lado, tem-se o TPC-C que é do tipo *on-line transaction processing* (OLTP) *benchmark*. Este é composto por 5 transações concorrentes de diferente complexidade e tem como objetivo simular o processo de venda de determinados produtos, onde entra componentes como os clientes, encomendas, armazéns, entre outros [2]. Desta forma, o TPC-C produz uma carga que pode ser ajustada mas que é caracterizada por ser bastante intensiva, no que toca ao consumo de recursos como o CPU ou a RAM. Por outro lado, tem-se as interrogações analíticas. Estas tem um comportamento completamente díspar do TPC-C uma vez que apenas são executadas ocasionalmente, utilizam um grande volume de dados e são apenas de leitura. Assim sendo, o objetivo é produzir os melhores resultados possíveis. Para tal, a métrica que usada para comparar a carga TPC-C será o débito conseguido. Por sua vez, as interrogações analíticas serão comparadas com base no tempo de resposta associado a cada uma.

De forma a uniformizar todo o processo, o ambiente de otimização foi o `Google Cloud Platform`, onde todos os testes foram realizados.

Primeiramente, encontrou-se uma configuração de referência que permitisse a comparação das várias iterações de otimização, de forma a perceber a evolução conseguida.

De seguida, procedeu-se ao processo de otimização vertical, onde foram utilizados mecanismos como a otimização do plano de execução, a introdução de mecanismos de redundância (e.g. índices, vistas, vistas materializadas, ...) ou mesmo o afinamento das configurações do PostgreSQL. Este processo foi realizado tanto para o TPC-C como para as interrogações analíticas.

Por fim, realizou-se otimização dita horizontal, onde se utilizou estratégias como replicação em *streaming,* com base nos próprios mecanismos do PostgreSQL, e *sharding* utilizando o *software* `Citus` de forma a permitir melhorias na performance em relação à configuração de referência.

2 Escolha da configuração de referência

A escolha da configuração de referência teria de possuir um hardware que aguentasse com um tamanho considerável de dados/processamento, no entanto sem nunca saturar o sistema. Tendo em vista que um dos requisitos passava por ter o tamanho BD superior ao tamanho da RAM achou-se que 7.5 GB de RAM (uma das opções oferecidas no serviço de cloud), permitiam um conjunto de dados substâncial (cerca de 8GB). Adicionalmente, dois CPU's permitiram que obter um sistema final equilibrado sem que fosse facilmente saturado.

Desta forma, fixou-se o hardware em:

- Machine type: custom (2 vCPUs, 7.5GB memory)
- CPU platform: Intel Haswell
- Disk: Standard persistent disk com 15GB

De seguida, para escolha da configuração de referência do tpc-c fez-se a seguinte análise:

Warehouses	Throughput	Response Time
2	1,823196671	0,004347319
4	3,421091569	0,004341552
8	6,897142139	0,004345746
16	13,82799325	0,004586068
25	21,14926059	0,004699823
32	27,49272391	0,005462359
40	33,65785355	0,005275807
45	37,78022197	0,010221966
50	43,80002262	0,016036814
55	45,00351716	0,021012219
60	52,22046428	0,022347377
64	55,69772785	0,03773738
70	60,8588139	0,040986698
80	67,36574087	0,074384953

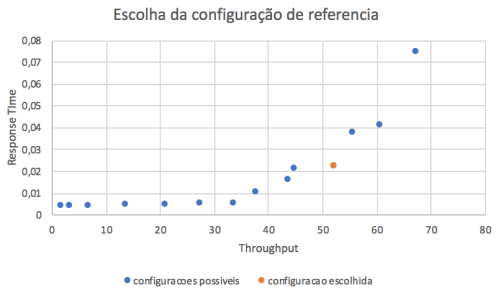


Figura 1: Dados recolhidos para a escolha da configuração de referência(à esquerda), gráfico à direita

Começou-se primeiro por fixar o tempo de execução do run. Verificou-se que em média seria preciso 5 minutos para aquecer a cache, e portanto decidiu-se ter 15 minutos de teste estáveis. Traduz-se então em 20 minutos de execução do run.

Para além disso, o número de clientes por warehouse foi fixado em 10, variando portanto o número total de clientes de acordo com o número de warehouses a testar.

Para os diferentes testes apresentados a cima foi automatizado o seu processo de criação da base de dados e popular, usando a script shell init.sh.

A escolha da configuração de referência é 60 Warehouses, traduzindo num número total de cliente 600, mas com o thinktime a true. O tamanho da base de dados é de 8.1GB, pouco maior que o tamanho da RAM escolhida.

3 Concretização do objetivo 2

Escolhida uma configuração de referência, e dado que o load dessa configuração demora algum tempo, foi tomada a decisão de fazer dump da base de dados depois de fazer o load, para rapidamente se obter a base de dados com essa configuração utilizando o restore, evitando assim a re-execução do load. Para realizar este passo foram tomadas as seguintes decisões:

- Para otimização do pg_dump:

  - opção Fc (formato do ficheiro)

- Para otimização do pg_restore:
  - opção jobs que permite tirar proveito do paralelismo.

Na otimização do pg_dump sem dúvida que tinha-se de utilizar o formato do ficheiro binário, dado que o tamanho do ficheiro gerado é bastante menor se fosse utilizado o ficheiro com os comandos SQL transcritos na integra. Sendo assim o comando utilizado foi:

```
time pg_dump -h localhost -F c tpcc > tpcc.dump
```

Houve uma maior preocupação na otimização do pg_restore uma vez que este ia ser utilizado bastante vezes pelo grupo de trabalho. Verificou-se uma grande diferença de tempo quando o grupo de trabalhou decidiu aumentar para 2 CPU, 7.5GB de RAM da máquina(17min para 4min). Depois dentro da configuração escolhida para a máquina final, verificou-se que a alteração do número de jobs influenciava apenas segundos. Foi então usado o seguinte comando:

```
time pg_restore -h localhost -d tpcc -F c -j 8 tpcc.dump
```

De forma a automatizar ainda mais o processo de restore, foi feita uma shell script que realiza todos os passos necessários para a concretização do restore.

Segundo passo que o grupo de trabalho tomou foi utilizar uma ferramenta de monitorização e diagnóstico do PostgreSQL, pgbadger, de forma a facilitar toda a análise do sistema. Para concretização deste passo, foi necessário ativar o sistema de log, e para isso foram alterados os seguintes parametros de configuração do postgreSQL:

- Where to log:
  - log_destination = 'stderr'
  - logging_collector = on
  - log_directory = 'pg_log'
  - log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
- When to Log:
  - log_min_duration_statement = 0 (para todas as transações serem analisadas no pgbager)
- What to Log:
  - log_checkpoints = on
  - log_connections = on
  - log_disconnections = on
  - log_duration = off
  - log_error_verbosity = default
  - log_line_prefix = '
  - log_lock_waits = on
  - log_statement = 'none'
  - log_temp_files = 0
- autovaccum parameters
  - log_autovacuum_min_duration = 0 (para todos os autovacuum serem analisados no pg-bader)

3.1 Interrogações TPC-C

Tendo então tudo operacional, foi feita a análise das interrogações do sistema e os mecanismo de redundância já existentes. Existem então 5 interrogações:

- neworder
- payment
- orderstatus
- delivery
- stocklevel

Tendo em conta a execução dos scripts *load.sh* e *run.sh*, foi feita uma análise do tamanho de cada uma das tabelas, bem como dos mecanismos de redundância já existentes. E ainda, uma análise do número de páginas que cada tabela ocupa bem como os índices que já existem. Esta informação foi obtida do pg_class, mais propriamente, do atributo relpages.

Table	Size	External Size
order_line	2627 MB	1116 MB
stock	2516 MB	386 MB
customer	1116 MB	178 MB
orders	278 MB	147 MB
history	198 MB	39 MB
new_order	51 MB	28 MB
item	16 MB	4416 kB
district	152 kB	64 kB
warehouse	48 kB	32 kB

Table	Size	External Size
order_line	2674 MB	1135 MB
stock	2516 MB	386 MB
customer	1115 MB	178 MB
orders	282 MB	149 MB
history	201 MB	39 MB
new_order	51 MB	28 MB
item	16 MB	4416 kB
district	152 kB	64 kB
warehouse	48 kB	32 kB

Figura 2: Tamanho tabelas e mecanismos redundantes tpc-c depois do load (à esquerda) e do run (à direita).

tabela	operação	paginas tabela	paginas indice pk	pagina indice ix	pagina indice key
order_line	load	193392	58906	41948	41948
	run	196939	59931	42677	42677
stock	load	272728	16454	16454	16454
	run	272728	16454	16454	16454
customer	load	120000	6933	10931	4939
	run	119958	6933	10931	4939
orders	load	16823	6933	6933	4939
	run	17046	7036	7036	5012
history	load	20455	-	-	4939
	run	20719	-	-	5002
new_order	load	2919	-	2082	1484
	run	2932	-	2091	1490
item	load	1539	276	-	276
	run	1539	276	-	276
district	load	11	4	-	4
	run	11	4	-	4
warehouse	load	2	2	-	2
	run	2	2	-	2

Figura 3: Tamanho em disco de cada tabela e índices já existentes

Está informação ajudará numa melhor escolha dos algoritmos pois rapidamente se percebe se cada operação da interrogação cabe em memoria e quantos acessos terão de ser feitos ao disco.

3.1.1 Neworder

Primeiramente, fez-se uma análise detalhada das operação à base de dados.

tabela	insert	update (tipo de dados atualizados)	update (atributos clausula where)	select (atributos a selecionar)	select (atributos clausula where)	order by	delete(atributos da clausula where)
order_line	✓	-	-	-	-	-	-
stock	✗	integer, integer	s_i_id,s_w_id	*	s_i_id, s_w_id	✗	-
		integer, integer	s_i_id,s_w_id				
customer	✗	-	-	*	c_id, c_w_id, c_d_id	✗	-
orders	✓	-	-	-	-	-	-
history	✗	-	-	-	-	-	-
new_order	✓	-	-	-	-	-	-
item	✗	-	-	*	i_id	✗	-
district	✗	integer	d_w_id, d_id	*	d_w_id, d_id		-
				d_next_o_id	d_w_id, d_id	✗	
warehouse	✗	-	-	*	w_id	-	-

Figura 4: Operações efetuadas na interrogação neworder

Em termos de plano de execução, gerou-se o pgbagder para a configuração de referência e com as configurações do postgreSQL até ao momento, a transação new_order mais lenta foi:

tpcc_neworder ('42', '2', '1774', '10', '0', '20384,89919,81750,47616,40303,21492,65152,23304,15796,80272,', '42,42,42,42,42,42,42,42,42,42,', '2,8,1,2,6,9,1,4,7,7,');

Com os valores desta transação foi verificado os planos de execução das operações desta interrogação e chegou-se a conclusão que, todos as interrogação usam os índices já criados. Mais ainda, os planos de execução de cada uma das operações são bastantes simples, portanto melhorias nesse sentido não se encontram.

A análise da tabela da figura 4 permite rapidamente tirar conclusões sobe que atributos é que poderá haver redundância, mais propriamente os atributos da clausula where. Começando então a analise para os mecanismos de redundância, existe então os seguintes índices:

- pk_stock (b-tree) sobe s_i_id e s_w_id
- pk_customer (b-tree) sobe c_id, c_w_id e c_d_id
- pk_item (b-tree) sobe i_id
- pk_district (btree) sobe d_w_id e d_id
- pk_warehouse (btree) w_id

Será então este mecanismo de redundância realmente mais eficiente que um seq. scan? Tendo em conta que em todos os casos, o que procuramos é um único valor, da-nos desde já uma pista que o uso de índices deverá trazer uma grande vantagem. Em particular, o uso de uma b-tree faz com que o número de acesso aos disco seja mesmo reduzido, ou seja, este é igual à altura da árvore. De forma a obter esta altura, foi utilizado um modulo adicional, pageinspect, que permite fazer uma analise mais baixo nivel. Através da query fornecida em link na secção Summarize keyspace of a B-Tree index, foi então possível concluir que o número de níveis das b-tree usadas são:

- pk_stock (b-tree) 3 níveis
- pk_customer (b-tree) 3 níveis
- pk_item (b-tree) 2 níveis
- pk_district (btree) 2 níveis
- pk_warehouse (btree) 1 nível

Tendo como exemplo a tabela stock, com a b-tree traduz-se em 3 disk I/O para encontrar o apontador do registo, mais 1 disk I/O para trazer o registo. O que comparando com o seq. scan se torna ridículo, iria ser necessário 272728 disk I/O. A única tabela que porventura poderá ser útil usar o seq. scan será warehouse, uma vez que o número de páginas da tabela difere em uma unidade da b-tree. Em todos os outros casos o número de paginas da tabela está numa ordem de grandeza bastante elevada, o que implicaria bastantes acessos ao disco. Em termos de espaço em memoria, verificamos que nesta interrogação só é utilizada uma tabela por operação, e pela analise da figura 1, não trazerá problemas em termos de não ser suficiente espaço em memoria, ou seja, haver trocas entre disco e memoria. Assim conclui-se que é imprescindível o uso de mecanismos de redundância.

Será então a estrutura b-tree a mais eficiente dos mecanismos de redundância? Estamos no caso em que o acesso é por chave primária, e então só produzirá um único resultado a pesquisa. Clustered index não seria uma boa aposta uma vez que só há um resultado, o esforço de ordenar seria inútil. Uma vez que temos que a operação é igualdade, poder-se-á optar por usar hash index. Para o caso em que o índice é sobe mais que 1 atributo, seria necessário criar um hash index para cada atributo e depois fazer join dos resultados. E cada utilização desse índice, podia assim devolver mais que um resultado, uma vez que só o conjunto é que é uma chave primária. Para este casos, que isto ia ser mais custoso que o número baixo de disk I/O das b-trees. Contudo quando há apenas um único atributo valerá a pena avaliar se hash index não será melhor. Estão nestas condições os seguintes casos:

- item - i_id
- warehouse - w_id

Analisando o plano de execução com b-tree e com hash surgem os seguintes resultados(tempo de execução):

- item:
  - b-tree:
    - tempo de planeamento: 0.865 ms
    - tempo de execução: 0.127 ms
  - hash:
    - tempo de planeamento: 34.743 ms
    - tempo de execução: 0.107 ms
- warehouse:
  - b-tree:
    - tempo de planeamento: 0.105 ms
    - tempo de execução: 0.051 ms
  - hash:
    - tempo de planeamento: 0.941 ms
    - tempo de execução: 0.162 ms

Tendo em conta que está é query mais frequente no sistema, qualquer pequena alteração que seja feita, irá produzir algum impacto, portanto decidiu-se criar um hash index para a tabela item sobe o atributo i_id. Na tabela warehouse, uma vez que está tem apenas 60 entradas e mantém este tamanho ao longo do tempo(só sofre updates), a b-tree mostra-se mais eficiente que a hash. Aqui verificou-se também o uso de seq. scan uma vez que a tabela é pequena, e mostrou poucas diferenças relativamente à b-tree.

Poderíamos optar também por vistas materializadas, mas o custo de manter atualizada, seria mais custoso que a b-tree. Portanto, a b-tree e hash-index são as estruturas escolhidas em termos de redundância.

Uma vez que esta é a transação que ocorro com mais frequência, e cada transação é composta por 3 updates e 3 inserts, na parte de otimização dos parâmetros de configuração do postgreSQL, os parâmetros de vacuum terão de ser analisados com cuidado. O uso de índices requer também atenção sobre o share_buffer uma vez que é necessário suportar para o caso da b-tree o tamanho de cada pagina(nodo da arvore) e para hash, o tamanho da hash. Temos que o tempo de execução das primeiras três transações mais lenta desta interrogação foi de 1s28ms, 1s23ms, 660ms.

3.1.2 Payment

Para a interrogação payment são efetuadas as seguintes operações à base de dados:

tabela	insert	update (tipo de dados atualizados)	update (atributos clausula where)	select (atributos a selecionar)	select (atributos clausula where)	order by	delete(atributos da clausula where)
order_line	✗	-	-	-	-	-	-
stock	✗	-	-	-	-	-	-
customer	✗	numeric, integer, numeric	c_w_id, c_d_id, c_id	* (dentro if(2))	c_last, c_w_id, c_d_id	c_w_id, c_d_id, c_last, c_first (com limit 1)	-
		text (dentro if(1))	c_w_id, c_d_id, c_id	*	c_id, c_w_id, c_d_id	✗	
orders	✗	-	-	-	-	-	-
history	✓	-	-	-	-	-	-
new_order	✗	-	-	-	-	-	-
item	✗	-	-	-	-	-	-
district	✗	numeric	d_w_id, d_id	*	d_w_id, d_id	✗	-
warehouse	✗	numeric	w_id	*	w_id	✗	-

Figura 5: Operações efetuadas na interrogação payment

Existem então os seguintes índices, para os atributos em questão:

- ix_customer (btree) sob c_w_id, c_d_id, c_last
- pk_customer (btree) sob c_w_id, c_d_id, c_id
- pk_district (btree) sob d_w_id, d_id
- pk_warehouse (btree) sob w_id

Esta interrogação engloba os índices já justificados na secção 3.1.1. Temos o acréscimo de ter uma operação order by,contudo, está está dentro de uma condição if. Foi feita a análise no pg_badger se as queries mais lentas desta transação verificavam a condição do if, e conclui-se que não. Temos também o uso de um indice que ainda não foi analisado, o ix_customer. Primeiro verificou-se se o plano de execução engloba este índice e conclui-se que sim. A b-tree é constituida por 3 níveis. Será então, o mecanismo mais eficiente? Sim, uma vez que engloba 3 atributos, fará 4 disk I/O e cada página do b-tree cabe em memoria.

Em termos de plano de execução, gerou-se o pg_bagder para a configuração de referência e com as configurações do postgreSQL até ao momento, a transação payment mais lenta foi:

tpcc_payment ('8', '8', cast('2437.98999' AS numeric(6, 2)), '8', '8', '1914', cast('' AS char(16)))

Com os valores desta transação foi verificado que para todos os casos em que existe uma b-tree, o plano de execução inclui essa estrutura. Mais ainda, os planos de execução de cada uma das operações são bastantes simples, portanto melhorias nesse sentido não se encontram.

Mais uma vez, na analise dos parâmetros de configuração, especial atenção ao vacuum, pois temos 4 updates e também no share_buffer e work_mem para concretização da operação order by. Temos que o tempo de execução das primeiras três transações mais lenta desta interrogação foi de 259ms, 258ms e 258ms, esperando alcançar melhorias com refinamentos nos parâmetros de vacuum, pois em relação aos selects não haverá muita margem de manobra graças ao uso dos mecanismos de redundância trazerem já uma melhoria enorme.

3.1.3 Orderstatus

Para a interrogação orderstatus são efetuadas as seguintes operações à base de dados:

tabela	insert	update (tipo de dados atualizados)	update (atributos clausula where)	select (atributos a selecionar)	select (atributos clausula where)	order by	delete(atributos da clausula where)
order_line	✗	-	-	* (dentro else(2))	ol_o_id, ol_d_id, ol_w_id	✗	-
stock	✗	-	-	-	-	-	-
customer	✗	-	-	* (dentro if(1))	c_last, c_w_id, c_d_id	c_w_id, c_d_id, c_last, c_first (com limit 1)	-
				* (dentro else(1))	c_id, c_d_id, c_w_id	✗	
orders	✗	-	-	* (dentro else(2))	o_c_id, o_d_id, o_w_id	o_id (com limit 1)	-
history	✗	-	-	-	-	-	-
new_order	✗	-	-	-	-	-	-
item	✗	-	-	-	-	-	-
district	✗	-	-	-	-	-	-
warehouse	✗	-	-	-	-	-	-

Figura 6: Operações efetuadas na interrogação orderstatus

Para analise do plano de execução e verificação dos algortimos escolhidos, a transação orderstatus mais lenta foi:

tpcc_orderstatus ('44', '6', '1', 'PRICALLYEING');

Em termos de plano de execução, valerá a pena prestar uma maior atenção aos atributos sob order by. Mais uma vez, o order by com 4 atributos não entra no conjunto de query mais lentas(tempo de execução entre 312ms -187ms). Já o orderby com um atributo sob a tabela orders, é praticamente executado sempre. Portanto, começou-se por tentar perceber, o número de linhas que o order by iria receber. Para isso, fez-se o seguinte:

```
select attname,  most_common_freqs from pg_stats
    where tablename='orders' and attname='o_c_id' or
    attname='o_d_id' or attname='o_w_id';
```

A ordem dos atributos na clausula where é o_c_id, o_d_id e o_w_id. Portanto, há primeiro um filtro sobre o atributo o_c_id e esse resultado é passado ao filtro sobre o_d_id e por fim o_w_id. É usada uma b-tree para este calculo intermédio. Pela análise do resultado do pg_stats e sabendo que a cardinalidade da tabela é 1806934 neste momento(a tabela orders sobre inserts portanto a cardinalidade não é constante), temos que o máximo de resultados que o o_c_id irá retornar, será o valor que aparece com mais frequência vezes a cardinalidade. Ou seja, 0.000733333*1806934 = 1325 linhas. Sobre este número de linhas ainda será filtrado o valor de o_d_id e o_w_id, portanto o order by receberá no min 1 linha e no máximo 1325. A correlation(0.000977788, 0.121337, 1, respetivamente) destes atributos, não é muito favorável, mas graças ao compromisso do número de linhas este não apresenta grandes custos. Por exemplo, na query mais lenta do pg_badger, temos está operação demorou 0.235ms. Para analise do work_mem deverá ser considerado o seguinte: tem-se que 1806934 linhas corresponde a 16823 páginas, portanto 1325 linhas são aproximadamente 12 páginas. Cada página do postgreSQL tem 8192 bytes o que traduz em 96Kb.

