

Universidade do Minho
Escola de Engenharia

Mestrado em Engenharia Informática

Otimização de uma base de dados

Administração de Bases de Dados

Grupo 9

Ano Letivo de 2021/2022

PG47164, Eduardo Coelho
PG47578, Pedro Veloso
PG47089, Carlos Preto
PG47398, Leonardo Marreiros
PG47559, Pedro Fernandes

Gualtar, Maio de 2022

Conteúdo

1	Introdução	5
2	Configuração <i>benchmark</i> TPC-C	6
2.1	Ambiente de teste	6
2.2	Obtenção do número de <i>warehouses</i>	6
2.3	Obtenção do número de clientes	6
2.4	Configuração base da base de dados	9
3	Otimização das configurações do PostgreSQL	10
3.1	Parâmetros de utilização de recursos	10
3.1.1	Shared-buffers [6]	10
3.1.2	Work_mem [17]	10
3.1.3	Huge Pages [7]	11
3.2	Parâmetros do WAL - <i>Write-ahead log</i>	14
3.2.1	Fsync [5]	14
3.2.2	Synchronous Commit [12]	14
3.2.3	wal_level [13] [14]	15
3.2.4	wal_sync_method [15]	16
3.2.5	full_pages_writes [10]	17
3.2.6	wal_buffers[11]	17
3.2.7	commit_delay [18]	17
3.2.8	commit_sibling [16]	18
3.2.9	checkpoint_timeout	19
3.2.10	checkpoint_completion_target	20
3.2.11	checkpoint_flush_after	21
3.2.12	checkpoint_warning	22
3.2.13	max_wal_size	22
3.2.14	min_wal_size [19]	23
3.2.15	archive_mode [20]	24
3.2.16	archive_command [21]	24
3.2.17	archive_timeout [22]	24
3.3	Parâmetros de Isolamento	25
3.4	Combinações	25
3.4.1	Combinações de <i>settings</i>	25
3.4.2	Combinações de <i>checkpoints</i>	26
3.4.3	Combinações de <i>archiving</i>	27
3.4.4	Combinação geral	27
4	Otimização das interrogações analíticas	28
4.1	Interrogação analítica A.1	28
4.1.1	Índices	30
4.1.2	Vista Materializada 1	33
4.1.3	Vista Materializada 2	35
4.1.4	Vista Materializada 2 com índices	36
4.1.5	Paralelismo	37
4.2	Interrogação analítica A.2	39
4.2.1	Índices	41
4.2.2	Vista Materializada 1	43
4.2.3	Vista Materializada 2	45

4.2.4	Vista Materializada 2 com Índice	46
4.2.5	Vista Materializada 3	47
4.2.6	Paralelismo	47
4.3	Interrogação analítica A.3	48
4.3.1	Índices	49
4.3.2	Vista Materializada 1	51
4.3.3	Vista Materializada 1 com índice	53
4.3.4	Paralelismo	54
4.4	Interrogação analítica A.4	54
4.4.1	Vista Materializada 1	56
4.4.2	Vista Materializada 1 com índice	57
4.4.3	Vista Materializada 2	57
4.4.4	Vista Materializada 2 com índices	58
4.4.5	Paralelismo	59
5	Replicação da base de dados [23]	60
5.1	Replicação lógica	60
6	Conclusão	62

Índice de figuras

2.1	Throughput por número de clientes	7
2.2	Tempo de resposta por número de clientes	8
2.3	Taxa de aborto por número de clientes	8
2.4	Resultados das métricas da configuração base	9
3.1	Utilização de recursos com a utilização de Work_mem = 1024MB	11
3.2	Memória Virtual [7]	11
3.3	Tabela de paginação [7]	12
3.4	Processo de <i>lookup</i> da um mapeamento de uma página [8]	12
3.5	Métricas com <i>fsync Off</i>	13
3.6	Métricas com <i>fsync Off</i>	14
3.7	Comparação entre os vários valores do synchronous commit	15
3.8	Valor <i>fsync_writethrough</i> indisponível na plataforma utilizada	16
3.9	Comparação entre os vários valores de <i>wal_buffers</i>	17
3.10	Comparação entre os vários valores de <i>commit_delay</i>	18
3.11	Comparação entre os vários valores de <i>commit_sibling</i>	19
3.12	Curva de escalabilidade <i>checkpoint timeout</i>	20
3.13	Curva de escalabilidade <i>checkpoint completion target</i>	21
3.14	Curva de escalabilidade <i>checkpoint_warning</i>	22
3.15	Curva de escalabilidade <i>Max_wal_size</i>	23
3.16	Curva de escalabilidade <i>Min_wal_size</i>	23
3.17	Métricas com combinações de <i>settings</i>	26
3.18	Métricas com combinações de <i>checkpoints</i>	26
4.1	Lista de índices já presentes na base de dados	28
4.2	Resultado da interrogação analítica A1	28
4.3	Resultado do <i>explain analyze</i> da interrogação analítica A1	29
4.4	Estatísticas do plano de execução original da interrogação analítica A1	29
4.5	Estatísticas do plano de execução original da interrogação analítica A1	30
4.6	EXPLAIN ANALYZE da interrogação analítica A1 com uso de índices	31

4.7	EXPLAIN ANALYZE interrogação analítica A1 com uso de índices nas colunas do GROUP BY	31
4.8	Estatísticas do plano de execução da interrogação analítica A1 com uso de índices nas colunas do GROUP BY	32
4.9	Estatísticas do plano de execução da interrogação analítica A1 com uso de índices nas colunas do GROUP BY	32
4.10	EXPLAIN ANALYZE interrogação analítica A1 com uso de uma vista materializada	33
4.11	Estatísticas do plano de execução da interrogação analítica A1 com uso de uma vista materializada	34
4.12	Estatísticas do plano de execução da interrogação analítica A1 com uso de uma vista materializada	34
4.13	Resultado da interrogação analítica A1 com uso de uma vista materializada	34
4.14	Resultado da interrogação analítica A1 com uso da nova vista materializada	35
4.15	Estatísticas do plano de execução da interrogação analítica A1 com uso da nova vista materializada	36
4.16	Estatísticas do plano de execução da interrogação analítica A1 com uso da nova vista materializada	36
4.17	Resultado da interrogação analítica A1 com uso da nova vista materializada	36
4.18	Resultado da interrogação analítica A1 com uso da nova vista materializada e de índices	37
4.19	Estatísticas do plano de execução da interrogação analítica A1 com uso da nova vista materializada e índices	37
4.20	Estatísticas do plano de execução da interrogação analítica A1 com uso da nova vista materializada e índices	37
4.21	Plano de execução da interrogação analítica A1 com 1 <i>worker</i>	38
4.22	Plano de execução da interrogação analítica A1 com 4 <i>workers</i>	38
4.23	Plano de execução da interrogação analítica A1 com 6 <i>workers</i>	39
4.24	Curva de escalabilidade do paralelismo da interrogação analítica A1	39
4.25	Execução da <i>query 2</i> original	40
4.26	Visualização da <i>query 2</i> original	41
4.27	Execução da <i>query 2</i> com índices criados	42
4.28	Visualização da <i>query 2</i> com índices criados	42
4.29	Execução da <i>query 2</i> com índices criados com <i>sequential scan</i> desativado	43
4.30	Visualização da <i>query 2</i> com índices criados com <i>sequential scan</i> desativado	43
4.31	Execução da <i>query 2</i> com índices e Vista Materializada 1	44
4.32	Visualização da <i>query 2</i> com índices e Vista Materializada 1	45
4.33	Execução da <i>query 2</i> com Vista Materializada 2	46
4.34	Visualização da <i>query 2</i> com Vista Materializada 2	46
4.35	Execução da <i>query 2</i> com Vista Materializada 3	47
4.36	Curva de escalabilidade do paralelismo da interrogação analítica A2	48
4.37	EXPLAIN ANALYZE da interrogação analítica A3	49
4.38	Estatísticas do plano de execução original da interrogação analítica A3	49
4.39	EXPLAIN ANALYZE da interrogação analítica A3 com índices	50
4.40	Estatísticas do plano de execução da interrogação analítica A3 com índices	50
4.41	EXPLAIN ANALYZE da interrogação analítica A3 com índices e <i>seqscan off</i>	51
4.42	Estatísticas do plano de execução da interrogação analítica A3 com índices e <i>seqscan off</i>	51
4.43	EXPLAIN ANALYZE da interrogação analítica A3 com vista Materializada 1	52
4.44	Estatísticas do plano de execução da interrogação analítica A3 com vista Materializada 1	52
4.45	EXPLAIN ANALYZE da interrogação analítica A3 com vista Materializada 1	52
4.46	Estatísticas do plano de execução da interrogação analítica A3 com vista Materializada 1	53
4.47	EXPLAIN ANALYZE da interrogação analítica A3 com índice na vista Materializada 1 .	53

4.48	Estatísticas do plano de execução da interrogação analítica A3 com índice na vista Materializada 1	53
4.49	Curva de escalabilidade do paralelismo da interrogação analítica A3	54
4.50	Estatísticas do plano de execução original da interrogação analítica A4	55
4.51	Estatísticas do plano de execução orginal da interrogação analítica A4	56
4.52	Estatísticas do plano da primeira otimização da interrogação analítica A4	57
4.53	Estatísticas do plano da segunda otimização da interrogação analítica A4	58
4.54	Tempos de Execução da Query A4	59
5.1	Replicação lógica de uma Base de Dados	60
5.2	Publicação "pub1" criada na base de dados a replicar	60
5.3	Replicação da base de dados	61

Índice de tabelas

2.1	Determinação do número de <i>warehouses</i>	6
2.2	Determinação do número de clientes	7
3.1	Resultados da alteração do parâmetro <i>shared_buffers</i>	10
3.2	Resultados da alteração do parâmetro <i>Work_mem</i>	10
3.3	Resultados da alteração do parâmetro <i>huge_pages</i>	13
3.4	Resultados da alteração do parâmetro <i>fsync</i>	14
3.5	Resultados da alteração do parâmetro <i>synchronous commit</i>	15
3.6	Resultados da alteração do parâmetro <i>wal_level</i>	16
3.7	Resultados da alteração do parâmetro <i>wal_sync_method</i>	16
3.8	Resultados da alteração do parâmetro <i>fullpages</i>	17
3.9	Resultados da alteração do parâmetro <i>wal_buffers</i>	17
3.10	Resultados da alteração do parâmetro <i>commit_delay</i>	18
3.11	Resultados da alteração do parâmetro <i>commit_sibling</i>	18
3.12	Resultados da alteração do parâmetro <i>checkpoint timeout</i>	19
3.13	Resultados da alteração do parâmetro <i>checkpoint completion target</i>	20
3.14	Resultados da alteração do parâmetro <i>checkpoint flush_after</i> com <i>shared_buffers</i> igual a 128MB	21
3.15	Resultados da alteração do parâmetro <i>checkpoint flush_after</i> com <i>shared_buffers</i> igual a 2048MB	21
3.16	Comparação do <i>checkpoint_warning</i>	22
3.17	Comparação do <i>max_wal_size</i>	22
3.18	Comparação do <i>min_wal_size</i>	23
3.19	Comparação Opções <i>archive_mode</i>	24
3.20	Comparação <i>archive_command</i>	24
3.21	Comparações <i>archive_timeout</i> no Modo <i>On</i>	24
3.22	Comparações <i>archive_timeout</i> no Modo <i>Always</i>	24
3.23	Níveis de Isolamento	25
3.24	Resultados com combinação de <i>settings</i>	25
3.25	Resultados com combinação de <i>checkpoints</i>	26
3.26	Resultados com combinação geral	27
3.27	Resultados finais com combinação geral	27
4.1	Tempo de execução com diferente número de <i>workers</i> - query 2	48
4.2	Tempo de execução com diferente número de <i>workers</i> - query 3	54

1 Introdução

Ao longo do presente relatório pretende-se descrever os processos realizados para melhorar o desempenho do *benchmark* TPC-C.

Este *benchmark*, pretende simular o sistema operacional de uma cadeia de lojas, suportando desta forma as operações diárias de gestão de vendas e stocks. Assim, contém um conjunto de interrogações analíticas que deverão ser executadas constantemente, e como tal deverão apresentar uma boa performance.

Na parte inicial do trabalho será abordada a instalação do *benchmark*, bem como a escolha de diferentes parâmetros base - número de *warehouses* e clientes - utilizados ao longo dos testes realizados. A escolha destes parâmetros deve seguir as características da máquina a utilizar ao longo do projeto, permitindo que, por exemplo, a quantidade de RAM disponível não seja suficiente para conter toda a base de dados utilizada pelo *benchmark*.

Posteriormente, serão abordadas as estratégias utilizadas para melhorar o desempenho do sistema de gestão de base de dados utilizada - *PostgreSQL*. Assim serão referidos os vários testes realizados, na definição dos parâmetros disponíveis e os resultados obtidos. No final dos vários testes, aborda-se ainda a performance obtida com o conjunto dos melhores parâmetros obtidos anteriormente, de modo a obter a melhor combinação possível.

Com a definição dos parâmetros concluída, observou-se as várias *queries* analíticas de modo a realizar a sua otimização. Nesta fase do trabalho, recorreu-se à redundância dos dados, com a criação de índices ou vistas materializadas, de modo a obter uma melhor performance. Para além disto, testou-se a utilização de paralelismo, com a definição de vários *workers* simultâneos, para testar a diferença de performance relativamente à utilização de apenas um *worker*.

No final do trabalho, ainda se irá abordar o processo de replicação da base de dados. Este processo poderá permitir a posterior utilização de diferentes tipos de algoritmos que permitam a melhoria do desempenho obtido.

2 Configuração *benchmark* TPC-C

Começamos por instalar e configurar o *benchmark* TPC-C com vista a obter uma configuração de referência em termos de *hardware*, número de *warehouses* e clientes.

2.1 Ambiente de teste

O primeiro problema que tivemos de ultrapassar foi definir a configuração de *hardware* para as máquinas virtuais a utilizar. Após alguma discussão entre os elementos do grupo e opinião dos docentes, além de testes alterando o número de clientes e *warehouses* obteve-se a configuração inicial.

A nível de núcleos e memória, optamos por utilizar 4vCPU's N1 e 8 GB respetivamente pois permitia uma boa margem para optimizações futuras e não utilizar uma quantidade exagerada de recursos monetários.

Em termos de disco, inicialmente pensamos utilizar um SSD de 50GB pois parecia suficiente para englobar a base de dados mesmo com um número elevado de *warehouses*. No entanto, ao analisar a documentação da *google cloud*, observou-se que a velocidade de disco é influenciada pelo seu tamanho. Assim, optou-se por utilizar um SSD de 500GB, que apresentava uma velocidade suficiente para a realização de testes.

Por fim, recorreu-se à instalação do *benchmark* TPC-C e respetiva configuração inicial. Assim, foi necessário garantir a criação de uma imagem da máquina com o estado básico da base de dados mesmo depois de a máquina ser apagada. Para tal, o grupo recorreu à criação de uma imagem de máquina disponibilizada na *google cloud*. Esta imagem, permitiu a criação de várias máquinas com o estado inicial da base de dados, sem manter a uma imagem ativa e os custos a tal associados, além de poupar o tempo de povoar a base de dados.

2.2 Obtenção do número de *warehouses*

Visto que iremos utilizar 8GB de RAM optou-se por escolher um tamanho da base de dados um pouco maior que este número de modo a obrigar a máquina a não poder colocar simplesmente toda a base de dados em RAM. Assim, como podemos observar na Tabela 2.1, foram realizados testes para diferentes números de *warehouses* obtendo-se o seu espaço ocupado. De modo a ir de acordo com o objetivo primeiramente mencionado, decidiu-se por optar na utilização de 96 *warehouses*.

Número de Warehouses	Ocupação da Base Dados (MB)
2	257
4	457
8	757
16	1740
32	3490
64	7316
96	10297
128	13718

Tabela 2.1: Determinação do número de *warehouses*

2.3 Obtenção do número de clientes

De seguida, passamos para a definição do número de clientes a utilizar. Para tal, tivemos em atenção métricas de desempenho, nomeadamente o *Throughput*, *Response Time*, *Abort Rate* e a carga do CPU da máquina (obtida com recurso à monitorização da *google cloud*). Para efeitos de teste, de modo a ter uma maior consistência de resultados, sempre que mudávamos o nº de clientes, criávava-mos uma

máquina nova a partir de uma imagem de instância. Em todos os testes foram utilizados 10 minutos de *measurement time*.

Nº Clientes	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
10	887.2131204	0.021334403	0.07195352
20	826.0099572	0.021495873	0.150993732
30	882.3758282	0.019022169	0.210887392
40	801.0430428	0.019532538	0.276624121
50	846.1837737	0.017573941	0.317745826
60	810.6915746	0.01755589	0.352526105
70	783.1620636	0.017245375	0.388082964
80	763.203401	0.017140775	0.409801034
90	779.0397169	0.016152309	0.432592454
100	677.6306795	0.017630294	0.46300284

Tabela 2.2: Determinação do número de clientes

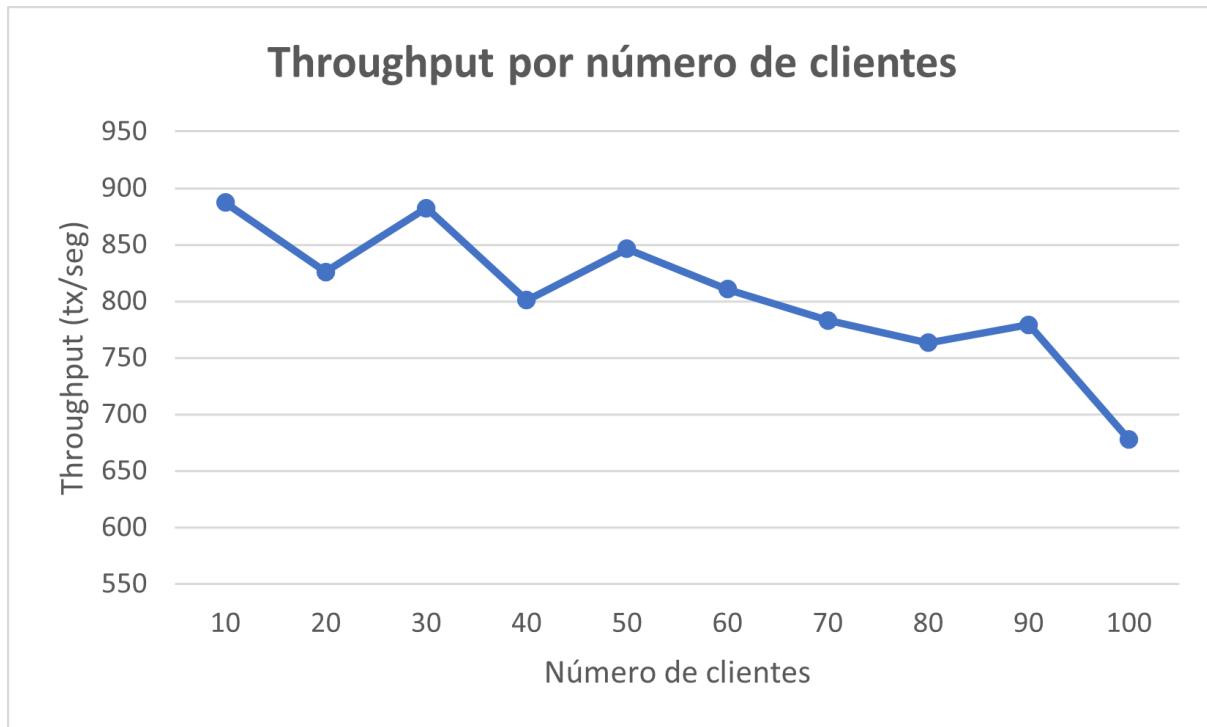


Figura 2.1: *Throughput* por número de clientes

Tempo de resposta por número de clientes

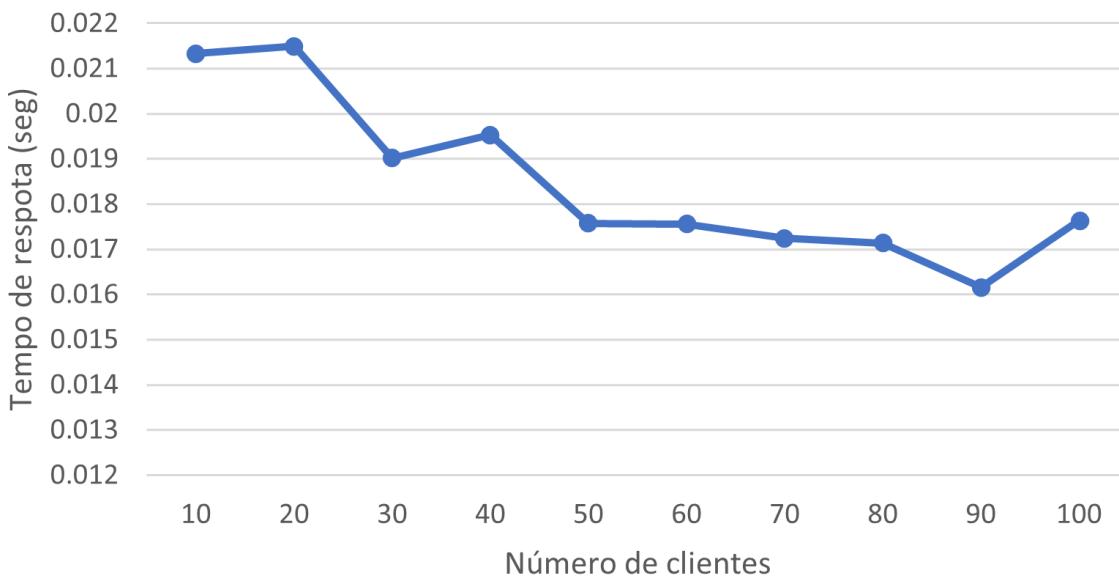


Figura 2.2: Tempo de resposta por número de clientes

Taxa de aborto por número de clientes

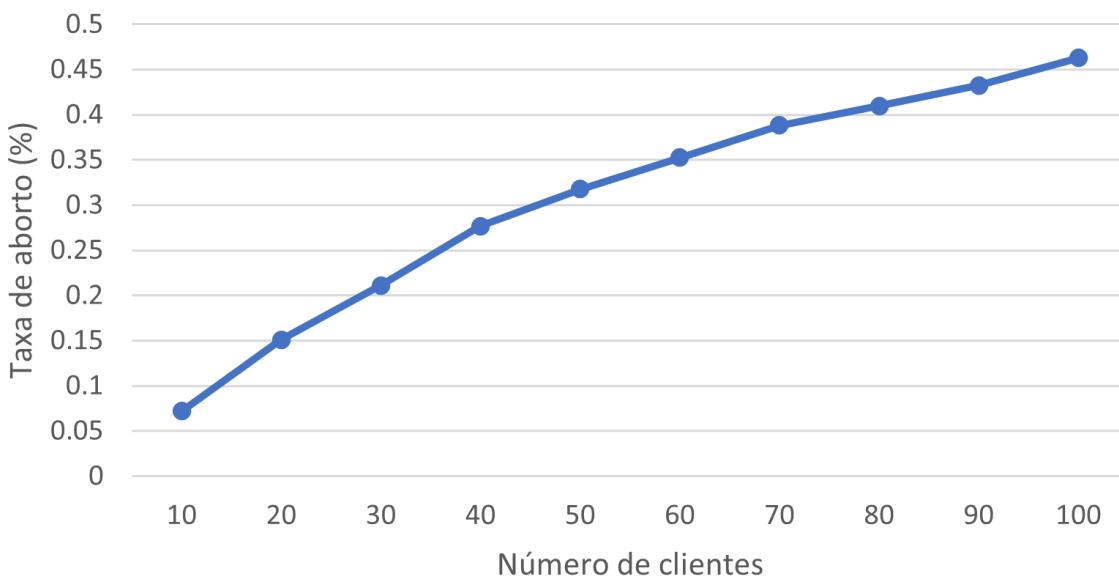


Figura 2.3: Taxa de aborto por número de clientes

Após uma cuidada análise destes dados, decidimos definir o número de clientes a 50, uma vez que proporciona um equilíbrio entre as várias métricas, mais especificamente, um alto *Throughput*, um tempo de resposta baixo, e uma taxa de aborto aceitável (figura 2.4).

