

**Escola de Engenharia
Universidade do Minho**

UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

Administração de Bases de Dados

Trabalho Prático de Grupo
Relatório de Desenvolvimento

Carolina Oliveira
pg47101

Carolina Santejo
pg47102

Manuel Moreira
pg47439

Raquel Costa
pg47600

Sara Dias
pg47667

3 de julho de 2022

Conteúdo

1	Introdução	8
2	Configuração de referência do <i>benchmark</i> TPC-C	9
2.1	Especificações da máquina escolhida	9
2.2	Configuração do número de <i>warehouses</i>	9
2.3	Configuração do número de clientes	10
2.4	Configuração de referência	13
3	Configuração do PostgreSQL	14
3.1	Settings	14
3.1.1	<i>fsync</i>	14
3.1.2	<i>synchronous_commit</i>	15
3.1.3	<i>wal_sync_method</i>	15
3.1.4	<i>full_page_writes</i>	16
3.1.5	<i>commit_delay</i>	16
3.1.6	<i>commit_siblings</i>	17
3.2	Checkpoints	17
3.2.1	<i>checkpoint_timeout</i>	17
3.2.2	<i>checkpoint_flush_after</i>	18
3.2.3	<i>checkpoint_warning</i>	18
3.2.4	<i>max_wal_size</i>	19
3.2.5	<i>min_wal_size</i>	19
3.3	Archiving	20
3.3.1	<i>archive_mode</i>	20
3.3.2	<i>archive_command</i>	20
3.3.3	<i>archive_timeout</i>	20
3.4	Isolamento	21
3.5	Combinação dos parâmetros	22

3.5.1	<i>Settings</i>	22
3.5.2	<i>Checkpoints</i>	23
3.5.3	<i>Archiving</i>	24
3.5.4	<i>Settings e Checkpoints</i>	25
3.5.5	<i>Settings e Archiving</i>	26
3.5.6	<i>Checkpoints e Archiving</i>	27
3.5.7	<i>Settings, Checkpoints e Archiving</i>	28
3.5.8	Configuração de referência	29
4	Análise e otimização das interrogações analíticas	30
4.1	Interrogacão A1	31
4.1.1	Primeira versão de vista materializada	33
4.1.2	Índice	34
4.1.3	Segunda versão de vista materializada	35
4.1.4	Paralelismo	36
4.2	Interrogacão A2	37
4.2.1	Primeira versão com índices	39
4.2.2	Vistas materializadas	40
4.2.3	Novo índice	44
4.2.4	Paralelismo	45
4.3	Interrogacão A3	46
4.3.1	Índices	47
4.3.2	Vista materializada	48
4.3.3	Paralelismo	49
4.4	Interrogacão A4	50
4.4.1	Índice	51
4.4.2	Vista materializada	52
5	Conclusão	55

6 Anexos	56
6.1 Anexo A - <i>Script</i> das dependências	56
6.2 Anexo B - <i>Script</i> do <i>setup</i> das VMs	56

Listas de Figuras

1	<i>Throughput</i> em função do número de clientes	11
2	Tempo de resposta em função do número de clientes	12
3	<i>Throughput</i> em função do tempo de resposta	12
4	Métricas da configuração de referência	13
5	Métricas da configuração de <i>settings</i> escolhida	23
6	Métricas da configuração de <i>checkpoints</i> escolhida	24
7	Métricas da configuração de <i>archiving</i> escolhida	25
8	Métricas da configuração de <i>settings</i> escolhida	30
9	Listagem dos índices presentes na base de dados	31
10	EXPLAIN ANALYZE da interrogação A1	32
11	EXPLAIN ANALYZE da interrogação A1	33
12	EXPLAIN ANALYZE da interrogação A1 com a primeira versão de vista materializada	34
13	EXPLAIN ANALYZE da interrogação A1 com índice	35
14	EXPLAIN ANALYZE da interrogação A1 com a segunda versão de vista materializada	36
15	EXPLAIN ANALYZE da interrogação A1 com 4 workers	37
16	EXPLAIN ANALYZE da interrogação A2	38
17	EXPLAIN ANALYZE da interrogação A2	39
18	EXPLAIN ANALYZE da interrogação A2 com a utilização do índice	40
19	EXPLAIN ANALYZE da interrogação A2 com a alteração da cláusula <i>group by</i>	41
20	EXPLAIN ANALYZE da interrogação A2 com a vista materializada	42
21	EXPLAIN ANALYZE da interrogação A2 com a vista materializada atualizada	43
22	EXPLAIN ANALYZE da interrogação A2 com a tentativa de novo índice	44
23	EXPLAIN ANALYZE da interrogação A2 com a tentativa de novo índice	44
24	EXPLAIN ANALYZE da interrogação A2 com paralelismo	45
25	EXPLAIN ANALYZE da interrogação A3	47