Para análise dos mecanismos de redundâncoa, existem então os seguintes índices, para os atributos em questão da causula where:

- pk_order_line (btree) sob ol_w_id, ol_d_id, ol_o_id e ol_number(neste caso só nos interessa os 3 primeiros)
- ix_customer (btree) sob c_w_id, c_d_id, c_last
- pk_customer (btree) sob c_w_id, c_d_id, c_id
- ix_orders (btree) sob o_w_id, o_d_id, o_c_id

Esta interrogação engloba alguns índices já justificados nas secções anteriores. Em relação ao select sobre a tabela order_line e orders, verificou-se que ambos usam o indice b-tree. Mais uma vez, o

uso de indice torna-se imprescendível graças ao tamanho da tabela ser enorme, 193392 e 16823 respetivamente, sem esquecer que existem inserts sobre estas duas tabela na transação new_order, portanto o número de páginas desta tabela está sempre a crescer. Seria impensável fazer um seq. scan neste caso, uma vez que estamos a procura de um número pequeno de resultados. Mais uma vez, b-tree torna-se a melhor opção pelo número de atributos que formam o índice, e por todas as vantagens em termos de disk I/O e memória que esta trás.

Está interrogação é formada por apenas selects, e apresenta-se com resultados ótimos no pg_badger graças ao uso dos mecanismos de redundância. Temos que o tempo de execução das primeiras três transações mais lenta desta interrogação foi de 312ms, 221ms e 190ms.

3.1.4 Delivery

Para a interrogação delivery são efetuadas as seguintes operações à base de dados:

tabela	insert	update (tipo de dados atualizados)	update (atributos clausula where)	select (atributos a selecionar)	select (atributos clausula where)	order by	delete(atributos da clausula where)
order_line	✗	timestamp without time zone (dentro if(1))	ol_w_id, ol_d_id, ol_o_id	* ((dentro if(1))	ol_w_id, ol_d_id, ol_o_id	✗	-
stock	✗	-	-	-	-	-	-
customer	✗	numeric, integer (dentro if(1))	c_w_id, c_d_id, c_id	-	-	-	-
orders	✗	integer (dentro if(1))	o_w_id, o_d_id, o_id	* ((dentro if(1))	o_w_id, o_d_id, o_id	✗	-
history	✗	-	-	-	-	-	-
new_order	✗	-	-	*	no_w_id, no_d_id	no_o_id (com limit 1)	no_w_id, no_d_id, no_o_id (dentro if(1))
item	✗	-	-	-	-	-	-
district	✗	-	-	-	-	-	-
warehouse	✗	-	-	-	-	-	-

Figura 7: Operações efetuadas na interrogação delivery

Para analise do plano de execução e verificação dos algortimos escolhidos, a transação orderstatus mais lenta foi:

tpcc_delivery ('20', '3');

De notar que está é a interrogação mais lenta do sistema, através da analise do pg_bagder. Com uma predominância quase de 90% nas 100 transações mais lentas. Começando com um tempo de 3s895ms e descendo para 1s a partir da 58 transação.

Pela análise da tabela, verifica-se que quando a condição do if é verdadeira, são feitos 3 updates, 1 delete e 2 selects, em vez de apenas um 1 select com order by. Esse if é executado praticamente em todas as vezes. Existem então os seguintes índices, para os atributos em questão da clausula where:

- ix_new_order (b-tree) sob no_w_id, no_d_id e no_o_id
- pk_orders (b-tree) sob o_w_id, o_d_id e o_id
- pk_order_line (b-tree) sob ol_w_id, ol_d_id, ol_o_id, ol_number(só usa os 3 primeiros atributos)
- pk_customer (btree) c_w_id, c_d_id e c_id

Está interrogação engloba alguns índices já justificados nas secções anteriores, com exceção do ix_new_order. Este índice encontra-se nas mesmas condições justificadas anteriormente, portanto, foi optado manter o uso deste índice.

O facto de ser a query mais lenta, verificou-se que estes resultados só são verificados no inicio da execução da script load.sh, e portanto esses tempos de execução não estão a ser considerados para o débito final do sitema. Não deixa de ser uma query com algum peso computacional, 3 updates e 1 delete que deverão ser considerados na analise do vacuum. Teremos um select com order by que será sempre executado, e portanto é importante garantir que a operação order by consegue ser feita toda em memória evitando trocas entre memória e disco que acarretam um custo de performance enorme. Tentou-se então, perceber qual seria o tamanho do resultado passado ao order by. Mais uma vez foi feita a análise do pg_stats dos atributos da clausula where, no_w_id e no_d_id. Haverá primeiro um filtro sobre o atributo no_w_id e esse resultado passado ao filtro sobre

no_d_id. É usada uma b-tree para este calculo intermédio. Pela análise do pg_stats e sabendo que a cardinalidade da tabela é 540593 neste momento(há inserts e deletes sobre a tabele portanto a cardinalidade não é constante), temos que o máximo de resultado que o no_w_id irá retornar será a maior frequência desse atributo vezes a cardinalidade. Ou seja, 0.0181 * 540593 = 9784 linhas. Este atributo apresenta uma correlation igual a 1, o que permitirá que todos os valores iguais estejam agrupados nas mesmas páginas o que torna este filtro bastante eficiente. Desde resultado é filtrado o valor de no_d_id pretendido. Portanto têm-se que order by receberá no máximo 9784 e no min 1 linha. Para 540593 linhas temos 2919 páginas então para 9784 teremos 52 páginas. Cada página do postgreSQL ocupa 8192 bytes, teremos então no máximo 416Kb. Este valor deverá ser tomado em consideração no estudo do paramêtro work_mem.

3.1.5 Stocklevel

Para a interrogação stocklevel são efetuadas as seguintes operações à base de dados:

tabela	insert	update (tipo de dados atualizados)	update (atributos clausula where)	select (atributos a selecionar)	select (atributos clausula where)	order by	delete(atributos da clausula where)
order_line	✗	-	-	count(distinct(s_i_id))	ol_w_id, ol_d_id, ol_o_id (between dois valores), s_w_id, s_i_id, s_quantity (menor que um valor)	-	-
stock	✗	-	-				-
customer	✗	-	-	-	-	-	-
orders	✗	-	-	-	-	-	-
history	✗	-	-	-	-	-	-
new_order	✗	-	-	-	-	-	-
item	✗	-	-	-	-	-	-
district	✗	-	-	*	d_w_id, d_id	-	-
warehouse	✗	-	-	-	-	-	-

Figura 8: Operações efetuadas na interrogação stocklevel

Pela análise do pg_badger a transação mais lenta desta interrogação foi:

tpcc_stocklevel ('5', '5', '13')

Dadas as operações que são feitas, podem-se utilizar os seguintes mecanismos de redundância:

- pk_district (btree) sob d_w_id e d_id
- pk_order_line (b-tree) sob ol_w_id, ol_d_id, ol_o_id, ol_number(só usa os 3 primeiros atributos)
- pk_stock (btree) sob s_w_id e s_i_id

O primeiro select sobre a tabela district usa o indice criado, e como vimos anteriormente é a melhor opção. Já no caso do segundo select, tem-se um join de duas tabelas, order_line e stock, e o respetivo plano de execução usando os valores da transação apresentada a cima é:

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1796.59..1796.60 rows=1 width=8) (actual time=269.082..269.082 rows=1 loops=1)
   ->  Nested Loop  (cost=0.99..1796.59 rows=3 width=4) (actual time=0.396..269.050 rows=8 loops=1)
         ->  Index Scan using pk_order_line on order_line  (cost=0.56..181.71 rows=98 width=8) (actual time=0.154..0.448 rows=184 loops=1)
               Index Cond: ((ol_w_id = 5) AND (ol_d_id = 5) AND (ol_o_id >= 2992) AND (ol_o_id <= 3011))
         ->  Index Scan using pk_stock on stock  (cost=0.43..16.48 rows=1 width=8) (actual time=1.459..1.459 rows=0 loops=184)
               Index Cond: ((s_w_id = 5) AND (s_i_id = order_line.ol_i_id))
               Filter: (s_quantity < 13)
               Rows Removed by Filter: 1
 Planning time: 32.544 ms
 Execution time: 269.136 ms
```

Uma vez que temos operação de filtro por intervalos temos que a estrutura mais eficiente para esse casos é b-tree, ou sejas, nas operações ol_o_id >= 2992 e ol_o_id <= 3011.(verificar com eles se o s_quantity está fora). A questão que se coloca agora é, será o mecanismo escolhido para join, NLJ, o melhor? De facto, o actual time=0.396..269.050 é bastante bom, uma vez que estão a ser usados índices para percorrer as tabelas.

A operação `distinct` implica especial atenção para o `work_mem`.

3.2 Interrogações analíticas

A análise das interrogações analíticas baseia-se na configuração de referência para gerar os tempos de execução da cada *query*. De facto, existem dois momentos em que as medições se revelaram bastante díspares, ainda que úteis consoante o paradigma de execução. Por um lado, tem-se a medição associada ao pior tempo, que corresponde à primeira vez em que a *query* é executada, uma vez que não é tirado partido da cache existente nos mecanismos do PostgreSQL. A justificação para a sua pertinência advém das características da *query* que se está a analisar. Efetivamente, como se trata de uma interrogação analítica, esta apenas será realizada esporadicamente e, consequentemente, não deverá ter os valores de que necessita em cache, pois muitas outras interrogações foram realizadas entretanto. Por outro lado, caso a *query* seja feita num ambiente que dispõe da vantagem de utilização da cache, então foram medidos tempos que tiram partido dessa funcionalidade, ou seja, que resultam de múltiplas execuções da mesma *query*. No entanto, dada a dificuldade na medição de tempos em que os dados não se encontram na cache, foram apenas preservados os melhores tempos após múltiplas execuções da mesmo interrogação.

Além disso, foi necessário avaliar o tipo de carga que o sistema estava a sofrer. Por um lado, o *benchmark* TPCC é uma carga contínua no sistema, baseada em interrogações tanto de escrita como de leitura em geral simples. Por outro lado, as interrogações analíticas caracterizam-se por serem apenas de leitura. No entanto, estas são apenas realizadas ocasionalmente e, normalmente, envolvem a interação com uma grande quantidade de dados. Assim sendo, percebeu-se que as configurações do PostgreSQL que se revelam mais promissoras podem variar tendo em conta o tipo de carga. Desta forma, serão consideradas configurações isoladas e diferentes tanto para o *benchmark* TPCC, como para as interrogações analíticas.

3.2.1 A1

A interrogação analíticas A1 (3.2.1) tem como objetivo calcular a perda anual se apenas fossem processados os pedidos em que a quantidade fosse superior à quantidade média vendida desse produto. Sendo que neste caso são analisados todos os produtos cujo nome começa por 'b'. O resultado da execução da interrogação é 39195450.265000000000.

Interrogação A1.

```
select sum(ol_amount) / 2.0 as avg_yearly
from order_line,
     (select i_id, avg(ol_quantity) as a
      from item, order_line
      where ol_i_id = i_id and i_data like 'b%'
      group by i_id) t
where ol_i_id = t.i_id and ol_quantity < t.a;
```

Desta forma, as medições base foram as seguintes, com base no plano de execução: 4551.065 ms.

Mais ainda, o plano escolhido pelo PostgreSQL para realizar a interrogação, conseguido através do commando `EXPLAIN` com o argumento `ANALYZE`, tem a seguinte constituição:

```
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
Aggregate  (cost=668951.32..668951.33 rows=1 width=32) (actual time=4550.849..4550.849 rows=1 loops=1)
-> Hash Join  (cost=274636.42..668818.19 rows=53248 width=3) (actual time=1127.414..4541.094 rows=53453 loops=1)
     Hash Cond: (order_line.ol_i_id = item.i_id)
     Join Filter: ((order_line.ol_quantity)::numeric < (avg(order_line_1.ol_quantity)))
     Rows Removed by Join Filter: 170628
     -> Seq Scan on order_line  (cost=0.00..353327.83 rows=15563083 width=11) (actual time=0.017..1614.488 rows=15563120 loops=1)
     -> Hash  (cost=274623.79..274623.79 rows=1010 width=36) (actual time=1127.298..1127.298 rows=1439 loops=1)
          Buckets: 2048 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 78kB
          -> Finalize GroupAggregate  (cost=1005.93..274613.69 rows=1010 width=36) (actual time=8.653..1126.023 rows=1439 loops=1)
               Group Key: item.i_id
               -> Gather Merge  (cost=1005.93..274596.02 rows=1010 width=36) (actual time=6.933..1122.264 rows=1439 loops=1)
                    Workers Planned: 1
                    Workers Launched: 1
                    -> Partial GroupAggregate  (cost=5.92..273482.38 rows=1010 width=36) (actual time=1.851..930.453 rows=720 loops=2)
                       Group Key: item.i_id
                       -> Nested Loop  (cost=5.92..273009.97 rows=92463 width=8) (actual time=0.143..907.752 rows=112040 loops=2)
                           ->  Parallel Index Scan using pk_item on item  (cost=0.29..3881.59 rows=594 width=4) (actual time=0.042..21.624 rows=720 loops=2)
```

```
                              Filter: (i_data ~~ 'b%'::text)
                              Rows Removed by Filter: 49280
                        ->  Bitmap Heap Scan on order_line order_line_1 (cost=5.63..451.50 rows=158 width=8) (actual time=0.056..1.186 rows=156 loops=1439)
                              Recheck Cond: (ol_i_id = item.i_id)
                              Heap Blocks: exact=86554
                              ->  Bitmap Index Scan on ix_order_line  (cost=0.00..5.59 rows=158 width=0) (actual time=0.031..0.031 rows=156 loops=1439)
                                    Index Cond: (ol_i_id = item.i_id)
Planning time: 0.486 ms
Execution time: 4551.065 ms
(26 rows)
```

Primeiramente, uma das otimizações que se considerou foi a remoção da declaração `LIKE`, visto que a sua eficiência depende muito da condição que é usada. Assim sendo, a condição `i_data like 'b%'`, que fazia *match* com todos os produtos cujo nome começa-se por 'b', passou a ser `LEFT(i_data, 1) = 'b'`, que verifica a igualdade entre a primeira letra do conteúdo da variável `i_data` e o 'b'. Os resultados foram favoráveis, uma vez que o tempo de execução passou a ser: 1556.688 ms. No entanto, o novo de plano de execução mudou bastante.

```
                                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------
Aggregate  (cost=426918.44..426918.45 rows=1 width=32) (actual time=1556.520..1556.520 rows=1 loops=1)
-> Nested Loop  (cost=1011.58..426852.29 rows=26458 width=3) (actual time=21.002..1542.386 rows=53453 loops=1)
    -> Finalize GroupAggregate  (cost=1005.95..161553.17 rows=500 width=36) (actual time=7.671..181.292 rows=1439 loops=1)
         Group Key: item.i_id
         ->  Gather Merge  (cost=1005.95..161544.42 rows=500 width=36) (actual time=6.252..176.812 rows=1439 loops=1)
               Workers Planned: 1
               Workers Launched: 1
               ->  Partial GroupAggregate  (cost=5.94..160488.16 rows=500 width=36) (actual time=1.477..829.664 rows=720 loops=2)
                     Group Key: item.i_id
                     ->  Nested Loop  (cost=5.94..160254.35 rows=45762 width=8) (actual time=0.138..806.199 rows=112040 loops=2)
                          ->  Parallel Index Scan using pk_item on item  (cost=0.29..4175.70 rows=294 width=4) (actual time=0.063..55.905 rows=720 loops=2)
                                Filter: ("left"((i_data)::text, 1) = 'b'::text)
                                Rows Removed by Filter: 49280
                          ->  Bitmap Heap Scan on order_line order_line_1  (cost=5.65..529.29 rows=159 width=8) (actual time=0.051..0.997 rows=156 loops=1439)
                                Recheck Cond: (ol_i_id = item.i_id)
                                Heap Blocks: exact=23094
                                ->  Bitmap Index Scan on ix_order_line  (cost=0.00..5.61 rows=159 width=0) (actual time=0.027..0.027 rows=156 loops=1439)
                                      Index Cond: (ol_i_id = item.i_id)
    -> Bitmap Heap Scan on order_line  (cost=5.62..530.06 rows=53 width=11) (actual time=0.714..0.936 rows=37 loops=1439)
         Recheck Cond: (ol_i_id = item.i_id)
         Filter: ((ol_quantity)::numeric < (avg(order_line_1.ol_quantity)))
         Rows Removed by Filter: 119
         Heap Blocks: exact=223983
         ->  Bitmap Index Scan on ix_order_line  (cost=0.00..5.61 rows=159 width=0) (actual time=0.025..0.025 rows=156 loops=1439)
               Index Cond: (ol_i_id = item.i_id)
Planning time: 0.501 ms
Execution time: 1556.688 ms
(27 rows)
```

Deste modo, apesar na melhoria do tempo de execução geral, o mesmo não se verifica para a declaração onde o `LIKE` foi substituído. Efetivamente, o que dantes demorava 21.624 ms passou a demorar 55.905 ms. Isto deve-se ao facto do predicado usado na declaração `LIKE` ter um *wildcard* no final do padrão, sítio onde não é tão problemático e, consequentemente, o `LIKE` consegue ser bastante eficiente. Mais ainda, a alternativa com recurso à função `LEFT` tem a desvantagem de invocar uma função numa clausula `WHERE` que é sempre desaconselhado.

Ainda assim, o tempo de execução melhorou e não foi resultante da mudança efetuada. Desta forma, resta perceber o que mais mudou no plano de execução de forma a justificar a melhoria. Uma das principais diferenças advém do cálculo da equação `ol_quantity < t.a`. No ínicio era efetuada como filtro num `hash join` que devido às condições/predicados que tinha que validar, era a operação mais pesarosa (`actual time=1127.414..4541.094`). No entanto, desta vez apesar de ser utilizado um `nested loop` (de evitar), este não tem condição nenhuma e como tal revela-se bastante rápido (`actual time=21.002..1542.386`). Mais ainda, a equação lógica surge como um filtro de um `bitmap heap scan` (`actual time=0.714..0.936`) que é executado em paralelo com as restantes operações, e o seu custo é quase nulo e como tal não influencia o tempo final da interrogação, uma vez que o outro fio de execução contempla tempos superiores (`actual time=7.671..181.292`).

Desta forma, com base na informação adquirida pela análise do plano de execução percebeu-se que era necessário favorecer o acesso aleatório às páginas com os dados, uma vez que isso refletia-se numa maior adoção por parte do PostgreSQL a mecanismos como os índices. Para tal, foi executado o comando `SET random_page_cost = 1.5`, seguido de um `VACUUM ANALYZE` de forma a que o PostgreSQL recalculasse as métricas de escolha dos planos de execução. Dado que a interrogação com a declaração `LIKE` teve tempos melhores, decidiu-se reverter a *query* para que ficasse igual à usada na medição de base, ou seja:

```
select sum(ol_amount) / 2.0 as avg_yearly
from order_line,
     (select i_id, avg(ol_quantity) as a
      from item, order_line
      where ol_i_id = i_id and i_data like 'b%'
      group by i_id) t
where ol_i_id = t.i_id and ol_quantity < t.a;
```

O resultado de executar a *query* foi o seguinte: 1011.842 ms.