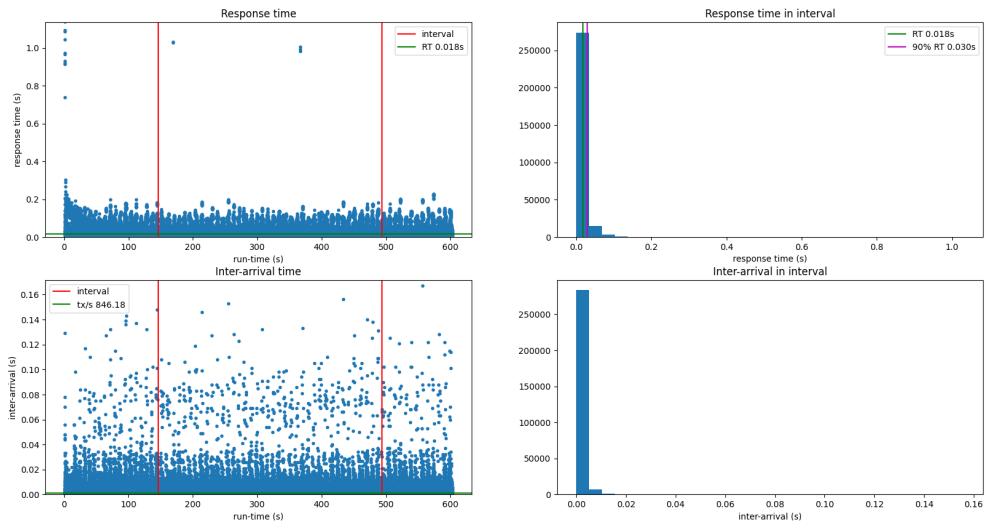


Figura 2.4: Resultados das métricas da configuração base

2.4 Configuração base da base de dados

Assim, durante a realização de todos os testes, utilizaremos uma máquina N1 da *google cloud* com 4 vCPUs, 8GB de RAM e 500GB de disco SSD, e a configuração base para a nossa base de dados tem 96 *warehouses* e 50 clientes.

3 Otimização das configurações do PostgreSQL

3.1 Parâmetros de utilização de recursos

3.1.1 Shared-buffers [6]

O *postgreSQL* utiliza *double buffering*, ou seja, utiliza o seu próprio *buffer* interno (*shared_buffers*) e o *buffer* do *kernel*. Este parâmetro define quanta memória do sistema será dedicada para *cache* do *postgreSQL*. De modo a garantir uma maior compatibilidade com os sistemas, por defeito, este parâmetro está pré-definido para o baixo valor de 128MB. Tendo em conta que a nossa base de dados ocupa aproximadamente 10GB, não queremos que todos dados caibam em *cache* na memória, pelo que iremos aumentar este *buffer* para 2048MB (aproximadamente 20% do espaço ocupado pela base de dados).

É esperado que aumentar este valor melhore a performance da base dados, principalmente quando se tratam de operações de leitura, uma vez que, nestes casos, a base de dados vai ter de ir ler ao disco menos vezes. No entanto, quando se tratam de operações de escrita, a performance pode ser prejudicada, uma vez que todo o conteúdo deste *buffer* tem de ser processado.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
2048MB	1004.2130647969938	0.015302789605930013	0.29552468515675856
128MB	846.1837737	0.017573941	0.317745826

Tabela 3.1: Resultados da alteração do parâmetro shared_buffers

3.1.2 Work_mem [17]

Este parâmetro, permite definir um valor máximo de memória a ser utilizada no espaço de trabalho de uma *query*. A definição deste parâmetro torna-se complexa na medida em que a sua configuração pode melhorar a performance obtida, uma vez que por exemplo o seu aumento pode diminuir o número de vezes em que é necessário escrever dados para ficheiros temporários, porém, a sua diminuição também pode originar melhores resultados, ao impedir a ocorrência de erros *out of memory*.

Como valor *default* este parâmetro utiliza 4MB, contudo é possível determinar um valor entre os 64kB e os 2147483647kB. Desta forma, realizaram-se testes para os seguintes valores:

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
64KB	790.6311568905326	0.0187981638340893	0.32117313627730293
4MB	771.5726286384586	0.019191800018585635	0.32249510176739316
8MB	776.5079310561958	0.01930892624796813	0.31384991249279204
64MB	802.0366412823591	0.018676339484234982	0.3134430794030726
512MB	836.7319896182313	0.01787229274036128	0.3167846642361519
1024MB	857.3200530523735	0.017548507923991166	0.31004679899485904

Tabela 3.2: Resultados da alteração do parâmetro Work_mem

Apesar de a utilização de um valor incremental para este parâmetro ter melhorado a performance visualizada, é possível que a utilização de valores elevados tornem-se inviáveis, na medida em que o consumo de RAM é excessivo, tal como se observa na figura 3.1. Para o teste apresentado de seguida, utilizou-se o parâmetro *Work_mem* com o valor 1024MB e executou-se seis *queries* A.2 simultâneas. Neste teste, visualizou-se um consumo excessivo de RAM (apenas 123MB dos 7948.4MB de RAM disponíveis se encontrava livre) com a utilização de apenas seis *queries*, tornando assim inviável, valores mais elevados para soluções que podem exigir centenas de *queries* simultâneas.

Figura 3.1: Utilização de recursos com a utilização de Work_mem = 1024MB

3.1.3 Huge Pages [7]

O Linux gera a memória do sistema com paginação. Isto significa que, ao ler e escrever na memória, o sistema operativo utiliza páginas, ou seja, mesmo que só precisemos de escrever 1 byte de dados para o disco, temos de escrever todos os dados da página para o disco.

De modo a perceber como é que a utilização de *Huge Pages* pode afetar a performance da base de dados então é preciso perceber como é que o Linux lida com a memória.

Todas as aplicações (e o próprio sistema operativo) correm em memória virtual. Assim, cada processo tem a impressão de que está a trabalhar sobre secções de memória grandes e contíguas.

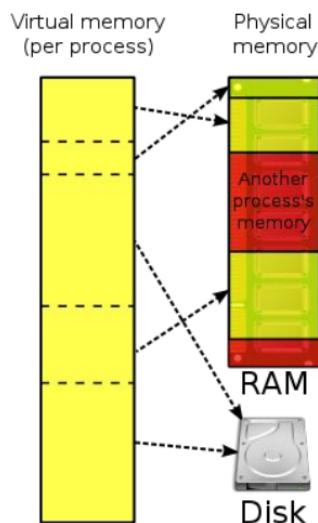


Figura 3.2: Memória Virtual [7]

Esta memória virtual funciona como uma abstração da memória física, na medida em que o sistema operativo a mapeia, utilizando uma tabela de paginação.

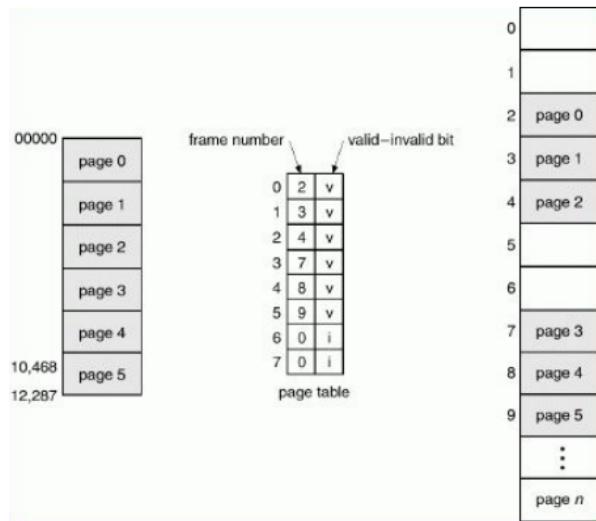


Figura 3.3: Tabela de paginação [7]

A tradução de endereços virtuais para endereços físicos é feita por um componente do processador, a unidade de gestão de memória (*MMU - Memory Management Unit*). A *MMU* utiliza uma *cache* para guardar o mapeamento das páginas utilizadas recentemente, chamada *TLB* (*Translation look-aside buffer*). Assim, sempre que é necessário aceder a uma página, o processador começa por procurar o mapeamento da memória virtual dessa página na *TLB*. Caso encontre esta página (*TLB hit*), então retorna o seu endereço físico (resultando num único acesso à memória). Caso contrário (*TLB miss*), é preciso percorrer toda a tabela de paginação para procurar pelo mapeamento desta página (resultando em dois acessos à memória).

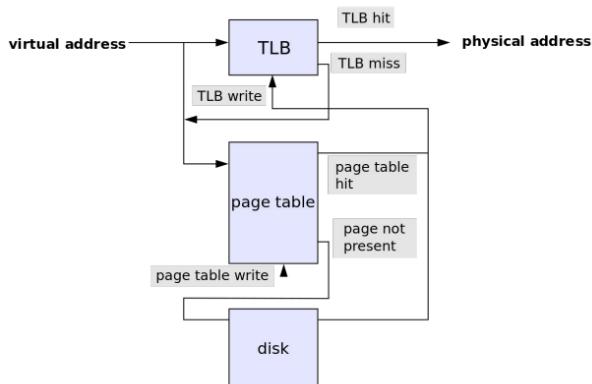


Figura 3.4: Processo de *lookup* da um mapeamento de uma página [8]

A *TLB*, normalmente, só tem capacidade para umas centenas de entradas de páginas. De modo a aumentar a eficiência desta *cache*, isto é, diminuir o número de *misses*, teríamos de aumentar a sua capacidade (dependente do *hardware*) ou aumentar o tamanho das páginas, o que reduz o número de páginas a mapear.

De modo a ativar a utilização de *Huge Pages* de modo eficiente, precisamos de calcular qual o uso de memória máximo da nossa base de dados. Para tal, iremos utilizar os seguintes comandos (após correr o *workload*) para obter este valor:

```
nyblen2000@benjamim:/home/ABD/tpc-c-0.1-SNAPSHOT$ ps -ef|grep postgres
postgres      690      1  0 13:18 ?          00:00:00 /usr/lib/postgresql/14/bin/postgres
-D /var/lib/postgresql/14/main -c config_file=/etc/postgresql/14/main/postgresql.conf
```

```

postgres      695      690  2 13:18 ?
postgres      696      690  0 13:18 ?
postgres      697      690  1 13:18 ?
postgres      698      690  0 13:18 ?
postgres      699      690  0 13:18 ?
postgres      700      690  0 13:18 ?
00:00:25 postgres: 14/main: checkpointer
00:00:05 postgres: 14/main: background writer
00:00:12 postgres: 14/main: walwriter
00:00:00 postgres: 14/main: autovacuum launcher
00:00:01 postgres: 14/main: stats collector
00:00:00 postgres: 14/main: logical replication
launcher
nyblen2+    4336     1812  0 13:35 pts/0    00:00:00 grep --color=auto postgres
nyblen2000@benjamim:/home/ABD/tpc-c-0.1-SNAPSHOT$ grep ^VmPeak /proc/690/status
VmPeak:    318348 kB

```

Como podemos observar, o *postgreSQL* utilizou 318348kB de memória. Após confirmar com o comando ”*grep Huge /proc/meminfo*” que o tamanho pré-definido das páginas **Huge** é 2048kB, podemos calcular o número de páginas que o *postgreSQL* vai utilizar com a fórmula:

$$N = VmPeak/tamanhoPaginaHuge \quad (1)$$

Neste caso, o *postgreSQL* necessita de aproximadamente 160 páginas *Huge*. Após mudar a configuração do *kernel* para utilizar 160 páginas *Huge* e alterar o parâmetro ***huge_pages*** do *postgreSQL* para ***on***, reiniciámos o *kernel* e o serviço do *postgreSQL* para aplicar as alterações.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
On	794.1530272711353	0.018537707371564294	0.324528652046191
Off	846.1837737	0.017573941	0.317745826

Tabela 3.3: Resultados da alteração do parâmetro *huge_pages*

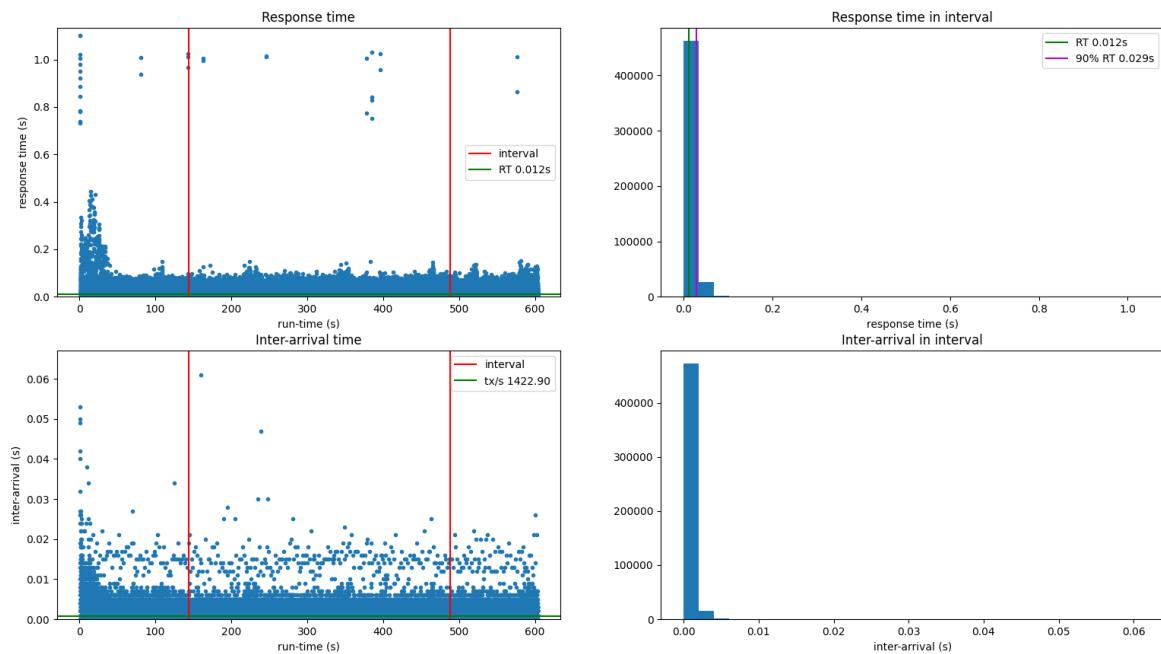


Figura 3.5: Métricas com *fsync Off*

Como podemos observar, os resultados foram o oposto do esperado. O *throughput* diminuiu e o tempo de resposta e a taxa de aborto pioraram ligeiramente. Após uma cuidada análise de alguns estudos [9], chegámos à conclusão que esta degradação da performance deve-se ao ***overhead*** introduzido na gestão de *misses* na *cache TLB*, quando a base de dados está a correr num ambiente virtual ao invés de um ambiente nativo dedicado.

3.2 Parâmetros do WAL - Write-ahead log

3.2.1 Fsync [5]

Este parâmetro, no seu valor *default (on)*, utiliza a chamada ao sistema *fsync()* ou outros métodos equivalentes para garantir que quaisquer *updates* são escritos fisicamente no disco. Ora, esta garantia permite à base de dados recuperar de um estado inconsistente depois de uma falha no sistema ou hardware.

Alterar este parâmetro para *off* pode melhorar a performance da base de dados uma vez que, para cada *update*, o cliente não precisa de esperar que a chamada ao sistema *fsync()* confirme a escrita no disco. No entanto, em sistemas reais, este parâmetro deve ser deixado inalterado para evitar a corrupção de dados.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
Off	1422.9037418096202	0.011936688483918618	0.15425218929295545
On	846.1837737	0.017573941	0.317745826

Tabela 3.4: Resultados da alteração do parâmetro *fsync*

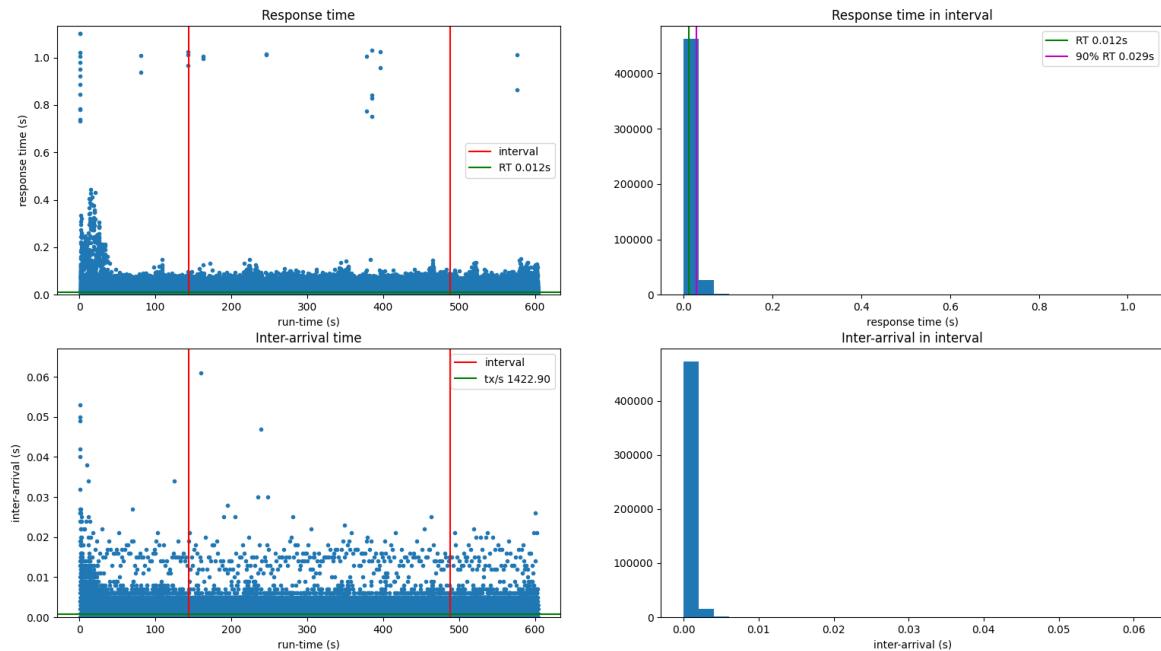


Figura 3.6: Métricas com *fsync Off*

3.2.2 Synchronous Commit [12]

O *synchronous commit*, é um parâmetro disponível no *postgreSQL* que permite identificar a quantidade do processamento de dados que deve estar completo, antes do servidor da base de dados retornar uma indicação de sucesso ao cliente.

Para especificar este parâmetro existem os seguintes valores: *on*, *off*, *remote_write*, *remote_apply* e *local*.

Ao utilizar este parâmetro com o valor *on*, o comportamento evidenciado é caracterizado pela espera da escrita do *Wall record* no disco, antes de notificar o cliente com a informação de sucesso. Porém, ao utilizar o valor *off* é evidenciado um *delay* entre o período em que a transação está segura contra a existência de falhas no servidor e o período em que o cliente foi notificado com a informação de sucesso. A utilização do valor *off*, ao contrário do parâmetro *fsync* não impõe um risco elevado para a consistência

da base de dados, na medida em que a ocorrência de falhas resulta na perda de transações efetuadas recentemente. Estas perdas, implicam obter um estado da base de dados igual ao observado no caso em que essas transações fossem abortadas corretamente.

Assim, desativar o *synchronous commit* torna-se benéfico em situações que a preocupação com a *performance* é superior com a certeza absoluta na realização de uma transação. Ao observar a tabela 3.5 é possível evidenciar um ganho na taxa de transferência ao desativar este atributo.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
Off	1144.6929744802082	0.014486506591622034	0.16558866477858528
On	867.5524312341391	0.017243370553050707	0.3137891414490988
local	811.7974226286893	0.018315663637744166	0.31974140664789596
remote_write	802.5827413718282	0.018450123264426205	0.32288648172102064
remote_apply	772.522386644718	0.019061203770848443	0.32608303384809706

Tabela 3.5: Resultados da alteração do parâmetro *synchronous commit*

Por outro lado, a distinção dos valores para os parâmetros *remote_write*, *remote_apply*, *local* e *on* não apresenta uma grande variação. Esta característica dá-se ao facto do parâmetro *synchronous_standby_names* estar definido como uma lista vazia, tornando a distinção entre eles irrelevante.

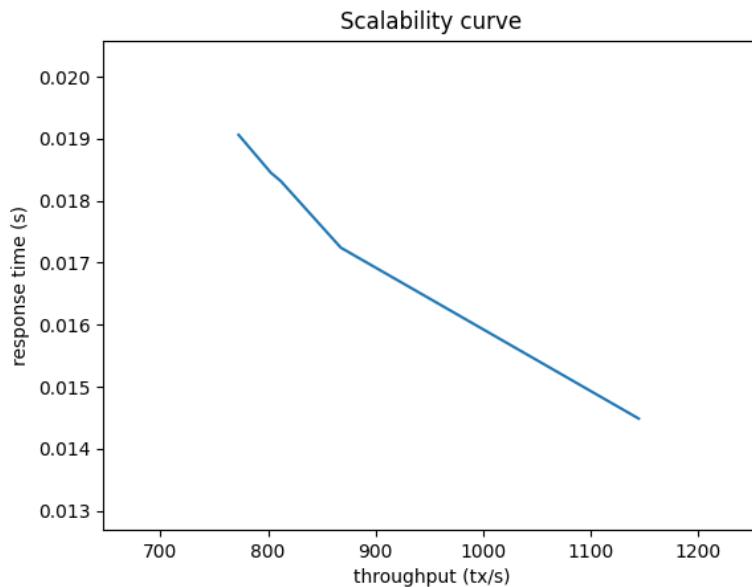


Figura 3.7: Comparação entre os vários valores do *synchronous commit*

3.2.3 wal_level [13] [14]

Este parâmetro é responsável por especificar o nível de informação escrita para o WAL. Assim, o *postgreSQL* disponibiliza três níveis diferentes - *minimal*, *replica* e *logical*.

O nível obtido por *default* é o nível *replica*, onde é escrita informação suficiente para suportar *WAL archiving* e replicação. Assim, este nível deve ser utilizado para permitir *WAL archiving* e *streaming replication*.

Um outro nível, que também permite replicação, é o *logical*. Este segundo nível, adiciona informação que permite suportar *logical decoding*. Desta forma, regista-se a mesma informação do nível *replica*, à qual se adiciona a informação necessária para a extração de conjuntos de mudanças lógicas no WAL. Assim, este nível implica o aumento do *WAL volume* em relação ao nível anterior.