26	EXPLAIN ANALYZE da interrogação A3 com índices	48
27	EXPLAIN ANALYZE da interrogação A3 com vista materializada	49
28	EXPLAIN ANALYZE da interrogação A3 com vista materializada	49
29	EXPLAIN ANALYZE da interrogação A3 com paralelismo	50
30	EXPLAIN ANALYZE da interrogação A4	51
31	EXPLAIN ANALYZE da interrogação A4 com índice	52
32	EXPLAIN ANALYZE da interrogação A4 materialized	54

Listas de Tabelas

1	Tamanho da base de dados em função do número de <i>warehouses</i>	10
2	Métricas medidas em função do número de clientes	11
3	Alteração de parâmetros na opção <i>fsync</i>	14
4	Alteração de parâmetros na opção <i>synchronous_commit</i>	15
5	Alteração de parâmetros na opção <i>wal_sync_method</i>	16
6	Alteração de parâmetros na opção <i>full_page_writes</i>	16
7	Alteração de parâmetros na opção <i>commit_delay</i>	17
8	Alteração de parâmetros na opção <i>commit_siblings</i>	17
9	Alteração de parâmetros na opção <i>checkpoint_timeout</i>	18
10	Alteração de parâmetros na opção <i>checkpoint_flush_after</i>	18
11	Alteração de parâmetros na opção <i>checkpoint_warning</i>	18
12	Alteração de parâmetros na opção <i>max_wal_size</i>	19
13	Alteração de parâmetros na opção <i>min_wal_size</i>	19
14	Alteração de parâmetros na opção <i>archive_mode</i>	20
15	Alteração de parâmetros na opção <i>archive_command</i>	20
16	Alteração de parâmetros na opção <i>archive_timeout</i>	21
17	Alteração de parâmetros no isolamento	21
18	Comparação das configurações - <i>settings</i>	22
19	Comparação das configurações - <i>Checkpoints</i>	23
20	Comparação das configurações - <i>Archiving</i>	24
21	Comparação das configurações - <i>Settings</i> e <i>Checkpoints</i>	26
22	Comparação das configurações - <i>Settings</i> e <i>Archiving</i>	27
23	Comparação das configurações - <i>Checkpoints</i> e <i>Archiving</i>	27
24	Comparação das configurações - <i>Settings</i> , <i>Checkpoints</i> e <i>Archiving</i>	28

1 Introdução

Este trabalho prático foi desenvolvido no âmbito da Unidade Curricular Administração de Bases de Dados, do perfil de Engenharia de Aplicações, lecionada no 2º semestre do 1º ano do Mestrado em Engenharia Informática.

O trabalho prático consiste na configuração, otimização e avaliação do *benchmark* TPC-C com alguns dados e interrogações adicionais. O TPC-C é um *benchmark* de processamento de transações que simula um sistema operacional de uma cadeia de lojas, suportando a operação diária de gestão de vendas e *stocks*. Para além disso, foram fornecidas interrogações analíticas adicionais, baseadas na adaptação do TPC-H.