```
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
Aggregate  (cost=290586.41..290586.43 rows=1 width=32) (actual time=1010.631..1010.631 rows=1 loops=1)
  ->  Nested Loop  (cost=1001.17..290450.34 rows=54428 width=3) (actual time=16.878..1003.554 rows=53453 loops=1)
        ->  Finalize GroupAggregate  (cost=1000.74..110676.07 rows=1010 width=36) (actual time=8.672..739.654 rows=1439 loops=1)
              Group Key: item.i_id
              ->  Gather Merge  (cost=1000.74..110658.40 rows=1010 width=36) (actual time=7.667..738.304 rows=1439 loops=1)
                    Workers Planned: 1
                    Workers Launched: 1
                    ->  Partial GroupAggregate  (cost=0.73..109544.76 rows=1010 width=36) (actual time=1.435..764.598 rows=720 loops=2)
                          Group Key: item.i_id
                          ->  Nested Loop  (cost=0.73..109072.15 rows=92502 width=8) (actual time=0.088..750.838 rows=112040 loops=2)
                                ->  Parallel Index Scan using pk_item on item  (cost=0.29..3189.09 rows=594 width=4)
                                                                               (actual time=0.037..15.256 rows=720 loops=2)
                                      Filter: (i_data ~~ 'b%'::text)
                                      Rows Removed by Filter: 49280
                                ->  Index Scan using ix_order_line on order_line order_line_1  (cost=0.43..176.63 rows=162 width=8)
                                                                                               (actual time=0.020..0.997 rows=156 loops=1439)
                                      Index Cond: (ol_i_id = item.i_id)
        ->  Index Scan using ix_order_line on order_line  (cost=0.43..177.44 rows=54 width=11) (actual time=0.107..0.179 rows=37 loops=1439)
              Index Cond: (ol_i_id = item.i_id)
              Filter: ((ol_quantity)::numeric < (avg(order_line_1.ol_quantity)))
              Rows Removed by Filter: 119
Planning time: 0.423 ms
Execution time: 1011.842 ms
(21 rows)
```

De facto, é possível verificar uma melhoria significativa em relação aos tempos conseguidos na medição de base, passando de 4551.065 ms para 1011.842 ms.

Após uma análise dos plano de execução atual percebeu-se que o fio de execução que termina com um `Finalize GroupAggregate` era o responsável pela era o responsável pela maioria do tempo de execução da interrogação. Assim sendo, ao analisar essa porção do plano percebeu-se que esta era responsável por realizar o cálculo correspondente à *subquery* presente na declaração `FROM`, ou seja:

```
SELECT i_id, AVG(ol_quantity) AS a
FROM item, order_line
WHERE ol_i_id = i_id AND i_data LIKE 'b%'
GROUP BY i_id
```

Desta forma, com vista a otimizar esta secção da interrogação principal decidiu-se utilizar uma vista materializada. De facto, esta é uma estratégia viável uma vez que as interrogações analíticas são realizadas esporadicamente e o seu cálculo não necessita que as informações se encontrem totalmente atualizadas, isto é, pode existir um pequeno atraso na atualização dos dados da vista que não afeta significativamente o resultado produzido. Consequentemente, a atualização da própria vista pode ser realizada em momentos de baixo tráfego de forma a minimizar o impacto das restantes operações. Na prática, realizou-se o seguinte comando para efetuar a criação da *view*:

```
CREATE MATERIALIZED VIEW mv_i_avg AS
   SELECT i_id, AVG(ol_quantity) AS a
   FROM item, order_line
   WHERE ol_i_id = i_id AND i_data LIKE 'b%'
   GROUP BY i_id
WITH NO DATA;
```

De seguida, foi necessário popular a vista. O comando que realiza esta ação é o `REFRESH MATERIALIZED VIEW mv_i_avg;`. Este deve ser repetido sempre que se quiser atualizar a vista e deve ser feito manualmente ou através de um mecanismos do tipo *trigger*. Os resultados obtidos foram os sequintes:

```
EXPLAIN ANALYZE SELECT SUM(ol_amount) / 2.0 AS avg_yearly
FROM order_line, mv_i_avg AS t
WHERE ol_i_id = t.i_id AND ol_quantity < t.a;
```

```
                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
Aggregate  (cost=224448.61..224448.62 rows=1 width=32) (actual time=1143.692..1143.692 rows=1 loops=1)
-> Nested Loop  (cost=0.43..224260.26 rows=75337 width=3) (actual time=9.282..1136.063 rows=53453 loops=1)
   -> Seq Scan on mv_i_avg t  (cost=0.00..23.39 rows=1439 width=11) (actual time=0.011..0.625 rows=1439 loops=1)
   -> Index Scan using ix_order_line on order_line  (cost=0.43..155.31 rows=52 width=11)
                                                    (actual time=0.467..0.784 rows=37 loops=1439)
         Index Cond: (ol_i_id = t.i_id)
         Filter: ((ol_quantity)::numeric < t.a)
         Rows Removed by Filter: 119
Planning time: 0.261 ms
Execution time: 1143.730 ms
(9 rows)
```

Contudo, ao contrário do que seria de esperar o tempo de execução não melhorou. Após alguma análise através do módulo `pg_stat_statements`, percebeu-se que a percentagem de *hits* de leitura na cache se encontrava bastante baixa, cerca de 9%.

```
SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit /
       nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;
```

```
                          query                            | calls | total_time | rows |    hit_percent
-----------------------------------------------------------+-------+------------+------+--------------------
 explain analyze select sum(ol_amount) / 2.0 as avg_yearly+|     1 | 1144.09285 |    0 | 9.0668249527542216
 from order_line, mv_i_avg as t                           +|       |            |      |
 where ol_i_id = t.i_id and ol_quantity < t.a              |       |            |      |
(2 rows)
```

Assim sendo, aumentou-se o tamanho do `shared_buffers` para 2GB de forma a que os dados coubessem todos em memória e assim tirar um maior partido da vista materializada. Após a mudança reiniciou-se o servidor e os resultados obtidos foram os seguintes:

```
explain analyze select sum(ol_amount) / 2.0 as avg_yearly
from order_line, mv_i_avg as t
where ol_i_id = t.i_id and ol_quantity < t.a;
```

```
                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
Aggregate  (cost=221704.96..221704.98 rows=1 width=32) (actual time=280.693..280.693 rows=1 loops=1)
-> Nested Loop  (cost=0.43..221519.99 rows=73989 width=3) (actual time=1.846..273.560 rows=53453 loops=1)
   -> Seq Scan on mv_i_avg t  (cost=0.00..23.39 rows=1439 width=11) (actual time=0.006..0.203 rows=1439 loops=1)
   -> Index Scan using ix_order_line on order_line  (cost=0.43..153.41 rows=51 width=11)
                                                    (actual time=0.113..0.186 rows=37 loops=1439)
          Index Cond: (ol_i_id = t.i_id)
          Filter: ((ol_quantity)::numeric < t.a)
          Rows Removed by Filter: 119
Planning time: 0.221 ms
Execution time: 280.734 ms
(9 rows)
```

Mais ainda, a melhoria significativa de 1143.730 ms para 280.734 ms vem acompanhada de uma mudança por completo na percentagem de *hits* na cache, passando de 9% para 100%.

```
SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit /
       nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;
```

```
                          query                           | calls | total_time | rows |     hit_percent
----------------------------------------------------------+-------+------------+------+----------------------
 explain analyze select sum(ol_amount) / 2.0 as avg_yearly+|     1 | 281.052345 |    0 | 100.0000000000000000
 from order_line, mv_i_avg as t                           +|       |            |      |
 where ol_i_id = t.i_id and ol_quantity < t.a               |       |            |      |
```

Concluindo, após algumas mudanças, nomeadamente, através da prioritização do uso de índices com a diminuição do custo do `random_page_cost`, usando vistas materializadas e um aumento do `shared_buffers` e consequente aumento da percentagem de *hits* na cache, conseguiu-se realizar melhorias consideráveis no tempo de execução da interrogação A1. De facto, passou-se de um tempo inicial de **4551.065 ms** para **280.734 ms**.

Query	**Tempo de execução em ms (com cache)**
Base	4551.065
Substituição do LIKE	1556.688
Alteração do random_page_cost	1011.842
Com vista materializada	1143.730
Aumento do shared_buffer	280.734

Tabela 1: Tabela resumo dos tempos de execução.

3.2.2 A2

A interrogação analítica tem como objetivo apresentar o número de clientes agrupados e ordenados segundo o número de encomendas efetuadas por estes. Este resultado representa então o conjunto das relações entre os clientes e o tamanho das suas encomendas é ordenado então pelo tamanho das encomendas contando assim quantos clientes efetuaram negócios semelhantes.

Apresenta-se então de seguida a interrogação *SQL* e um excerto de um possível resultado.

Interrogação A2.

```
select c_count, count(*) as custdist
from (select c_id, count(o_id)
         from customer left outer join orders
         on (
                c_w_id = o_w_id
                and c_d_id = o_d_id
                and c_id = o_c_id
                and o_carrier_id > 8)
         group by c_id) as c_orders (c_id, c_count)
group by c_count
order by custdist desc, c_count desc;
```

Excerto do resultado obtido.

```
 c_count | custdist
---------+----------
       0 |      900
     122 |       99
     116 |       93
     120 |       92
```

```
     123 |        83
     119 |        82
     118 |        81
     124 |        80
     117 |        79
     125 |        78
     126 |        76
     115 |        76
     112 |        76
     121 |        69
     113 |        69
     114 |        67
     129 |        65
     111 |        61
     128 |        58
     130 |        56
     127 |        53
     110 |        51
```

Assim, as medições base foram as seguintes:

- Sem cache (primeira execução): 2103.162 ms
- Com cache (após várias execuções): 1877.166 ms

Recorrendo ao comando `EXPLAIN` disponibilizado pelo PostgreSQL, é possível observar com detalhe a estratégia de execução utilizada pelo motor de bases de dados para a resolução da interrogação. Adicionalmente, foi adicionado um argumento `ANALYZE` possibilitando assim a análise dos tempos de execução da interrogação.

Apresenta-se então o plano de execução:

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=239985.26..239985.76 rows=200 width=16) (actual time=1877.077..1877.080 rows=59 loops=1)
   Sort Key: (count(*)) DESC, c_orders.c_count DESC
   Sort Method: quicksort  Memory: 27kB
   ->  GroupAggregate  (cost=239953.12..239977.62 rows=200 width=16) (actual time=1876.616..1877.051 rows=59 loops=1)
         Group Key: c_orders.c_count
         ->  Sort  (cost=239953.12..239960.62 rows=3000 width=8) (actual time=1876.593..1876.746 rows=3000 loops=1)
               Sort Key: c_orders.c_count DESC
               Sort Method: quicksort  Memory: 237kB
               ->  Subquery Scan on c_orders  (cost=239719.85..239779.85 rows=3000 width=8) (actual time=1875.509..1876.186 rows=3000 loops=1)
                     ->  HashAggregate  (cost=239719.85..239749.85 rows=3000 width=12) (actual time=1875.507..1875.916 rows=3000 loops=1)
                           Group Key: customer.c_id
                           ->  Hash Right Join  (cost=178290.00..230719.85 rows=1800000 width=8) (actual time=1022.305..1560.369 rows=1800000 loops=1)
                                 Hash Cond: ((orders.o_w_id = customer.c_w_id) AND (orders.o_d_id = customer.c_d_id) AND (orders.o_c_id = customer.c_id))
                                 ->  Seq Scan on orders  (cost=0.00..39323.00 rows=244680 width=16) (actual time=149.881..198.785 rows=251809 loops=1)
                                       Filter: (o_carrier_id > 8)
                                       Rows Removed by Filter: 1548191
                                 ->  Hash  (cost=138000.00..138000.00 rows=1800000 width=12) (actual time=871.993..871.994 rows=1800000 loops=1)
                                       Buckets: 131072  Batches: 32  Memory Usage: 3457kB
                                       ->  Seq Scan on customer  (cost=0.00..138000.00 rows=1800000 width=12) (actual time=0.049..529.138 rows=1800000
                                                                  loops=1)
 Planning time: 0.445 ms
 Execution time: 1877.166 ms
(21 rows)
```

De seguida, analisou-se a informação fornecida pelo comando `EXPLAIN` no que toca à construção da `hash`. Deste modo, observou-se que o número de *batches* se encontrava bastante elevado, o que é indicativo de que a memória `work_mem` (memória de trabalho) utilizada para a execução de operações por exemplo de ordenação ou construção de tabelas de `hash`, é insuficiente para a operação a realizar.

Como resultado desta observação e com base no número de *batches*, estimou-se então que o tamanho do `work_mem` deveria de passar de 4MB (*default* do postgreSQL) para 128MB, permitindo assim que a `hash` seja construida de uma só vez.

Apresenta-se então de seguida o novo plano de execução.

```
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
Sort  (cost=121458.11..121458.61 rows=200 width=16) (actual time=1645.589..1645.592 rows=59 loops=1)
   Sort Key: (count(*)) DESC, c_orders.c_count DESC
   Sort Method: quicksort  Memory: 27kB
   ->  GroupAggregate  (cost=121425.97..121450.47 rows=200 width=16) (actual time=1645.078..1645.571 rows=59 loops=1)
         Group Key: c_orders.c_count
         ->  Sort  (cost=121425.97..121433.47 rows=3000 width=8) (actual time=1645.042..1645.242 rows=3000 loops=1)
               Sort Key: c_orders.c_count DESC
               Sort Method: quicksort  Memory: 237kB
               ->  Subquery Scan on c_orders  (cost=121147.70..121252.70 rows=3000 width=8) (actual time=1640.739..1644.499 rows=3000 loops=1)
                     ->  Finalize GroupAggregate  (cost=121147.70..121222.70 rows=3000 width=12) (actual time=1640.738..1644.072 rows=3000 loops=1)
                           Group Key: customer.c_id
                           ->  Sort  (cost=121147.70..121162.70 rows=6000 width=12) (actual time=1640.730..1641.944 rows=9000 loops=1)
                                 Sort Key: customer.c_id
                                 Sort Method: quicksort  Memory: 806kB
                                 ->  Gather  (cost=120141.18..120771.18 rows=6000 width=12) (actual time=1630.155..1637.707 rows=9000 loops=1)
                                       Workers Planned: 2
                                       Workers Launched: 2
                                       ->  Partial HashAggregate  (cost=119141.18..119171.18 rows=3000 width=12) (actual time=1624.595..1625.324
                                       rows=3000
                                                            loops=3)
                                             Group Key: customer.c_id
                                             ->  Merge Left Join  (cost=61960.77..115391.25 rows=749987 width=8) (actual time=783.320..1332.249 rows=600000
                                                            loops=3)
                                                   Merge Cond: ((customer.c_w_id = orders.o_w_id) AND (customer.c_d_id = orders.o_d_id)
                                                                 AND (customer.c_id = orders.o_c_id))
                                                   ->  Parallel Index Only Scan using pk_customer on customer  (cost=0.43..44232.13 rows=749987 width=12)
                                                   (actual time=0.031..186.590 rows=600000 loops=3)
                                                         Heap Fetches: 0
                                                   ->  Sort  (cost=61960.34..62591.09 rows=252300 width=16) (actual time=783.266..812.002 rows=251757 loops=3)
                                                         Sort Key: orders.o_w_id, orders.o_d_id, orders.o_c_id
                                                         Sort Method: quicksort  Memory: 17948kB
                                                         ->  Seq Scan on orders  (cost=0.00..39323.00 rows=252300 width=16) (actual time=0.033..557.902
                                                                                 rows=251809 loops=3)
                                                               Filter: (o_carrier_id > 8)
                                                               Rows Removed by Filter: 1548191
 Planning time: 0.438 ms
 Execution time: 1645.747 ms
(31 rows)
```

Observa-se então que com o incremento na "memória de trabalho" do PostgreSQL, este calcula imediatamente um plano de execução bastante diferente, o que é indicativo de que esta mudança deu uma maior liberdade de otimização recorrendo agora a várias operações paralelas.

Uma vez que no passo anterior foram tomadas decisões no que toca ao tamanho do *buffer* `work_mem`, procedeu-se também à verificação de se o tamanho do *buffer* `shared_buffers` seria o mais indicado.

Através dos comandos `select pg_stat_statements_reset()` e `SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit / nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;` é possível ter uma visão da *performance* da *cache* do sistema. Com o primeiro comando, é possível efetuar o *reset* desta estrutura de dados por forma a isolar a operação a ser analisada. O ultimo comando, permite visualizar os dados recolhidos sobre o comportamento da *query* no que toca à *cache*.

Assim, com o valor pre-estabelecido pelo PostgreSQL para o *shared_buffers* (128MB), obtem-se o seguinte resultado.

```
                         query                          | calls | total_time  | rows |     hit_percent
--------------------------------------------------------+-------+-------------+------+---------------------
 explain analyse select   c_count, count(*) as custdist+|     1 | 1646.371527 |    0 | 18.6397853209730257
 from                                                  +|       |             |      |
     (select c_id, count(o_id)                         +|       |             |      |
 from customer left outer join orders on (             +|       |             |      |
    c_w_id = o_w_id                                    +|       |             |      |
    and c_d_id = o_d_id                                +|       |             |      |
    and c_id = o_c_id                                  +|       |             |      |
    and o_carrier_id > 8)                              +|       |             |      |
      group by c_id) as c_orders (c_id, c_count)       +|       |             |      |
 group by c_count                                      +|       |             |      |
 order by custdist desc, c_count desc                   |       |             |      |
 select pg_stat_statements_reset()                      |     1 |    0.106665 |    1 |
(2 rows)
```

Como se pode observar, a percentagem de *hits* na *cache* encontra-se nos 18 porcento, o que é indicativo que o espaço alocado para `shared_buffers` não é o indicado.

Deste modo, procedeu-se então a uma série de testes para verificar qual o melhor valor a utilizar para o `shared_buffers` tendo sempre em conta a percentagem de *hits* na *cache*. Assim, com um *shared_buffers* de tamanho 800MB, obtêm-se os seguintes resultados.

```
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=121418.69..121419.19 rows=200 width=16) (actual time=1212.335..1212.340 rows=59 loops=1)
```

```
  Sort Key: (count(*)) DESC, c_orders.c_count DESC
  Sort Method: quicksort  Memory: 27kB
  ->  GroupAggregate  (cost=121386.55..121411.05 rows=200 width=16) (actual time=1211.823..1212.296 rows=59 loops=1)
        Group Key: c_orders.c_count
        ->  Sort  (cost=121386.55..121394.05 rows=3000 width=8) (actual time=1211.817..1212.000 rows=3000 loops=1)
              Sort Key: c_orders.c_count DESC
              Sort Method: quicksort  Memory: 237kB
              ->  Subquery Scan on c_orders  (cost=121108.29..121213.29 rows=3000 width=8) (actual time=1208.571..1211.388 rows=3000 loops=1)
                    ->  Finalize GroupAggregate  (cost=121108.29..121183.29 rows=3000 width=12) (actual time=1208.570..1211.113 rows=3000 loops=1)
                          Group Key: customer.c_id
                          ->  Sort  (cost=121108.29..121123.29 rows=6000 width=12) (actual time=1208.563..1209.501 rows=9000 loops=1)
                                Sort Key: customer.c_id
                                Sort Method: quicksort  Memory: 806kB
                                ->  Gather  (cost=120101.76..120731.76 rows=6000 width=12) (actual time=1196.809..1206.607 rows=9000 loops=1)
                                      Workers Planned: 2
                                      Workers Launched: 2
                                      ->  Partial HashAggregate  (cost=119101.76..119131.76 rows=3000 width=12) (actual time=1187.747..1188.146 rows=3000
                                                                  loops=3)
                                            Group Key: customer.c_id
                                            ->  Merge Left Join  (cost=61925.87..115351.75 rows=750002 width=8) (actual time=662.965..1002.909 rows=600000
                                                                  loops=3)
                                                  Merge Cond: ((customer.c_w_id = orders.o_w_id) AND (customer.c_d_id = orders.o_d_id)
                                                                AND (customer.c_id = orders.o_c_id))
                                                  ->  Parallel Index Only Scan using pk_customer on customer  (cost=0.43..44232.46 rows=750002 width=12)
                                                  (actual time=0.042..122.921 rows=600000 loops=3)
                                                        Heap Fetches: 0
                                                  ->  Sort  (cost=61925.45..62555.30 rows=251940 width=16) (actual time=662.900..682.868 rows=251603 loops=3)
                                                        Sort Key: orders.o_w_id, orders.o_d_id, orders.o_c_id
                                                        Sort Method: quicksort  Memory: 17948kB
                                                        ->  Seq Scan on orders  (cost=0.00..39323.00 rows=251940 width=16) (actual time=0.015..549.558
                                                                                  rows=251809 loops=3)
                                                              Filter: (o_carrier_id > 8)
                                                              Rows Removed by Filter: 1548191
 Planning time: 0.467 ms
 Execution time: 1212.540 ms
(31 rows)
```

Com este novo valor, verificou-se uma melhoria no tempo de execução bem como a percentagem de *hit rate* passou a ser 100 porcento.

```
                         query                           | calls | total_time  | rows |      hit_percent
---------------------------------------------------------+-------+-------------+------+----------------------
 explain analyse select   c_count, count(*) as custdist+|     1 | 1213.222642 |    0 | 100.0000000000000000
 from                                                   +|       |             |      |
     (select c_id, count(o_id)                          +|       |             |      |
 from customer left outer join orders on (              +|       |             |      |
    c_w_id = o_w_id                                     +|       |             |      |
    and c_d_id = o_d_id                                 +|       |             |      |
    and c_id = o_c_id                                   +|       |             |      |
    and o_carrier_id > 8)                               +|       |             |      |
      group by c_id) as c_orders (c_id, c_count)        +|       |             |      |
 group by c_count                                       +|       |             |      |
 order by custdist desc, c_count desc                    |       |             |      |
 select pg_stat_statements_reset()                       |     1 |    0.084229 |    1 |
(2 rows)
```

De seguida, através de uma observação mais em detalhe do plano de execução atual, este ainda se encontra a determinar como método mais eficiente um `seq scan` no contexto de uma condição de comparação do tipo

$$x > y$$

. Assim, a utilização de um índice seria a opção a adotar para obter uma melhor *performance*. Adicionalmente, recorreu-se também à opção `CLUSTER` por forma a tirar o melhor partido deste índice uma vez que a maioria das linhas é eliminada pela condição de comparação.