A diferença entre estes dois níveis pode ser observada na seguinte tabela:

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
replica	818.889247566783	0.018155759751336983	0.318515084516083
logical	752.0097948623172	0.019633271127085065	0.3236355136546028

Tabela 3.6: Resultados da alteração do parâmetro *wal_level*

Por fim, o *postgreSQL* permite um terceiro nível para este parâmetro - *minimal*. Ao definir o *wal_level* como *minimal* nenhuma informação é registada sobre a transação que criou a informação. Assim, a realização de operações é efetuada de forma mais rápida. Porém, para poder utilizar este nível é necessário recorrer à alteração de outros parâmetros. A alteração destes parâmetros desliga a capacidade de replicar a base de dados, sendo por isso impossível replicar uma base de dados ao utilizar o *wal_level* como *minimal*.

Com base nesta característica decidiu-se deixar de lado esta opção, de modo a não limitar a replicação da base de dados no futuro.

3.2.4 wal_sync_method [15]

Este parâmetro é utilizado para selecionar o método utilizado durante os *Wal updates* no disco. Assim, este parâmetro torna-se irrelevante se a opção de *fsync* estiver desativada, uma vez que os *Wal updates* não terão a sua escrita forçada para o disco.

Para este parâmetro o *postgreSQL* oferece as seguintes opções:

- **open_datasync**: escreve no ficheiro WAL, utilizando o função *open* e a opção O_DSYNC;
- **fdatasync**: recorre à função *fdatasync()* para cada *commit* realizado;
- **fsync**: recorre à função *fsync()* para cada *commit* realizado;
- **fsync_writethrough**: recorre à função *fsync()* para cada *commit* realizado e força a escrita de valores na *cache* para disco;
- **open_sync**: escreve no ficheiro WAL, utilizando o função *open* e a opção O_SYNC.

Apesar da possibilidade de escolha entre as opções acima descritas, é possível que dependendo da plataforma algumas das opções não estejam disponíveis. Por exemplo, nos testes realizados não foi possível escolher a opção *fsync_writethrough* devido à sua indisponibilidade. Este facto foi evidenciado no ficheiro de *logs* (figura 3.8) resultante da sua escolha.

```
2022-05-03 17:18:44.831 GMT [5095] LOG: invalid value for parameter "wal_sync_method": "fsync_writethrough"
2022-05-03 17:18:44.831 GMT [5095] HINT: Available values: fsync, fdatasync, open_sync, open_datasync.
2022-05-03 17:18:44.831 UTC [5095] FATAL: configuration file "/etc/postgresql/14/main/postgresql.conf" contains e>
```

Figura 3.8: Valor *fsync_writethrough* indisponível na plataforma utilizada

Assim, para os parâmetro disponíveis pela plataforma, obteve-se os seguintes resultados:

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
open_datasync	823.3546993158083	0.01799046209505016	0.32018372191485617
fdatasync	804.5334879939044	0.01846795059808824	0.33067042764317733
fsync	683.4699775355783	0.020772053631472765	0.3532181420788795
open_sync	586.9133225635818	0.023877644078530713	0.36083225388601037

Tabela 3.7: Resultados da alteração do parâmetro *wal_sync_method*

Apesar deste valor poder ser escolhido pelo utilizador, durante a instalação, o *postgreSQL*, determina a melhor opção para o sistema operativo. Assim, a alteração deste parâmetro não oferece uma melhoria significativa nos resultados evidenciados.

3.2.5 full_pages_writes [10]

Este parâmetro, no seu valor *default (on)*, faz com que o servidor *PostgreSQL* escreva o conteúdo de cada página do disco no WAL, na primeira modificação a essa página após um *checkpoint*. Segundo as recomendações da documentação do *PostgreSQL*, mudar este parâmetro pode causar a corrupção de dados, tal como o *fsync*, pelo que deve ser mantido a *on*.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
Off	1245.6862099096124	0.012871491882717396	0.24535214689853865
On	846.1837737	0.017573941	0.317745826

Tabela 3.8: Resultados da alteração do parâmetro *fullpages*

3.2.6 wal_buffers[11]

Este parâmetro indica a quantidade de memória partilhada (*shared buffers*) utilizada para dados do WAL que ainda não foram escritos para o disco. Por *default*, este valor é definido para 3% do valor do *shared_buffers*. Como este parâmetro, por *default*, tem o valor de 128MB, então o *wal_buffers default* tem o valor de 3.84MB.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
2MB	804.888288236478	0.018492772831849556	0.3171048265901594
3.84MB	846.1837737	0.017573941	0.317745826
8MB	810.7810030585208	0.018363025705030223	0.31702247890507523
16MB	879.3767642188533	0.017180882132937867	0.30466029253466725

Tabela 3.9: Resultados da alteração do parâmetro *wal_buffers*

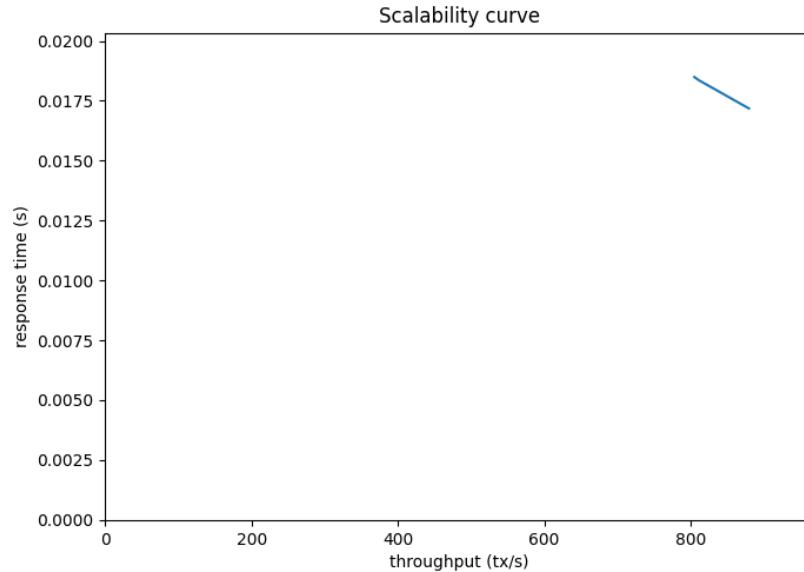


Figura 3.9: Comparaçāo entre os vārios valores de *wal_buffers*

3.2.7 commit_delay [18]

Este parâmetro define um atraso, em micro segundos, que ocorre entre o *commit* de transação e antes de um *WAL flush* para o disco. Ora, aumentar este valor pode beneficiar o *throughput* de um *group*

commit, uma vez que permite um maior número de transações realizarem *commit* num único *WAL flush*. Uma vez que este parâmetro tem a desvantagem de adicionar um atraso a cada *WAL flush*, este atraso só é aplicado quando existem pelo menos o mesmo número de transações ativas que foram definidas no parâmetro *commit_siblings*.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
0	846.1837737	0.017573941	0.317745826
200	854.4444539842536	0.017551086235655863	0.30878277478728944
400	841.6064711275634	0.017795560216389398	0.3115110334320937
1000	743.8453064301038	0.019711304768066353	0.326620056897511
2000	893.1280007558257	0.016881371851338964	0.3073848927412157

Tabela 3.10: Resultados da alteração do parâmetro *commit_delay*

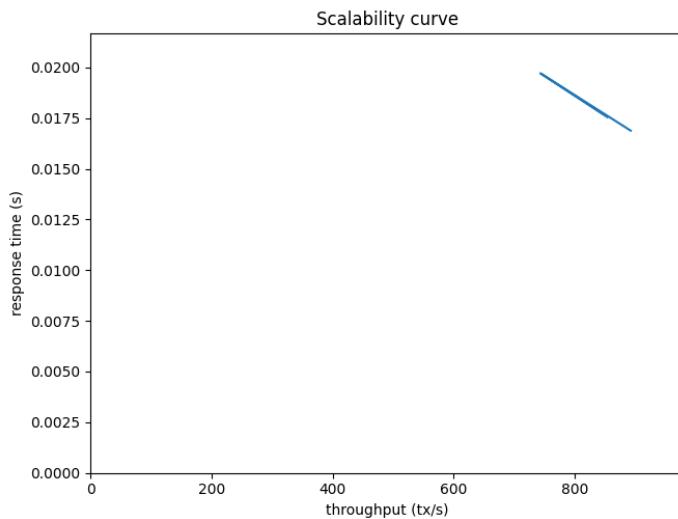


Figura 3.10: Comparação entre os vários valores de *commit_delay*

3.2.8 *commit_sibling* [16]

Este parâmetro é responsável por definir o número mínimo de transações concorrentes que ocorrem antes de se realizar o *commit_delay*. Assim, ao aumentar este valor espera-se que a probabilidade de uma outra transação ficar pronta para o *commit* durante o intervalo aumente.

Desta forma, este parâmetro aceita como valores um inteiro na gama de 0-1000, tendo como *default* o valor 5. Apesar do intervalo de valores aceitáveis ser extenso, é recomendado, na documentação do *PostgreSQL*, utilizar valores entre 3 e 8. Assim, realizou-se os seguintes testes:

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
3	785.5273935202036	0.01896853412694964	0.31657371345998725
5	807.7515908758211	0.01841857754561769	0.31869353011243134
8	794.2977546585896	0.01860009947774186	0.32326706229282676

Tabela 3.11: Resultados da alteração do parâmetro *commit_sibling*

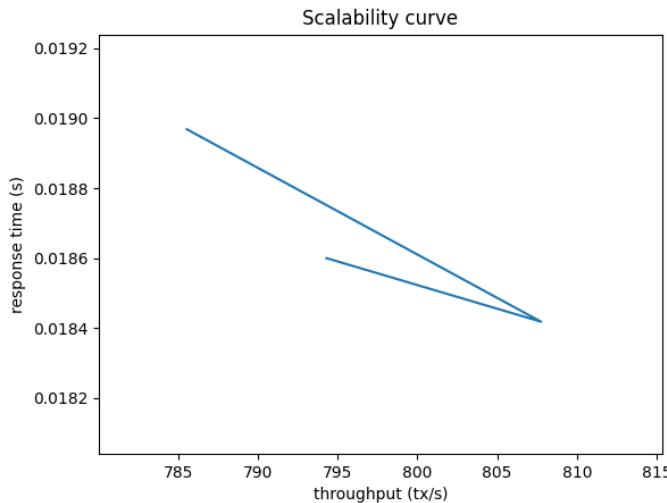


Figura 3.11: Comparação entre os vários valores de *commit_sibling*

3.2.9 *checkpoint_timeout*

O *checkpoint_timeout* determina o tempo máximo entre WAL *checkpoints* automáticos e tem como objetivo garantir que o WAL antes de algum ponto no tempo não seja mais necessário para recuperação, reduzindo os requisitos de espaço em disco e o tempo de recuperação da base de dados.

No entanto, fazer *checkpoints* muito frequentemente permitiria manter apenas uma pequena quantidade de WAL e a recuperação seria muito rápida em caso de falha mas tornaria as escritas assíncronas para os ficheiros de dados em escritas síncronas, reduzindo a taxa de transferência.

No geral, o valor *default* de 5 minutos é bastante baixo e valores entre 30 minutos e 1 hora são bastante comuns. Contudo, uma vez que o tempo de medição dos testes é de 10 minutos, foram experimentados alguns valores menores que 10, para poderem ser sentidas eventuais mudanças.

Pela análise da tabela seguinte podemos verificar que nenhuma opção melhorou os valores da configuração inicial.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
30s	834.5472296380774	0.0178693797687203	0.31653478307015576
1min	806.2633076660754	0.01849104667529182	0.3165624559715886
2min	817.7697163586506	0.01812108923375702	0.32050335721795925
4min	802.2438589574369	0.01844172961115088	0.32357326793969077
8min	804.4986552001782	0.018494183116486727	0.3202710559263932

Tabela 3.12: Resultados da alteração do parâmetro *checkpoint timeout*

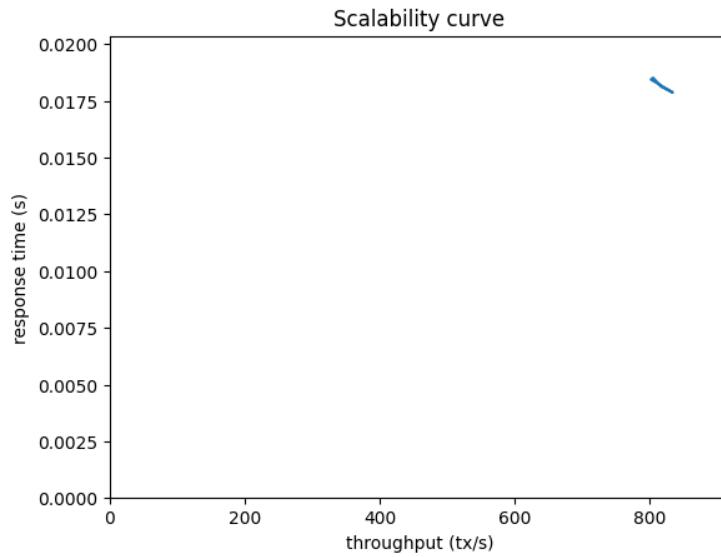


Figura 3.12: Curva de escalabilidade *checkpoint timeout*

3.2.10 `checkpoint_completion_target`

Este parâmetro especifica o objetivo de conclusão do *checkpoint*, como uma fração do tempo total entre os *checkpoints*, isto é, o tempo gasto a limpar *dirty buffers* durante o *checkpoint*, como fração do intervalo do *checkpoint*.

Foram experimentados vários valores sendo que o valor para o qual foi obtido maior *throughput* foi 1.0. No entanto, com este valor, não é deixado nenhum tempo para *overhead* na conclusão do *checkpoint* pelo que não seria sensato utilizá-lo.

De igual modo, alguns dos valores abaixo de 0.5 obteram bons resultados, no entanto, reduzir o valor *default* de 0.9 não é recomendado pois faz com que o *checkpoint* seja concluído mais rapidamente. Isso resulta numa taxa mais alta de I/O durante o *checkpoint* seguido de um período de menos I/O entre a conclusão do *checkpoint* e o próximo *checkpoint* agendado, causando picos de I/O indesejados.

Sendo assim, achamos melhor não alterar o valor *default* 0.9, já que os possíveis riscos não compensavam as ligeiras melhorias.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
0.0	862.0497044343269	0.017460173418121318	0.30692624671086777
0.2	860.006091473629	0.01729138260650002	0.3165415707622969
0.4	867.259933186438	0.017319830053403356	0.3073433139488677
0.6	820.9165700844405	0.01810994811734105	0.3182298803704108
0.8	772.4762460124169	0.01919590364120782	0.32070975574810157
1.0	883.5517738283125	0.01702953492285389	0.3081898143898775

Tabela 3.13: Resultados da alteração do parâmetro *checkpoint completion target*

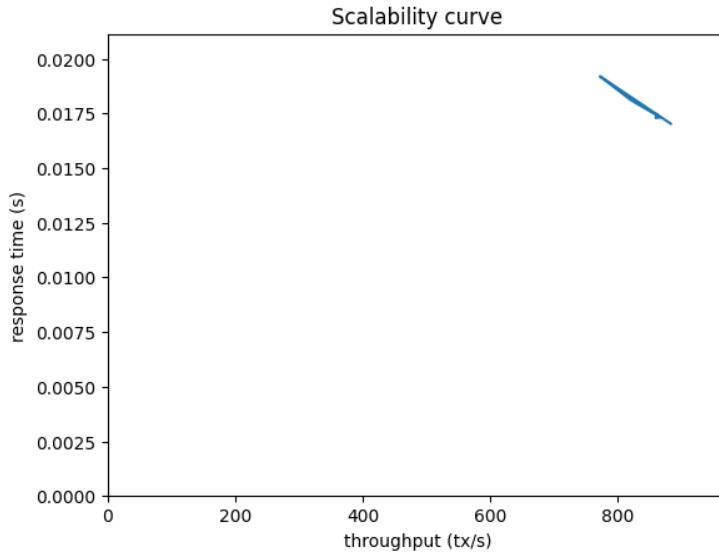


Figura 3.13: Curva de escalabilidade *checkpoint completion target*

3.2.11 `checkpoint_flush_after`

`Checkpoint.flush_after` permite forçar o sistema operativo a dar *flush* para disco das páginas escritas na altura do *checkpoint* após um número configurável de bytes. Caso contrário, essas páginas podem ser mantidas no cache de páginas do sistema operativo, causando paragens quando o *fsync* é emitido no final de um *checkpoint*.

Este parâmetro geralmente ajuda a reduzir a latência de transações, mas também pode ter um efeito adverso no desempenho; especialmente para cargas de trabalho maiores que *shared_buffers*, mas menores que o *cache* de páginas do SO.

Para este parâmetro testamos 3 opções tanto com *shared_buffers* igual a 128MB como a 2048MB. Deste modo pudemos verificar o seu comportamento em diferentes configurações. Em ambos os casos o melhor valor para este parâmetro foi 512kB.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
0	814.6407905175229	0.018327842440863412	0.31655998277397696
512kB	874.8488374746158	0.017246368212404155	0.30570984929764283
1MB	758.040769286177	0.019565141373713877	0.3212204239905264

Tabela 3.14: Resultados da alteração do parâmetro `checkpoint_flush_after` com *shared_buffers* igual a 128MB

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
0	921.8148132190098	0.01653647995113986	0.29921364729209476
512kB	1049.7471574892609	0.014662814536135875	0.2938587794562399
1MB	898.2543927760041	0.01681254682247479	0.3099509774681928

Tabela 3.15: Resultados da alteração do parâmetro `checkpoint_flush_after` com *shared_buffers* igual a 2048MB

3.2.12 checkpoint_warning

Este parâmetro serve para enviar uma mensagem de *log* para o servidor na eventualidade de os intervalos entre *checkpoints* causados pela ocupação total dos ficheiro de segmento for inferior ao valor aqui assinalado. No caso da opção 0, significa a desativação destes avisos. O valor por defeito utilizado para esta métrica é 30s. Através da tabela abaixo é possível verificar que a opção de 60s é favorável face à *default* em termos de performance.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
0	791.205137	0.018700537	0.322702867
15s	800.2834039	0.01846848	0.325322474
30s	846.1837737	0.017573941	0.317745826
60s	900.1759805	0.016670877	0.311355152
120s	768.6159703	0.019342026	0.319906695

Tabela 3.16: Comparaçao do *checkpoint_warning*

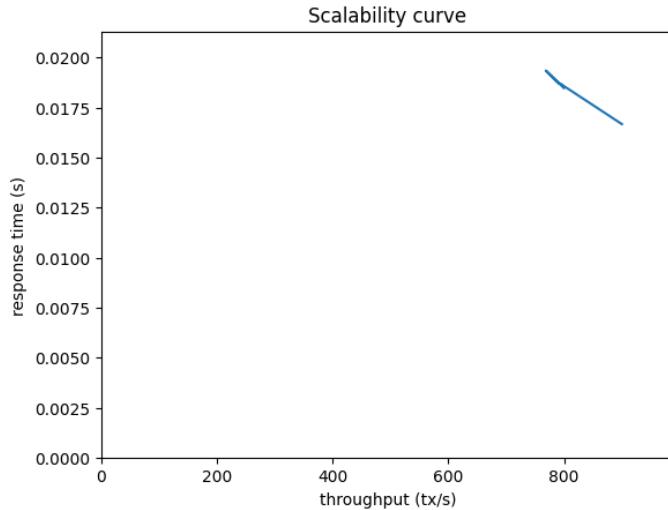


Figura 3.14: Curva de escalabilidade *checkpoint_warning*

3.2.13 max_wal_size

Este valor serve para marcar o tamanho máximo que o *WAL* pode crescer durante *checkpoints* automáticos. Este limite no entanto é considerado um *soft limit*, uma vez que é possível que o *WAL* em determinadas circunstâncias ultrapasse este valor. O valor por defeito para esta métrica é 1GB. Através dos testes efetuados pela equipa, foi possível verificar que o valor que mais beneficia a performance é o 4GB.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
0.5GB	703.8270802	0.02063153	0.336973517
1GB	846.1837737	0.017573941	0.317745826
2GB	908.1898412	0.01661371	0.306828224
4GB	1036.758635	0.015173931	0.268976614

Tabela 3.17: Comparaçao do *max_wal_size*

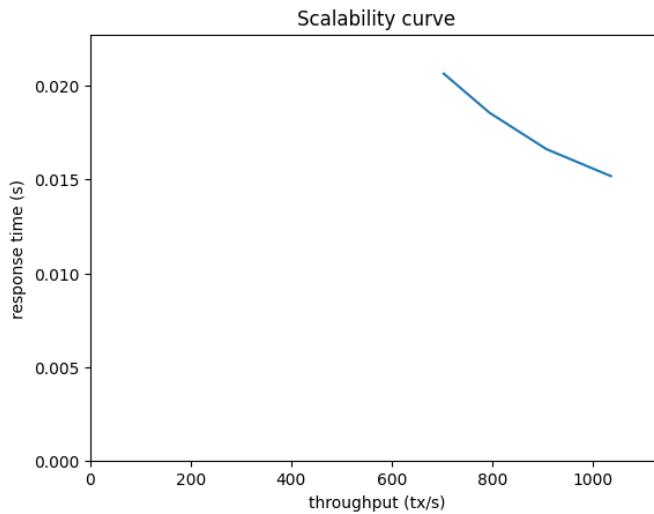


Figura 3.15: Curva de escalabilidade *Max_wal_size*

3.2.14 min_wal_size [19]

Ficheiros *WAL* antigos são sempre reciclados para serem usados num futuro *checkpoint*, em vez de serem removidos, desde que a utilização do disco por parte do *WAL* seja inferior a este valor. No caso desta métrica o valor por defeito é 80MB. Olhando para a tabela abaixo, constata-se que a utilização de 160MB será benéfica em termos de aumento de performance.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
40MB	823.5790561	0.01809483	0.317745826
80MB	846.1837737	0.017573941	0.319398782
160MB	863.6833915	0.017371547	0.311519341
320MB	791.6354842	0.01865382	0.325258536

Tabela 3.18: Comparação do min_wal_size

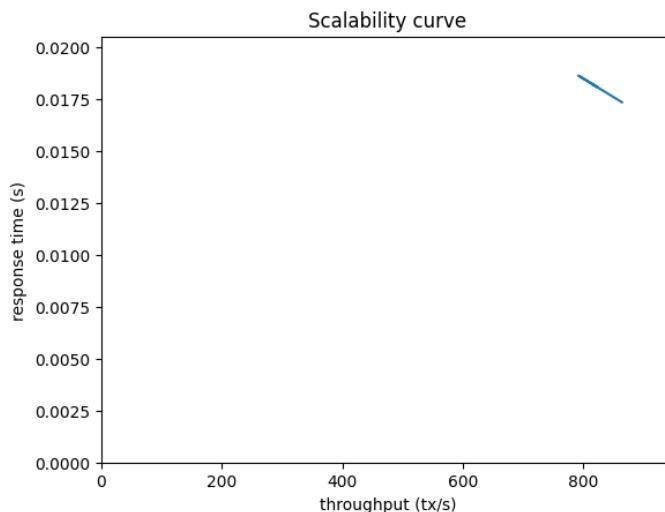


Figura 3.16: Curva de escalabilidade *Min_wal_size*

3.2.15 archive_mode [20]

O parâmetro *archive_mode* permite o arquivo de ficheiros WAL através do comando *archive_command*. Para permitir tal armazenamento, é necessário que o *wal_level* esteja definido, no mínimo, como *replica*. Quando este parâmetro é ativado, segmentos WAL (*Write-Ahead Logging*) que estejam completos são armazenados pela opção *archive_command*. Para além do modo *off*, é também possível definir dois modos adicionais, sendo esses o modo *on* e o modo *always*. A principal diferença entre estes dois últimos modos é que quando se define o modo *always*, o arquivador de WAL é também ativado durante o *standby mode*.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
<i>Off</i>	786.905053	0.0191429	0.3078616
<i>On</i>	767.838878	0.01962944	0.3130123
<i>Always</i>	709.652106	0.0214897	0.303204

Tabela 3.19: Comparaçāo Opções *archive_mode*

Como se pode observar, a melhor opção para este comando é o valor *Off*, uma vez que as restantes opções fazem com que o número de operações a ser realizadas durante a execução aumente.