Este relatório está dividido em 3 capítulos. O primeiro, [Configuração de referência do *benchmark* TPC-C](#) diz respeito à instalação e configuração do *benchmark* TPC-C, de forma a obter uma configuração referência, em termos de *hardware*, número de *warehouses* e número de clientes. Em seguida, utilizando a configuração definida, foi procurado otimizar o desempenho da carga transacional tendo em conta, principalmente, os parâmetros de configuração do PostgreSQL, [Configuração do PostgreSQL](#). O terceiro capítulo, [Análise e otimização das interrogações analíticas](#), tem como objetivo otimizar o desempenho das interrogações analíticas tendo em conta, principalmente, os respetivos planos, a redundância e o paralelismo.

2 Configuração de referência do *benchmark* TPC-C

2.1 Especificações da máquina escolhida

O primeiro passo para o desenvolvimento deste trabalho consiste em determinar as especificações de máquina em que vai ser instalado, configurado e testado o *benchmark* TPC-C. Para tal, optou-se por utilizar uma máquina localizada em Iowa, USA, da zona *central1-a*, da série *e2-standard-2* que dispõe de 2 CPUs e 8GB de memória RAM. Foi utilizado um disco SSD de 200GB de forma a garantir que a base de dados tem capacidade suficiente para realizar os vários testes necessários. Como sistema operativo foi escolhido o Ubuntu, mais concretamente, a versão 18.04. Adicionalmente, foi concedido também o acesso a todas as APIs do *Google Cloud* de modo a facilitar a utilização das mesmas, caso necessário.

De forma complementar, com o objetivo de diminuir o tempo de configuração do *benchmark*, foi criado um *bucket* utilizando a *Cloud Storage* que a *Google Cloud Platform* disponibiliza. Neste *bucket* foram armazenados *scripts* desenvolvidos pelo grupo para automatizar o processo de *setup* da base de dados, assim como um *backup* da mesma, para que não fosse necessário realizar a criação e povoamento das tabelas sempre que é utilizada uma nova máquina, que era um processo demorado. Além disso, esta estratégia permitiu ao grupo reduzir custos relativamente ao saldo disponibilizado, uma vez que já não era necessário ter uma máquina, ou várias, permanentemente, no *Google Cloud*.

Para além dos *scripts* fornecidos pela equipa docente, aos quais fizemos algumas alterações, nesta fase foi criado e armazenado no *bucket* um *script* adicional para a instalação de dependências, como por exemplo o *maven*. Este *script* pode ser consultado em [Anexo A - Script das dependências](#)

2.2 Configuração do número de *warehouses*

Depois de definidas as especificações da máquina com que iríamos trabalhar, procedemos à identificação do número de *warehouses* que deveriam estar presentes na nossa base de dados.

Para isso, o grupo criou bases de dados com diferentes números de *warehouses*, alterando o parâmetro *tpcc.number.warehouses* do ficheiro de configuração. Os resultados deste estudo encontram-se apresentados na tabela 1.

O grupo procurou escolher um número de *warehouses* que originasse uma base de dados com tamanho igual ou superior ao da memória RAM (8GB), para que a taxa de ocupação da mesma fosse aproximadamente 100%. Desta forma, temos garantia que a base de dados não consegue ser carregada completamente para RAM, dando oportunidade de observar melhorias na segunda fase do trabalho. Por essa razão e analisando a tabela 1, foi decidido utilizar uma base de dados com 80 *warehouses*, resultando numa base de dados que ocupa 8685MB.

Número de <i>warehouses</i>	Tamanho da base de dados
2	251MB
8	902MB
16	1773MB
32	3492MB
64	6954MB
70	7603MB
80	8685MB
90	9773 MB
100	11GB

Tabela 1: Tamanho da base de dados em função do número de *warehouses*

No final e como sugerido pela equipa docente, foi criado um *backup* comprimido dessa base de dados, recorrendo ao comando *pg_dump*. Este foi armazenado no *bucket* criado, com o objetivo de tornar os próximos testes mais rápidos, devido a não ser necessário realizar o *load* da base de dados sempre que utilizamos uma máquina nova. Desta forma, apenas tem de se restaurar o *backup*, permitindo assim que o *setup* da base de dados seja mais rápido e, consequentemente, menos custoso.