Este *clustered index* foi criado com os seguintes comandos.

```
create index idx_o_carrier_id on orders(o_carrier_id);

cluster orders using idx_o_carrier_id;
```

Apresenta-se então o novo plano de execução e relatório de *hit rate* na *cache*.

```
                                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=91568.58..91569.08 rows=200 width=16) (actual time=879.909..879.912 rows=59 loops=1)
   Sort Key: (count(*)) DESC, c_orders.c_count DESC
   Sort Method: quicksort  Memory: 27kB
   ->  GroupAggregate  (cost=91536.44..91560.94 rows=200 width=16) (actual time=879.408..879.888 rows=59 loops=1)
         Group Key: c_orders.c_count
         ->  Sort  (cost=91536.44..91543.94 rows=3000 width=8) (actual time=879.404..879.599 rows=3000 loops=1)
               Sort Key: c_orders.c_count DESC
               Sort Method: quicksort  Memory: 237kB
               ->  Subquery Scan on c_orders  (cost=91258.18..91363.18 rows=3000 width=8) (actual time=875.080..878.822 rows=3000 loops=1)
                     ->  Finalize GroupAggregate  (cost=91258.18..91333.18 rows=3000 width=12) (actual time=875.078..878.445 rows=3000 loops=1)
                           Group Key: customer.c_id
                           ->  Sort  (cost=91258.18..91273.18 rows=6000 width=12) (actual time=875.071..876.115 rows=9000 loops=1)
                                 Sort Key: customer.c_id
```

```
                                Sort Method: quicksort  Memory: 806kB
                                ->  Gather  (cost=90251.66..90881.66 rows=6000 width=12) (actual time=860.695..873.078 rows=9000 loops=1)
                                      Workers Planned: 2
                                      Workers Launched: 2
                                      ->  Partial HashAggregate  (cost=89251.66..89281.66 rows=3000 width=12) (actual time=852.836..853.328 rows=3000
                                                                  loops=3)
                                            Group Key: customer.c_id
                                            ->  Merge Left Join  (cost=32082.19..85501.63 rows=750005 width=8) (actual time=372.495..709.805 rows=600000
                                                                  loops=3)
                                                  Merge Cond: ((customer.c_w_id = orders.o_w_id) AND (customer.c_d_id = orders.o_d_id)
                                                               AND (customer.c_id = orders.o_c_id))
                                                  ->  Parallel Index Only Scan using pk_customer on customer  (cost=0.43..44232.54 rows=750005 width=12)
                                                  (actual time=0.034..145.378 rows=600000 loops=3)
                                                        Heap Fetches: 0
                                                  ->  Sort  (cost=32081.77..32710.57 rows=251520 width=16) (actual time=372.442..393.478 rows=251400 loops=3)
                                                        Sort Key: orders.o_w_id, orders.o_d_id, orders.o_c_id
                                                        Sort Method: quicksort  Memory: 17948kB
                                                        ->  Index Scan using idx_o_carrier_id on orders  (cost=0.43..9520.03 rows=251520 width=16)
                                                        (actual time=0.045..95.440 rows=251809 loops=3)
                                                              Index Cond: (o_carrier_id > 8)
 Planning time: 0.678 ms
 Execution time: 880.135 ms
(30 rows)
```

```
                        query                         | calls | total_time | rows |     hit_percent
------------------------------------------------------+-------+------------+------+----------------------
 explain analyse select   c_count, count(*) as custdist+|     1 | 881.031668 |    0 | 100.0000000000000000
 from                                                 +|       |            |      |
     (select c_id, count(o_id)                        +|       |            |      |
 from customer left outer join orders on (            +|       |            |      |
    c_w_id = o_w_id                                   +|       |            |      |
    and c_d_id = o_d_id                               +|       |            |      |
    and c_id = o_c_id                                 +|       |            |      |
    and o_carrier_id > 8)                             +|       |            |      |
      group by c_id) as c_orders (c_id, c_count)      +|       |            |      |
 group by c_count                                     +|       |            |      |
 order by custdist desc, c_count desc                  |       |            |      |
 select pg_stat_statements_reset()                     |     1 |   0.176687 |    1 |
(2 rows)
```

Como se pode observar, a utilização do `clustered index` fez com que o `seq scan` já não fosse escolhido como melhor opção para a execução da interrogação. Para além disso, observou-se também uma melhoria significativa na performance bem como o *hit rate* se mantem nos 100 porcento.

Neste novo plano de execução, verificou-se também que o grande peso de execução encontra-se na interrogação aninhada que reune a informação das tabelas *customer* e *orders*. Deste modo, a implementação de uma vista materializada sobre esta seria uma mais valia para o desempenho uma vez que permite que este resultado seja calculado em momentos com um menor número de pedidos à base de dados mas também beneficia a experiência de utilzação sendo assim possível apresentar resultados ao utilizador muito mais rápido.

Para a criação da vista materializada recorreu-se então ao seguinte comando.

```
CREATE MATERIALIZED VIEW customers_orders_view AS
SELECT c_id, count(o_id) AS c_count FROM customer
    LEFT OUTER JOIN orders ON (
        c_w_id = o_w_id
        AND c_d_id = o_d_id
        AND c_id = o_c_id
        AND o_carrier_id > 8)
    GROUP BY c_id
WITH NO DATA;
```

Através do comando `REFRESH` é possível, caso apenas a vista se encontre criada, popular esta com os respetivos dados ou recalcular o resultado a devolver pela vista. Este deverá ser utilizado em momentos de menor carga na base de dados por forma a não prejudicar o desempenho desta uma vez que se trata de uma operação pesada.

Para a integração desta vista na interrogação analítica, foi substituida a interrogação aninhada pelo acesso à vista materializada criada anteriormente. Apresenta-se de seguida a interrogação analítica modificada.

```
select   c_count, count(*) as custdist
from
```

```
    (select * from customers_orders_view) as c_orders (c_id, c_count)
group by c_count
order by custdist desc, c_count desc;
```

Uma vez que, apenas é necessário proceder à ordenação e agrupamento dos resultados devolvidos pela vista materializada, o seu tempo de execução é reduzido para menos de 1 ms. Consegue-se um tempo de execução tão baixo devido à reduzida dimensão do resultado devolvido pela vista materializada, sendo a operação de ordenação bastante rápida também.

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Sort  (cost=65.13..65.32 rows=76 width=16) (actual time=0.895..0.898 rows=76
        loops=1)
   Sort Key: (count(*)) DESC, customers_orders_view.c_count DESC
   Sort Method: quicksort  Memory: 28kB
   ->  HashAggregate  (cost=62.00..62.76 rows=76 width=16)
                      (actual time=0.856..0.867 rows=76 loops=1)
         Group Key: customers_orders_view.c_count
         ->  Seq Scan on customers_orders_view  (cost=0.00..47.00 rows=3000
               width=8) (actual time=0.008..0.228 rows=3000 loops=1)
 Planning time: 0.158 ms
 Execution time: 0.944 ms
(8 rows)
```

Tendo em conta a memória utilizada indicada no plano de execução, não serão necessárias quaisquer alterações no que diz respeito ao tamanho dos buffers *work_mem* e *shared_buffers*.

Query	**Tempo de execução em ms (com cache)**
Base	1877.166
Aumento work_mem	1645.747
Aumento shared_buffers	1212.540
Clustered index para idx_o_carrier_id	880.135
Vista materializada	0.944

Tabela 2: Tabela resumo dos tempos de execução.

3.2.3 A3

A interrogação analíticas A3 tem como objetivo calcular a receita conjunta dos produtos que tenham uma quantidade entre 1 e 10 (inclusive), um preço entre 1 e 400000 (inclusive) e o seu nome, ou começa por 'a' e foi retirado do armazém 1, 2 ou 3, ou começa por 'b' e foi retirado do armazém 1, 2 ou 4, ou começa por 'c' e veio do armazém 1, 3 ou 5. O resultado da execução da interrogação tem como receita 52317037.63 unidades monetárias.

```
select sum(ol_amount) as revenue
from order_line, item
where
(ol_i_id = i_id and    or (ol_i_id = i_id and    or (ol_i_id = i_id and
 i_data like 'a%' and      i_data like 'b%' and      i_data like 'c%' and
 ol_quantity >= 1 and      ol_quantity >= 1 and      ol_quantity >= 1 and
 ol_quantity <= 10 and     ol_quantity <= 10 and     ol_quantity <= 10 and
```

```
i_price between 1 and      i_price between 1 and      i_price between 1 and
400000 and                 400000 and                 400000 and
ol_w_id in (1,2,3))        ol_w_id in (1,2,4))        ol_w_id in (1,5,3));
```

Desta forma, as medições base foram as seguintes: 551.449 ms.

Mais ainda, o plano escolhido pelo PostgreSQL para realizar a interrogação, conseguido através do commando `EXPLAIN` com o argumento `ANALYZE`, tem a seguinte constituição:

```
                                                                          QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------------
Finalize Aggregate  (cost=290513.28..290513.29 rows=1 width=32) (actual time=445.309..445.310 rows=1 loops=1)
->  Gather  (cost=290513.06..290513.27 rows=2 width=32) (actual time=445.285..445.986 rows=3 loops=1)
     Workers Planned: 2
     Workers Launched: 2
     ->  Partial Aggregate  (cost=289513.06..289513.07 rows=1 width=32) (actual time=437.204..437.205 rows=1 loops=3)
          ->  Hash Join  (cost=62254.20..289478.69 rows=13747 width=3) (actual time=157.622..435.298 rows=11638 loops=3)
               Hash Cond: (order_line.ol_i_id = item.i_id)
               Join Filter: (((item.i_data ~~ 'a%'::text) AND (order_line.ol_w_id = ANY ('{1,2,3}'::integer[]))) OR
                             ((item.i_data ~~ 'b%'::text) AND (order_line.ol_w_id = ANY ('{1,2,4}'::integer[]))) OR
                             ((item.i_data ~~ 'c%'::text) AND (order_line.ol_w_id = ANY ('{1,5,3}'::integer[]))))
               Rows Removed by Join Filter: 7715
               ->  Parallel Bitmap Heap Scan on order_line (cost=58415.32..283082.74 rows=974122 width=11) (actual time=116.025..325.968 rows=431943 loops=3)
                    Recheck Cond: ((ol_w_id = ANY ('{1,2,3}'::integer[])) OR
                                   (ol_w_id = ANY ('{1,2,4}'::integer[])) OR
                                   (ol_w_id = ANY ('{1,5,3}'::integer[])))
                    Filter: ((ol_quantity >= 1) AND (ol_quantity <= 10))
                    Heap Blocks: exact=6154
                    ->  BitmapOr  (cost=58415.32..58415.32 rows=2465867 width=0) (actual time=122.526..122.526 rows=0 loops=1)
                          ->  Bitmap Index Scan on pk_order_line  (cost=0.00..19040.53 rows=828697 width=0) (actual time=48.864..48.864 rows=778370 loops=1)
                                Index Cond: (ol_w_id = ANY ('{1,2,3}'::integer[]))
                          ->  Bitmap Index Scan on pk_order_line  (cost=0.00..19076.20 rows=830253 width=0) (actual time=39.165..39.165 rows=778642 loops=1)
                                Index Cond: (ol_w_id = ANY ('{1,2,4}'::integer[]))
                          ->  Bitmap Index Scan on pk_order_line  (cost=0.00..18545.17 rows=806917 width=0) (actual time=34.491..34.491 rows=776681 loops=1)
                                Index Cond: (ol_w_id = ANY ('{1,5,3}'::integer[]))
               ->  Hash  (cost=3789.00..3789.00 rows=3990 width=55) (actual time=41.485..41.485 rows=4477 loops=3)
                    Buckets: 8192 (originally 4096)  Batches: 1 (originally 1)  Memory Usage: 445kB
                    ->  Seq Scan on item  (cost=0.00..3789.00 rows=3990 width=55) (actual time=0.021..40.319 rows=4477 loops=3)
                          Filter: ((i_price >= '1'::numeric) AND
                                   (i_price <= '400000'::numeric) AND
                                   ((i_data ~~ 'a%'::text) OR (i_data ~~ 'b%'::text) OR (i_data ~~ 'c%'::text)))
                          Rows Removed by Filter: 95523
Planning time: 0.643 ms
Execution time: 446.157 ms
(27 rows)
```

Primeiramente, devido ao tamanho da *query* em questão decidiu-se organiza-la de forma a facilitar a sua compreensão, apesar desta alteração não trazer melhoria no que toca ao tempo de execução.

```
select sum(ol_amount) as revenue
from order_line, item
where
 ol_i_id = i_id and
 ol_quantity between 1 and 10 and
 i_price between 1 and 400000 and
 ((
   i_data like 'a%' and
   ol_w_id in (1,2,3)
  ) or (
   i_data like 'b%' and
   ol_w_id in (1,2,4)
  ) or (
   i_data like 'c%' and
   ol_w_id in (1,5,3)
 ));
```

De seguida, analisou-se o plano de execução escolhido pelo PostgreSQL. De facto, este encontra-se bastante otimizado uma vez que dá prioridade aos índices, usa `hash join`, paraleliza a execução e usa `bitmap` quando pretende varrer um número considerável de tuplos de um tabela, em vez de usar *scan* sequencial ou por índices. No entanto, ao analisar o fio de execução do `parallel bitmap heap scan` verificou-se que algo podia ser melhorado.

```
->  Parallel Bitmap Heap Scan on order_line (cost=58415.32..283082.74 rows=974122 width=11) (actual time=116.025..325.968 rows=431943 loops=3)
      Recheck Cond: ((ol_w_id = ANY ('{1,2,3}'::integer[])) OR
                     (ol_w_id = ANY ('{1,2,4}'::integer[])) OR
                     (ol_w_id = ANY ('{1,5,3}'::integer[])))
      Filter: ((ol_quantity >= 1) AND (ol_quantity <= 10))
      Heap Blocks: exact=6154
      ->  BitmapOr  (cost=58415.32..58415.32 rows=2465867 width=0) (actual time=122.526..122.526 rows=0 loops=1)
            ->  Bitmap Index Scan on pk_order_line  (cost=0.00..19040.53 rows=828697 width=0) (actual time=48.864..48.864 rows=778370 loops=1)
                  Index Cond: (ol_w_id = ANY ('{1,2,3}'::integer[]))
            ->  Bitmap Index Scan on pk_order_line  (cost=0.00..19076.20 rows=830253 width=0) (actual time=39.165..39.165 rows=778642 loops=1)
                  Index Cond: (ol_w_id = ANY ('{1,2,4}'::integer[]))
            ->  Bitmap Index Scan on pk_order_line  (cost=0.00..18545.17 rows=806917 width=0) (actual time=34.491..34.491 rows=776681 loops=1)
                  Index Cond: (ol_w_id = ANY ('{1,5,3}'::integer[]))
```

Assim sendo, esta porção do plano de execução está responsável por filtrar todas as `order_line` que sejam do armazém 1, 2, 3, 4 ou 5. Contudo, está a fazê-lo de forma bastante ineficiente, pois a lógica usada envolve que o `ol_w_id` pertença a {1,2,3}, a {1,2,4} ou a {1,5,3}. Mais ainda, o resultado é exatamente igual a pertencer a {1,2,3,4,5} e esta condição já permite fazer a filtragem de uma vez, ao contrário da implementada. Esta por sua vez obriga a fazer 3 `bitmap index scan` diferentes e, posteriormente, a conjugar o resultado. Para implementar a nova visão tem de se criar um índice no `ol_w_id` de forma a que o PostgreSQL consiga percorrer a coluna da forma mais eficiente possível, ou seja, usando um `index scan`.

```
explain analyze select sum(ol_amount) as revenue
from order_line, item
where
   ol_i_id = i_id and
   ol_w_id in (1,2,3,4,5) and
   ol_quantity between 1 and 10 and
   i_price between 1 and 400000 and
   ((
      i_data like 'a%' and
      ol_w_id in (1,2,3)
   ) or (
      i_data like 'b%' and
      ol_w_id in (1,2,4)
   ) or (
      i_data like 'c%' and
      ol_w_id in (1,5,3)
   ));
```

```
                                                                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
Finalize Aggregate  (cost=116411.23..116411.24 rows=1 width=32) (actual time=372.763..372.763 rows=1 loops=1)
  ->  Gather  (cost=116411.01..116411.22 rows=2 width=32) (actual time=371.975..376.849 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate  (cost=115411.01..115411.02 rows=1 width=32) (actual time=366.121..366.121 rows=1 loops=3)
              ->  Hash Join  (cost=3851.55..115407.41 rows=1439 width=3) (actual time=60.768..364.040 rows=11638 loops=3)
                    Hash Cond: (order_line.ol_i_id = item.i_id)
                    Join Filter: (((item.i_data ~~ 'a%'::text) AND (order_line.ol_w_id = ANY ('{1,2,3}'::integer[]))) OR
                                 ((item.i_data ~~ 'b%'::text) AND (order_line.ol_w_id = ANY ('{1,2,4}'::integer[]))) OR
                                 ((item.i_data ~~ 'c%'::text) AND (order_line.ol_w_id = ANY ('{1,5,3}'::integer[]))))
                    Rows Removed by Join Filter: 7715
                    ->  Parallel Index Scan using ix_ol_w_id on order_line (cost=0.43..111343.08 rows=81226 width=11)
                                                                            (actual time=0.062..234.153 rows=431943 loops=3)
                          Index Cond: (ol_w_id = ANY ('{1,2,3,4,5}'::integer[]))
                          Filter: ((ol_quantity >= 1) AND (ol_quantity <= 10))
                    ->  Hash  (cost=3789.00..3789.00 rows=4969 width=55) (actual time=60.570..60.571 rows=4477 loops=3)
                          Buckets: 8192  Batches: 1  Memory Usage: 445kB
                          ->  Seq Scan on item  (cost=0.00..3789.00 rows=4969 width=55) (actual time=0.030..59.397 rows=4477 loops=3)
                                Filter: ((i_price >= '1'::numeric) AND
                                         (i_price <= '400000'::numeric) AND
                                         ((i_data ~~ 'a%'::text) OR (i_data ~~ 'b%'::text) OR (i_data ~~ 'c%'::text)))
                                Rows Removed by Filter: 95523
Planning time: 0.872 ms
Execution time: 376.951 ms
(19 rows)
```

Posteriormente, foi considerada a possibilidade de introduzir uma vista materializada de forma a otimizar a secção mais demorada, ou seja, a que se encontra responsável por filtrar todas as

`order_line` que pertencem ao armazém 1, 2, 3, 4 ou 5. A seguir, preencheu-se a vista e executou-se a *query*. O resultado melhorou mas apenas em cerca de 150 ms.