3.2.16 archive_command [21]

Corresponde ao comando *shell* a ser executado para arquivar um segmento de WAL completo. Esta opção só deve ser definida quando o comando *archive_mode* esteja no modo *On* ou *Always*, e como foi visto anteriormente, essas opções não contribuem para o aumento da *performance*, pelo que, como seria de esperar, os resultados também não foram melhores.

Opção	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
%p	770.4532	0.020045	0.34078
%f	765.15432	0.0021885	0.34913

Tabela 3.20: Comparaçāo *archive_command*

3.2.17 archive_timeout [22]

Este parâmetro irá forçar a escrita do próximo ficheiro WAL, caso um novo ficheiro WAL não tenha começado dentro de um determinado número de segundos. Não é aconselhável definir um tempo de *archive_timeout* muito baixo, uma vez que poderá sobrecarregar o local de armazenamento, sendo por isso melhor definir tempos de cerca de um minuto.

Modo On

Tempo	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
60 segundos	756.046267	0.02020008	0.299616
80 segundos	737.838878	0.02046	0.31500

Tabela 3.21: Comparações *archive_timeout* no Modo *On*

Modo Always

Tempo	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
60 segundos	739.01809	0.020360	0.316022
80 segundos	767.3765	0.01969	0.31304

Tabela 3.22: Comparações *archive_timeout* no Modo *Always*

Analizando os resultados obtidos, percebe-se que deixando o comando do *archive mode Off*, se obtém melhores resultados, pelo que essa será a opção a ser considerada para as combinações finais.

3.3 Parâmetros de Isolamento

Para terminar os testes sobre os parâmetros do *PostgreSQL*, decidimos avaliar o impacto do nível de isolamento no comportamento das transações. O *PostgreSQL* suporta os níveis *read committed*, *repeatable read* e *serializable*. Também é possível testar o nível *read uncommitted* no entanto, o *PostgreSQL* não implementa este nível, que permite *dirty reads*, e em vez disso utiliza o valor *default read committed*.

Nível de isolamento	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
Read committed	846.18377	0.01757	0.317745
Repeatable read	896.0583	0.0167880	0.306967
Serializable	892.25427	0.016825	0.309172

Tabela 3.23: Níveis de Isolamento

Antes de testar estes parâmetros prevemos que, no mínimo, os valores de tempo de resposta e taxa de aborto iriam subir visto que tanto o nível *serializable* como o *repeatable read* são mais rigorosos na forma como tratam as transações. No entanto, não só estes valores não sofreram grandes alterações como a taxa de transferência aumentou em ambos os níveis.

3.4 Combinações

3.4.1 Combinações de *settings*

Melhor combinação:

```
shared_buffers : 2048 MB
work_mem : 1024MB
huge pages : Off
fsync : Off
wal_level : replica
full_pages_writes = Off
synchronous_commit = Off
wal_buffers = 16MB
commit_delay = 2000
commit_siblings = 5
```

Configurações	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
Default	846.18377	0.01757	0.317745
Combinações Settings	1350.62452	0.012591	0.1557098

Tabela 3.24: Resultados com combinação de *settings*

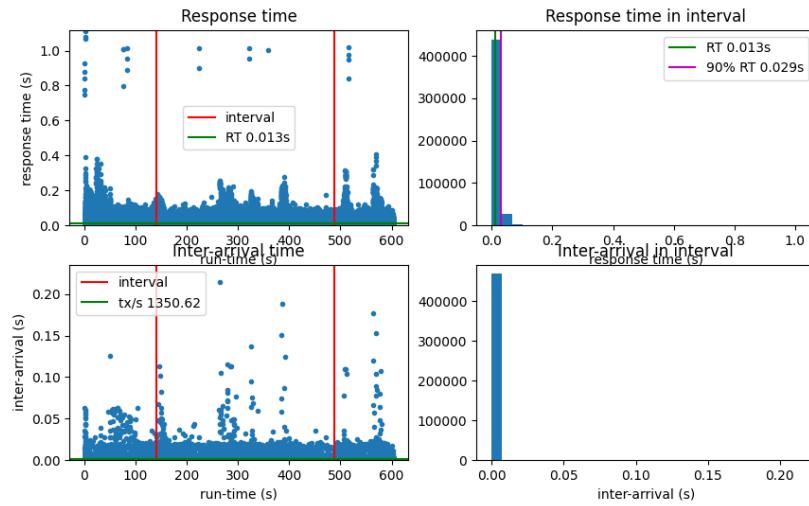


Figura 3.17: Métricas com combinações de *settings*

3.4.2 Combinações de *checkpoints*

Melhor combinação:

```
checkpoint_timeout = 5min
checkpoint_completion_target = 0.9
checkpoint_flush_after = 512 kB
checkpoint_warning = 60s
max_wal_size = 4GB
min_wal_size = 160MB
```

Configurações	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
<i>Default</i>	846.18377	0.01757	0.317745
Combinações <i>Checkpoints</i>	867.647101	0.017763	0.2852001

Tabela 3.25: Resultados com combinação de *checkpoints*

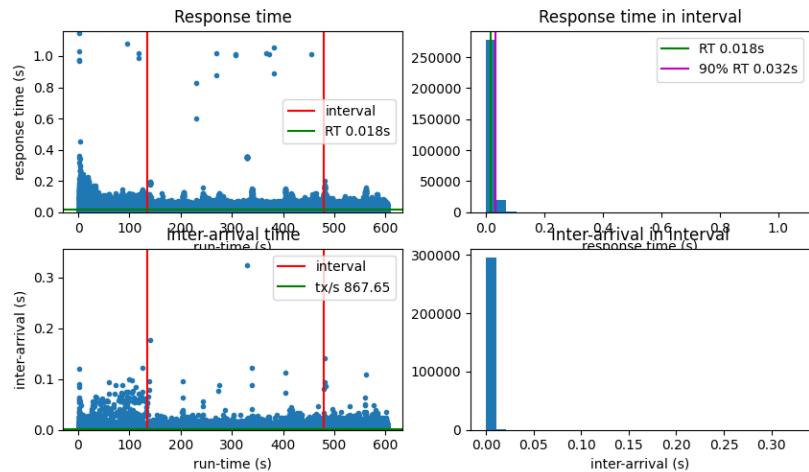


Figura 3.18: Métricas com combinações de *checkpoints*

3.4.3 Combinações de *archiving*

No que toca a esta tentativa de otimização, os valores obtidos para todos os testes foram inferiores ao valores obtidos para os valores de referência, de modo que esta tentativa de melhoria acabou por ser ignorada.

3.4.4 Combinação geral

combinação geral:

```
shared_buffers : 2048 MB
work_mem : 1024MB
huge pages : Off
fsync : Off
wal_level : replica
full_pages_writes = Off
synchronous_commit = Off
wal_buffers = 16MB
commit_delay = 2000
commit_siblings = 5
checkpoint_timeout = 5min
checkpoint_completion_target = 0.9
checkpoint_flush_after = 512 kB
checkpoint_warning = 60s
max_wal_size = 4GB
min_wal_size = 160MB
default_transaction_isolation = repeatable read
```

Configurações	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
Default	846.18377	0.01757	0.317745
Combinação Geral	1692.703909	0.0102355	0.151963

Tabela 3.26: Resultados com combinação geral

Esta nova configuração permite uma grande otimização do desempenho da carga transacional em relação à predefinida pelo *PostgreSQL*, no entanto, por questões de segurança escolheu-se ligar o *fsync* e utilizar os valores *default* do parâmetro *work_mem*, de modo a evitar possíveis erros de *out of memory*. Os resultados finais foram os seguintes:

Configurações	Taxa de transferência (tx/seg)	Tempo de resposta (seg)	Taxa de aborto (%)
Default	846.18377	0.01757	0.317745
Combinação Geral	1436.944796	0.011940	0.16044

Tabela 3.27: Resultados finais com combinação geral

Desta forma, comparando com a configuração de referência, é possível observar que conseguimos aumentar o valor de taxa de transferência de 846.18 para 1436.94 tx/s. Juntamente, o tempo de resposta diminuiu ligeiramente e a taxa de aborto reduziu de 0.31 para 0.16%. Sendo assim, com um ganho de cerca de 40%, otimizamos com sucesso o desempenho da carga transacional.

4 Otimização das interrogações analíticas

Para esta fase, foi necessário adaptar o TPC-C para o TPC-H para a criação e povoamento das novas tabelas.

schemaname	tablename	indexname	tablespace	indexdef
public	customer	keycustomer		CREATE UNIQUE INDEX keycustomer ON public.customer USING btree (key)
public	district	keydistrict		CREATE UNIQUE INDEX keydistrict ON public.district USING btree (key)
public	history	keyhistory		CREATE UNIQUE INDEX keyhistory ON public.history USING btree (key)
public	item	keyitem		CREATE UNIQUE INDEX keyitem ON public.item USING btree (key)
public	new_order	keyneworder		CREATE UNIQUE INDEX keyneworder ON public.new_order USING btree (key)
public	order_line	keyorderline		CREATE UNIQUE INDEX keyorderline ON public.order_line USING btree (key)
public	orders	keyorders		CREATE UNIQUE INDEX keyorders ON public.orders USING btree (key)
public	stock	keystock		CREATE UNIQUE INDEX keystock ON public.stock USING btree (key)
public	warehouse	keywarehouse		CREATE UNIQUE INDEX keywarehouse ON public.warehouse USING btree (key)
public	customer	pk_customer		CREATE UNIQUE INDEX pk_customer ON public.customer USING btree (c_w_id, c_d_id, c_i_id)
public	customer	ix_customer		CREATE INDEX ix_customer ON public.customer USING btree (c_w_id, c_d_id, c_last)
public	district	pk_district		CREATE UNIQUE INDEX pk_district ON public.district USING btree (d_w_id, d_id)
public	item	pk_item		CREATE UNIQUE INDEX pk_item ON public.item USING btree (i_id)
public	order_line	pk_order_line		CREATE UNIQUE INDEX pk_order_line ON public.order_line USING btree (ol_w_id, ol_d_id, ol_o_id, ol_number)
public	order_line	ix_order_line		CREATE INDEX ix_order_line ON public.order_line USING btree (ol_i_id)
public	orders	pk_orders		CREATE INDEX pk_orders ON public.orders USING btree (o_w_id, o_d_id, o_id)
public	orders	ix_orders		CREATE INDEX ix_orders ON public.orders USING btree (o_w_id, o_d_id, o_c_id)
public	new_order	ix_new_order		CREATE INDEX ix_new_order ON public.new_order USING btree (no_w_id, no_d_id, no_o_id)
public	stock	pk_stock		CREATE UNIQUE INDEX pk_stock ON public.stock USING btree (s_w_id, s_i_id)
public	stock	ix_stock		CREATE INDEX ix_stock ON public.stock USING btree (s_i_id)
public	warehouse	pk_warehouse		CREATE UNIQUE INDEX pk_warehouse ON public.warehouse USING btree (w_id)

Figura 4.1: Lista de índices já presentes na base de dados

4.1 Interrogação analítica A.1

```
SELECT c_id, c_last, sum(ol_amount) AS revenue, c_city, c_phone, n_name
FROM customer, orders, order_line, nation
WHERE c_id = o_c_id
    AND c_w_id = o_w_id
    AND c_d_id = o_d_id
    AND ol_w_id = o_w_id
    AND ol_d_id = o_d_id
    AND ol_o_id = o_id
    AND o_entry_d >= '2022-01-01 00:00:00.000000'
    AND o_entry_d <= ol_delivery_d
    AND n_nationkey = ascii(substr(c_state, 1, 1)) % 24
GROUP BY c_id, c_last, c_city, c_phone, n_name
ORDER BY revenue DESC;
```

Esta interrogação analítica tem como objetivo obter os clientes que gastaram mais dinheiro em encomendas. Ao executar esta interrogação analítica obtemos o seguinte resultado:

c_id	c_last	revenue	c_city	c_phone	n_name
2305	EINGEINGESE	104369.00	ni5tielzavb6ni5tielz	2n7qshmp6b312n7q	ARGENTINA
2693	ANTIAFABLE	103194.00	e9rlm1j2aoyre9r1mlj2	1t8k70lx3mcn1lt8	ARGENTINA
2464	ABLEESEEING	102941.00	15di4t1zxju3215di4t1	3fub9juujhpy3fub	CHINA
2525	ESEESEABLE	102866.00	zzbitnsbmhzezzb1tnsb	v6q36nlbd771v6q3	ARGENTINA
2518	OUGHTEINGEING	101647.00	xs4keogbkursxs4keogb	19zixnjq4edds19z	ARGENTINA
2450	EINGPRIPRES	101212.00	jc3vln98xzmbjc3vln98	1rv6og2qmt7gb1rv	ARGENTINA
2899	BARPRIATION	101112.00	qe2ziadixt0hge2ziad1	iv8axfeagbxriv8a	RUSSIA

Figura 4.2: Resultado da interrogação analítica A1

De modo a verificar o plano de execução que o *PostgreSQL* seleciona para esta interrogação analítica, utilizamos o EXPLAIN ANALYZE:

HTML	SOURCE	HINTS	INew!	STATS
Per node type stats				
node type	count	sum of times	% of query	
Finalize GroupAggregate	1	3,373.735 ms	3.7 %	
Gather Merge	1	6,771.643 ms	7.4 %	
Hash	1	0.030 ms	0.0 %	
Hash Join	1	5,864.905 ms	6.4 %	
Materialize	1	1,016.934 ms	1.1 %	
Merge Join	1	4,257.352 ms	4.6 %	
Parallel Hash	1	589.224 ms	0.6 %	
Parallel Hash Join	1	7,520.185 ms	8.2 %	
Parallel Seq Scan	2	11,083.168 ms	12.1 %	
Partial GroupAggregate	1	4,615.364 ms	5.0 %	
Seq Scan	2	1,931.923 ms	2.1 %	
Sort	4	44,714.920 ms	48.7 %	

Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
customer	1	5,549.240 ms	6.0 %
Parallel Seq Scan	1	5,549.240 ms	100.0 %
nation	1	1,005.466 ms	1.1 %
Seq Scan	1	1,005.466 ms	100.0 %
order_line	1	5,533.928 ms	6.0 %
Parallel Seq Scan	1	5,533.928 ms	100.0 %
orders	1	926.457 ms	1.0 %
Seq Scan	1	926.457 ms	100.0 %

Figura 4.5: Estatísticas do plano de execução original da interrogação analítica A1

Como podemos ver, o *seq scan* na tabela *orders* em *o_entry_d* seleciona 2880000 linhas. Ora, em casos em que são selecionadas mais que 5-10% das linhas de uma tabela (neste caso corresponde a todas as linhas da tabela), um *seq scan* é mais eficiente do que um *index scan*, uma vez que terá de fazer menos operações de *IO* no disco.

Também podemos verificar que o *PostgreSQL* não está a usar nenhum índice. Acreditamos que se deve ao mesmo motivo, uma vez que estão a ser selecionadas um grande número de linhas em cada condição.

4.1.1 Índices

Ao criar o índice em *o_entry_d* e ao obrigar o *PostgreSQL* a usar índices, ao mudar o uso de *seq scan* para *off* no ficheiro de configuração, obtemos o seguinte resultado:

4.1.2 Vista Materializada 1

De modo a otimizar a operação ORDER BY revenue DESC;, decidimos criar uma vista materializada para conseguirmos dar *cache* do valor *sum(ol_amount)* para cada *customer*.

```
create materialized view a1_mv as SELECT c_id, c_last,
sum(ol_amount) AS revenue, c_city, c_phone, n_name
  FROM customer, orders, order_line, nation
 WHERE c_id = o_c_id
   AND c_w_id = o_w_id
   AND c_d_id = o_d_id
   AND ol_w_id = o_w_id
   AND ol_d_id = o_d_id
   AND ol_o_id = o_id
   AND o_entry_d >= '2022-01-01 00:00:00.000000'
   AND o_entry_d <= ol_delivery_d
   AND n_nationkey = ascii(substr(c_state, 1, 1)) % 24
 GROUP BY c_id, c_last, c_city, c_phone, n_name
```

Ao adaptar a interrogação analítica para funcionar com esta vista materializada:

```
SELECT c_id, c_last, revenue, c_city, c_phone, n_name
FROM a1_mv
ORDER BY revenue DESC;
```

Obtemos o seguinte resultado:

```
QUERY PLAN
-----
Gather Merge (cost=299274.09..579257.00 rows=2399686 width=89) (actual time=1652.643..3028.276 rows=2880000 loops=1)
  Workers Planned: 2
  Workers Launched: 2
    -> Sort (cost=298274.07..301273.67 rows=1199843 width=89) (actual time=1452.175..1834.870 rows=960000 loops=3)
        Sort Key: revenue DESC
        Sort Method: external merge Disk: 137584kB
        Worker 0: Sort Method: external merge Disk: 70264kB
        Worker 1: Sort Method: external merge Disk: 71904kB
        -> Parallel Seq Scan on a1_mv (cost=0.00..54091.43 rows=1199843 width=89) (actual time=0.051..184.768 rows=960000 loops=3)
Planning Time: 0.316 ms
Execution Time: 3167.417 ms
(11 rows)
```

Figura 4.10: EXPLAIN ANALYZE interrogação analítica A1 com uso de uma vista materializada

Podemos ver que houve um salto de performance enorme, obtendo um tempo de execução de 3167.417ms. Porém, como iremos explicar a seguir, esta solução não é ideal.

Colocando o plano de execução na ferramenta *explain depesz*:

No entanto, esta é uma solução irrealista e inflexível, na medida em que estamos a colocar a maior parte da interrogação analítica dentro da vista materializada, incluindo condições com valores específicos que não podem ser alterados sem criar uma nova vista materializada. Assim, o grupo decidiu procurar outros modos de otimizar esta interrogação analítica.

4.1.3 Vista Materializada 2

Uma segunda abordagem para a utilização de vistas materializadas para a otimização desta interrogação analítica consiste em guardar os gastos de cada cliente para cada dia distinto. De seguida, a interrogação analítica só tem de somar os gastos de cada cliente dos dias que são posteriores à data pretendida, ao invés de ter de somar todos os gastos das compras de todos os clientes desde sempre. Assim, criámos a seguinte vista materializada:

```
create materialized view a1_mv as SELECT c_id, c_last, DATE(o_entry_d) as dia,
sum(ol_amount) AS revenue, c_city, c_phone, n_name
FROM customer, orders, order_line, nation
WHERE c_id = o_c_id
    AND c_w_id = o_w_id
    AND c_d_id = o_d_id
    AND ol_w_id = o_w_id
    AND ol_d_id = o_d_id
    AND ol_o_id = o_id
    AND o_entry_d <= ol_delivery_d
    AND n_nationkey = ascii(substr(c_state, 1, 1)) % 24
GROUP BY dia, c_id, c_last, c_city, c_phone, n_name
```

Após adaptar a interrogação analítica para:

```
SELECT c_id, c_last, SUM(revenue) as revenue_total, c_city, c_phone, n_name
FROM a1_mv
WHERE dia >= DATE('2022-01-01 00:00:00.000000')
GROUP BY c_id, c_last, c_city, c_phone, n_name
ORDER BY revenue_total DESC;
```

Obtemos o seguinte resultado:

```
QUERY PLAN
Sort (cost=1359847.85..1367048.19 rows=2880138 width=117) (actual time=8402.915..9163.820 rows=2880000 loops=1)
  Sort Key: (sum(revenue)) DESC
  Sort Method: external merge Disk: 279616kB
-> Finalize GroupAggregate (cost=305407.64..696446.07 rows=2880138 width=117) (actual time=1479.678..5280.875 rows=2880000 loops=1)
    Group Key: c_id, c_last, c_city, c_phone, n_name
    -> Gather Merge (cost=305407.64..618442.31 rows=2400116 width=117) (actual time=1479.636..2765.110 rows=2880000 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial GroupAggregate (cost=304407.61..340409.35 rows=1200058 width=117) (actual time=1386.898..2379.619 rows=960000 loops=3)
            Group Key: c_id, c_last, c_city, c_phone, n_name
            -> Sort (cost=304407.61..307407.76 rows=1200058 width=89) (actual time=1386.833..1550.550 rows=960000 loops=3)
                Sort Key: c_id, c_last, c_city, c_phone, n_name
                Sort Method: external merge Disk: 70584kB
                Worker 0: Sort Method: external merge Disk: 66656kB
                Worker 1: Sort Method: external merge Disk: 142520kB
                -> Parallel Seq Scan on a1_mv (cost=0.00..60180.72 rows=1200058 width=89) (actual time=342.299..559.891 rows=960000 loops=3)
                    Filter: (dia >= '2022-01-01':date)

Planning Time: 0.473 ms
JIT:
  Functions: 30
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 5.835 ms, Inlining 215.216 ms, Optimization 486.927 ms, Emission 323.054 ms, Total 1031.033 ms
Execution Time: 9330.603 ms
(23 rows)
```

Figura 4.14: Resultado da interrogação analítica A1 com uso da nova vista materializada

Como podemos verificar, obtemos um tempo de execução bastante melhor de 9330.603 ms. Ao analisar o novo plano de execução com a ferramenta *explain depesz*:


```

-----  

 QUERY PLAN  

-----  

Sort  (cost=1056693.97..1063893.97 rows=2880000 width=117) (actual time=5889.677..6648.577 rows=2880000 loops=1)
  Sort Key: (sum(revenue)) DESC
  Sort Method: external merge Disk: 279616kB
    -> GroupAggregate  (cost=0.56..393328.99 rows=2880000 width=117) (actual time=143.950..2779.282 rows=2880000 loops=1)
        Group Key: c_id, c_last, c_city, c_phone, n_name
        -> Index Scan using a1_mv_group on a1_mv  (cost=0.56..314128.99 rows=2880000 width=89) (actual time=143.908..1126.043 rows=2880000 loops=1)
              Filter: (dia >= '2022-01-01':date)
Planning Time: 0.507 ms
JIT:
  Functions: 7
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 1.412 ms, Inlining 13.292 ms, Optimization 79.481 ms, Emission 50.723 ms, Total 144.908 ms
Execution Time: 6794.301 ms
(13 rows)