2.3 Configuração do número de clientes

Após ter sido definido o número de *warehouses*, falta estudar o número de clientes que devemos ter.

Para este estudo, foi utilizado o *backup* anteriormente criado para realizar o *setup* da base de dados, na qual fomos alterando o parâmetro *tpcc.numclients* do ficheiro de configuração. Com estes testes procuramos encontrar um número de clientes que maximize o ***throughput***, ao mesmo tempo que apresenta um tempo de resposta e *abort rate* relativamente baixos, considerando também a relação entre o *throughput* e tempo de resposta.

Na tabela 2 estão apresentados os resultados desses testes. Para facilitar a interpretação dos resultados, também foram criados 3 gráficos com os valores obtidos.

Número de clientes	Throughput (tx/s)	Response time (s)	Abort rate (%)
20	139.857	0.0071	0.0026
30	210.602	0.0084	0.0045
40	281.199	0.0083	0.0059
50	348.693	0.0139	0.0102
60	416.942	0.0122	0.0118
70	479.962	0.0203	0.0207
80	539.326	0.0222	0.0247
90	558.709	0.0348	0.0371
100	584.200	0.0333	0.0373
110	611.273	0.0318	0.0379
120	544.111	0.0318	0.0370
130	558.195	0.0358	0.0370

Tabela 2: Métricas medidas em função do número de clientes

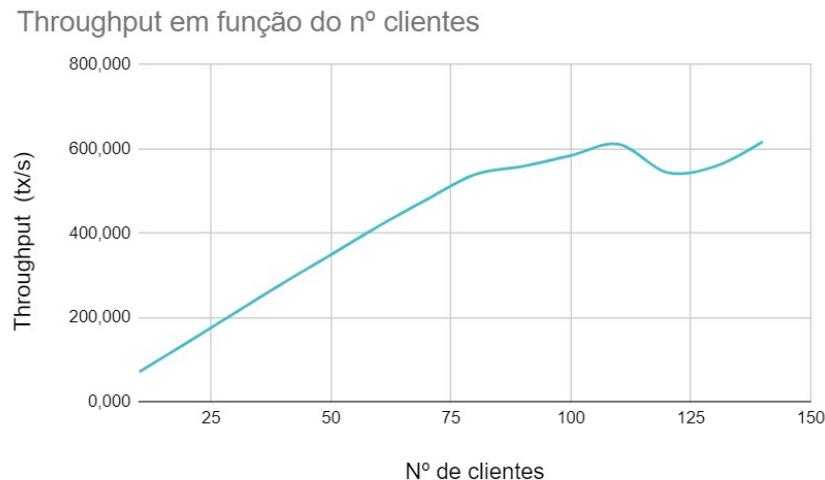
Figura 1: *Throughput* em função do número de clientes



Figura 2: Tempo de resposta em função do número de clientes

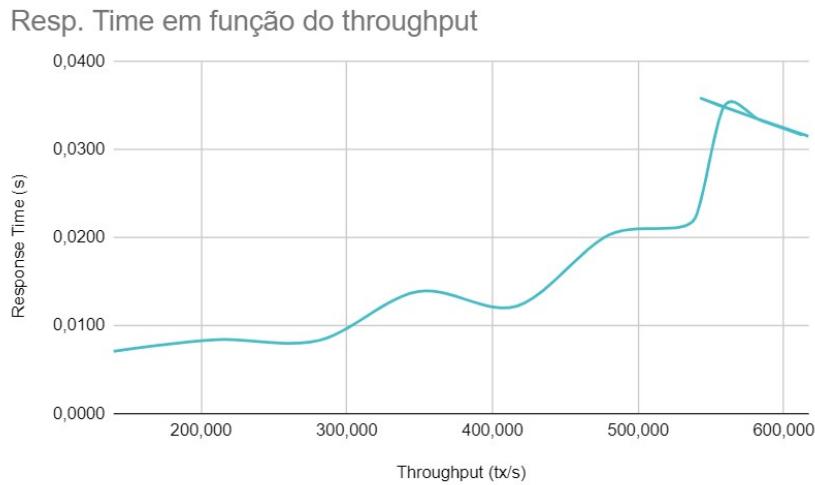


Figura 3: *Throughput* em função do tempo de resposta

Analisando os resultados apresentados acima e seguindo a sugestão do professor, o grupo optou por definir o número de clientes como 80. Apesar deste valor não ser o que apresentava maior *throughput*, foi escolhido para, nas próximas fases, haver possibilidade de melhoria da *performance* do *benchmark*.