```
create materialized view mv_order_line_w as
    select ol_i_id, ol_quantity, ol_w_id, ol_amount
    from order_line
    where ol_w_id in (1, 2, 3, 4, 5) and ol_quantity between 1 and 10
with no data;
```

```
refresh materialized view mv_order_line_w ;
```

```
                                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
Aggregate  (cost=10000029308.87..10000029308.88 rows=1 width=32) (actual time=291.462..291.462 rows=1 loops=1)
  ->  Hash Join  (cost=10000005458.40..10000029211.26 rows=39043 width=3) (actual time=37.651..286.717 rows=34913 loops=1)
        Hash Cond: (mv_order_line_w.ol_i_id = item.i_id)
        Join Filter: (((item.i_data ~~ 'a%'::text) AND (mv_order_line_w.ol_w_id = ANY ('{1,2,3}'::integer[]))) OR
                      ((item.i_data ~~ 'b%'::text) AND (mv_order_line_w.ol_w_id = ANY ('{1,2,4}'::integer[]))) OR
                      ((item.i_data ~~ 'c%'::text) AND (mv_order_line_w.ol_w_id = ANY ('{1,5,3}'::integer[]))))
        Rows Removed by Join Filter: 23144
        ->  Seq Scan on mv_order_line_w  (cost=10000000000.00..10000020351.30 rows=1295830 width=11) (actual time=0.062..110.301 rows=1295830 loops=1)
        ->  Hash  (cost=5396.29..5396.29 rows=4969 width=55) (actual time=37.561..37.561 rows=4477 loops=1)
              Buckets: 8192  Batches: 1  Memory Usage: 445kB
              ->  Index Scan using pk_item on item  (cost=0.29..5396.29 rows=4969 width=55) (actual time=0.011..36.695 rows=4477 loops=1)
                    Filter: ((i_price >= '1'::numeric) AND
                             (i_price <= '400000'::numeric) AND
                             ((i_data ~~ 'a%'::text) OR (i_data ~~ 'b%'::text) OR (i_data ~~ 'c%'::text)))
                    Rows Removed by Filter: 95523
Planning time: 0.443 ms
Execution time: 291.524 ms
(13 rows)
```

Por último, verificou-se a percentagem de *hits* (usando o `pg_stat_statements`) e esta revelou-se bastante baixa (25%).

```
SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit /
       nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;
```

```
              query                | calls | total_time | rows |  hit_percent
-----------------------------------+-------+------------+------+---------------
 sum(ol_amount) as revenue        +|     1 | 292.143762 |    0 | 25.6593943
 from mv_order_line_w, item       +|       |            |      |
 where                            +|       |            |      |
     ol_i_id = i_id and           +|       |            |      |
     i_price between 1 and 400000 +|       |            |      |
     and                          +|       |            |      |
     ((                           +|       |            |      |
         i_data like 'a%' and     +|       |            |      |
         ol_w_id in (1,2,3)       +|       |            |      |
     ) or (                       +|       |            |      |
         i_data like 'b%' and     +|       |            |      |
         ol_w_id in (1,2,4)       +|       |            |      |
     ) or (                       +|       |            |      |
         i_data like 'c%' and     +|       |            |      |
         ol_w_id in (1,5,3)       +|       |            |      |
     ))                            |       |            |      |
```

A solução passou por aumentar o `shared_buffers` para 400MB, pois é o tamanho suficiente para obter 100% de acerto. O que resultou num tempo de execução de 199.397727 ms.

```
              query                | calls | total_time | rows |  hit_percent
-----------------------------------+-------+------------+------+--------------
```

```
select sum(ol_amount) as revenue  +|     1 | 199.397727 |    0 |    100.0
from mv_order_line_w, item        +|       |            |      |
where                             +|       |            |      |
    ol_i_id = i_id and            +|       |            |      |
    i_price between 1 and 400000  +|       |            |      |
    and                           +|       |            |      |
    ((                            +|       |            |      |
        i_data like 'a%' and      +|       |            |      |
        ol_w_id in (1,2,3)        +|       |            |      |
    ) or (                        +|       |            |      |
        i_data like 'b%' and      +|       |            |      |
        ol_w_id in (1,2,4)        +|       |            |      |
    ) or (                        +|       |            |      |
        i_data like 'c%' and      +|       |            |      |
        ol_w_id in (1,5,3)        +|       |            |      |
    ))                             |       |            |      |
```

Concluindo, após algumas mudanças, nomeadamente, através da adoção do `index scan` , usando vistas materializadas e um aumento do `shared_buffers` e consequente aumento da percentagem de *hits* na cache, conseguiu-se realizar algumas melhorias no tempo de execução da interrogação A2, apesar de a complexidade das condições da interrogação serem um ponto que não permitiu avanços significativos na *performance*. Assim sendo, passou-se de um tempo inicial de **551.449 ms** para **199.397727 ms**.

Query	Tempo de execução em ms (com cache)
Base	4551.065
Index Scan para ol_w_id	376.951
Com vista materializada	291.524
Aumento do shared_buffer	199.398

Tabela 3: Tabela resumo dos tempos de execução.

3.2.4 A4

A interrogação analítica A4 tem como objetivo calcular a cardinalidade de clientes tendo em consideração a sua localização que nunca efetuaram qualquer compra que têm um saldo superior ao saldo médio de todos os utilizadores.

O resultado a apresentar será uma lista com o número de clientes que nunca efetuaram qualquer compra mas têm um saldo superior ao saldo médio das contas de todos os clientes. O código do país será representado pelo primeiro carácter do contacto do cliente.

Apresenta-se então o código SQL e o resultado obtido.

Interrogação A4.

```
select substr(c_state,1,1) as country,
        count(*) as numcust,
        sum(c_balance) as totacctbal
from customer
where substr(c_phone,1,1) in ('1','2','3','4','5','6','7')
        and c_balance > (select avg(c_BALANCE)
                         from customer
                         where c_balance > 0.00
                                 and substr(c_phone,1,1) in
                                         ('1','2','3','4','5','6','7'))
```

```
        and not exists (select *
                        from orders
                        where o_c_id = c_id
                              and o_w_id = c_w_id
                              and o_d_id = c_d_id)
group by substr(c_state,1,1)
order by substr(c_state,1,1);
```

Resultado da interrogação analítica.

```
 country | numcust | totacctbal
---------+---------+-------------
 1       |    7068 | 52652380.83
 2       |     206 |  1514152.00
 3       |     191 |  1446323.40
 4       |     201 |  1490141.00
 5       |     209 |  1545490.00
 6       |     216 |  1640616.00
 7       |     222 |  1683354.00
 8       |     190 |  1384349.00
 9       |     200 |  1463936.70
 a       |     206 |  1524765.59
 b       |     204 |  1527313.00
 c       |     214 |  1555323.00
 d       |     207 |  1572585.00
 e       |     235 |  1761991.00
 f       |     203 |  1520224.00
 g       |     189 |  1404803.00
 h       |     218 |  1659722.00
 i       |     195 |  1451841.00
 j       |     215 |  1640634.00
 k       |     210 |  1574428.00
 l       |     237 |  1791905.00
 m       |     210 |  1567472.00
 n       |     226 |  1684762.00
 o       |     195 |  1464285.00
 p       |     224 |  1633525.00
 q       |     216 |  1571079.00
 r       |     200 |  1480945.00
 s       |     201 |  1504549.00
 t       |     213 |  1564688.00
 u       |     216 |  1615076.87
 v       |     194 |  1456247.01
 w       |     209 |  1548782.00
 x       |     206 |  1511419.00
 y       |     216 |  1610684.00
 z       |     197 |  1476881.00
```

Através do comando EXPLAIN com o argumento analyse foi possível então extrair o plano de execução determinado pelo PostgreSQL com as configurações por defeito.

```
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=398147.29..398147.32 rows=1 width=72) (actual time=1356.451..1356.451 rows=0 loops=1)
   Group Key: (substr((customer.c_state)::text, 1, 1))
   InitPlan 1 (returns \$1)
     ->  Aggregate  (cost=141152.11..141152.12 rows=1 width=32) (actual time=627.934..627.934 rows=1 loops=1)
           ->  Gather  (cost=1000.00..141152.10 rows=1 width=5) (actual time=1.930..627.115 rows=3630 loops=1)
                 Workers Planned: 2
```

```
               Workers Launched: 2
               ->  Parallel Seq Scan on customer customer_1  (cost=0.00..140152.00 rows=1 width=5) (actual time=1.512..611.169 rows=1210 loops=3)
                     Filter: ((c_balance > 0.00) AND (substr((c_phone)::text, 1, 1) = ANY ('{1,2,3,4,5,6,7}'::text[])))
                     Rows Removed by Filter: 598790
   ->  Sort  (cost=256995.17..256995.17 rows=1 width=37) (actual time=1356.450..1356.450 rows=0 loops=1)
         Sort Key: (substr((customer.c_state)::text, 1, 1))
         Sort Method: quicksort  Memory: 25kB
         ->  Nested Loop Anti Join  (cost=0.43..256995.16 rows=1 width=37) (actual time=1356.443..1356.443 rows=0 loops=1)
               ->  Seq Scan on customer  (cost=0.00..167806.21 rows=21070 width=20) (actual time=632.889..1347.332 rows=1816 loops=1)
                     Filter: ((c_balance > \$1) AND (substr((c_phone)::text, 1, 1) = ANY ('{1,2,3,4,5,6,7}'::text[])))
                     Rows Removed by Filter: 1798184
               ->  Index Only Scan using ix_orders on orders  (cost=0.43..4.23 rows=1 width=12) (actual time=0.004..0.004 rows=1 loops=1816)
                     Index Cond: ((o_w_id = customer.c_w_id) AND (o_d_id = customer.c_d_id) AND (o_c_id = customer.c_id))
                     Heap Fetches: 1816
 Planning time: 0.588 ms
 Execution time: 1356.640 ms
(22 rows)
```

Como se pode observar, as operações mais demoradas encontram-se relacionadas com a operação de comparação de valores fixos com o atributo *c_balance* da tabela *customer*.

Desta forma, uma vez que em otimizações anteriores a ordenação dos dados levou a melhorias bastante significativas, adotou-se também nesta situação a mesma estratégia. Assim, por forma a que fosse possível a ordenação dos dados, procedeu-se à criação de um índice sobre a coluna da tabela pela qual a ordenação será efetuada.

Para a operação de criação do índice e ordenação dos dados recorreu-se aos seguintes comandos SQL.

```
CREATE INDEX idx_c_balance ON customer(c_balance);

CLUSTER customer USING idx_c_balance;
```

Apresenta-se então o novo plano de execução.

```
                                                                 QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=154394.87..154394.90 rows=1 width=72) (actual time=81.451..81.451 rows=0 loops=1)
   Group Key: (substr((customer.c_state)::text, 1, 1))
   InitPlan 1 (returns \$0)
     ->  Aggregate  (cost=4.46..4.47 rows=1 width=32) (actual time=5.773..5.773 rows=1 loops=1)
           ->  Index Scan using idx_c_balance on customer customer_1  (cost=0.43..4.46 rows=1 width=5) (actual time=0.063..5.152 rows=3630 loops=1)
                 Index Cond: (c_balance > 0.00)
                 Filter: (substr((c_phone)::text, 1, 1) = ANY ('{1,2,3,4,5,6,7}'::text[]))
                 Rows Removed by Filter: 2492
   ->  Sort  (cost=154390.39..154390.40 rows=1 width=37) (actual time=81.450..81.450 rows=0 loops=1)
         Sort Key: (substr((customer.c_state)::text, 1, 1))
         Sort Method: quicksort  Memory: 25kB
         ->  Nested Loop Anti Join  (cost=0.85..154390.38 rows=1 width=37) (actual time=81.443..81.443 rows=0 loops=1)
               ->  Index Scan using idx_c_balance on customer  (cost=0.43..65337.43 rows=21000 width=20) (actual time=5.807..8.413 rows=1816 loops=1)
                     Index Cond: (c_balance > \$0)
                     Filter: (substr((c_phone)::text, 1, 1) = ANY ('{1,2,3,4,5,6,7}'::text[]))
                     Rows Removed by Filter: 1248
               ->  Index Only Scan using ix_orders on orders  (cost=0.43..4.24 rows=1 width=12) (actual time=0.040..0.040 rows=1 loops=1816)
                     Index Cond: ((o_w_id = customer.c_w_id) AND (o_d_id = customer.c_d_id) AND (o_c_id = customer.c_id))
                     Heap Fetches: 1816
 Planning time: 25.711 ms
 Execution time: 81.543 ms
(21 rows)
```

Como se pode observar, a ordenação dos dados produziu um resultado bastante positivo reduzindo drasticamente o tempo de execução. Verificou-se também que qualquer um dos acessos já se encontra a tirar partido de índices previamente criados. Finalmente, apenas se testou se o tamanho do *shared_buffers* seria o ideal verificando-se que a percentagem de acerto na cache se encontrava nos 100%.

Query	**Tempo de execução em ms (com cache)**
Base	1356.640
Cluster index para idx_c_balance	81.543

Tabela 4: Tabela resumo dos tempos de execução.

3.3 Parâmetros de configuração PostgreSQL

Na escolha do valor mais adequado para os parâmetros selecionados para tentar otimizar, esse será o que vai no sentido da seta representada em todos os gráficos, uma vez que está representa o sentido de otimização. Caso não exista nenhum valor no sentido desta, foi mantido o valor default. A linha a verde na tabela identifica o valor escolhido.

3.3.1 Share_Buffer

Apesar de básico, a análise deste parâmetro não podia deixar de ser feita uma vez que este é responsável por suportar todas às páginas necessárias em memória. Foram então feitos os seguintes testes:

Share Buffer / Warehouses	Throughput	Response Time
50	43,80002262	0,016036814
55	45,00351716	0,021012219
60	52,22046428	0,022347377
64	55,69772785	0,03773738
70	60,8588139	0,040986698
80	67,36574087	0,074384953
128MB	52,22046428	0,022347377
1GB	52,80700765	0,01730566
2GB	49,18283291	0,020699635
3GB	52,44788872	0,021669324
4GB	49,50054637	0,022141148
6GB	52,44296765	0,025467658

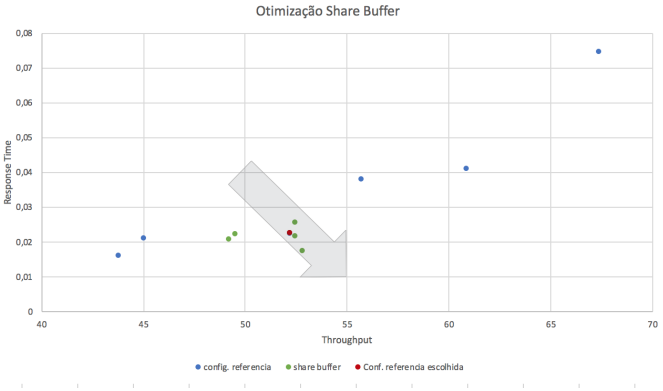


Figura 9: Dados recolhidos da análise do share_buffer (à esquerda) gráfico (à direita).

3.3.2 Work_Mem

Pela análise das interrogações do tpc-c, todas as operações que envolviam trabalhar no buffer work_mem, por exemplo, order by e distinct, no máximo iriam necessitar de 96Kb e 416kb. Fez-se então uma variação dentro deste intervalo para obter o débito máximo.

Warehouses/ Work mem	Throughput	Response Time
50	43,80002262	0,016036814
55	45,00351716	0,021012219
60	52,22046428	0,022347377
64	55,69772785	0,03773738
70	60,8588139	0,040986698
80	67,36574087	0,074384953
100Kb	53,34339453	0,021592578
500Kb	52,11653013	0,018885278
600Kb	52,62988372	0,018816135
700Kb	52,44884456	0,018686365
1MB	52,40879248	0,019673871

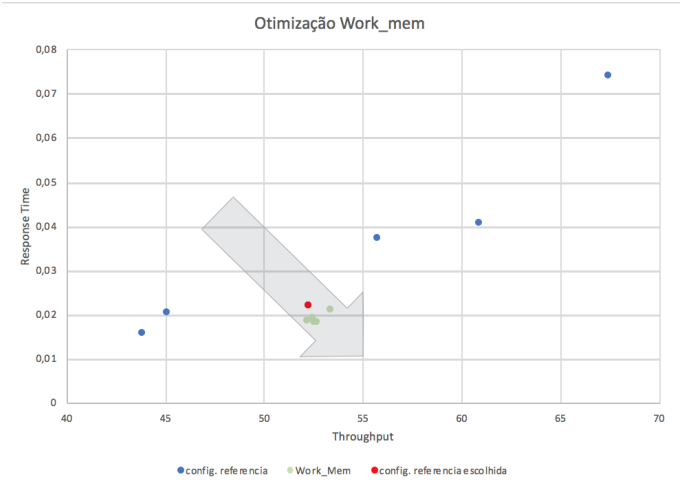


Figura 10: Dados recolhidos da análise do work_mem (à esquerda) gráfico (à direita).

Em todos os casos de teste, verificou-se através do pg_badger na seção Temp File, que todas as operações feitas no work_mem são feitas em memória, uma vez que não há criação de ficheiros temporários.

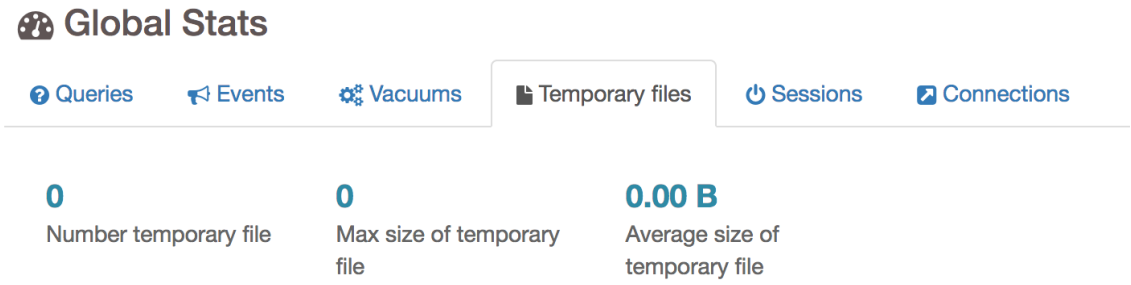


Figura 11: Exemplo pg_badger temp file para 700Kb

3.3.3 Vacuum

Com o intuito de ter uma ideia de quantos dead tuples estão a ser gerados durante a execução, foi desativado o auto-vacuum e obteve-se então os seguintes resultados:

```
tpcc-db=# select relname, n_tup_ins, n_tup_upd, n_tup_del, n_live_tup, n_dead_tup, autovacuum_count from pg_stat_all_tables where relname='order_line' or relname='stock' or relname=
'customer' or relname='orders' or relname='history' or relname='new_order' or relname='item' or relname='district' or relname='warehouse';
  relname   | n_tup_ins | n_tup_upd | n_tup_del | n_live_tup | n_dead_tup | autovacuum_count
------------+-----------+-----------+-----------+------------+------------+------------------
 customer   |   1800000 |     56156 |         0 |    1799980 |      16801 |                0
 district   |       600 |     55704 |         0 |        600 |        764 |                0
 history    |   1823815 |         0 |         0 |    1823342 |        473 |                0
 item       |    100000 |         0 |         0 |     100000 |          0 |                0
 new_order  |    568412 |         0 |     25630 |     541687 |      26725 |                0
 order_line |  15582159 |    217933 |         0 |   15570244 |      76602 |                0
 orders     |   1828423 |     25630 |         0 |    1827317 |       4042 |                0
 stock      |   6000000 |    568374 |         0 |    6000000 |     164648 |                0
 warehouse  |        60 |     24566 |         0 |         60 |         60 |                0
(9 rows)
```

Figura 12: Análise do número de dead tuples criados após 20 min run.sh

Pela análise dos resultados acima percebe-se a necessidade de ativar o auto-vacuum, dado os milhares de dead tuples criados durante 20 min de execução. Sem esquecer também do analyze que é responsável por atualizar as estatísticas de forma a que o query planner faça a melhor escolha de plano de execução.

Sabe-se que há dois parâmetros fundamentais que são responsáveis por ativar o vacuum, autovacuum_vacuum_threshold e autovacuum_vacuum_scale_factor, bem como o analyze, autovacuum_analyze_threshold e autovacuum_analyze_scale_factor. Estes são ativados quando:

`n_dead_tuples = threshold + reltuples* scale_factor.`

Para além disso o auto_vacuum fica ativo enquanto não ultrapassar os valores definido nos parâmetros autovacuum_vacuum_cost_delay e autovacuum_vacuum_cost_limit.

Os primeiros testes que se fez de forma a obter uma ideia de como o sistema se comporta foi:

- teste 1:
  - autovacuum_vacuum_threshold = 1000
  - autovacuum_vacuum_scale_factor = 0
  - autovacuum_analyze_threshold = 1000
  - autovacuum_analyze_scale_factor = 0
  - autovacuum_vacuum_cost_delay = 20ms (default)
  - autovacuum_vacuum_cost_limit = 200 (default)
- teste 2:
  - autovacuum_vacuum_threshold = 10000

- autovacuum_vacuum_scale_factor = 0
- autovacuum_analyze_threshold = 10000
- autovacuum_analyze_scale_factor = 0
- autovacuum_vacuum_cost_delay = 20ms (default)
- autovacuum_vacuum_cost_limit = 200 (default)

Teoricamente, no teste 1, iria ser feito um vacuum e analyze a cada 1000 linhas dead e no teste 2 a cada 10000. Estes foram os resultados obtidos:

relname	n_tup_ins	n_tup_upd	n_tup_del	n_live_tup	n_dead_tup	autovacuum_count
customer	1800000	56079	0	1800012	17751	0
district	600	55788	0	600	0	21
history	1826409	0	0	1826351	58	0
item	100000	0	0	100000	0	0
new_order	568306	0	25607	542355	25951	0
order_line	15582213	217687	0	15574006	70600	0
orders	1828325	25607	0	1827955	3440	0
stock	6000000	568548	0	5996045	154116	0
warehouse	60	26641	0	60	0	21

relname	n_tup_ins	n_tup_upd	n_tup_del	n_live_tup	n_dead_tup	autovacuum_count
customer	1800000	55400	0	1800028	8729	1
district	600	55334	0	600	4403	0
history	1823736	0	0	1823249	487	0
item	100000	0	0	100000	0	0
new_order	568228	0	25270	542429	0	2
order_line	15581078	215120	0	15488234	0	2
orders	1828242	25270	0	1827699	4469	0
stock	6000000	566257	0	5949054	0	2
warehouse	60	25106	0	60	1034	0

Figura 13: auto-vacuum feitos teste 1 (à esquerda) auto-vacuum feitos teste 2 (à direita).

Pela análise dos resultados, o grupo de trabalho tentou perceber o que realmente estava a acontecer uma vez que havia tabelas que não sofriam vaccum durante toda a execução. Tem-se a certeza que o processo auto-vacuum foi inicializado através do comando:

```
ps -ef  | grep -i vacuum
```

```
postgres: autovacuum launcher process
postgres: autovacuum worker process   tpcc-db
postgres: autovacuum worker process   tpcc-db
postgres: autovacuum worker process   tpcc-db
```

Figura 14: Launcher e 3 worker responsaveis por auto-vacuum

Portanto, temos a certeza que o auto-vacuum está ativo, está a escuta, sendo assim, o que impedirá de o fazer vacuum das tabelas? Provavelmente porque as tabelas devem ter quase sempre locks. Para verificar que isso realmente acontece, com o parâmetro log_autovacuum_min_duration = 0 todos os autovacuum que forem skip graças a um lock, aparecerá essa informação no ficheiro de log. Analisado o ficheiro de log temos que não aparece nenhuma entrada "skipping VACUUM of .. — lock not available", portanto, não é um problema de locks.

Segunda opção será, os parâmetros autovacuum_vacuum_cost_delay e autovacuum_vacuum_cost_limit não estarem a permitir que seja feito vacuum. Portanto foi feito um teste com os seguintes parâmetros:

- autovacuum_vacuum_threshold = 100
- autovacuum_vacuum_scale_factor = 0
- autovacuum_analyze_threshold = 100
- autovacuum_analyze_scale_factor = 0
- autovacuum_vacuum_cost_delay = 10
- autovacuum_vacuum_cost_limit = 2000

relname	n_tup_ins	n_tup_upd	n_tup_del	n_live_tup	n_dead_tup	autovacuum_count
customer	1800000	33632	0	1799580	5260	6
district	600	33522	0	600	8748	10
history	1814324	0	0	1811507	2123	5
item	100000	0	0	100000	0	0
new_order	554790	0	15185	537159	4764	10
order_line	15455815	128879	0	15394945	45998	7
orders	1814836	15185	0	1812269	4691	10
stock	6000000	316121	0	6004068	136024	1
warehouse	60	14846	0	60	2006	7

Figura 15: Análise tipo de tuplos existentes e auto-vacuum feito durante o run

E realmente verifica-se que o número de auto-vacuum feitos durante a execução aumentou substancialmente.

Uma vez que restam dead tuple no final da execução, verificou-se se o vacuum termina de limpar as tabelas. Com recurso ao analyze para atualizar as estatísticas, verificou-se que isso estava a acontecer.

Depois, foi feito então vários testes de forma a trazer um equilíbrio entre throughtput/response time e o número total dead tuples existentes no final da execução.

Os melhores resultados em termos de menor número de dead tuples existentes no final da execução foram:

- autovacuum_vacuum_threshold = 100
- autovacuum_vacuum_scale_factor = 0
- autovacuum_analyze_threshold = 100
- autovacuum_analyze_scale_factor = 0
- autovacuum_vacuum_cost_delay = 10
- autovacuum_vacuum_cost_limit = 1000

relname	n_tup_ins	n_tup_upd	n_tup_del	n_live_tup	n_dead_tup	autovacuum_count
customer	1800000	43538	0	1799555	0	15
district	600	43643	0	600	0	17
history	1820324	0	0	1819860	10	2
item	100000	0	0	100000	0	0
new_order	561567	0	19616	541402	0	17
order_line	15517462	166575	0	15481044	7023	6
orders	1821588	19616	0	1821002	0	16
stock	6000000	439163	0	5996749	114973	1
warehouse	60	20484	0	60	0	12

(9 rows)

Figura 16: Análise tipo de tuplos existentes e auto-vacuum feito durante o run

Apenas a tabela stock apresenta um grande número de tuplos dead. Esta configuração resultou num throughtput de 51,4481826 tx/s e tempo de resposta 0,070658273s.

- autovacuum_vacuum_threshold = 50
- autovacuum_vacuum_scale_factor = 0

- autovacuum_analyze_threshold = 50
- autovacuum_analyze_scale_factor = 0
- autovacuum_vacuum_cost_delay = 10
- autovacuum_vacuum_cost_limit = 1000

relname	n_tup_ins	n_tup_upd	n_tup_del	n_live_tup	n_dead_tup	autovacuum_count
customer	1800000	54750	0	1796767	0	11
district	600	54896	0	600	0	18
history	1824757	0	0	1824109	43	5
item	100000	0	0	100000	0	0
new_order	567362	0	24680	540742	0	18
order_line	15575982	210001	0	15528937	0	11
orders	1827391	24680	0	1825422	0	16
stock	6000000	556196	0	5943836	0	4
warehouse	60	25352	0	60	0	14

Figura 17: Análise tipo de tuplos existentes e auto-vacuum feito durante o run

Apenas a tabela history fica com tuplos dead. Esta configuração resultou num throughtput de 50,88243094 tx/s e tempo de resposta 0,062966767s.

O resultados de todos os testes foi:

Vacuum(threshold)/ Warehouses	Vacuum(scale_factor)/ Warehouses	max_workers/ Warehouses	naptime/ Warehouses	cost_delay/ Warehouse	cost_limit/ Warehouses	Throughput	Response Time	n_dead_tuples(s/ vacuum-290115)
50						43,8000226	0,016036814	
55						45,0035172	0,021012219	
60						52,2204643	0,022347377	
64						55,6977279	0,03773738	
70						60,8588139	0,040986698	
80						67,3657409	0,074384953	
1000	0	default(3)	default(1min)	default(20ms)	default(200)	52,7234409	0,030620391	271916
10000	0	default(3)	default(1min)	default(20ms)	default(200)	49,7565585	0,016375103	19122
100	0	default(3)	default(1min)	10ms	2000	47,1636643	0,083457966	209614
100	0	default(3)	default(1min)	10ms	1000	51,4481826	0,070658273	122006
500	0	default(3)	default(1min)	10ms	1000	51,5620444	0,043960803	43628
50	0	default(3)	default(1min)	10ms	1000	50,8824309	0,062966767	43
50	0	6	default(1min)	10ms	1000	50,6846159	0,070163931	57118
default(50)	default(0.2)	default(3)	default(1min)	10ms	1000	49,8321881	0,016838134	294030
default(50)	default(0.2)	default(3)	default(1min)	default(20ms)	default(200)	52,2964443	0,017732558	274446

Figura 18: Dados recolhidos para análise do auto-vacuum

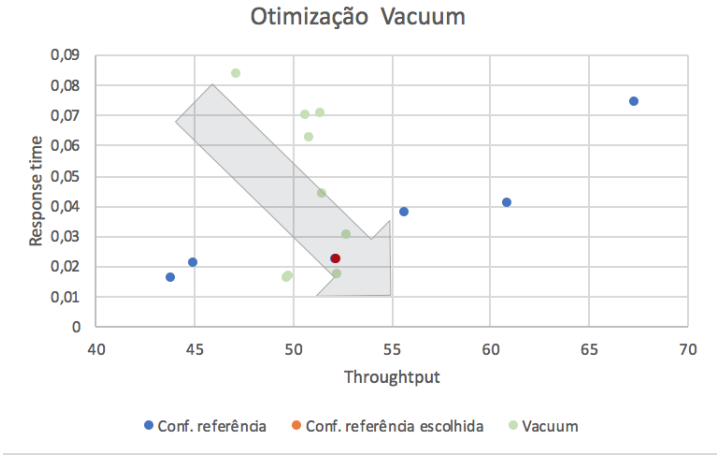


Figura 19: Gráfico dos repetivos dados

Num contexto real em que o sistema ia estar permanentemente a correr, inevitavelmente o vacuum das tabelas teria de acontecer para libertar o espaço em disco ocupado pelos dead tuples. Mas uma vez que este trabalho assenta numa análise de 20 minutos, foi decidiu manter as configurações default uma vez que são estas que apresentam melhores resultados.

3.3.4 Paramêtros de Checkpoints

Os valores default dos parâmetros de ckeckpoint mais relevantes são:

- checkpoint_timeout = 5min
- max_wal_size = 1GB

Checkpoint é feito quando checkpoint_timeout é alcançado ou max_wal_size é ultrapassado. Num sistema com uma grande carga de escritas, temos que valores de max_wal_size baixos rapidamente serão alcançados. Portanto se queremos que o checkpoint seja feito com base no tempo, o parâmetro max_wal_size terá de tomar valores altos de forma a que esse tamanho nao seja alcançado em menos de checkpoint_timeout. Portanto, quando se pretende fazer a análise com base no tempo, fixou-se 30GB, um valor por alto. Durante todo este processo foi verificado através do ficheiro de log que está a ser gerado se o checkpoint foi ativado com base no tempo ou no tamanho WAL. Com base nisto, primeiro variou-se apenas os dois primeiros parâmetros e obteve-se estes resultados:

checkpoint_timeout/ Warehouses	max_wal_size/ Warehouses	checkout ativado por	tamanho pg_wal	Throughput	Response Time
50				43,8000226	0,016036814
55				45,0035172	0,021012219
60				52,2204643	0,022347377
64				55,6977279	0,03773738
70				60,8588139	0,040986698
80				67,3657409	0,074384953
2min	30GB	time	1.5GB	52,5294079	0,027834694
5min	2MB	size WAL	16M	52,2650715	0,031810445
5min	50MB	size WAL	48MB	52,3058143	0,018703371
5min	70MB	size WAL	225MB	52,5785418	0,020504561
5min	100MB	size WAL	95MB	52,9045878	0,015998397
5min	200MB	size WAL	193MB	53,06746	0,015013078
5min	300MB	size WAL	289MB	52,5490704	0,015399604
5min	500MB	size WAL	497M	52,2193868	0,017029048
5min	1GB	size WAL	1.1GB	52,7366523	0,017824585
5min	30GB	time	2.5GB	52,4238849	0,026928673
5min	2GB	size WAL	2.0GB	52,363264	0,030210006
7min	30GB	time	2.9GB	52,074531	0,027333036

Figura 20: Dados recolhidos da análise dos parâmetros de checkpoint

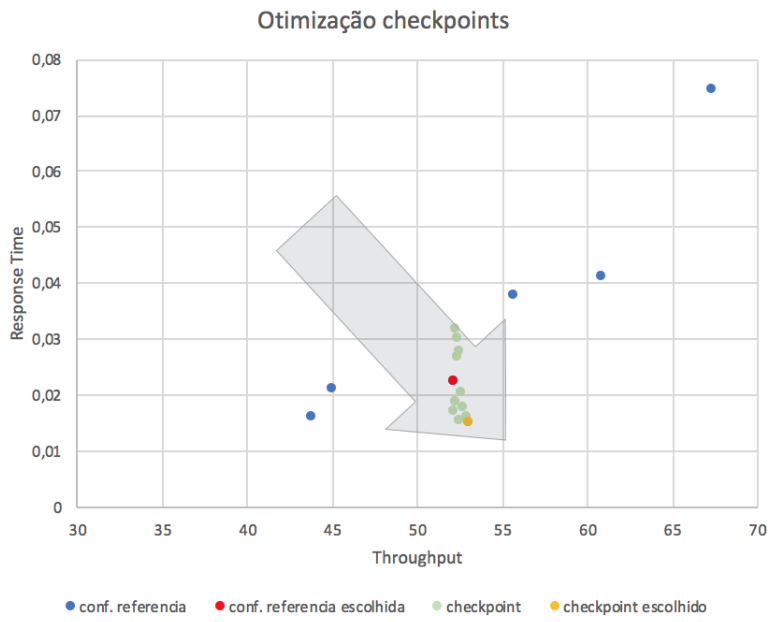


Figura 21: Gráfico dos respetivos dados

Verifica-se que os melhores resultados são quando é ativado o checkpoint através max_wal_size (xlog) e como foi fixado este valor na ordem dos MB, o custo de trazer MB para disco é menor que checkpoint ser ativado através de timeout (time) e agora ter de trazer para disco todos os dados dirty durante esse período de tempo(provavelmente na ordem dos GB). Temos que a carga do sistema é constante, isto é está sempre sobrecarregado, portanto é preferível indo escrevendo aos poucos no disco e muitas vezes, do que escrever GB de uma só vez. A prova disso são os resultados obtidos.

3.3.5 wal_buffers

Sendo este um sistema com bastantes escritas é necessário analisar o paramêtro wal_buffer, uma vez que caso este encha impossibilita a inserção de WAL até o que estiver no buffer for trazido

para disco.

Foram feitos os seguintes testes:

Warehouses/ Wal_buffers	Throughput	Response Time
50	43,8000226	0,016036814
55	45,0035172	0,021012219
60	52,2204643	0,022347377
64	55,6977279	0,03773738
70	60,8588139	0,040986698
80	67,3657409	0,074384953
default(32MB)	52,2964443	0,017732558
40MB	52,3826029	0,017021619
50MB	52,2924429	0,017531337
60MB	52,590046	0,017911474
70MB	52,5464736	0,022160348

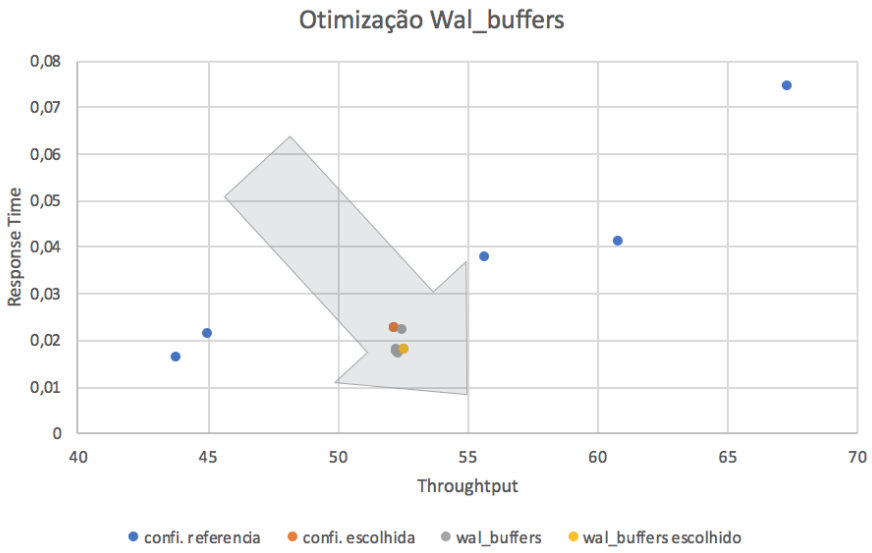


Figura 22: Dados recolhidos da análise do wal_buffers(em cima) e gráfico (em baixo).

3.3.6 full_page_writes, fsync e synchronous_commit

Estes parâmetros asseguram as propriedades ACID, e portanto têm de ser mantido a on.

Para testar se realmente há um impacto na performance, foi colocado o synchronous_commit a off . O resultado foi o seguinte:

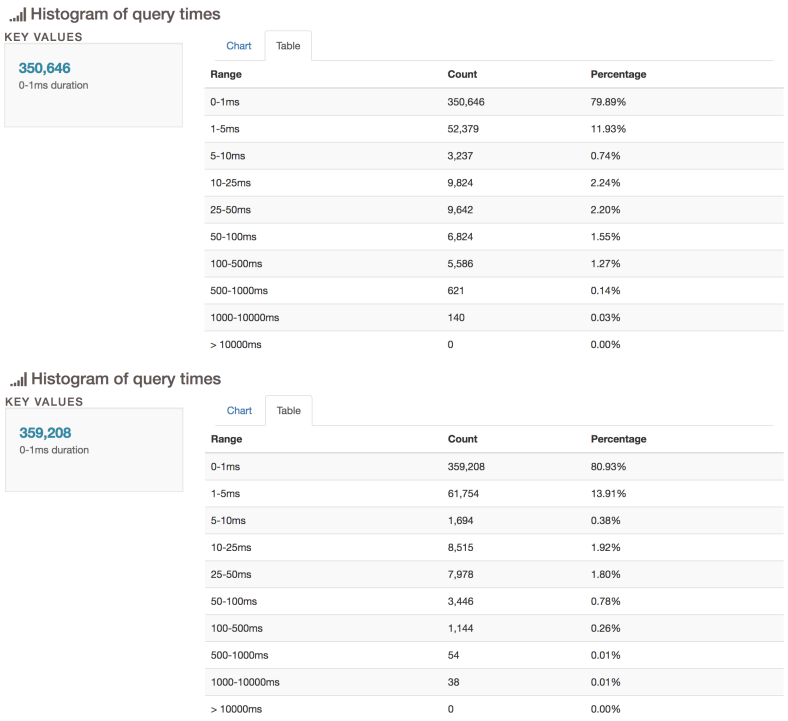
- synchronous_commit: off
  - throughput(tx/s): 52.2768389472
  - response_time(s): 0.0206333676065
  - abort_rate: 0.00251005370975
- synchronous_commit: on
  - throughput(tx/s): 52.6155666357

- response_time(s): 0.0183629671271
- abort_rate: 0.00184368737475

Nem valerá a pena colocar a questão se será mais eficiente ter um backup dos dados, e permitir que o sistema corra com `synchronous_commit` a off e caso haja uma falha no sistema tem-se sempre o backup.

3.3.7 Comparação de resultados

Dado por concluído à análise dos parâmetros de configuração do PostgreSQL e tirando partindo das informações obtidas no pgbadger é feita a comparação em termos de tempos de execução entre a configuração escolhida como referência e a configuração obtida no final com todas as otimizações conseguidas.



Range	Count	Percentage
0-1ms	350,646	79.89%
1-5ms	52,379	11.93%
5-10ms	3,237	0.74%
10-25ms	9,824	2.24%
25-50ms	9,642	2.20%
50-100ms	6,824	1.55%
100-500ms	5,586	1.27%
500-1000ms	621	0.14%
1000-10000ms	140	0.03%
> 10000ms	0	0.00%

Range	Count	Percentage
0-1ms	359,208	80.93%
1-5ms	61,754	13.91%
5-10ms	1,694	0.38%
10-25ms	8,515	1.92%
25-50ms	7,978	1.80%
50-100ms	3,446	0.78%
100-500ms	1,144	0.26%
500-1000ms	54	0.01%
1000-10000ms	38	0.01%
> 10000ms	0	0.00%

Figura 23: Histograma do tempo de execução das interrogações na (em cima) gráfico (em baixo).

Os resultados devolvidos pelo TPC-C foram:

- configuração de referencia:
  - throughput(tx/s): 52.220464276738895
  - response_time(s): 0.022347377341929237
  - abort_rate: 0.011696081364
- configuração otimizada:
  - throughput(tx/s): 52.7325855459
  - response_time(s): 0.0159561261572
  - abort_rate: 0.00177408250808

4 Replicação em Streaming

A secção 3 teve como principal objetivo otimizar a configuração de referência verticalmente, ou seja, apenas alterando as interrogações, os planos de execução, as configurações do PostgreSQL e utilizando mecanismos de redundância (e.g. índices e vistas materializadas). Contudo, nesta secção será dada prioridade à otimização horizontal, isto é, aos mecanismos de replicação, *sharding* e/ou processamento distribuído.

Desta forma, de entre os vários caminhos possíveis optou-se por utilizar a replicação como mecanismo de otimização primordial. Assim sendo, utilizou-se a funcionalidade de *streaming replication* para criar uma arquitetura Master/Slave que permite a replicação da BD entre duas máquinas. De facto, consegue-se distribuir o tráfego de maneira a não sobrecarregar uma das máquinas e, consequentemente, melhorar a performance da base de dados. Mais ainda, caso seja necessário é possível ter parâmetros de configuração do PostgreSQL diferentes de máquina para máquina, podendo-se ajustar de melhor forma ao tipo de carga em questão.

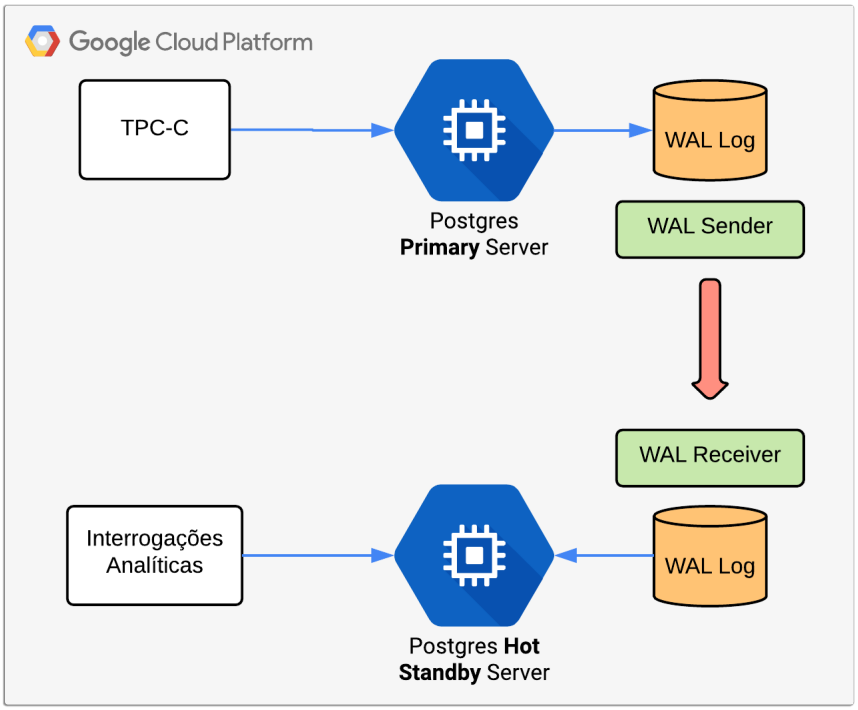


Figura 24: Arquitetura da replicação em streaming no GCP.

Em detalhe, uma das máquinas virtuais da Google Cloud Platform ficou responsável por alojar o servidor primário do PostgreSQL. Ainda, tem como funcionalidade produzir os *logs Write-Ahead Logging* (WAL) e enviar o seu conteúdo pela rede para o servidor *standby*. Este processo permite que o *standby* esteja mais atualizado em relação ao envio de *logs* baseado em ficheiros uma vez que não é preciso esperar que o ficheiro de *log* esteja completo para o enviar. De forma, a permitir a ligação entre os dois servidores, os seguintes parâmetros foram alterados no servidor primário:

Parâmetros do servidor primário.