```

Figura 4.18: Resultado da interrogação analítica A1 com uso da nova vista materializada e de índices

Como podemos observar, conseguimos uma pequena melhoria na performance, obtendo um tempo de execução de 6794.301ms. Ao colocar este novo plano de execução na ferramenta *explain depesz*:

	HTML	SOURCE	HINTS	!NEW!	STATS	
#	exclusive	inclusive	rows x	rows	loops	node
1.	3,869.295	6,648.577	↑ 1.0	2,880,000	1	→ Sort (cost=1.056.693.97..1.063.893.97 rows=2,880,000 width=117) (actual time=5,889.677..6,648.577 rows=2,880,000 loops=1) Sort Key: (sum(revenue)) DESC Sort Method: external merge Disk: 279,616kB
2.	1,653.239	2,779.282	↑ 1.0	2,880,000	1	→ GroupAggregate (cost=0.56..393,328.99 rows=2,880,000 width=117) (actual time=143.950..2,779.282 rows=2,880,000 loops=1) Group Key: c_id, c_last, c_city, c_phone, n_name
3.	1,126.043	1,126.043	↑ 1.0	2,880,000	1	→ Index Scan using a1_mv_group on a1_mv (cost=0.56..314,128.99 rows=2,880,000 width=89) (actual time=143.908..1,126.043 rows=2,880,000 loops=1) Filter: (dia >= '2022-01-01':date)

Figura 4.19: Estatísticas do plano de execução da interrogação analítica A1 com uso da nova vista materializada e índices

	HTML	SOURCE	HINTS	!NEW!	STATS																
Per node type stats																					
<table border="1"> <thead> <tr> <th>node type</th> <th>count</th> <th>sum of times</th> <th>% of query</th> </tr> </thead> <tbody> <tr> <td>GroupAggregate</td> <td>1</td> <td>1,653.239 ms</td> <td>24.9 %</td> </tr> <tr> <td>Index Scan</td> <td>1</td> <td>1,126.043 ms</td> <td>16.9 %</td> </tr> <tr> <td>Sort</td> <td>1</td> <td>3,869.295 ms</td> <td>58.2 %</td> </tr> </tbody> </table>					node type	count	sum of times	% of query	GroupAggregate	1	1,653.239 ms	24.9 %	Index Scan	1	1,126.043 ms	16.9 %	Sort	1	3,869.295 ms	58.2 %	
node type	count	sum of times	% of query																		
GroupAggregate	1	1,653.239 ms	24.9 %																		
Index Scan	1	1,126.043 ms	16.9 %																		
Sort	1	3,869.295 ms	58.2 %																		
Per table stats																					
<table border="1"> <thead> <tr> <th>Table name</th> <th>Scan count</th> <th>Total time</th> <th>% of query</th> </tr> <tr> <th>scan type</th> <th>count</th> <th>sum of times</th> <th>% of table</th> </tr> </thead> <tbody> <tr> <td>a1_mv</td> <td>1</td> <td>1,126.043 ms</td> <td>16.9 %</td> </tr> <tr> <td>Index Scan</td> <td>1</td> <td>1,126.043 ms</td> <td>100.0 %</td> </tr> </tbody> </table>						Table name	Scan count	Total time	% of query	scan type	count	sum of times	% of table	a1_mv	1	1,126.043 ms	16.9 %	Index Scan	1	1,126.043 ms	100.0 %
Table name	Scan count	Total time	% of query																		
scan type	count	sum of times	% of table																		
a1_mv	1	1,126.043 ms	16.9 %																		
Index Scan	1	1,126.043 ms	100.0 %																		

Figura 4.20: Estatísticas do plano de execução da interrogação analítica A1 com uso da nova vista materializada e índices

Podemos ver que adicionar os índices removeu o impacto na performance que a operação GROUP BY estava a causar. Assim, conseguimos obter uma grande melhoria na performance, passando de 88132.242ms para 6794.301ms. Estamos bastante satisfeitos com este resultado, uma vez que conseguimos diminuir o tempo de execução para aproximadamente 7% do tempo de execução original.

4.1.5 Paralelismo

Outro modo de otimizar esta interrogação analítica é aumentar (ou diminuir) o nível de paralelismo que está a ser utilizado, ao definir o número de *workers* do PostgreSQL. Relembrando que o tempo de execução original (2 *workers*) era 88132.242ms. Ao mudar o número de *workers* para 1:


```

QUERY PLAN
Sort  (cost=6142746.86..6160730.13 rows=7193306 width=195) (actual time=91952.649..93003.080 rows=2880000 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC
  Sort Method: external merge  Disk: 279616kB
-> GroupAggregate (cost=2183162.87..3258227.20 rows=7193306 width=195) (actual time=60736.950..88794.051 rows=2880000 loops=1)
    Group Key: customer.c_id, customer.c_last, customer.c_city, customer.c_phone, nation.n_name
    -> Gather Merge (cost=2183162.87..3060411.29 rows=7193306 width=166) (actual time=60736.851..79125.380 rows=24476106 loops=1)
      Workers Planned: 6
      Workers Launched: 6
      -> Sort  (cost=2182162.77..2185159.98 rows=1198884 width=166) (actual time=59920.933..62631.762 rows=3496587 loops=7)
        Sort Key: customer.c_id, customer.c_last, customer.c_city, customer.c_phone, nation.n_name
        Sort Method: external merge  Disk: 321458kB
        Worker 0: Sort Method: external merge  Disk: 339569kB
        Worker 1: Sort Method: external merge  Disk: 345024kB
        Worker 2: Sort Method: external merge  Disk: 332040kB
        Worker 3: Sort Method: external merge  Disk: 341369kB
        Worker 4: Sort Method: external merge  Disk: 358680kB
        Worker 5: Sort Method: external merge  Disk: 329723kB
-> Hash Join  (cost=1670041.20..1864422.88 rows=1198884 width=166) (actual time=33924.982..42727.488 rows=3496587 loops=7)
  Hash Cond: ((ascii(substr((customer.c_state)::text, 1, 1)) % 24) = nation.n_nationkey)
-> Parallel Hash Join  (cost=1670027.38..1840078.76 rows=1410452 width=65) (actual time=32405.631..35578.457 rows=3496587 loops=7)
  Hash Cond: ((orders.o_c_id = customer.c_id) AND (orders.o_w_id = customer.c_w_id) AND (orders.o_d_id = customer.c_d_id))
    -> Merge Join  (cost=1455438.38..1592115.38 rows=1358778 width=23) (actual time=13891.418..22618.924 rows=3496587 loops=7)
      Merge Cond: ((order_line.ol_w_id = orders.o_w_id) AND (order_line.ol_d_id = orders.o_d_id) AND (order_line.ol_o_id = orders.o_id))
      Join Filter: (orders.o_entry_d <= order_line.ol_delivery_d)
      -> Sort  (cost=965447.38..975649.76 rows=4422932 width=23) (actual time=9607.068..11267.190 rows=3496587 loops=7)
        Sort Key: order_line.ol_id, order_line.ol_d_id, order_line.ol_o_id
        Sort Method: external merge  Disk: 124992kB
        Worker 0: Sort Method: external merge  Disk: 124648kB
        Worker 1: Sort Method: external merge  Disk: 117456kB
        Worker 2: Sort Method: external merge  Disk: 130808kB
        Worker 3: Sort Method: external merge  Disk: 121440kB
        Worker 4: Sort Method: external merge  Disk: 132528kB
        Worker 5: Sort Method: external merge  Disk: 120288kB
-> Parallel Seq Scan on order_line  (cost=0.00..350215.51 rows=4079351 width=23) (actual time=1.053..4208.336 rows=3496587 loops=7)
-> Materialize  (cost=489991.00..504389.56 rows=2879712 width=24) (actual time=4274.281..6930.415 rows=5964619 loops=7)
-> Sort  (cost=489991.00..497190.28 rows=2879712 width=24) (actual time=4274.266..5407.462 rows=2879668 loops=7)
  Sort Key: orders.o_w_id, orders.o_d_id, orders.o_id
  Sort Method: external merge  Disk: 95872kB
  Worker 0: Sort Method: external merge  Disk: 95872kB
  Worker 1: Sort Method: external merge  Disk: 95872kB
  Worker 2: Sort Method: external merge  Disk: 95872kB
  Worker 3: Sort Method: external merge  Disk: 95872kB
  Worker 4: Sort Method: external merge  Disk: 95872kB
  Worker 5: Sort Method: external merge  Disk: 95872kB
-> Seq Scan on orders  (cost=0.00..62916.00 rows=2879712 width=24) (actual time=0.081..1229.977 rows=2880000 loops=7)
  Filter: (o_entry_d >= '2022-01-01 00:00:00'::timestamp without time zone)
-> Parallel Hash  (cost=197759.00..197759.00 rows=576000 width=70) (actual time=5965.981..5965.983 rows=411429 loops=7)
  Buckets: 65536 Batches: 128 Memory Usage: 2912kB
-> Parallel Seq Scan on customer  (cost=0.00..197759.00 rows=576000 width=70) (actual time=2.258..5718.228 rows=411429 loops=7)
  Buckets: 1024 Batches: 1 Memory Usage: 10kB
-> Hash  (cost=11.70..11.70 rows=170 width=108) (actual time=1519.209..1519.211 rows=170 width=108)
  Buckets: 256 Batches: 1 Memory Usage: 10kB
-> Seq Scan on nation  (cost=0.00..11.70 rows=170 width=108) (actual time=1518.976..1519.187 rows=170 width=108)

Planning Time: 1.482 ms
JIT:
  Functions: 311
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 90.394 ms, Inlining 1438.194 ms, Optimization 6202.828 ms, Emission 2966.459 ms, Total 10697.874 ms
Execution Time: 93233.285 ms
(58 rows)

```

Figura 4.23: Plano de execução da interrogação analítica A1 com 6 workers

Mais uma vez, o tempo de execução é pior que o original, tendo um valor de 93233.285ms. Por último, geramos um gráfico que permite visualizar a curva de escalabilidade do paralelismo para a interrogação analítica A1:

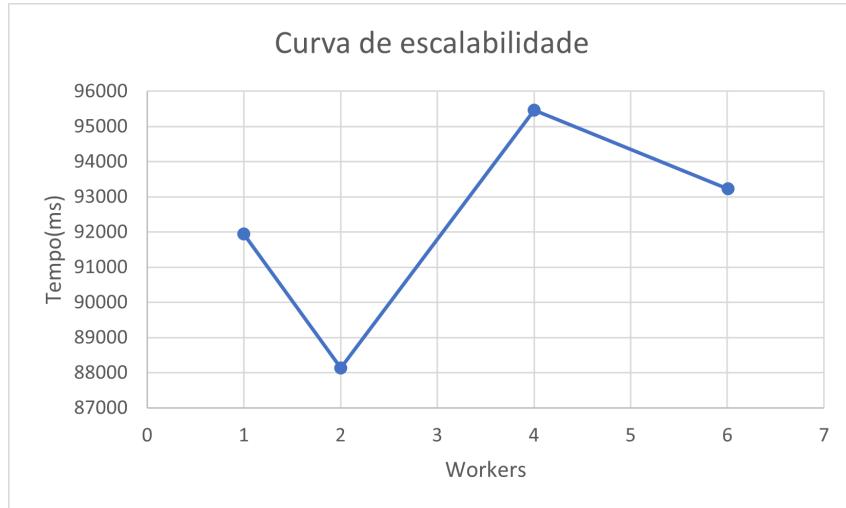


Figura 4.24: Curva de escalabilidade do paralelismo da interrogação analítica A1

Assim, chegámos à conclusão que não podemos tirar partido do paralelismo para otimizar ainda mais a interrogação analítica A1.

4.2 Interrogação analítica A.2

```
WITH revenue (supplier_no, total_revenue) AS (
```

```

SELECT mod((s_w_id * s_i_id), 10000) AS supplier_no,
       sum(ol_amount) AS total_revenue
  FROM order_line, stock
 WHERE ol_i_id = s_i_id
       AND ol_supply_w_id = s_w_id
       AND ol_delivery_d >= '2022-01-01 00:00:00.000000'
 GROUP BY mod((s_w_id * s_i_id), 10000))
SELECT su_suppkey, su_name, su_address, su_phone, total_revenue
  FROM supplier, revenue
 WHERE su_suppkey = supplier_no
       AND total_revenue IN (
           SELECT total_revenue
             FROM revenue
            ORDER BY 1 DESC
            LIMIT 100)
 ORDER BY su_suppkey;

```

Esta interrogação analítica identifica os 100 fornecedores que mais contribuíram para a receita global por ordem do seu número identificador num dado período de tempo. Começamos por analisar o plano de execução criado pelo *PostgreSQL* com o comando EXPLAIN ANALYZE.

Assim sendo, o grupo deparou-se com um tempo de execução de 30480.656ms. Como tal, procedeu-se à otimização começando pela criação de índices.

```

QUERY PLAN
-----
Merge Join  (cost=2390235.08..2398685.08 rows=480000 width=103) (actual time=30217.723..30460.494 rows=99 loops=1)
  Merge Cond: (supplier.su_suppkey = revenue.supplier_no)
    CTE revenue
      -> Finalize GroupAggregate  (cost=1992660.62..2247876.26 rows=960000 width=36) (actual time=29813.625..30081.825 rows=10000 loops=1)
          Group Key: (mod((stock.s_w_id * stock.s_i_id), 10000))
          -> Gather Merge  (cost=1992660.62..2216676.26 rows=1920000 width=36) (actual time=29813.586..30062.320 rows=30000 loops=1)
              Workers Planned: 2
              Workers Launched: 2
              -> Sort  (cost=1991660.60..1994060.60 rows=960000 width=36) (actual time=29729.332..29731.173 rows=10000 loops=3)
                  Sort Key: (mod((stock.s_w_id * stock.s_i_id), 10000))
                  Sort Method: quicksort  Memory: 1791kB
                  Worker 0:  Sort Method: quicksort  Memory: 1791kB
                  Worker 1:  Sort Method: quicksort  Memory: 1791kB
                  -> Partial HashAggregate  (cost=1774535.57..1870021.76 rows=960000 width=36) (actual time=29241.201..29725.333 rows=10000 loops=3)
                      Group Key: (mod((stock.s_w_id * stock.s_i_id), 10000))
                      Planned Partitions: 128  Batches: 129  Memory Usage: 4241kB  Disk Usage: 89608kB
                      Worker 0:  Batches: 129  Memory Usage: 4241kB  Disk Usage: 57864kB
                      Worker 1:  Batches: 129  Memory Usage: 4241kB  Disk Usage: 57864kB
                      -> Parallel Hash Join  (cost=551989.00..1207995.02 rows=10071832 width=7) (actual time=21674.689..25841.026 rows=8158702 loops=3)
                          Hash Cond: ((order_line.ol_i_id = stock.s_i_id) AND (order_line.ol_supply_w_id = stock.s_w_id))
                          -> Parallel Seq Scan on order_line  (cost=0.00..436901.72 rows=10197358 width=11) (actual time=0.907..7975.619 rows=8158702 loops=3)
                              Filter: (ol_delivery_d >= '2022-01-01 00:00:00'::timestamp without time zone)
                          -> Parallel Hash  (cost=476364.00..476364.00 rows=4000000 width=8) (actual time=1575.501..11575.503 rows=3200000 loops=3)
                              Buckets: 131072  Batches: 128  Memory Usage: 4032kB
                              -> Parallel Seq Scan on stock  (cost=0.00..476364.00 rows=4000000 width=8) (actual time=165.607..10636.650 rows=3200000 loops=3)
                          -> Sort  (cost=986.39..1011.39 rows=10000 width=71) (actual time=367.493..368.410 rows=9921 loops=1)
                              Sort Key: supplier.su_suppkey
                              Sort Method: quicksort  Memory: 1791kB
                              -> Seq Scan on supplier  (cost=0.00..322.00 rows=10000 width=71) (actual time=363.363..364.948 rows=10000 loops=1)
-> Materialize  (cost=141372.43..143772.43 rows=480000 width=36) (actual time=29850.185..29850.222 rows=100 loops=1)
  -> Sort  (cost=141372.43..142572.43 rows=480000 width=36) (actual time=29850.181..29850.197 rows=100 loops=1)
      Sort Key: revenue.supplier_no
      Sort Method: quicksort  Memory: 29kB
      -> Hash Semi Join  (cost=55893.01..32953.01 rows=480000 width=36) (actual time=29846.491..29850.145 rows=100 loops=1)
          Hash Cond: (revenue.total_revenue = revenue.i_total_revenue)
          -> CTE Scan on revenue  (cost=0.00..19200.00 rows=960000 width=36) (actual time=29813.630..29815.153 rows=10000 loops=1)
          -> Hash  (cost=55891.76..55891.76 rows=100 width=32) (actual time=32.830..32.833 rows=100 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 12kB
              -> Limit  (cost=55890.51..55890.76 rows=100 width=32) (actual time=32.760..32.782 rows=100 loops=1)
                  -> Sort  (cost=55890.51..58290.51 rows=960000 width=32) (actual time=32.756..32.767 rows=100 loops=1)
                      Sort Key: revenue.i_total_revenue DESC
                      Sort Method: top-N heapsort  Memory: 32kB
                      -> CTE Scan on revenue_1  (cost=0.00..19200.00 rows=960000 width=32) (actual time=0.003..30.429 rows=10000 loops=1)

Planning Time: 0.599 ms
JIT:
  Functions: 90
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 11.471 ms, Inlining 242.394 ms, Optimization 499.812 ms, Emission 272.973 ms, Total 1026.650 ms
Execution Time: 30480.656 ms
(49 rows)

```

Figura 4.25: Execução da *query 2* original

Per node type stats			
node type	count	sum of times	% of query
CTE Scan	2	0.000 ms	0.0 %
Finalize GroupAggregate	1	19.505 ms	0.1 %
Gather Merge	1	331.147 ms	1.1 %
Hash	1	0.051 ms	0.0 %
Hash Semi Join	1	2.159 ms	0.0 %
Limit	1	0.015 ms	0.0 %
Materialize	1	0.025 ms	0.0 %
Merge Join	1	241.862 ms	0.8 %
Parallel Hash	1	938.853 ms	3.1 %
Parallel Hash Join	1	6,289.904 ms	20.6 %
Parallel Seq Scan	2	18,612.269 ms	61.1 %
Partial HashAggregate	1	3,884.307 ms	12.8 %
Seq Scan	1	364.948 ms	1.2 %
Sort	4	11.692 ms	0.0 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
order_line	1	7,975.619 ms	26.2 %
Parallel Seq Scan	1	7,975.619 ms	100.0 %
stock	1	10,636.650 ms	34.9 %
Parallel Seq Scan	1	10,636.650 ms	100.0 %
supplier	1	364.948 ms	1.2 %
Seq Scan	1	364.948 ms	100.0 %

Figura 4.26: Visualização da *query 2* original

4.2.1 Índices

Como primeira tentativa, o grupo decidiu investir algum tempo na criação de índices. Ao analisar a interrogação reparamos que esta filtrava as linhas da tabela *order_line* cuja data de entrega fosse superior a um determinado valor. Além disso, era feita a união com as tabelas *order_line* e *stock* onde o *ol_supply_w_id* fosse igual a *s_w_id*. Finalmente, era feita a união com a tabela *supplier* pela *su_suppkey*. Assim sendo, e atendendo à estrutura da *query*, o grupo decidiu proceder à criação dos índices que se encontram em seguida:

```
CREATE INDEX q3.ol_supply_w_id ON order_line (ol_supply_w_id);
CREATE INDEX q3.s_w_id ON stock (s_w_id);
CREATE INDEX q3.ol_delivery_d ON order_line (ol_delivery_d);
CREATE INDEX q3.su_suppkey ON supplier (su_suppkey);
```


vista ter de ser atualizada raramente. Com isto, deixa de ser necessário não só aceder à tabela *stock* como ao cálculo em questão durante a execução da interrogação.

Foi criada a seguinte vista materializada:

```
CREATE MATERIALIZED VIEW q2_1 AS SELECT s_w_id, s_i_id, mod((s_w_id * s_i_id), 10000)
AS supplier_no FROM stock;
```

Assim, para podermos utilizar esta vista, foi necessário alterar o código para o seguinte:

```
WITH revenue (supplier_no, total_revenue) AS (
    SELECT supplier_no, sum(ol_amount) AS total_revenue
    FROM order_line, q2_1
    WHERE ol_i_id = s_i_id AND ol_supply_w_id = s_w_id
        AND ol_delivery_d >= '2022-01-01 00:00:00.000000'
    GROUP BY supplier_no)
SELECT su_suppkey, su_name, su_address, su_phone, total_revenue
FROM supplier, revenue
WHERE su_suppkey = supplier_no
    AND total_revenue IN (SELECT total_revenue
    FROM revenue ORDER BY 1 DESC LIMIT 100)
ORDER BY su_suppkey;
```

Após a execução, foi verificado que o impacto desta vista não foi muito relevante, passando apenas de 15240.197ms para 14693.870ms, o que não foi de acordo com as nossas expectativas.

The screenshot shows the Oracle SQL query plan for the execution of query 2. The plan starts with a Sort operation on the 'revenue' view, followed by a Gather Merge operation on the 'order_line' table. It then involves several Hash Joins and Hash Semi Joins between the 'supplier' table and the 'revenue' view. The plan also includes CTE Scans and a CTE Scan on the 'revenue' view. The execution time is shown as 14683.870 ms, and the total number of rows processed is 47.