2.4 Configuração de referência

Concluídos estes testes, o grupo optou por utilizar, como configuração de referência para o resto do projeto, uma máquina de **2CPUs**, **8GB de memória RAM**, **200GB de disco**, e uma base de dados com **80 warehouses** e **80 clientes**.

Na imagem seguinte, estão apresentadas as métricas obtidas na execução da carga transacional com a configuração escolhida:

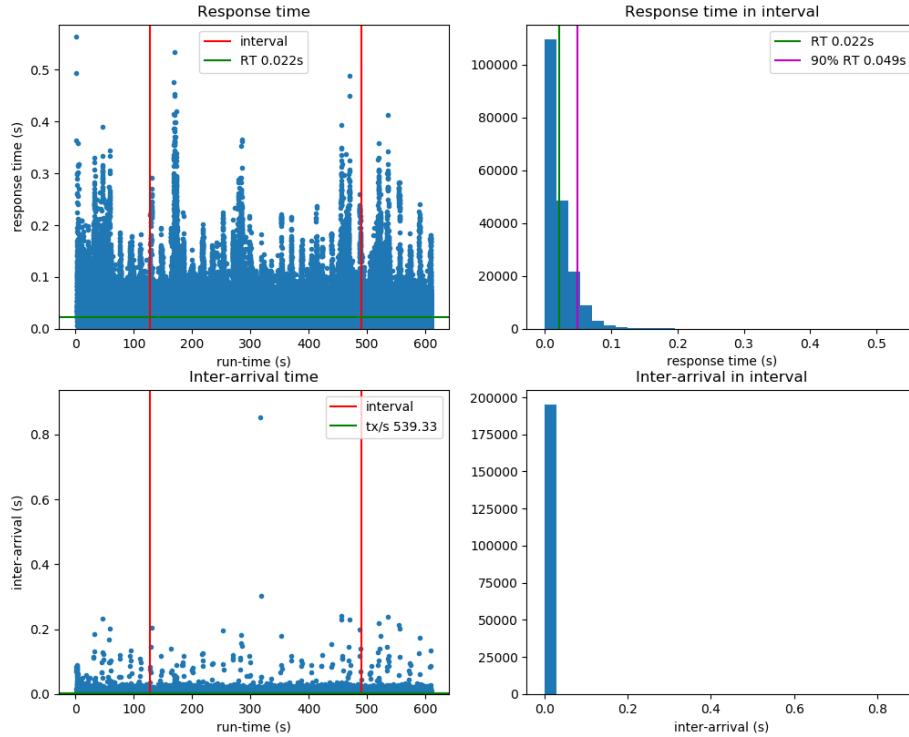


Figura 4: Métricas da configuração de referência

No final desta primeira fase do trabalho, foi atualizado o *bucket* para ter os parâmetros relativos ao número de *warehouses* e de clientes, de acordo com aqueles que foram escolhidos. Para além disso, foi também feito um novo *script*, [Anexo B - Script do setup das VMs](#), para automatizar o *setup* das máquinas virtuais, que para além de correr os *scripts* de instalação do *PostgreSQL* e de instalação de dependências, também realiza o *restore* de base de dados a partir do *backup* criado.

3 Configuração do PostgreSQL

Nesta secção, vai ser estudado e testado o impacto que algumas configurações do *PostgreSQL* têm no desempenho do nosso *benchmark*.