```
listen_addresses = '*'
wal_level = hot_standby
synchronous_commit = local
max_wal_senders = 2
wal_keep_segments = 10
```

```
synchronous_standby_names = 'pgslave001'
```

Já o servidor *standby* necessita de solicitar a sincronização. O feito é conseguido através do seguinte script:

Script para cópia dos dados do servidor primário para o standby.

```
#! /bin/bash
if [ "$#" -ne 1 ]; then
    echo "Usage: $0 MASTER_IP" >&2
    exit 1
fi
rm -r tpcc-db
mkdir tpcc-db
chmod 700 tpcc-db/
pg_basebackup -h $1 -U replicator -p 5432 -D tpcc-db -P -Xs -R
```

Posteriormente, é necessário configurar os seguintes parâmetros mas desta vez no servidor *standby*:

Parâmetros do servidor standby.

```
wal_level = hot_standby
synchronous_commit = local
max_wal_senders = 2
wal_keep_segments = 10
synchronous_standby_names = 'pgslave001'
hot_standby = on
```

Desta forma, o *standby* conecta-se ao primário, que envia os registos WAL mal são gerados.

Informação sobre o processo de replicação.

```
tpcc-db=# select * from pg_stat_replication;
  usename   | application_name |  client_addr  | client_port |   state   |  sent_lsn  | sync_state
------------+------------------+---------------+-------------+-----------+------------+------------
 replicator | walreceiver      | 35.237.197.74 |       38552 | streaming | 1/7E118900 | async
(1 row)
```

O processo de replicação é realizado assincronamente desta forma pode existir algum atraso na transmissão da informação, o que no pior caso pode obrigar a reinicializar o servidor *standby* para que este fique operacional. No entanto, a latência entre as duas máquina é bastante reduzida e como tal não foi necessário tomar nenhuma medida para prevenir a perda de informação. De facto, a latência existente entre o registo WAL enviado pelo primário e recebido pelo *standby* é praticamente inexistente uma vez que os identificadores presentes na figura 4 são praticamente os mesmos (salvo alguma porção resultante do desfazamento na execução dos comandos). Mais ainda, é possível verificar que o servidor primário consegue enviar os *logs* de forma bastante eficiente, uma vez que a diferença entre os identificadores é praticamente inexistente (figura 4).

Verificação latência entre o servidor primário e o /standby/.

```
[PRIMARIO]
tpcc-db=# select sent_lsn from pg_stat_replication;
  sent_lsn
------------
 3/3A9B10F8
(1 row)

[STANDBY]
tpcc-db=# select * from pg_last_wal_receive_lsn();
 pg_last_wal_receive_lsn
```

```
-------------------------
       3/3AD840E0
(1 row)
```

Verificação da carga do servidor primário.

```
tpcc-db=# select sent_lsn from pg_stat_replication;
  sent_lsn
------------
 3/2D740A90
(1 row)

tpcc-db=# select pg_current_wal_lsn();
 pg_current_wal_lsn
--------------------
     3/2DD04888
(1 row)
```

Tendo em conta os diferentes paradigmas de execução (*benchmark* TPC-C e interrogações analíticas) e o objetivo de separar adequadamente o tráfego com base nestes, revelou-se necessário interagir com o servidor *standby*. Assim sendo, o servidor *standby* assumiu o estatuto de *hot standby* de forma a permitir realizar apenas interrogações de leitura. Desta forma, torna-se possível aliviar o servidor primário das interrogações analíticas, pois estas passam a ser realizadas ao servidor *standby*.

Na prática, os resultados obtidos vieram satisfazer o objetivo de melhorar os tempos de resposta das interrogações analíticas, uma vez que os recursos da máquina não eram utilizados abusivamente pelo TPC-C. Cada uma das interrogações A1, A2, A3 e A4 foram executadas no minuto 7, 10, 13, 16, respetivamente. De facto, esta restrição foi imposta de forma a haver uma comparação mais justa entre os tempos da arquitetura com e sem replicação.

Interrogações analíticas automatizadas.

```
#! /bin/bash
sleep 420
echo "A1"
psql -h localhost -f a1.sql -d tpcc-db > /dev/null
sleep 180
echo "A2"
psql -h localhost -f a2.sql -d tpcc-db > /dev/null
sleep 180
echo "A3"
psql -h localhost -f a3.sql -d tpcc-db > /dev/null
sleep 180
echo "A4"
psql -h localhost -f a4.sql -d tpcc-db > /dev/null
```

Desta forma, a interrogação **A1** passou de **7m37s** para **4s451ms**, a A2 passou de **16s149ms** para **11s580ms**, a A3 de **6s426ms** para **598ms** e por fim a A4 passou de um tempo de execução de **13s670ms** para **2s274ms**. Contudo, de realçar que o tempo de resposta resultante do `showtpc` desceu de 0.05ms para 0.02ms. O que significa que mais do que uma influência no débito total, as interrogações analíticas encontravam-se a perturbar o tempo que o servidor demorava a responder.

Slowest individual queries

Rank	Duration	Query
1	7m37s	SELECT sum(ol_amount) / 2.0 AS avg_yearly FROM order_line, (SELECT i_id, avg(ol_quantity) AS a FROM item, order_line WHERE ol_i_id = i_id AND i_data LIKE 'b%' GROUP BY i_id) t WHERE ol_i_id = t.i_id AND ol_quantity < t.a; [Date: 2018-12-27 11:44:43 - Database: tpcc-db - User: afonscosta - Remote: ::1 - Application: psql]
2	16s149ms	SELECT c_count, count(*) AS custdist FROM (SELECT c_id, count(o_id) FROM customer LEFT OUTER JOIN orders ON (c_w_id = o_w_id AND c_d_id = o_d_id AND c_id = o_c_id AND o_carrier_id > 8) GROUP BY c_id) AS c_orders (c_id, c_count) GROUP BY c_count ORDER BY custdist DESC, c_count DESC; [Date: 2018-12-27 11:45:21 - Database: tpcc-db - User: afonscosta - Remote: ::1 - Application: psql]
3	13s670ms	SELECT substr(c_state, 1, 1) AS country, count(*) AS numcust, sum(c_balance) AS totacctbal FROM customer WHERE substr(c_phone, 1, 1) IN ('1', '2', '3', '4', '5', '6', '7') AND c_balance > (SELECT avg(c_BALANCE) FROM customer WHERE c_balance > 0.00 AND substr(c_phone, 1, 1) IN ('1', '2', '3', '4', '5', '6', '7')) AND NOT EXISTS (SELECT * FROM orders WHERE o_c_id = c_id AND o_w_id = c_w_id AND o_d_id = c_d_id) GROUP BY substr(c_state, 1, 1) ORDER BY substr(c_state, 1, 1); [Date: 2018-12-27 11:47:19 - Database: tpcc-db - User: afonscosta - Remote: ::1 - Application: psql]
4	6s426ms	SELECT sum(ol_amount) AS revenue FROM order_line, item WHERE (ol_i_id = i_id AND i_data LIKE 'a%' AND ol_quantity >= 1 AND ol_quantity <= 10 AND i_price BETWEEN 1 AND 400000 AND ol_w_id IN (1, 2, 3)) OR (ol_i_id = i_id AND i_data LIKE 'b%' AND ol_quantity >= 1 AND ol_quantity <= 10 AND i_price BETWEEN 1 AND 400000 AND ol_w_id IN (1, 2, 4)) OR (ol_i_id = i_id AND i_data LIKE 'c%' AND ol_quantity >= 1 AND ol_quantity <= 10 AND i_price BETWEEN 1 AND 400000 AND ol_w_id IN (1, 5, 3)); [Date: 2018-12-27 11:46:11 - Database: tpcc-db - User: afonscosta - Remote: ::1 - Application: psql]

Figura 25: Tempos das interrogações analíticas na arquitetura de referência.

Slowest individual queries

Rank	Duration	Query
1	11s580ms	SELECT c_count, count(*) AS custdist FROM (SELECT c_id, count(o_id) FROM customer LEFT OUTER JOIN orders ON (c_w_id = o_w_id AND c_d_id = o_d_id AND c_id = o_c_id AND o_carrier_id > 8) GROUP BY c_id) AS c_orders (c_id, c_count) GROUP BY c_count ORDER BY custdist DESC, c_count DESC; [Date: 2018-12-29 17:47:05 - Database: tpcc-db - User: afonscosta - Remote: ::1 - Application: psql]
2	4s451ms	SELECT sum(ol_amount) / 2.0 AS avg_yearly FROM order_line, (SELECT i_id, avg(ol_quantity) AS a FROM item, order_line WHERE ol_i_id = i_id AND i_data LIKE 'b%' GROUP BY i_id) t WHERE ol_i_id = t.i_id AND ol_quantity < t.a; [Date: 2018-12-29 17:43:53 - Database: tpcc-db - User: afonscosta - Remote: ::1 - Application: psql]
3	2s274ms	SELECT substr(c_state, 1, 1) AS country, count(*) AS numcust, sum(c_balance) AS totacctbal FROM customer WHERE substr(c_phone, 1, 1) IN ('1', '2', '3', '4', '5', '6', '7') AND c_balance > (SELECT avg(c_BALANCE) FROM customer WHERE c_balance > 0.00 AND substr(c_phone, 1, 1) IN ('1', '2', '3', '4', '5', '6', '7')) AND NOT EXISTS (SELECT * FROM orders WHERE o_c_id = c_id AND o_w_id = c_w_id AND o_d_id = c_d_id) GROUP BY substr(c_state, 1, 1) ORDER BY substr(c_state, 1, 1); [Date: 2018-12-29 17:53:10 - Database: tpcc-db - User: afonscosta - Remote: ::1 - Application: psql]
4	598ms	SELECT sum(ol_amount) AS revenue FROM order_line, item WHERE (ol_i_id = i_id AND i_data LIKE 'a%' AND ol_quantity >= 1 AND ol_quantity <= 10 AND i_price BETWEEN 1 AND 400000 AND ol_w_id IN (1, 2, 3)) OR (ol_i_id = i_id AND i_data LIKE 'b%' AND ol_quantity >= 1 AND ol_quantity <= 10 AND i_price BETWEEN 1 AND 400000 AND ol_w_id IN (1, 2, 4)) OR (ol_i_id = i_id AND i_data LIKE 'c%' AND ol_quantity >= 1 AND ol_quantity <= 10 AND i_price BETWEEN 1 AND 400000 AND ol_w_id IN (1, 5, 3)); [Date: 2018-12-29 17:50:08 - Database: tpcc-db - User: afonscosta - Remote: ::1 - Application: psql]

Figura 26: Tempos das interrogações analíticas no servidor standby.

Além disso, o débito do TPC-C desceu.

Métrica	Configuração de Referência	Replicação com Streaming
Débito	51.645205861020024	49.66876248991671
Tempo de resposta	0.055311857528101895	0.02303024644201115
Taxa de aborto	0.0230076953176	0.0588752196837

Tabela 5: Tabela de comparação de resultados do `showtpc`.

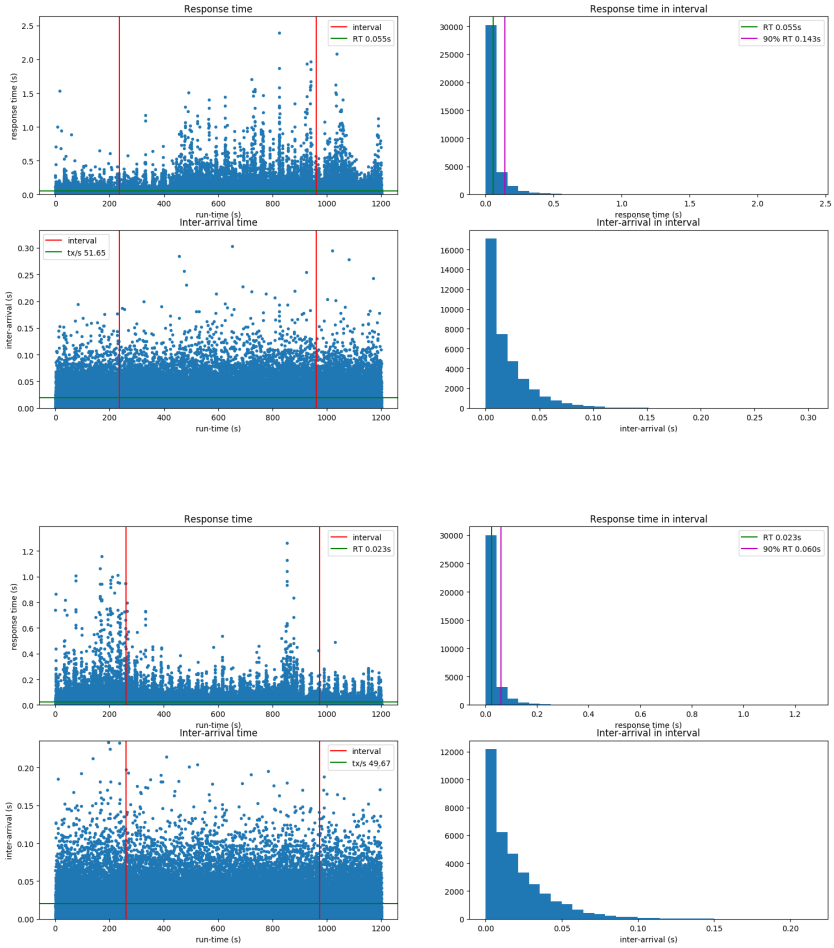


Figura 27: Resultado do `showtpc` para a configuração de referência (em cima) e para a arquitetura com replicação (em baixo).

Dado que as interrogações analíticas não correram na mesma máquina que o *benchmark* era de esperar que houvesse um aumento no débito do TPC-C, pois mais recursos estariam disponíveis. Contudo, por um lado, é de salientar que as interrogações analíticas são unicamente de leitura. Adicionalmente, a maior parte dos tipos de locks possíveis no PostgreSQL permite que as leituras possam ser realizadas em paralelo, logo as *queries* analíticas em pouco afetavam o débito do TPC-C, a não ser na utilização dos recursos da máquina. Por outro lado, uma razão viável é a própria existência da replicação via *streaming*, uma vez que necessita de mais dados para implementar a sincronização, ou seja, precisa de partilhar, entre o primário e o *standby*, um maior número de ficheiros WAL. Desta forma, foram realizadas várias tentativas com diferentes valores de `max_wal_size` (1GB foi o usado por omissão para os valores apresentados em cima [26]) de forma a aumentar o débito do TPC-C, uma vez que este parâmetro está responsável por limitar o tamanho do WAL entre checkpoints automáticos. [3]

max_wal_size	Débito	Tempo de resposta	Taxa de aborto
100MB	49,81434654	0,011185299	0.0483853834714
200MB	50,16974989	0,011716983	0.0473272765688
500MB	49,06877292	0,016791068	0.0658395676366
1GB	49,66876249	0,023030246	0.0588752196837
2GB	52,29564533	0,035429848	0.00629362941791
3GB	46,92126073	0,037287939	0.110795454545

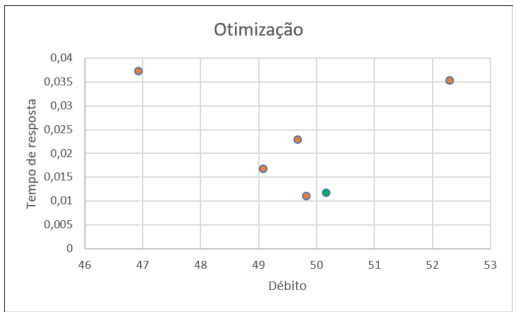


Figura 28: Análise realizada ao `max_wal_size`.

Assim sendo, foi escolhida a arquitetura com 200MB de `max_wal_size` pois é a que tem uma melhor relação entre o débito e o tempo de resposta.

Por fim, realçar que no futuro a introdução de um balanceador de carga (e.g. `pgpool`) que permitisse distribuir as interrogações com base na sua funcionalidade (escrita/leitura) pelos servidores existentes, seria uma estratégia a considerar. No caso em particular deste projeto, esta estratégia não foi abordada uma vez que o benchmark TPC-C apenas utiliza transações e o `pgpool` não consegue perceber quais as instruções que são de escrita ou leitura e como tal iria mandar todas para o servidor primário. Por sua vez as interrogações analíticas seria repartidas pelo primário e pelo standby. No entanto, o mais correto seria mandar para o standby uma vez que este se encontra bastante mais livre. Assim sendo, numa situação dita normal, a arquitetura apresentada a seguir seria uma solução a ter em conta.

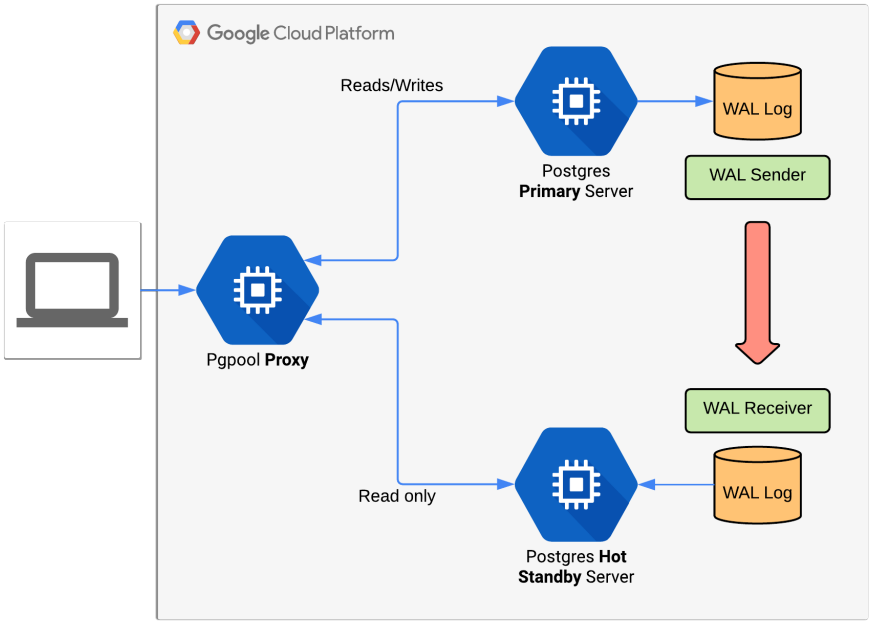


Figura 29: Arquitetura da replicação em streaming no GCP com `Pgpool`.

5 Sharding

Sharding é um tipo de particionamento de bases de dados que separa estas em fragmentos de dados independentes, pequenos, rápidos e fáceis de gerir. De uma forma mais técnica, *sharding* é uma técnica de partição horizontal. Na prática, este termo é utilizado para qualquer estratégia de partição de uma base de dados com o principal objetivo de a tornar mais fácil de gerir.

O conceito por de trás da técnica de *sharding* é o facto de que sendo o crescimento do tamanho de uma base de dados e do número de transações linear, o crescimento do tempo de resposta irá crescer não de forma linear mas sim exponencialmente.

Para além disto, uma base de dados de grandes dimensões exige máquinas com grande capacidade de processamento. Por outro lado, os fragmentos de dados podem ser distribuídos comodamente por servidores bastante menos dispendiosos.

Adicionalmente, esta possibilidade de alocação dos dados em diferentes máquinas permite também em alguns casos tirar partido da localização destas. Uma vez que, no caso de estudo é guardada informação relativa a clientes, uma possibilidade seria distribuir a informação destes tendo em conta a localização geográfica do cliente, fornecendo assim um serviço potencialmente melhor.

Em alguns casos, este processo pode revelar-se também bastante complexo se os dados não se encontrarem bem estruturados, dando assim origem a fragmentos de dados bastante difíceis de gerir.

Neste trabalho prático, para a implementação desta técnica de particionamento de bases de dados foi utilizada uma ferramenta *open-source* chamada `Citus Data`. Este produto de software é uma extensão para o PostgreSQL que distribui não só a base de dados em questão mas também as interrogações efetuadas a esta pelos multiplos nodos.

5.1 Configuração do serviço

Para este projeto, serão utilizadas 3 instâncias do PostgreSQL, uma do tipo *master* e as restantes do tipo *worker*. O *master* será responsável pela gestão dos fragmentos de dados bem como a distribuição das interrogações pelos diferentes nodos. Os *workers* apenas albergam os diferentes fragmentos de dados.

Antes de inicializar qualquer serviço, será necessário definir a permissões de acesso do serviço. Para isso, no ficheiro de configuração *postgresql.conf* será necessário introduzia a seguinte regra.