Figura 4.31: Execução da *query 2* com índices e Vista Materializada 1

Per node type stats			
node type	count	sum of times	% of query
CTE Scan	2	0.000 ms	0.0 %
Finalize GroupAggregate	1	17.430 ms	0.1 %
Gather Merge	1	337.676 ms	2.3 %
Hash	2	2.072 ms	0.0 %
Hash Join	1	0.104 ms	0.0 %
Hash Semi Join	1	1.215 ms	0.0 %
Limit	1	0.015 ms	0.0 %
Parallel Hash	1	998.946 ms	6.8 %
Parallel Hash Join	1	6,350.330 ms	43.3 %
Parallel Seq Scan	2	2,847.171 ms	19.4 %
Partial HashAggregate	1	3,780.375 ms	25.8 %
Seq Scan	1	331.119 ms	2.3 %
Sort	3	308.636 ms	2.1 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
order_line	1	2,059.137 ms	14.0 %
Parallel Seq Scan	1	2,059.137 ms	100.0 %
q2_1	1	788.034 ms	5.4 %
Parallel Seq Scan	1	788.034 ms	100.0 %
supplier	1	331.119 ms	2.3 %
Seq Scan	1	331.119 ms	100.0 %

Figura 4.32: Visualização da *query* 2 com índices e Vista Materializada 1

4.2.3 Vista Materializada 2

Insatisfeitos com o resultado anterior, o grupo decidiu inserir mais lógica da interrogação na vista materializada. Reparamos que era feita a união com as colunas da tabela *stock* (*ol_i_id = s_i_id and ol_supply_w_id = s_w_id*). Assim sendo, passamos a fazer esta união diretamente na vista materializada, prevenindo o filtro dessas operações em tempo de execução.

```
CREATE MATERIALIZED VIEW q2_2
AS SELECT (mod((s_w_id * s_i_id),10000))
          AS supplier_no, ol_delivery_d, ol_amount from stock, order_line
WHERE ol_i_id = s_i_id and ol_supply_w_id = s_w_id;
```

A interrogação a ser executada passa a ser a seguinte:

```
WITH revenue (supplier_no, total_revenue) AS (
    SELECT supplier_no, sum(ol_amount) AS total_revenue
    FROM q2_2
    WHERE ol_delivery_d >= '2022-01-01 00:00:00.000000'
    GROUP BY supplier_no)
SELECT su_suppkey, su_name, su_address, su_phone, total_revenue
FROM supplier, revenue
WHERE su_suppkey = supplier_no
    AND total_revenue IN (SELECT total_revenue
    FROM revenue ORDER BY 1 DESC LIMIT 100)
ORDER BY su_suppkey;
```

Esta alteração teve um impacto positivo no tempo de execução da interrogação analítica passando para 5161.055ms.

```

-----  

        QUERY PLAN  

-----  

Sort  (cost=340052.59..340064.02 rows=4572 width=103) (actual time=5151.408..5155.084 rows=99 loops=1)
  Sort Key: supplier.su_supkey
  Sort Method: quicksort  Memory: 38kB
  CTE revenue
    -> Finalize GroupAggregate (cost=336086.63..338472.10 rows=9145 width=36) (actual time=5091.401..5123.445 rows=10000 loops=1)
        Group Key: q2_2.supplier_no
      -> Gather Merge (cost=336086.63..338220.61 rows=18290 width=36) (actual time=5091.363..5103.634 rows=30000 loops=1)
          Workers Planned: 2
          Workers Launched: 2
        -> Sort (cost=335086.61..335109.47 rows=9145 width=36) (actual time=5065.327..5067.350 rows=10000 loops=3)
            Sort Key: q2_2.supplier_no
            Sort Method: quicksort  Memory: 1791kB
            Worker 0: Sort Method: quicksort  Memory: 1791kB
            Worker 1: Sort Method: quicksort  Memory: 1791kB
          -> Partial HashAggregate (cost=334370.61..334484.92 rows=9145 width=36) (actual time=4945.385..5060.587 rows=10000 loops=3)
              Group Key: q2_2.supplier_no
              Batches: 5  Memory Usage: 4273kB  Disk Usage: 7456kB
              Worker 0: Batches: 5  Memory Usage: 4273kB  Disk Usage: 7488kB
              Worker 1: Batches: 5  Memory Usage: 4273kB  Disk Usage: 12408kB
            -> Parallel Seq Scan on q2_2 (cost=0.00..283378.72 rows=10198378 width=7) (actual time=6.049..1477.035 rows=8158702 loops=3)
                Filter: (ol_delivery_d >= '2022-01-01 00:00:00'::timestamp without time zone)
-> Hash Join (cost=981.92..1302.55 rows=4572 width=103) (actual time=5147.713..5151.367 rows=99 loops=1)
  Hash Cond: (revenue.supplier_no = supplier.su_supkey)
-> Hash Semi Join (cost=534.92..792.68 rows=4572 width=36) (actual time=5125.409..5128.998 rows=100 loops=1)
  Hash Cond: (revenue.total_revenue = revenue_1.total_revenue)
    -> CTE Scan on revenue (cost=0.00..182.90 rows=9145 width=36) (actual time=5091.405..5092.840 rows=10000 loops=1)
-> Hash (cost=533.67..533.67 rows=100 width=32) (actual time=33.979..33.983 rows=100 loops=1)
  Buckets: 1024  Batches: 1  Memory Usage: 12kB
    -> Limit (cost=532.42..532.67 rows=100 width=32) (actual time=33.909..33.930 rows=100 loops=1)
      -> Sort (cost=532.42..555.28 rows=9145 width=32) (actual time=33.906..33.915 rows=100 loops=1)
        Sort Key: revenue.total_revenue DESC
        Sort Method: top-N heapsort  Memory: 32kB
      -> CTE Scan on revenue revenue_1 (cost=0.00..182.90 rows=9145 width=32) (actual time=0.002..31.597 rows=10000 loops=1)
-> Hash (cost=322.00..322.00 rows=10000 width=71) (actual time=22.218..22.219 rows=10000 loops=1)
  Buckets: 16384  Batches: 1  Memory Usage: 1144kB
    -> Seq Scan on supplier (cost=0.00..322.00 rows=10000 width=71) (actual time=17.479..20.038 rows=10000 loops=1)
Planning Time: 0.335 ms
JIT:
  Functions: 62
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 9.556 ms, Inlining 0.000 ms, Optimization 3.631 ms, Emission 49.948 ms, Total 63.134 ms
Execution Time: 5161.055 ms
(42 rows)

```

Figura 4.33: Execução da query 2 com Vista Materializada 2

Per node type stats			
node type	count	sum of times	% of query
CTE Scan	2	0.992 ms	0.0 %
Finalize GroupAggregate	1	19.811 ms	0.4 %
Gather Merge	1	36.284 ms	0.7 %
Hash	2	2.234 ms	0.0 %
Hash Join	1	0.150 ms	0.0 %
Hash Semi Join	1	2.175 ms	0.0 %
Limit	1	0.015 ms	0.0 %
Parallel Seq Scan	1	1,477.035 ms	28.7 %
Partial HashAggregate	1	3,583.552 ms	69.5 %
Seq Scan	1	20.038 ms	0.4 %
Sort	3	12.798 ms	0.2 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
q2_2	1	1,477.035 ms	28.7 %
Parallel Seq Scan	1	1,477.035 ms	100.0 %
supplier	1	20.038 ms	0.4 %
Seq Scan	1	20.038 ms	100.0 %

Figura 4.34: Visualização da query 2 com Vista Materializada 2

4.2.4 Vista Materializada 2 com Índice

Ainda em busca de uma melhoria decidimos usar um índice para a vista materializada definida anteriormente. Após analisar o EXPLAIN ANALIZE obtido anteriormente reparámos que o *supplier_no* era utilizado na *Group Key* :

```

...
Group Key: q2_2.supplier_no
...

```

Assim pensámos que poderia ajudar criar um índice para *supplier_no* na vista materializada 2, como o seguinte:

```
CREATE INDEX q2_mv2_2 ON q2_2(supplier_no);
```

Após a execução da interrogação indicada anteriormente para utilizar a vista materializada 2, foi verificado que não houveram melhorias, ficando o valor do tempo praticamente inalterado. Olhando depois para o EXPLAIN ANALYZE verificou-se que este índice não tinha sido utilizado pelo *postgres*, não se tornando útil e contrariando o pensamento de possível melhoria.

4.2.5 Vista Materializada 3

Chegando à última tentativa de otimizar esta *query*, procedemos à criação de uma nova vista materializada. Decidimos então materializar toda a parte superior da interrogação, visto que a maior parte do tempo passado pela *query* se verifica no GROUP BY aí presente. Assim, a vista materializada resultante encontra-se em seguida:

```
CREATE MATERIALIZED VIEW q2_3
AS SELECT mod((s_w_id * s_i_id), 10000) AS supplier_no,
sum(ol_amount) AS total_revenue
FROM order_line, stock
WHERE ol_i_id = s_i_id
    AND ol_supply_w_id = s_w_id
    AND ol_delivery_d >= '2022-01-01 00:00:00.000000'
GROUP BY mod((s_w_id * s_i_id), 10000);

WITH revenue (supplier_no, total_revenue) AS (
    SELECT * FROM q2_3)
SELECT su_suppkey, su_name, su_address, su_phone, total_revenue
FROM supplier, revenue
WHERE su_suppkey = supplier_no
AND total_revenue IN (SELECT total_revenue
FROM revenue ORDER BY 1 DESC LIMIT 100)
ORDER BY su_suppkey;
```

```
QUERY PLAN
Sort  (cost=1844.51..1857.01 rows=5000 width=103) (actual time=10.927..10.937 rows=99 loops=1)
  Sort Key: supplier.su_suppkey
  Sort Method: quicksort  Memory: 38kB
  CTE revenue
    -> Seq Scan on q2_3  (cost=0.00..155.00 rows=10000 width=10) (actual time=0.011..0.690 rows=10000 loops=1)
    -> Hash Join  (cost=1031.69..1382.32 rows=5000 width=103) (actual time=8.809..10.896 rows=99 loops=1)
      Hash Cond: (revenue.supplier_no = supplier.su_suppkey)
      -> Hash Semi Join  (cost=584.69..866.57 rows=5000 width=36) (actual time=4.528..6.568 rows=100 loops=1)
        Hash Cond: (revenue.total_revenue = revenue_1.total_revenue)
        -> CTE Scan on revenue  (cost=0.00..200.00 rows=10000 width=36) (actual time=0.014..0.796 rows=10000 loops=1)
        -> Hash  (cost=583.44..583.44 rows=100 width=32) (actual time=4.493..4.494 rows=100 loops=1)
          Buckets: 1024  Batches: 1  Memory Usage: 12kB
          -> Limit  (cost=582.19..582.44 rows=100 width=32) (actual time=4.458..4.471 rows=100 loops=1)
            -> Sort  (cost=582.19..607.19 rows=10000 width=32) (actual time=4.455..4.461 rows=100 loops=1)
              Sort Key: revenue_1.total_revenue  DESC
              Sort Method: top-N heapsort  Memory: 32kB
            -> CTE Scan on revenue_1  (cost=0.00..200.00 rows=10000 width=32) (actual time=0.002..2.493 rows=10000 loops=1)
              Buckets: 16384  Batches: 1  Memory Usage: 1144kB
              -> Seq Scan on supplier  (cost=0.00..322.00 rows=10000 width=71) (actual time=0.008..2.267 rows=10000 loops=1)
Planning Time: 0.575 ms
Execution Time: 11.075 ms
(22 rows)
```

Figura 4.35: Execução da *query* 2 com Vista Materializada 3

No entanto, apesar do tempo de execução mínimo, ao inserir na vista materializada a condição sobre a data, a interrogação perde toda a sua flexibilidade uma vez que o utilizador deixa de poder escolher qualquer data que pretenda para esta *query*, restringindo-se a uma única. Além disso, como a tabela *order_line* não é uma tabela imutável, caso não existisse a atualização frequente da vista, os dados materializados ficariam rapidamente desatualizados, retirando a elasticidade à interrogação analítica.

4.2.6 Paralelismo

Igualmente ao que foi feito nas restantes interrogações, outro modo de possivelmente otimizar esta interrogação analítica é aumentar (ou diminuir) o nível de paralelismo que está a ser utilizado, ao definir o número de *workers* do *PostgreSQL*. Estes testes foram feitos sob a *query* original, sem otimizações.

Workers	Tempo de resposta (ms)
1	16074,281
2	14525,679
4	14227,621
6	14752,194

Tabela 4.1: Tempo de execução com diferente número de *workers* - *query 2*

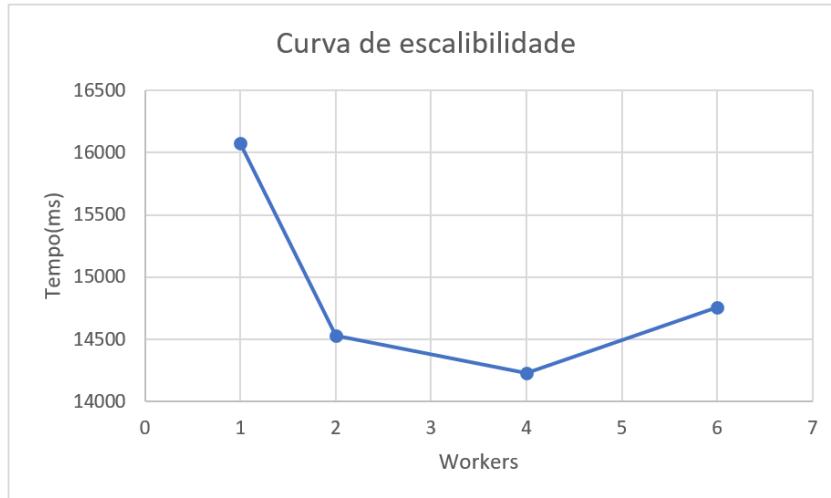


Figura 4.36: Curva de escalabilidade do paralelismo da interrogação analítica A2

Pela análise da tabela, vemos que quando há apenas um *worker* o tempo de execução é maior, o que seria de esperar pois não há distribuição de carga. Quanto aos restantes valores, com número de *workers* igual a 2,4 e 6 a diferença observada é mínima, isto verifica-se porque o ganho que se obtém por distribuir o trabalho, é perdido depois no *overhead* de criar mais *workers*.

4.3 Interrogação analítica A.3

```
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 trim(i_data) LIKE '%b'
        AND ol_i_id = i_id
    GROUP BY i_id) t
WHERE ol_i_id = t.i_id
    AND ol_quantity < t.a;
```

Para esta interrogação analítica foi inicialmente analisado o plano de execução criado pelo *PostgreSQL* com o comando EXPLAIN ANALYZE.

Assim sendo, o resultado do tempo de execução da *query* após o aquecimento da *cache* foi de 8012.956ms. Como tal, procedeu-se à otimização começando pela criação de índices.

```

QUERY PLAN
Aggregate (cost=1081602.66..1081602.67 rows=1 width=32) (actual time=8008.296..8008.609 rows=1 loops=1)
  -> Hash Join (Cost=459145.78..1077579.91 rows=1609100 width=3) (actual time=8008.272..8008.585 rows=0 loops=1)
    Hash Cond: (order_line.ol_i_id = item.i_id)
    Join Filter: ((order_line.ol_quantity)::numeric < (avg(order_line.ol_quantity)))
    Rows Removed by Join Filter: 620847
    -> Seq Scan on order_line (cost=0.00..554183.06 rows=24476106 width=11) (actual time=0.076..2504.306 rows=24476106 loops=1)
    -> Hash (cost=458895.78..458895.78 rows=20000 width=36) (actual time=3054.974..3055.285 rows=2542 loops=1)
      Buckets: 32768 Batches: 1 Memory Usage: 358kB
      -> Finalize GroupAggregate (cost=453578.79..458695.78 rows=20000 width=36) (actual time=3049.525..3054.578 rows=2542 loops=1)
      Group Key: item.i_id
      -> Gather Merge (cost=453578.79..458245.78 rows=40000 width=36) (actual time=3049.499..3051.762 rows=7626 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Sort (cost=452578.76..452628.76 rows=20000 width=36) (actual time=3031.593..3031.867 rows=2542 loops=3)
          Sort Key: item.i_id
          Sort Method: quicksort Memory: 295kB
          Worker 0: Sort Method: quicksort Memory: 295kB
          Worker 1: Sort Method: quicksort Memory: 295kB
          -> Partial HashAggregate (cost=450949.99..451149.99 rows=20000 width=36) (actual time=3029.556..3030.606 rows=2542 loops=3)
            Group Key: item.i_id
            Batches: 1 Memory Usage: 1297kB
            Worker 0: Batches: 1 Memory Usage: 1297kB
            Worker 1: Batches: 1 Memory Usage: 1297kB
            -> Parallel Hash Join (cost=2715.47..440893.12 rows=2011375 width=8) (actual time=535.513..2922.185 rows=206949 loops=3)
              Hash Cond: (order_line.ol_i_id = item.i_id)
              -> Parallel Seq Scan on order_line order_line_1 (cost=0.00..411405.78 rows=10198378 width=8) (actual time=0.046..1214.611 rows=8158702 loops=3)
              -> Parallel Hash (cost=2568.41..2568.41 rows=11765 width=4) (actual time=535.216..535.218 rows=847 loops=3)
                Buckets: 32768 Batches: 1 Memory Usage: 416kB
                -> Parallel Seq Scan on item (cost=0.00..2568.41 rows=11765 width=4) (actual time=406.978..431.515 rows=847 loops=3)
                  Filter: (TRIM(BOTH FROM i_data) ~~ '%5':;text)
                  Rows Removed by Filter: 3246
Planning Time: 0.357 ms
JIT:
  Functions: 70
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 9.374 ms, Inlining 250.729 ms, Optimization 686.939 ms, Emission 281.745 ms, Total 1228.786 ms
Execution Time: 8012.956 ms
(37 rows)

```

Figura 4.37: EXPLAIN ANALYZE da interrogação analítica A3

Per node type stats

node type	count	sum of times	% of query
Aggregate	1	0.024 ms	0.0 %
Finalize GroupAggregate	1	2.816 ms	0.0 %
Gather Merge	1	19.895 ms	0.2 %
Hash	1	0.707 ms	0.0 %
Hash Join	1	2,448.994 ms	30.6 %
Parallel Hash	1	103.703 ms	1.3 %
Parallel Hash Join	1	1,172.356 ms	14.6 %
Parallel Seq Scan	2	1,646.126 ms	20.6 %
Partial HashAggregate	1	108.421 ms	1.4 %
Seq Scan	1	2,504.306 ms	31.3 %
Sort	1	1.261 ms	0.0 %

Per table stats

Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
item	1	431.515 ms	5.4 %
Parallel Seq Scan	1	431.515 ms	100.0 %
order_line	2	3,718.917 ms	46.4 %
Parallel Seq Scan	1	1,214.611 ms	32.7 %
Seq Scan	1	2,504.306 ms	67.3 %

Figura 4.38: Estatísticas do plano de execução original da interrogação analítica A3

4.3.1 Índices

Como primeira tentativa, o grupo decidiu testar a influência de alguns índices. Atendendo à estrutura da interrogação, verificamos que eram feitos filtros utilizando os valores *i_data* e *ol_amount* pelo que criamos índices nestas duas colunas para tentar facilitar a sua procura e desta forma reduzir o tempo de execução.

```

CREATE INDEX q3_i_data on item(i_data);
CREATE INDEX q3.ol_quantity on order_line(ol_quantity);

```

```

QUERY PLAN
Aggregate  (cost=1081602.66..1081602.67 rows=1 width=32) (actual time=8093.956..8094.286 rows=1 loops=1)
  > Hash Join  (cost=459145.78..1077579.91 rows=1609100 width=3) (actual time=8093.921..8094.250 rows=0 loops=1)
    Hash Cond: (order_line.ol_i_id = item.i_id)
    Join Filter: ((order_line.ol_quantity)::numeric < (avg(order_line.ol_quantity)))
    Rows Removed by Join Filter: 620847
    > Seq Scan on order_line  (cost=0.00..554183.06 rows=24476106 width=11) (actual time=0.060..2670.573 rows=24476106 loops=1)
    > Hash  (cost=458895..78..458895.78 rows=20000 width=36) (actual time=3035.738..3036.065 rows=2542 loops=1)
      Buckets: 32768  Batches: 1  Memory Usage: 358KB
      > Finalize GroupAggregate  (cost=453578..79..458695.78 rows=20000 width=36) (actual time=3030.962..3035.516 rows=2542 loops=1)
        Group Key: item.i_id
        > Gather Merge  (cost=453578..79..458245.78 rows=40000 width=36) (actual time=3030.929..3033.196 rows=7626 loops=2)
          Workers Planned: 2
          Workers Launched: 2
          > Sort  (cost=452578..76..452628.76 rows=20000 width=36) (actual time=3009.327..3009.709 rows=2542 loops=3)
            Sort Key: item.i_id
            Sort Method: quicksort  Memory: 295kB
            Worker 0: Sort Method: quicksort  Memory: 295kB
            Worker 1: Sort Method: quicksort  Memory: 295kB
            > Partial HashAggregate  (cost=450949.99..451149.99 rows=20000 width=36) (actual time=3007.069..3008.324 rows=2542 loops=3)
              Hash Cond: (order_line.ol_i_id = item.i_id)
              Buckets: 32768  Batches: 1  Memory Usage: 1297KB
              Worker 0: Batches: 1  Memory Usage: 1297KB
              Worker 1: Batches: 1  Memory Usage: 1297KB
              > Parallel Hash Join  (cost=2715.47..440893.12 rows=2011375 width=8) (actual time=453.133..2885.442 rows=206949 loops=3)
                Hash Cond: (order_line.ol_i_id = item.i_id)
                > Parallel Seq Scan on order_line order_line_1  (cost=0.00..411405.78 rows=10198378 width=8) (actual time=0.052..1290.291 rows=8158702 loops=3)
                  Buckets: 32768  Batches: 1  Memory Usage: 384KB
                  > Parallel Seq Scan on item  (cost=0.00..2568.41 rows=11765 width=4) (actual time=416.681..452.279 rows=847 loops=3)
                    Filter: (TRIM(BOTH FROM i.data) ~~ '#\b'::text)
                    Rows Removed by Filter: 32486
Planning Time: 0.422 ms
JIT:
  Functions: 70
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 11.128 ms, Inlining 280.502 ms, Optimization 619.276 ms, Emission 348.085 ms, Total 1258.990 ms
Execution Time: 8098.808 ms
(37 rows)

```

Figura 4.39: EXPLAIN ANALYZE da interrogação analítica A3 com índices

Per node type stats			
node type	count	sum of times	% of query
Aggregate	1	0.036 ms	0.0 %
Finalize GroupAggregate	1	2.320 ms	0.0 %
Gather Merge	1	23.487 ms	0.3 %
Hash	1	0.549 ms	0.0 %
Hash Join	1	2,387.612 ms	29.5 %
Parallel Hash	1	0.529 ms	0.0 %
Parallel Hash Join	1	1,142.343 ms	14.1 %
Parallel Seq Scan	2	1,742.570 ms	21.5 %
Partial HashAggregate	1	122.882 ms	1.5 %
Seq Scan	1	2,670.573 ms	33.0 %
Sort	1	1.385 ms	0.0 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
item	1	452.279 ms	5.6 %
Parallel Seq Scan	1	452.279 ms	100.0 %
order_line	2	3,960.864 ms	48.9 %
Parallel Seq Scan	1	1,290.291 ms	32.6 %
Seq Scan	1	2,670.573 ms	67.4 %

Figura 4.40: Estatísticas do plano de execução da interrogação analítica A3 com índices

Como se pode ver, a interrogação teve um tempo de execução de 8098.808ms, não havendo diferenças em relação à iteração anterior. Analisando o plano de execução podemos verificar que nenhum índice foi utilizado, incluindo aqueles que já existiam (Figura 4.1). Uma vez que estávamos convencidos que os índices deveriam melhorar o tempo de execução da interrogação, decidimos alterar o *seqscan* para *off* no ficheiro de configuração do PostgreSQL, para ser forçado o uso de índices.

```

QUERY PLAN
Aggregate (cost=2285330.99..2295321.00 rows=1 width=32) (actual time=2303.301..2308.787 rows=1 loops=1)
  > Nested Loop (cost=1001.18..2291308.24 rows=1609100 width=3) (actual time=2303.285..2308.770 rows=1 loops=1)
    > Finalize GroupAggregate (cost=1000.74..865533.07 rows=20000 width=36) (actual time=310.607..1755.066 rows=2542 loops=1)
      Group Key: item.i_id
      > Gather Merge (cost=1000.74..865183.07 rows=20000 width=36) (actual time=309.428..1749.368 rows=2542 loops=1)
        Workers Planned: 1
        Workers Launched: 1
        > Partial GroupAggregate (cost=0.73..861933.06 rows=20000 width=36) (actual time=271.375..1870.220 rows=1271 loops=2)
          Group Key: item.i_id
          > Nested Loop (cost=0.73..847535.12 rows=2839589 width=8) (actual time=269.818..1826.897 rows=310424 loops=2)
            > Parallel Index Scan using pk_item on item (cost=0.29..4175.70 rows=11765 width=4) (actual time=269.728..320.029 rows=1271 loops=2)
              Filter: (TRIM(BOTH FROM i_data) ~~ '%b'::text)
              Rows Removed by Filter: 48729
            > Index Scan using ix_order_line on order_line order_line_1 (cost=0.44..69.27 rows=241 width=8) (actual time=0.016..1.147 rows=244 loops=2542)
              Index Cond: (ol_i_id = item.i_id)
        > Index Scan using ix_order_line on order_line (cost=0.44..70.48 rows=80 width=11) (actual time=0.217..0.217 rows=0 loops=2542)
          Index Cond: (ol_i_id = item.i_id)
          Filter: ((ol_quantity)::numeric < (avg(order_line_1.ol_quantity)))
          Rows Removed by Filter: 244
Planning Time: 1.108 ms
 JIT:
  Functions: 34
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 4.606 ms, Inlining 104.476 ms, Optimization 266.115 ms, Emission 168.039 ms, Total 543.236 ms
Execution Time: 2311.941 ms
(25 rows)

```

Figura 4.41: EXPLAIN ANALYZE da interrogação analítica A3 com índices e *seqscan off*

Per node type stats			
node type	count	sum of times	% of query
Aggregate	1	0.017 ms	0.0 %
Finalize GroupAggregate	1	5.698 ms	0.2 %
Gather Merge	1	0.000 ms	0.0 %
Index Scan	2	2,009.451 ms	87.0 %
Nested Loop	2	51.121 ms	2.2 %
Parallel Index Scan	1	320.029 ms	13.9 %
Partial GroupAggregate	1	43.323 ms	1.9 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
item	1	320.029 ms	13.9 %
Parallel Index Scan	1	320.029 ms	100.0 %
order_line	2	2,009.451 ms	87.0 %
Index Scan	2	2,009.451 ms	100.0 %

Figura 4.42: Estatísticas do plano de execução da interrogação analítica A3 com índices e *seqscan off*

Ora, como esperado, o tempo de execução passou de 8098.808ms para 2311.941ms. Embora não tenham sido utilizados os índices que criamos para a interrogação, foram utilizados os índices já existentes (*pk_item*, *ix_order_line*).