Como o objetivo é otimizar a carga transacional, o grupo focou-se em testar a maioria das configurações da secção ***Write Ahead Logs (WAL)***, testando para cada uma delas vários valores possíveis para cada parâmetro, e em seguida fazer uma breve explicação de qual das opções consideramos ser melhor para o nosso problema.

Cada teste foi realizado isoladamente, ou seja, apenas o parâmetro em questão foi alterado, os restantes foram mantidos com o valor *default*. Entre cada teste, o *PostgreSQL* foi também reiniciado. Para além destes cuidados, todos os testes foram realizados em máquinas com a mesma configuração e com a base de dados no mesmo ponto de partida, de forma a garantir que os resultados eram o menos enviesados possível.

No final, será discutida qual a combinação destes parâmetros que iremos utilizar na próxima secção, tendo como objetivo a otimização do desempenho da carga transacional.

De seguida apresentaremos brevemente cada um dos parâmetros testados e qual o seu impacto na *performance* das transações.

3.1 Settings

3.1.1 *fsync*

O valor *default* deste parâmetro é *on*, o que faz com que o *PostgreSQL* garanta que os *updates* estão de certeza escritos no disco, chamando funções de sincronização. Isto permite que a base de dados seja recuperável no caso de ocorrer alguma falha no sistema ou no *hardware*.

Quando se procede à alteração do seu estado para *off*, o que acontece é que o sistema vai dizer que o *update* já se encontra feito, mesmo que não esteja, adiando assim a escrita dos dados no disco. Esta opção, em termos de *performance*, é mais favorável. No entanto, caso ocorra uma falha no sistema, os dados são irrecuperáveis, ou pelo menos as últimas transações que não foram escritas no disco. Desta forma, é expectável que o valor de *throughput* na execução do *benchmark* com o *fsync* desligado seja maior em relação a ter a sincronização ligada.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
<i>on</i>	527.96	0.0313	0.0311
<i>off</i>	534.70	0.027	0.0296

Tabela 3: Alteração de parâmetros na opção *fsync*

Tal como esperado, a não sincronização dos dados resulta numa melhor *performance*.

Apesar disso, não é aconselhável deixar este parâmetro em *off* pelos motivos anteriormente referidos. Como isso poderia deixar a base de dados incoerente, optamos por deixar a sincronização ligada, para evitar os riscos associados a desligar este parâmetro.

3.1.2 *synchronous_commit*

Este parâmetro especifica se o *commit* de uma transação espera, ou não, que o *WAL record* seja escrito em disco antes de enviar uma mensagem de sucesso ao cliente.

Por exemplo, no caso do parâmetro se encontrar '*off*', o *commit* não espera que o *WAL record* seja copiado para disco para confirmar que a transação foi um sucesso. Se o seu valor for '*on*', o *commit* da transação vai sempre esperar até que os dados tenham sido escritos em disco.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'on'	533.0529	0.0284	0.0300
'off'	538.777	0.0246	0.0273
'local'	516.6376	0.03539	0.0344
'remote_apply'	532.8181	0.0282	0.0299
'remote_write'	530.6556	0.0288	0.0289

Tabela 4: Alteração de parâmetros na opção *synchronous_commit*

Como seria de esperar, quando este parâmetro está desligado verifica-se uma *performance* maior, visto que não é necessário esperar que o *WAL record* seja copiado para o disco para poder confirmar o sucesso de uma transação.

De forma contrária ao *fsync*, desligar este parâmetro não cria o risco da base de dados se tornar inconsistente. A única desvantagem é que o *crash* do sistema pode levar a que algumas transações recentes, dadas como *committed*, sejam perdidas. No entanto, o estado da base de dados será igual a se essas transações tivessem sido abortadas de forma "limpa".

O grupo tomou a decisão de optar por alterar o valor deste parâmetro para '*off*' devido à melhoria de *performance* que se verifica, tendo consciência que poderiam haver algumas transações que poderiam ser perdidas.