```
# Uncomment listen_addresses for the changes to take effect
listen_addresses = '*'
```

Adicionalmente, no ficheiro de configuração *pg_hba.conf* foi necessário adicionar uma regra que garantisse a autenticação dos *workers* como fidedignos.

```
# Allow unrestricted access to nodes in the local network. The following ranges
# correspond to 24, 20, and 16-bit blocks in Private IPv4 address spaces.
host    all             all             35.0.0.0/8              trust
```

Passando agora às configurações relacionadas com a ligação da extensão ao PostgreSQL, será necessário apenas adicionar a biblioteca *citus* através do ficheiro *postgresql.conf* bem como proceder à criação da extensão *citus* em cada um dos nodos recorrendo ao comando *psql*.

label

```
# preload citus extension
shared_preload_libraries = 'citus'
```

Comando SQL para a criação da extensão *citus*

```
CREATE EXTENSION citus;
```

Uma vez efetuadas todas as configurações acima em todas as máquinas (*master* e *workers*), as máquinas *workers* estão neste momento prontas e não necessitarão de mais nenhuma intervenção.

Continua-se então a configuração da máquina *master*.

Uma vez que, o *master* será responsável por efetuar toda a gestão dos fragmentos de dados, este terá de ter conhecer a localização das máquinas *worker*, mais concretamente, os seus endereços IP e a porta de acesso à instância do PostgreSQL.

Através do comando `SELECT * from master_add_node('35.231.246.171', 5432);` fornecido pela extensão Citus, é possível adicionar o IP e respetiva porta das máquinas a serem usadas como *workers*. É disponibilizado também o comando `SELECT * FROM master_get_active_worker_nodes();` que permite verificar a informação inserida sobre as máquinas *worker*, devolvendo o seguinte resultado.

```
tpcc-db=# SELECT * FROM master_get_active_worker_nodes();
  node_name    | node_port
---------------+-----------
35.237.39.148  |      5432
35.231.246.171 |      5432
(2 rows)
```

De salientar que o processo de adição dos endereços IP não foi incluido nos *scripts* automáticos uma vez que as máquinas na Google Cloud Platform não mantêm um IP fixo.

Uma vez verificada toda a informação introduzida, passemos à indicação ao PostgreSQL para a inicialização do processo de *sharding*. No que diz respeito à criação das tabelas na base de dados, estas devem ser criadas exatamente como se se tratasse de uma base de dados normal. De salientar que para o processo de sharding, é necessário que as tabelas previamente criadas não contenham qualquer tipo de informação.

Por forma a tornar as tabelas prontas a receber informação e que estes comecem imediatamente a ser distribuídos pelos diferentes nodos, recorre-se aos seguintes comandos.

```
SELECT master_create_distributed_table(
  'customer', 'c_w_id', 'hash'
);

SELECT master_create_worker_shards('customer', 16, 2);
```

Com o primeiro comando, é indicado ao PostgreSQL que a tabela deverá ser dividia em blocos pelos diversos nodos. O primeiro argumento desta função indica o nome da tabela, o segundo o atributo da tabela a utilizar na construção e pesquisa da zona de dados pretendida num determinado momento e finalmente, o último argumento indica que as partições serão organizadas seguindo uma estrutura *hash*. Relevante ainda que, após a execução deste comando, nenhuma informação foi enviada ainda para nenhum dos *workers*.

De salientar que, o atributo escolhido para a construção das partições terá de ser constituinte de todas as *constraints* do tipo `PRIMARY KEY` e `UNIQUE` pelo que procedeu-se à remoção da *constraint* unique inicialmente associada ao atributo *key* para que fosse possível a implementação do processo de *sharding*. Uma vez que, esta é uma limitação da estensão *Citus*, sem esta modificação no esquema das tabelas da base de dados, não seria de qualquer forma a implementação de *sharding*.

Com o comando seguinte, irão já ser criados os fragmentos de informação para a tabela indicada. No primeiro documento será indicado o nome da tabela, no segundo o número de fragmentos em que a tabela indicada no argumento anterior deverá ser dividida e finalmente o número de *workers* pelo qual os fragmentos devem replicados.

Encontra-se em anexo um ficheiro *SQL* que contém todos os comando executados no processo acima descrito. É possível observar-se neste ficheiro que os atributos selecionados para o processo de partição de cada uma das tabelas foram propositadamente escolhidos por forma a que os

dados associados a cada uma das *warehouses* se encontrem presentes próximos ou até no mesmo fragmento de dados.

Neste momento, é possível iniciar o processo de população da base de dados, sendo estes já distribuídos em fragmentos pelos diferentes nodos.

Durante o processo de carregamento dos dados foi possível verificar o aumento gradual dos tamanhos das tabelas fragmentadas nas máquinas através do seguinte comando.

```
SELECT
   relname as "Table",
   pg_size_pretty(pg_total_relation_size(relid)) As "Size",
   pg_size_pretty(pg_total_relation_size(relid) -
        pg_relation_size(relid)) as "External Size"
   FROM pg_catalog.pg_statio_user_tables
   ORDER BY pg_total_relation_size(relid) DESC;
```

Obtendo o seguinte resultado.

```
tpcc-db=# SELECT
   relname as "Table",
   pg_size_pretty(pg_total_relation_size(relid)) As "Size",
   pg_size_pretty(pg_total_relation_size(relid) - pg_relation_size(relid)) as "External Size"
   FROM pg_catalog.pg_statio_user_tables ORDER BY pg_total_relation_size(relid) DESC;
       Table       |  Size   | External Size
-------------------+---------+---------------
 stock_102124      | 249 MB  | 36 MB
 stock_102120      | 249 MB  | 36 MB
 stock_102134      | 207 MB  | 29 MB
 stock_102135      | 207 MB  | 29 MB
 stock_102131      | 207 MB  | 29 MB
 stock_102132      | 165 MB  | 23 MB
 stock_102130      | 165 MB  | 23 MB
 stock_102121      | 165 MB  | 23 MB
 stock_102127      | 165 MB  | 23 MB
 stock_102126      | 123 MB  | 17 MB
 stock_102123      | 123 MB  | 17 MB
 stock_102122      | 123 MB  | 17 MB
 stock_102133      | 123 MB  | 17 MB
 customer_102012   | 111 MB  | 17 MB
 customer_102008   | 111 MB  | 17 MB
 customer_102022   | 92 MB   | 14 MB
 customer_102019   | 92 MB   | 14 MB
 customer_102023   | 92 MB   | 14 MB
 stock_102129      | 82 MB   | 11 MB
 stock_102128      | 82 MB   | 11 MB
 order_line_102088 | 80 MB   | 30 MB
 order_line_102092 | 80 MB   | 30 MB
tpcc-db=# \q
[MarcoSilva@replication-worker ~]$
```

Figura 30: Demonstração da execução do processo de *sharding* da base de dados.

A obtenção de medições nesta arquitetura tornou-se bastante difícil uma vez que o executável *load.sh* leva entre 11 e 12 horas a carregar os dados para os nodos com um total de 600 clientes (60 *warehouses*, 10 clientes por *warehouse*).

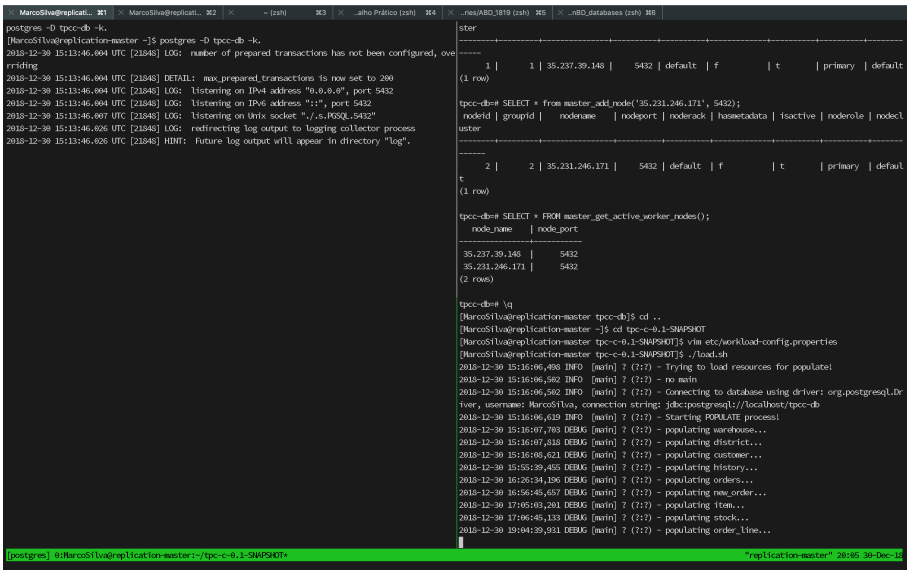


Figura 31: Demonstração da execução bastante demorada do *load.sh*.

De qualquer das formas, segundo a documentação da ferramenta, seriam esperados resultados 10 vezes superiores aos obtidos com uma base de dados singular. Como forma de prova para a realização do processo de *sharding*, foram sendo apresentadas ao longo desta secção demonstrações em diferentes fases do processo.

6 Conclusão

O processo de otimização do *benchmark* TPC-C envolveu diferentes estratégias, como o afinamento de configurações, acrescento de novas estruturas ou mesmo a alteração de outras. Sendo que todos estes componentes residiram num ambiente remoto (GCP), facilmente replicável, que levou a uma maior flexibilidade nos testes que permitiram chegar a algumas das conclusões descritas neste documento.

As interrogações do TPC-C apresentavam-se já bastantes otimizadas e foi feita uma análise detalhada que comprova isso. Já em relação aos parâmetros do PostgreSQL foi feita uma seleção dos que poderiam trazer benefícios ao tipo deste sistema. Neste ponto foram encontradas melhorias que apesar de bastante substanciais, são otimizações que noutra escala podem vir a revelar-se significativas.

A replicação com *streaming* introduziu uma nova frente de possibilidades no que toca à otimização do *benchmark* TPC-C e das interrogações analíticas. De facto, torna-se possível aplicar toda a otimização realizada verticalmente (e.g. vistas materializadas, índices, ...) mas personalizando consoante o tipo de carga que cada servidor vai sofrer. De realçar que o ganho no tempo de execução das interrogações analíticas foi em quase todos os casos de pelo menos um ordem de grandeza. Além disso, esta arquitetura permite escalabilidade através da inserção de novos servidores standby, disponibilidade caso o servidor principal vá abaixo um hot standby pode substituir e ainda permite a introdução de novas estratégias como o `pgpool` para realizar balanceamento de carga.

A arquitetura baseada em *sharding* não foi terminada neste projeto, contudo merece, sem dúvida, seguimento em futuras iterações do projeto, uma vez que permitirá escalar a memória, processamento e armazenamento através da adição de múltiplas máquinas. De realçar que esta estratégia encaixa na perfeição com a replicação por streaming, uma vez que é possível ter *sharding* no servidor principal para otimizar as ações que tenham escritas/leituras, mas ao mesmo tempo ter vários servidores *standby* para a carga mais analítica, ou seja, baseada em leituras. Assim, tirar-se-ia proveito das vantagens de ambas as soluções.

Concluindo, o processo de otimização da base de dados, possibilitada pelo TPC-C, foi conseguido e inúmeras melhorias foram abordadas tanto no *benchmark* como nas próprias interrogações analíticas. As estratégias de otimização horizontal abriram novas possibilidades. Algumas concretizadas, com consequentes melhorias nos resultados da configuração de referência. Outras apenas com uma potencial melhoria na *performance*, sendo que esta terá de ser explorada em desenvolvimentos futuros.

7 Anexos

7.1 init.sh

```
#! /bin/bash

initdb -D tpcc-db
sleep 5
postgres -D tpcc-db -k. &
sleep 5
createdb -h localhost tpcc-db
cd tpc-c-0.1-SNAPSHOT/etc/sql/postgresql/
for i in createtable.sql createindex.sql sequence.sql *0*;
do cat \$i | psql -h localhost tpcc-db; done
cd ../../../
sudo ./load.sh
```

7.2 dump.sh

```
#! /bin/bash
time pg_dump -h localhost -Fc tpcc-db > tpcc.dump
```

7.3 restore.sh

```
#! /bin/bash

#PRESERVAÇÃO DO FICHEIRO DE CONFIGURACAO DO POSTGRESQL
cp tpcc-db/postgresql.conf ~/

# REMOÇÃO BASE DE DADOS ANTERIOR
rm -r tpcc-db

# INICIALIZAÇÃO DA NOVA BD
initdb -D tpcc-db

echo '========== CHECKING IF POSTGRES PORT IS USED =========='
sudo netstat -anp | grep ':5432 '
echo '======================================================'

# POSTGRES EM BG
tmux new -d -s postgres

tmux send-keys -t postgres.0 "postgres -D tpcc-db -k." ENTER

echo '========== CREATEDB INITIALIZING IN ==================='
for i in {5..1}
do
        echo "\$i"
        sleep 1
done

createdb -h localhost tpcc-db
```

```
# RESTORE COM RECURSO AO FICHEIRO tpcc.dump
echo '========== RESTORE INTIALIZED ========================='
time pg_restore -h localhost -d tpcc-db -Fc -j 8 tpcc.dump

#PRESERVACAO DO FICHEIRO DE CONFIGURACAO DO POSTGRESQL
mv ~/postgresql.conf ~/tpcc-db/

# TERMINAR A SESSÃO TMUX QUE ESTA CORRER O POSTGRES
tmux send-keys -t postgres "C-c"
tmux kill-session -t postgres
```

7.4 to-bucket.sh

```
#! /bin/bash

if [ "\$#" -ne 1 ]; then
  echo "Usage: \$0 FILENAME" >&2
  exit 1
fi

gsutil cp \$1 gs://abd-tpcc-storage/
```

7.5 from-bucket.sh

```
#! /bin/bash

if [ "\$#" -ne 1 ]; then
  echo "Usage: \$0 FILENAME" >&2
  exit 1
fi

gsutil cp gs://abd-tpcc-storage/\$1 .
```

7.6 sharding_tables.sql

```
select master_create_distributed_table('customer', 'c_w_id', 'hash');
select master_create_distributed_table('district', 'd_w_id', 'hash');
select master_create_distributed_table('history', 'h_w_id', 'hash');
select master_create_distributed_table('item', 'i_id', 'hash');
select master_create_distributed_table('new_order', 'no_w_id', 'hash');
select master_create_distributed_table('order_line', 'ol_w_id', 'hash');
select master_create_distributed_table('orders', 'o_w_id', 'hash');
select master_create_distributed_table('stock', 's_w_id', 'hash');
select master_create_distributed_table('warehouse', 'w_id', 'hash');

SELECT master_create_worker_shards('customer', 16, 2);
SELECT master_create_worker_shards('district', 16, 2);
SELECT master_create_worker_shards('history', 16, 2);
SELECT master_create_worker_shards('item', 16, 2);
SELECT master_create_worker_shards('new_order', 16, 2);
SELECT master_create_worker_shards('order_line', 16, 2);
SELECT master_create_worker_shards('orders', 16, 2);
```

```
SELECT master_create_worker_shards('stock', 16, 2);
SELECT master_create_worker_shards('warehouse', 16, 2);
```

7.7 init_sharding.sql (master)

```
rm -r tpcc-db

initdb -D tpcc-db

cp postgresql.conf tpcc-db/

cp pg_hba.conf tpcc-db/

# POSTGRES EM BG
tmux new -d -s postgres

tmux send-keys -t postgres.0 "postgres -D tpcc-db -k." ENTER

sleep 1

createdb -h localhost tpcc-db

cat sharding_tables.sql | psql -h localhost tpcc-db

cd tpc-c-0.1-SNAPSHOT/etc/sql/postgresql/

for i in createtable.sql createindex.sql sequence.sql *0*; do cat \$i
                | psql -h localhost tpcc-db; done
```

7.8 init_sharding.sql (worker)

```
rm -r tpcc-db

initdb -D tpcc-db

cp postgresql.conf tpcc-db/

cp pg_hba.conf tpcc-db/

# POSTGRES EM BG
tmux new -d -s postgres

tmux send-keys -t postgres.0 "postgres -D tpcc-db -k." ENTER

sleep 1

createdb -h localhost tpcc-db

cat sharding_tables.sql | psql -h localhost tpcc-db
```

Referências

[1] DB-Engines. Db-engines ranking. https://db-engines.com/en/ranking, 2018. [Online; acedido a 30-Dezembro-2018].

[2] TPC. Tpc-c. http://www.tpc.org/tpcc/, 2018. [Online; acedido a 30-Dezembro-2018].

[3] PostgreSQL. 18.5.2. checkpoints. https://www.postgresql.org/docs/9.5/runtime-config-wal.html#GUC-MAX-WAL-SIZE, 2018. [Online; acedido a 30-Dezembro-2018].