4.3.2 Vista Materializada 1

De modo a tentar melhorar a performance, não recorrendo à desativação do *seqscan*, procedemos à criação de uma vista materializada. Observando a *query* inicial podemos verificar que existe uma outra *query* aninhada no interior da interrogação A.3. Deste modo criou-se uma vista materializada para acomodar esta *query* da seguinte forma:

```

CREATE MATERIALIZED VIEW materialized_view_q3_1 AS
  SELECT i_id, avg(ol_quantity) AS a
    FROM item, order_line
   WHERE trim(i_data) LIKE '%b'
     AND ol_i_id = i_id
  GROUP BY i_id;

```

Para então correr a interrogação A.3 com a modificação alterada, foi necessário proceder ao seguinte modo:

```

EXPLAIN ANALYZE SELECT sum(ol_amount) / 2.0 AS avg_yearly
  FROM order_line, (SELECT * FROM materialized_view_q3_1) t
 WHERE ol_i_id = t.i_id
   AND ol_quantity < t.a;

```

```

QUERY PLAN
Finalize Aggregate (cost=454768.84..454768.85 rows=1 width=32) (actual time=2919.916..2925.891 rows=1 loops=1)
  -> Gather (cost=454768.62..454768.83 rows=2 width=32) (actual time=2919.657..2925.878 rows=3 loops=1)
    Workers Planned: 2
    Workers Launched: 2
      -> Partial Aggregate (cost=453768.62..453768.63 rows=1 width=32) (actual time=2898.686..2898.690 rows=1 loops=3)
        -> Hash Join (cost=71.20..453555.58 rows=85215 width=3) (actual time=2898.658..2898.661 rows=0 loops=3)
          Hash Cond: (order_line.ol_i_id = materialized_view_q3_1.i_id)
          Join Filter: ((order_line.ol_quantity)::numeric < materialized_view_q3_1.a)
          Rows Removed by Join Filter: 206949
          -> Parallel Seq Scan on order_line  (cost=0.00..411405.78 rows=10198378 width=11) (actual time=0.047..1252.940 rows=8158702 loops=3)
            -> Hash (cost=39.42..39.42 rows=2542 width=9) (actual time=17.575..17.576 rows=2542 loops=3)
              Buckets: 4096  Batches: 1  Memory Usage: 134kB
              -> Seq Scan on materialized_view_q3_1  (cost=0.00..39.42 rows=2542 width=9) (actual time=0.023..0.373 rows=2542 loops=3)

Planning Time: 0.613 ms
JIT:
  Functions: 44
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 8.630 ms, Inlining 0.000 ms, Optimization 3.925 ms, Emission 44.843 ms, Total 57.398 ms
Execution Time: 2928.694 ms
(19 rows)

```

Figura 4.43: EXPLAIN ANALYZE da interrogação analítica A3 com vista Materializada 1

Per node type stats

node type	count	sum of times	% of query
Finalize Aggregate	1	0.013 ms	0.0 %
Gather	1	27.188 ms	0.9 %
Hash	1	17.203 ms	0.6 %
Hash Join	1	1,628.145 ms	55.6 %
Parallel Seq Scan	1	1,252.940 ms	42.8 %
Partial Aggregate	1	0.029 ms	0.0 %
Seq Scan	1	0.373 ms	0.0 %

Per table stats

Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
materialized_view_q3_1	1	0.373 ms	0.0 %
Seq Scan	1	0.373 ms	100.0 %
order_line	1	1,252.940 ms	42.8 %
Parallel Seq Scan	1	1,252.940 ms	100.0 %

Figura 4.44: Estatísticas do plano de execução da interrogação analítica A3 com vista Materializada 1

Em seguida, com o objetivo de verificar se a remoção do *sequential scan* se demonstraria benéfico, como tinha ocorrido anteriormente, procedeu-se à alteração do *seqscan* para *off*. Desta vez não houveram grandes melhorias passando apenas de 2928.694ms para 2813.141ms. Apesar do PostgreSQL ter utilizado um dos índices, isto não se provou ser benéfico.

```

QUERY PLAN
Aggregate (cost=10001258489.43..10001258489.44 rows=1 width=32) (actual time=2811.442..2811.444 rows=1 loops=1)
  -> Nested Loop (cost=10000000000.44..10001257978.13 rows=204517 width=3) (actual time=2811.421..2811.422 rows=0 loops=1)
    -> Seq Scan on materialized_view_q3_1 (cost=10000000000.00..10000000039.42 rows=2542 width=9) (actual time=0.015..1.443 rows=2542 loops=1)
    -> Index Scan using ix_order_line on order_line  (cost=0.44..494.06 rows=80 width=11) (actual time=1.055..1.055 rows=0 loops=2542)
      Index Cond: (ol_i_id = materialized_view_q3_1.i_id)
      Filter: ((ol_quantity)::numeric < materialized_view_q3_1.a)
      Rows Removed by Filter: 244
Planning Time: 13.897 ms
JIT:
  Functions: 9
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 1.576 ms, Inlining 17.405 ms, Optimization 69.062 ms, Emission 39.259 ms, Total 127.301 ms
Execution Time: 2813.141 ms
(13 rows)

```

Figura 4.45: EXPLAIN ANALYZE da interrogação analítica A3 com vista Materializada 1

node type	count	sum of times	% of query
Aggregate	1	0.022 ms	0.0 %
Index Scan	1	2,681.810 ms	95.4 %
Nested Loop	1	128.169 ms	4.6 %
Seq Scan	1	1.443 ms	0.1 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
materialized_view_q3_1	1	1.443 ms	0.1 %
Seq Scan	1	1.443 ms	100.0 %
order_line	1	2,681.810 ms	95.4 %
Index Scan	1	2,681.810 ms	100.0 %

Figura 4.46: Estatísticas do plano de execução da interrogação analítica A3 com vista Materializada 1

4.3.3 Vista Materializada 1 com índice

Finalmente, para tentar minimizar o tempo de execução decidiu-se criar um índice para a vista materializada criada anteriormente. Para tal, foi necessário proceder à sua criação do seguinte modo:

```
CREATE INDEX q3_materialized_view_q3_1_i_id ON materialized_view_q3_1(i_id);
```

Após a execução verificou-se que este índice foi ignorado e não houveram alterações no tempo de execução. Para a última tentativa desligou-se novamente o *seqscan* e verificou-se uma ligeira melhoria para 2779.094ms.

```
QUERY PLAN
Aggregate (cost=1258541.42..1258541.43 rows=1 width=32) (actual time=2777.383..2777.384 rows=1 loops=1)
  > Nested Loop (cost=0.72..1258030.12 rows=204517 width=3) (actual time=2777.363..2777.364 rows=0 loops=1)
    -> Index Scan using q3_materialized_view_q3_1_i_id on materialized_view_q3_1 (cost=0.28..91.41 rows=2542 width=9) (actual time=0.014..1.969 rows=2542 loops=1)
      Index Cond: (ol_i_id = materialized_view_q3_1.i_id)
      Filter: ((ol_quantity)::numeric < materialized_view_q3_1.a)
      Rows Removed By Filter: 244
Planning Time: 0.400 ms
JIT:
  Functions: 9
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 1.574 ms, Inlining 17.530 ms, Optimization 64.438 ms, Emission 38.755 ms, Total 122.298 ms
Execution Time: 2779.094 ms
(13 rows)
```

Figura 4.47: EXPLAIN ANALYZE da interrogação analítica A3 com índice na vista Materializada 1

Per node type stats			
node type	count	sum of times	% of query
Aggregate	1	0.020 ms	0.0 %
Index Scan	2	2,653.275 ms	95.5 %
Nested Loop	1	124.089 ms	4.5 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
materialized_view_q3_1	1	1.969 ms	0.1 %
Index Scan	1	1.969 ms	100.0 %
order_line	1	2,651.306 ms	95.5 %
Index Scan	1	2,651.306 ms	100.0 %

Figura 4.48: Estatísticas do plano de execução da interrogação analítica A3 com índice na vista Materializada 1

Concluindo, a criação das vistas materializadas não se verificou útil neste caso visto não só precisarem de ser atualizadas de tempos em tempos como também não obtiveram o melhor tempo de execução. Sendo assim, a melhor otimização conseguida foi a com utilização de índices com o *seqcan* a *off*.

4.3.4 Paralelismo

Também para esta interrogação foi tido em conta o aumento ou diminuição do paralelismo como uma outra forma de otimização. Para isso foi necessário redefinir o número de *workers* do *PostgreSQL*. Os resultados aqui presentes são referentes à *query* original desprovida de outras otimizações.

Workers	Tempo de resposta (ms)
1	7871,178
2	7867,15
4	7887,805
6	8061,999

Tabela 4.2: Tempo de execução com diferente número de *workers* - *query* 3

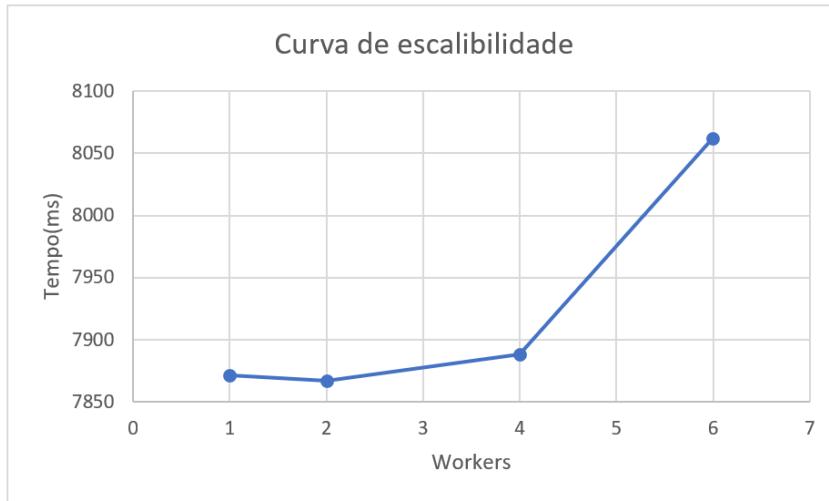


Figura 4.49: Curva de escalabilidade do paralelismo da interrogação analítica A3

Analisando a tabela e o gráfico é possível ver que a utilização de 1, 2 ou 4 *workers* pouco afeta a interrogação em termos de tempo de execução, isto pode dever-se ao facto de que o aumento do paralelismo não compensa/compensa pouco o *overhead* que a criação de novos *workers* injeta. Com 6 *workers* o ganho de paralelismo é tão pequeno que o custo da *overhead* se sobrepuja, prejudicando a performance global da *query*.

4.4 Interrogação analítica A.4

```

SELECT su_name, count(*) AS numwait
FROM supplier, order_line 11, orders, stock, nation
WHERE ol_o_id = o_id
    AND ol_w_id = o_w_id
    AND ol_d_id = o_d_id
    AND ol_w_id = s_w_id
    AND ol_i_id = s_i_id
    AND mod((s_w_id * s_i_id), 10000) = su_suppkey
    AND 11.ol_delivery_d > o_entry_d
    AND NOT EXISTS (
        SELECT *
        FROM order_line 12
        WHERE 12.ol_o_id = 11.ol_o_id
            AND 12.ol_w_id = 11.ol_w_id
            AND 12.ol_d_id = 11.ol_d_id
    )

```

```

        AND l2.ol_delivery_d > l1.ol_delivery_d)
AND su_nationkey = n_nationkey
AND n_name = 'GERMANY'
GROUP BY su_name
ORDER BY numwait DESC, su_name;

```

A interrogação analítica A4 tem como objetivo determinar os fornecedores que enviaram os itens de uma encomenda para um determinado país, que neste caso é a Alemanha. Utilizando o comando EXPLAIN ANALYZE, percebe-se que a *query* demora cerca de 35610 ms a executar.

```

QUERY PLAN
Sort  (cost=1606501.10..1606526.10 rows=10000 width=34) (actual time=35375.499..35601.702 rows=396 loops=1)
Sort Key: (count(*)) DESC, supplier.su_name
Sort Method: quicksort Memory: 55kB
-> Finalize GroupAggregate  (cost=1603129.52..1605836.71 rows=10000 width=34) (actual time=35271.587..35601.399 rows=396 loops=1)
    Group Key: supplier.su_name
-> Gather Merge  (cost=1603129.52..1605836.71 rows=20000 width=34) (actual time=35271.118..35600.988 rows=1188 loops=1)
    Workers Planned: 2
    Workers Launched: 2
-> Parallel GroupAggregate  (cost=1602129.50..1602328.19 rows=10000 width=34) (actual time=35165.151..35245.381 rows=396 loops=3)
    Group Key: supplier.su_name
-> Sort  (cost=1602129.50..1602162.40 rows=13159 width=26) (actual time=35165.068..35216.694 rows=203941 loops=3)
    Sort Key: supplier.su_name
    Sort Method: external merge Disk: 8344KB
    Worker 0: Sort Method: external merge Disk: 6400KB
    Worker 1: Sort Method: external merge Disk: 6880KB
-> Nested Loop Anti Join  (cost=1145058.56..1601229.17 rows=13159 width=26) (actual time=19588.572..34347.284 rows=203941 loops=3)
    -> Parallel Hash Join  (cost=1145057.99..1410425.41 rows=19738 width=46) (actual time=19586.969..20142.633 rows=308466 loops=3)
        Hash Cond: ((orders.o_id = l1.ol_o_id) AND (orders.o_w_id = l1.ol_w_id) AND (orders.o_d_id = l1.ol_d_id))
        Join Filter: (l1.ol_delivery_d > orders.o_entry_d)
-> Parallel Seq Scan on nation  (cost=0.00..1000000.00 rows=1000000 width=34) (actual time=679..896.182 rows=960000 loops=3)
-> Parallel Hash Scan on stock  (cost=1143442.08..1143442.08 rows=59252 width=50) (actual time=18384.158..18384.167 rows=308466 loops=3)
    Buckets: 65536 Batchsize: 32 (originally 4) Memory Usage: 3104KB
-> Parallel Hash Join  (cost=502325.92..1143442.09 rows=59252 width=50) (actual time=14752.356..18086.564 rows=308466 loops=3)
    Hash Cond: ((l1.ol_w_id = stock.s_w_id) AND (l1.ol_i_id = stock.s_i_id))
    -> Parallel Seq Scan on order_line 11  (cost=0.00..411405.78 rows=10198378 width=24) (actual time=0.050..4202.794 rows=8158702 loops=3)
    -> Parallel Hash  (cost=501972.97..501972.97 rows=23530 width=34) (actual time=7803.823..7803.831 rows=121160 loops=3)
        Buckets: 1024 Batches: 8 (originally 4) Memory Usage: 3104KB
    -> Hash Scan on stock  (cost=0.00..476364.00 rows=4000000 width=8) (actual time=491..6373.940 rows=3200000 loops=3)
-> Hash  (cost=372.23..372.23 rows=59 width=30) (actual time=849.866..849.870 rows=396 loops=3)
    Buckets: 1024 Batches: 1 Memory Usage: 33KB
-> Hash Join  (cost=12.14..372.23 rows=59 width=30) (actual time=845.293..849.787 rows=396 loops=3)
    Hash Cond: (supplier.su_nationkey = nation.n_nationkey)
-> Seq Scan on supplier  (cost=0.00..1000000.00 rows=1000000 width=34) (actual time=0.269..3.415 rows=1000000 loops=3)
-> Hash  (cost=12.12..12.12 rows=1 width=4) (actual time=844.970..844.970.870 rows=1 loops=3)
    Buckets: 1024 Batches: 1 Memory Usage: 9kB
-> Seq Scan on nation  (cost=0.00..12.12 rows=1 width=4) (actual time=844.962..844.965 rows=1 loops=3)
    Filter: (n_name = 'GERMANY')::bpchar
    Rows Removed by Filter: 24
-> Index Scan using pk_order_line on order_line 12  (cost=0.56..12.99 rows=3 width=20) (actual time=0.045..0.045 rows=0 loops=925397)
Index Cond: ((ol_w_id = l1.ol_w_id) AND (ol_d_id = l1.ol_d_id) AND (ol_o_id = l1.ol_o_id))
Filter: (ol_delivery_d > l1.ol_delivery_d)
Rows Removed by Filter: 8
Planning Time: 9.086 ms
 JIT:
 Functions: 180
 Options: Inlining true, Optimization true, Expressions true, Deforming true
 Timing: Generation 28.975 ms, Inlining 220.436 ms, Optimization 1548.509 ms, Emission 762.854 ms, Total 2560.874 ms
Execution Time: 35610.892 ms

```

Figura 4.50: Estatísticas do plano de execução original da interrogação analítica A4

De modo a conseguir perceber quais os pontos mais críticos da *query*, voltou a recorrer-se à ferramenta <https://explain.depesz.com/>. Como se pode observar, a *query* percorre as tabelas *nation*, *order_line*, *orders*, *stock* e *supplier* na sua totalidade, logo, o grupo tentou ajustar a *query*, de modo a que tal não se sucedesse. Além disso, verifica-se que o trabalho mais demorado na *query* é realizado sobre as tabelas *order_line*, *orders* e *stocks*, pelo que será necessário focar a análise nessas mesmas.

Per node type stats			
node type	count	sum of times	% of query
Finalize GroupAggregate	1	0.411 ms	0.0 %
Gather Merge	1	355.607 ms	1.0 %
Hash	2	0.090 ms	0.0 %
Hash Join	2	480.021 ms	1.3 %
Index Scan	1	13,880.955 ms	39.0 %
Nested Loop Anti Join	1	323.696 ms	0.9 %
Parallel Hash	2	399.003 ms	1.1 %
Parallel Hash Join	2	6,942.223 ms	19.5 %
Parallel Seq Scan	3	11,472.916 ms	32.2 %
Partial GroupAggregate	1	28.687 ms	0.1 %
Seq Scan	2	848.380 ms	2.4 %
Sort	2	869.713 ms	2.4 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
nation	1	844.965 ms	2.4 %
Seq Scan	1	844.965 ms	100.0 %
order_line	2	18,083.749 ms	50.8 %
Index Scan	1	13,880.955 ms	76.8 %
Parallel Seq Scan	1	4,202.794 ms	23.2 %
orders	1	896.182 ms	2.5 %
Parallel Seq Scan	1	896.182 ms	100.0 %
stock	1	6,373.940 ms	17.9 %
Parallel Seq Scan	1	6,373.940 ms	100.0 %
supplier	1	3.415 ms	0.0 %
Seq Scan	1	3.415 ms	100.0 %

Figura 4.51: Estatísticas do plano de execução original da interrogação analítica A4

4.4.1 Vista Materializada 1

Numa primeira análise, o grupo reparou que um dos *scans* mais pesados ocorria na tabela *stock*. As entradas da tabela *stock* apenas eram apenas usadas para realizar o cálculo $\text{mod}((s_w_id * s_i_id), 1000)$, portanto, o grupo optou por criar uma vista materializada, que armazenasse essa mesma informação. De forma a minimizar o tempo gasto na igualdade *s_number = su_suppkey*, ordenou-se a vista materializada segundo a coluna *s_value*.

```

CREATE MATERIALIZED VIEW A4_MV2 AS
SELECT s_i_id, s_w_id, mod((s_w_id * s_i_id), 10000) as s_value
FROM stock
ORDER BY s_value;

SELECT su_name, count(*) AS numwait
  FROM supplier, order_line l1, orders, A4_MV1, nation
 WHERE ol_o_id = o_id
   AND ol_w_id = o_w_id
   AND ol_d_id = o_d_id
   AND ol_w_id = s_w_id
   AND ol_i_id = s_i_id
   AND s_value = su_suppkey
   AND l1.ol_delivery_d > o_entry_d
   AND NOT EXISTS (
      SELECT *
        FROM order_line l2
       WHERE l2.ol_o_id = l1.ol_o_id
         AND l2.ol_w_id = l1.ol_w_id
         AND l2.ol_d_id = l1.ol_d_id
         AND l2.ol_delivery_d > l1.ol_delivery_d)
;
```

```

        AND su_nationkey = n_nationkey
        AND n_name = 'GERMANY'
    GROUP BY su_name
    ORDER BY numwait DESC, su_name;

```

4.4.2 Vista Materializada 1 com índice

De modo a melhorar ainda mais a performance, procedeu-se à criação de um índice para a vista materializada referida anteriormente. Assim, tendo em conta a igualdade $s_value = su_suppkey$, o grupo pensou que seria melhor criar um índice sobre a coluna s_value .

```
CREATE INDEX A4_i1 ON A4_MV1 (s_value);
```