3.1.3 *wal_sync_method*

Este parâmetro é usado para forçar *WAL updates* para o disco. É de notar que, se o parâmetro *fsync* estiver *off* esta configuração é irrelevante, uma vez que os updates do ficheiro *WAL* nunca vão ser forçados. Por este motivo, todos os testes foram realizados com essa especificação ligada.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'open datasync'	527.0976	0.0316	0.0325
'fdatasync'	542.8489	0.0194	0.0214
'fsync'	543.4067	0.0223	0.0243
'open sync'	514.2349	0.0372	0.0391

Tabela 5: Alteração de parâmetros na opção *wal_sync_method*

Como se pode observar na tabela 5, o melhor resultado foi obtido utilizando a opção '*fsync*'. Desta forma, podemos concluir que o valor *default* deste parâmetro seria a melhor escolha.

3.1.4 *full_page_writes*

Quando este parâmetro se encontra ativo, o servidor escreve o conteúdo completo de uma página do disco para WAL, durante a sua primeira modificação após um *checkpoint*. É possível concluir que desligando este parâmetro, poderá levar a uma melhor *performance*, assim como se pode verificar na tabela apresentada a seguir. No entanto, é de notar que o facto deste parâmetro se encontrar desligado pode levar à corrupção de dados.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'on'	528.2563	0.0302	0.0319
'off'	533.6066	0.0282	0.0282

Tabela 6: Alteração de parâmetros na opção *full_page_writes*

Desta forma, apesar da opção '*off*' apresentar melhores valores para as métricas medidas, consideramos que seria mais importante manter a consistência e integridade da base de dados.

3.1.5 *commit_delay*

Este parâmetro define o *delay*, em microssegundos, que o servidor espera até fazer o *flush* do WAL para memória. Quando esse tempo é aumentando, é provável que o *throughput* também aumente, já que permite que seja feito o *commit* de um número maior de transações no mesmo WAL *flush*. No entanto, quanto mais se aumenta o *commit delay*, mais aumenta também a latência.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'100'	535.91	0.0244	0.0264
'200'	538.104	0.0272	0.0295
'500'	539.016	0.0195	0.018
'1000'	544.72	0.0249	0.0276
'2000'	538.31	0.0266	0.0294

Tabela 7: Alteração de parâmetros na opção *commit_delay*

Como era de esperar, o desempenho do programa aumentou com o aumento do tempo de *delay*. No entanto, é de notar que quando o *delay* é definido em mais de 1000 microsegundos, o balanço entre o aumento da *performance* e a latência fica contraprodutivo. Assim, o valor de atraso de 1000 foi o que apresentou melhores resultados no desempenho.

3.1.6 commit_siblings

Este parâmetro diz respeito ao número mínimo de transações ativas concorrentes necessárias para que possa iniciar o *commit_delay*. Um valor maior deste parâmetro torna mais provável que pelo menos uma transação fique pronta para realizar *commit* durante o intervalo *delay*.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'5'	532.9775	0.0281	0.0295
'10'	536.0561	0.0298	0.0310
'15'	531.3243	0.0291	0.0297
'20'	515.2294	0.0346	0.0335

Tabela 8: Alteração de parâmetros na opção *commit_siblings*

Analizando as métricas medidas na tabela acima, conseguimos concluir que obtemos melhor *performance* quando este parâmetro está a '10', por isso decidimos manter este valor para a nossa configuração do *PostgreSQL*.

3.2 Checkpoints

3.2.1 *checkpoint_timeout*

O objetivo deste parâmetro é definir um tempo máximo entre *checkpoints* WAL automáticos. Os seus valores encontram-se entre 30 segundos e um dia, sendo o valor *default* correspondente a '5min'. Aumentando o valor deste parâmetro, é possível aumentar o intervalo de tempo necessário para recuperação de uma falha.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'30s'	487.8570	0.0349	0.0341
'1min'	522.9985	0.0338	0.0334
'2min'	520.9743	0.0353	0.0351

Tabela 9: Alteração de parâmetros na opção *checkpoint_timeout*

Partindo dos testes efetuados e dos resultados obtidos, o grupo chegou à conclusão que seria benéfico escolher a opção de '1min' para a configuração do *PostgreSQL*, uma vez que foi aquele que demonstrou uma melhor *performance*.