Ao voltar a executar a *query*, com o auxílio do EXPLAIN ANALYZE, percebe-se que o tempo de execução baixou significativamente, passando a *query* a executar em cerca de 16904 ms, o que corresponde a menos de metade do tempo obtido ao executar a *query* original, demonstrando assim que a otimização é de facto válida.

```

QUERY PLAN
Sort  (cost=975863.40..975888.40 rows=10000 width=34) (actual time=16465.060..16896.829 rows=396 loops=1)
  Sort Key: (count(*)) DESC, supplier.su_name
  Sort Method: quicksort Memory: 55KB
-> Finalize GroupAggregate (cost=972492.16..975199.01 rows=10000 width=34) (actual time=16371.013..16896.528 rows=396 loops=1)
    Group Key: supplier.su_name
    Sort Method: quicksort Memory: 55KB
-> Gather Merge (cost=972492.16..974999.01 rows=20000 width=34) (actual time=16370.722..16896.121 rows=1188 loops=1)
  Workers Planned: 2
  Workers Actually Used: 2
-> Parallel GroupAggregate (cost=971492.14..971690.49 rows=10000 width=34) (actual time=16344.064..16429.211 rows=396 loops=3)
  Group Key: supplier.su_name
  Sort Method: external merge Disk: 9376kB
  Worker 0: Sort Method: external merge Disk: 6160kB
  Worker 1: Sort Method: external merge Disk: 6088kB
-> Nested Loop Anti Join (cost=107851.91..107851.91 rows=0 width=34) (actual time=6136.855..61544.719 rows=203941 loops=3)
  -> Parallel Hash Join (cost=107851.91..107851.91 rows=0 width=34) (actual time=6136.855..61544.719 rows=203941 loops=3)
    Hash Cond: ((11.ol_w_id = a4.mv1.s_w_id) AND (11.ol_i_id = a4.mv1.s_l_id))
    -> Parallel Seq Scan on order_line l1  (cost=0.00..411405.78 rows=203941 width=26) (actual time=6136.855..61544.719 rows=203941 loops=3)
    -> Parallel Hash (cost=107500.96..107500.96 rows=23530 width=34) (actual time=1534.689..1534.694 rows=121160 loops=3)
      Buckets: 65536 (originally 65536) Batches: 8 (originally 1) Memory Usage: 3744kB
      -> Hash Join (cost=372.96..107500.96 rows=23530 width=34) (actual time=720.955..1446.621 rows=121160 loops=3)
        Hash Cond: (a4.mv1.s_value = supplier.su_supkey)
        -> Parallel Seq Scan on mv1  (cost=0.00..411405.78 rows=4000000 width=12) (actual time=0.025..354.384 rows=3200000 loops=3)
        -> Hash (cost=3.22..3.9723 rows=1 width=1) (actual time=714.442..714.445 rows=396 loops=3)
          Buckets: 1024 Batches: 1 Memory Usage: 33kB
          -> Hash Join (cost=12.14..372.23 rows=59 width=30) (actual time=711.273..714.340 rows=396 loops=3)
            Hash Cond: (supplier.su_nationkey = nation.n_nationkey)
            -> Seq Scan on supplier  (cost=0.00..322.00 rows=10000 width=34) (actual time=0.012..1.597 rows=10000 loops=3)
            -> Hash (cost=12.12..12.12 rows=1 width=4) (actual time=711.228..711.230 rows=1 loops=3)
              Buckets: 1024 Batches: 1 Memory Usage: 9kB
              -> Seq Scan on nation  (cost=0.00..12.12 rows=1 width=4) (actual time=711.215..711.219 rows=1 loops=3)
                Filter: (n_name = 'GERMANY'::bpchar)
                Rows Removed by Filter: 8
-> Index Scan using pk_orders on orders  (cost=0.43..2.64 rows=1 width=20) (actual time=0.005..0.005 rows=1 loops=925397)
  Index Cond: ((ol_w_id = 11.ol_w_id) AND (ol_d_id = 11.ol_d_id) AND (ol_o_id = 11.ol_o_id))
  Filter: (11.ol_delivery_d > o_entry_d)
-> Index Scan using pk_order_line on order_line l12  (cost=0.56..8.25 rows=3 width=20) (actual time=0.014..0.014 rows=0 loops=925397)
  Index Cond: ((ol_w_id = 11.ol_w_id) AND (ol_d_id = 11.ol_d_id) AND (ol_o_id = 11.ol_o_id))
  Filter: (ol_delivery_d > 11.ol_delivery_d)
  Rows Removed by Filter: 8
Planning Time: 2.215 ms
 JIT:
  Functions: 162
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 19.201 ms, Inlining 203.089 ms, Optimization 1206.395 ms, Emission 721.776 ms, Total 2150.460 ms
Execution Time: 16904.292 ms

```

Figura 4.52: Estatísticas do plano da primeira otimização da interrogação analítica A4

4.4.3 Vista Materializada 2

De seguida, o grupo decidiu realizar o pré-processamento da união das tabelas, de modo a que a carga computacional diminuisse. Para tal, todas as condições de igualdade da *query* foram colocadas na vista materializada, com a exceção da condição $n_name = 'GERMANY'$, visto que a inclusão dessa mesma igualdade na vista materializada iria diminuir bastante a flexibilidade da *query*.

Ainda dentro da vista materializada, ordenou-se também os valores pela coluna $ol_delivery_d$, de modo a que quando a condição $l2.ol_delivery_d > A4_MV2.ol_delivery_d$ fosse executada, a filtragem fosse mais rápida.

```

CREATE MATERIALIZED VIEW A4_MV2 AS
SELECT s_w_id, s_i_id, ol_o_id, ol_w_id, ol_d_id, ol_delivery_d, n_name, su_name
FROM supplier, order_line l1, orders, A4_MV1, nation
WHERE ol_o_id = o_id
      AND ol_w_id = o_w_id
      AND ol_d_id = o_d_id

```

```

AND ol_w_id = s_w_id
AND ol_i_id = s_i_id
AND s_value = su_suppkey
AND l1.ol_delivery_d > o_entry_d
AND su_nationkey = n_nationkey
ORDER BY n_name, ol_delivery_d DESC;

SELECT su_name, count(*) AS numwait
FROM A4_MV2
WHERE NOT EXISTS (
    SELECT *
    FROM order_line l2
    WHERE l2.ol_o_id = A4_MV2.ol_o_id
        AND l2.ol_w_id = A4_MV2.ol_w_id
        AND l2.ol_d_id = A4_MV2.ol_d_id
        AND l2.ol_delivery_d > A4_MV2.ol_delivery_d)
    AND n_name = 'GERMANY'
GROUP BY su_name
ORDER BY numwait DESC, su_name;

```

4.4.4 Vista Materializada 2 com índices

Analisando a vista materializada produzida, o grupo verificou que a criação de índices para certas colunas poderia trazer um enorme impacto na performance da *query*. Uma vez que se tem a comparação $l2.ol_delivery_d > A4_MV2.ol_delivery_d$, criou-se um índice sobre a coluna *ol_delivery_d* da vista materializada. Posteriormente, e como temos a igualdade $n.name = 'GERMANY'$, optou-se por criar um outro índice sobre a coluna *n.name*. Por fim, e como a *query* agrupa os resultados por *su_name*, optou-se por criar um índice sobre a coluna *su_name*.

```

CREATE INDEX A4_i2 ON A4_MV2 (ol_delivery_d);
CREATE INDEX A4_i3 ON A4_MV2 (n_name);
CREATE INDEX A4_i4 ON A4_MV2 (su_name);

```

The screenshot shows the Oracle SQL Developer Explain Plan window. The query plan is as follows:

- Sort** (cost=922604.52..922605.02 rows=200 width=112) (actual time=4179.759..4180.236 rows=396 loops=1)
 Sort Key: (count(*)) DESC, a4_mv2.su_name
 Sort Method: quicksort Memory: 55kB
 -> Final GroupAggregate (cost=922178.42..922596.88 rows=200 width=112) (actual time=4085.651..4179.919 rows=396 loops=1)
 Group Key: a4_mv2.su_name
 -> Gather Merge (cost=922178.42..922592.88 rows=400 width=112) (actual time=4085.327..4179.450 rows=1188 loops=1)
 Workers Planned: 2
 Workers Launched: 2
 -> Partial GroupAggregate (cost=921178.40..921546.68 rows=200 width=112) (actual time=4060.020..4148.733 rows=396 loops=3)
 Group Key: a4_mv2.su_name
 -> Sort (cost=921178.40..921300.49 rows=48838 width=104) (actual time=4059.903..4116.639 rows=203941 loops=3)
 Sort Key: a4_mv2.su_name
 Sort Method: external merge Disk: 5848kB
 Worker 0: Sort Method: external merge Disk: 5720kB
 Worker 1: Sort Method: external merge Disk: 10072kB
 -> Nested Loop Anti Join (cost=1381.00..914700.96 rows=48838 width=104) (actual time=356.692..3210.897 rows=203941 loops=3)
 -> Parallel Bitmap Heap Scan on a4_mv2 (cost=1380.44..334812.31 rows=50961 width=124) (actual time=25.323..111.692 rows=308466 loops=3)
 Recheck Cond: (n_name = 'GERMANY'::bpchar)
 Heap Blocks: exact-3570
 -> Bitmap Index Scan on a4_i3 (cost=0.00..1349.87 rows=122307 width=0) (actual time=40.624..40.624 rows=925397 loops=1)
 Index Cond: (n_name = 'GERMANY'::bpchar)
 -> Index Scan using pk_order_line on order_line l2 (cost=0.56..11.46 rows=3 width=20) (actual time=0.008..0.008 rows=0 loops=925397)
 Index Cond: ((ol_w_id = a4_mv2.ol_w_id) AND (ol_d_id = a4_mv2.ol_d_id) AND (ol_o_id = a4_mv2.ol_o_id))
 Filter: (ol_delivery_d > a4_mv2.ol_delivery_d)
 Rows Removed by Filter: 8
 Planning Time: 0.407 ms
 JIT:
 Functions: 48
 Options: Inlining true, Optimization true, Expressions true, Deforming true
 Timing: Generation 7.513 ms, Inlining 225.213 ms, Optimization 473.097 ms, Emission 293.611 ms, Total 999.433 ms
 Execution Time: 4183.741 ms

Figura 4.53: Estatísticas do plano da segunda otimização da interrogação analítica A4

Como se pode observar, o tempo de execução da *query* foi de 4183 ms, que comparativamente ao tempo de execução da *query* original de 35610 ms, demonstra o sucesso das otimizações realizadas. Uma outra possível otimização seria colocar praticamente toda a *query* dentro de uma vista materializada, apenas deixando fora da vista a condição de igualdade do nome do país e os respetivo *GROUP_BY* e *ORDER_BY*. Porém, tal implementação iria restringir bastante a *query*, visto que esta apenas seria útil para casos semelhantes ao original, ou seja, tornaria a *query* muito pouco flexível, pelo que o grupo decidiu não implementá-la.

4.4.5 Paralelismo

Uma outra maneira de otimizar a interrogação analítica passa por variar o nível de paralelismo. Para verificar o seu impacto na *query*, foi necessário alterar o número de *workers* disponíveis, através do comando:

```
SET max_parallel_workers_per_gather = M;
```

podendo M tomar os valores 1, 2 (valor *default*), 4 e 8. Foram realizadas várias execuções da *query*, onde se registaram os tempos de execução médios para cada situação. Como se pode observar na Figura 4.54, nota-se uma diferença considerável entre executar a *query* com 1 *worker* ou com 4 *workers*, visto que o tempo de execução reduziu cerca de 10s. Porém, a diferença entre ter 4 ou 8 *workers* é praticamente nula, pelo que se conclui que, na *query* 4, é possível tirar partido do paralelismo, sendo que o número ideal de *workers* a considerar deverá ser de 4 *workers*.

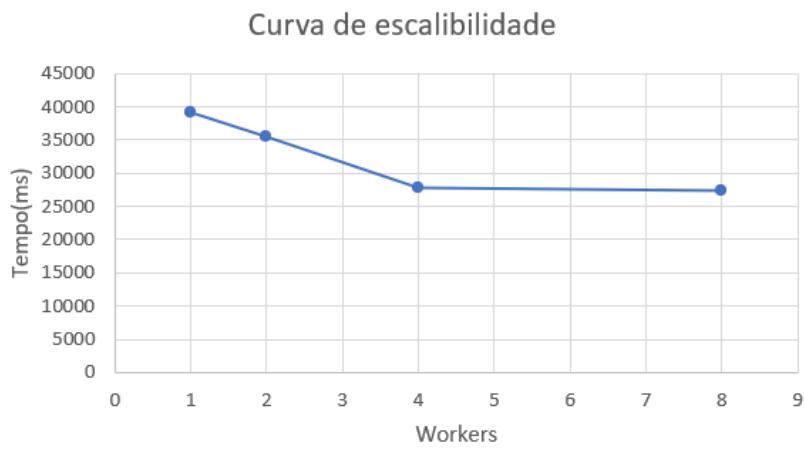


Figura 4.54: Tempos de Execução da Query A4

5 Replicação da base de dados [23]

Por fim, decidiu-se implementar a replicação da base de dados. Apesar de não se esperar melhorias de performance com a aplicação da replicação, decidiu-se utilizar esta funcionalidade disponibilizada pelo *PostgreSQL* de modo a permitir a aplicação de algoritmos de *load balance* ou de *heartbeat* no futuro. Com a aplicação destes algoritmos, será possível obter melhores resultados em *queries* de leitura, uma vez que estas poderão ser distribuídas pelas várias cópias existentes, para além de ser possível obter uma maior tolerância a falhas, ao substituir a base de dados que falhar por uma das replicas existentes.

5.1 Replicação lógica

Umas das possibilidades de replicação que o *PostgreSQL* oferece é a replicação lógica. Esta replicação oferece a possibilidade de utilizar mecanismos de sincronização, de modo a replicar modificações efetuadas numa determinada base de dados.

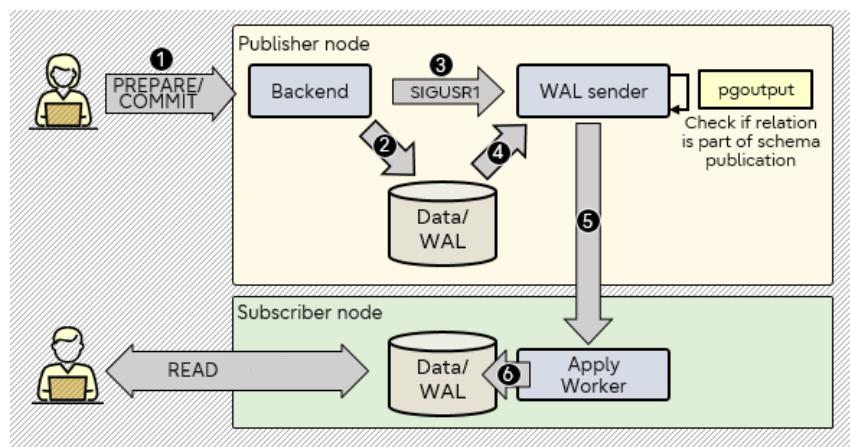


Figura 5.1: Replicação lógica de uma Base de Dados

A utilização deste tipo de replicação baseia-se no controlo refinado por ela fornecida, permitindo, por exemplo, decidir sobre que partes de uma base de dados se pretende replicar ou sincronizar. Este tipo de replicação permite, assim, replicar apenas algumas das tabelas presentes numa determinada base de dados, em vez de replicar a base de dados completa.

Assim, para realizar a replicação procedeu-se à criação de uma publicação, na base de dados a replicar e à respetiva subscrição, na base de dados secundária. Porém, antes de se proceder à replicação da base de dados foi necessário ter em atenção a definição de alguns dos parâmetros, como por exemplo, o parâmetro *wal_level*, que se definiu com o valor *logical*.

```
CREATE PUBLICATION pub1 FOR ALL TABLES;
```

Publication pub1						
Owner	All tables	Inserts	Updates	Deletes	Truncates	Via root
postgres	t	t	t	t	t	f

Figura 5.2: Publicação "pub1" criada na base de dados a replicar

```
CREATE SUBSCRIPTION sub1
  CONNECTION 'host=10.140.0.7 port=5432 dbname=tpcc user=postgres password=postgres'
  PUBLICATION pub1;
```

```
2022-05-25 17:14:05.635 UTC [5148] LOG: logical replication table synchronization worker for subscription "sub1", table "orders" has finished
2022-05-25 17:14:05.650 UTC [5155] LOG: logical replication table synchronization worker for subscription "sub1", table "supplier" has started
2022-05-25 17:14:05.997 UTC [5155] LOG: logical replication table synchronization worker for subscription "sub1", table "supplier" has finished
2022-05-25 17:14:06.006 UTC [5156] LOG: logical replication table synchronization worker for subscription "sub1", table "customer" has started
2022-05-25 17:14:12.842 UTC [719] LOG: checkpoints are occurring too frequently (8 seconds apart)
2022-05-25 17:14:12.842 UTC [719] HINT: Consider increasing the configuration parameter "max_wal_size".
2022-05-25 17:14:21.005 UTC [719] LOG: checkpoints are occurring too frequently (9 seconds apart)
```

Figura 5.3: Replicação da base de dados

Tal como se pode observar pelas figuras acima, foi configurada a replicação de todas as tabelas da base de dados, para uma outra base de dados utilizada para redundância.

Após esta fase, foi necessário testar o comportamento da execução de *queries* na base de dados replicada, de modo a perceber o impacto das alterações efetuadas no desempenho. Tal como esperado, a replicação impôs uma carga de trabalho superior nas operações, devido às operações extras necessárias para a realização da replicação, implicando a obtenção de resultados inferiores. Esta característica deve-se ao facto de que a replicação deve ser utilizada para garantir alta disponibilidade, em conjunto com mecanismos de que garantam a substituição de uma base de dados no caso da ocorrência de falhas na principal. Para além disto, de modo a que a replicação possa ser utilizada para melhorar os tempos de resposta, deve ser for conjugada com mecanismos de *load balancing*, de modo a melhorar, principalmente os tempos relativos à execução de *queries* de leitura.

6 Conclusão

Dado por terminado o trabalho prático, pretende-se agora fazer uma revisão do trabalho feito bem como uma avaliação dos resultados obtidos.

Após instalar o *benchmark TPC-C* o primeiro problema a resolver foi encontrar uma configuração para a máquina virtual onde o serviço iria correr que tivesse um bom balanceamento entre performance suficiente e poupança de recursos.

Neste sentido, convergimos para uma máquina com *hardware* em que houvesse espaço para otimização viável, isto é, uma máquina não muito fraca, e uma máquina que não consumisse muitos recursos monetários (4 cores, 8GB RAM). Quanto ao tamanho e qualidade do disco acabamos por escolher um disco com a capacidade máxima (SSD 500GB) dado que este tipo de disco quanto maior, mais rápido seria, o que era uma métrica que nos interessava maximizar.

O passo seguinte consistiu em determinar o número de *warehouses* e clientes ideal para obter uma configuração de referência. Dada a vasta quantidade de possibilidades possível, o grupo decidiu primeiro determinar o número de *warehouses* e só depois o número de clientes. O método de escolha para o número de *warehouses* consistiu em atingir uma ocupação da base de dados ligeiramente superior à quantidade de RAM de modo a obrigar a máquina a não poder colocar simplesmente toda a base de dados em RAM. Escolhemos 96 *warehouses* que ocupava 10297MB. Quanto ao número de clientes, tal como referido, este foi escolhido utilizando os 96 *warehouses*. Para determinar o número ideal, procuramos um equilíbrio entre as várias métricas, mais especificamente, um alto *Throughput*, um tempo de resposta baixo, e uma taxa de aborto aceitável. 50 clientes revelou ser uma quantidade que cumpria estes requisitos.

Tendo agora a configuração de referência, o grupo debruçou-se sobre a problemática de otimizar o desempenho da carga transacional tendo em conta, principalmente, os parâmetros de configuração do *PostgreSQL*. Para isto, optamos por alterar todos os parâmetros das secções *Settings*, *Checkpoints* e *Archiving* presentes no ficheiro de configuração do *PostgreSQL*. Este processo permitiu-nos determinar quais os parâmetros que maximizariam a taxa de transferência. No entanto, várias vezes, apesar dos valores para um certo parâmetro serem apelativos, não era recomendado, ou sensato, utilizá-los. Ainda assim, conseguimos atingir um valor de taxa de transferência consideravelmente superior à configuração de referência (de 846 tx/seg para 1436.94 tx/seg).

Seguidamente, procuramos otimizar o desempenho das interrogações analíticas. O objetivo seria minimizar o tempo de resposta de cada *query*. Para isso, foi útil o comando EXPLAIN ANALYZE e o *website explain depesz* para observar qual o plano de execução escolhido pelo *PostgreSQL* na execução de uma interrogação. A estratégia para redução do tempo de resposta foi idêntica para cada *query* começando por identificar quais os pontos em que a interrogação passava mais tempo e introduzir índices e/ou vistas materializadas para tentar otimizar estes desempenhos. Os resultados obtidos foram bastante satisfatórios tendo conseguido diminuir consideravelmente o tempo de resposta de cada interrogação analítica.

Finalmente, o grupo decidiu avaliar o impacto da replicação da base de dados. Optou-se por realizar uma replicação lógica criando o *publisher* e *subscriptions* necessários. Tínhamos previsto que esta replicação não iria trazer quaisquer benefícios em termos de performance, o que acabou por se verificar uma vez que para melhorar os tempos relativos à execução de interrogações de leitura seria necessário a implementação de mecanismos de balanceamento de carga.

Com isto, podemos concluir que consideramos que houve um balanço positivo do trabalho realizado dado que as dificuldades sentidas foram superadas e foram cumpridos todos os requisitos. A maior dificuldade sentida acabou por ser o consumo de tempo necessário dado operações como o povoamento da base de dados e o elevado número de vezes que se executou a carga transacional da base de dados (10 minutos por execução). Este problema foi mitigado com a criação de imagens de máquina com a base de dados já povoada.

O desenvolvimento deste projeto permitiu ao grupo aplicar e expandir os conceitos lecionados nas aulas e melhorar o conhecimento do funcionamento interno do *PostgreSQL*.

Referências

- [1] H. Garcia-Molina, J. Ullman and J. Widom. Database Systems: The Complete Book. Prentice-Hall, 2006 (2 nd Edition).
- [2] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan-Kaufmann, 1993.
- [3] M. Tamer Özsu, P. Valduriez. Principles of Distributed Database System. Springer, 2011. (3 rd Edition).
- [4] PostgreSQL Documentation, <https://www.postgresql.org/docs/14/static/index.html>
- [5] PostgresSQL Documentation, <https://postgresqlco.nf/doc/en/param/fsync/>
- [6] PostgresSQL Documentation, https://postgresqlco.nf/doc/en/param/shared_buffers/
- [7] Fernando Laudares Camargos, José Zechel, PostgreSQL and Hugepages: Working with an abundance of memory in modern servers, 2019
- [8] https://en.wikipedia.org/wiki/Memory_management_unit
- [9] Xiaolin Wang, Taowei Luo, Jingyuan Hu, Zhenlin Wang & Yingwei Luo, Evaluating the impacts of hugepage on virtual machines, 2017
- [10] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/full_page_writes/
- [11] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/wal_buffers/
- [12] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/synchronous_commit/
- [13] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/max_wal_senders/
- [14] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/wal_level/
- [15] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/wal_sync_method/
- [16] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/commit_siblings/
- [17] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/work_mem/
- [18] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/commit_delay/
- [19] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/min_wal_size/
- [20] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/archive_mode/
- [21] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/archive_command/
- [22] PostgreSQL Documentation, https://postgresqlco.nf/doc/en/param/archive_timeout/
- [23] FUJITSU Enterprise Postgres Documentation, <https://www.postgresql.fastware.com/blog/logical-replication-of-all-tables-in-schema-in-postgresql-15>