3.2.2 *checkpoint_flush_after*

Este parâmetro tem como objetivo, após um determinado tamanho de dados escritos durante um *checkpoint*, forçar o sistema operativo a guardar os mesmos num nível mais abaixo na memória. Desta forma, é possível reduzir a quantidade de dados na *cache* do *kernel* e diminuir a latência das transações.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'0'	544.0151	0.0198	0.0223
'512kB'	534.9926	0.0277	0.0292
'1024kB'	512.6039	0.0363	0.0362

Tabela 10: Alteração de parâmetros na opção *checkpoint_flush_after*

Analizando os resultados obtidos, verifica-se que existe uma melhor *performance* quando este parâmetro está desligado (opção '0').

3.2.3 *checkpoint_warning*

Este parâmetro tem como objetivo escrever uma mensagem para o *log* do servidor, nos casos em que o preenchimento de segmentos de ficheiros WAL acontece com um intervalo de tempo entre eles menor do que o fornecido.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'0'	513.8140	0.0353	0.0357
'10s'	535.5967	0.0254	0.0259
'30s'	523.7673	0.0312	0.0309
'60s'	548.2357	0.0182	0.0218
'100s'	538.0632	0.0276	0.0299

Tabela 11: Alteração de parâmetros na opção *checkpoint_warning*

Analizando os resultados obtidos, verifica-se que a opção '60s' permitiu obter a melhor *performance*.

3.2.4 max_wal_size

Este parâmetro permite definir um valor máximo para o tamanho do WAL durante os *checkpoints* automáticos. Este trata-se de um limite *soft*, uma vez que o tamanho do WAL pode exceder este valor quando está perante situações especiais como *heavy load*, ou então quando um *archive_command* falha. Ao incrementar este parâmetro incrementa-se também o intervalo de tempo para recuperação de uma falha.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'1GB'	526.4944	0.0334	0.0330
'2GB'	501.2231	0.0369	0.0347
'4GB'	510.2470	0.0358	0.0327

Tabela 12: Alteração de parâmetros na opção *max_wal_size*

Tendo em conta os resultados obtidos, o grupo verificou uma melhor *performance* quando se utilizou a opção de '1GB'.

3.2.5 min_wal_size

O *min_wal_size* é o parâmetro que nos possibilita escolher o valor mínimo do tamanho do WAL. Desde que este se mantenha abaixo desse valor mínimo, os WAL antigos são reciclados para uso futuro, em vez de serem removidos. Isto pode ser usado para assegurar que o espaço necessário é reservado para lidar com picos de utilização como, por exemplo, ao correr *large batch jobs*.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'80MB'	507.3891	0.0307	0.0333
'160MB'	543.2124	0.0214	0.0239
'320MB'	541.5812	0.0224	0.0244
'640MB'	540.3852	0.0227	0.0245

Tabela 13: Alteração de parâmetros na opção *min_wal_size*

Através dos resultados obtidos, o grupo chegou à conclusão que a opção de 160MB era aquela que produzia resultados com uma melhor *performance*.

3.3 Archiving

3.3.1 archive_mode

Quando o *archive mode* está ativado, segmentos WAL completos são enviados para o *archive storage*. Para além da opção *off*, que o desativa, existem também as opções *always* e *on*, que funcionam de maneira igual num modo operacional normal. No entanto, quando a opção escolhida é *always*, o arquivador WAL está também ativado durante os modos de recuperação e *standby*.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'always'	536.8351	0.0286	0.0296
'on'	527.6394	0.0325	0.0324
'off'	540.3845	0.0260	0.0281

Tabela 14: Alteração de parâmetros na opção *archive mode*

Através dos resultados obtidos, o grupo chegou à conclusão que a opção '*off*' era aquela que produzia resultados com uma melhor *performance*.

3.3.2 archive_command

O *archive command* permite definir o comando *shell* que será chamado para arquivar um segmento completo do ficheiro WAL. No caso da opção '%p', sempre que aparecer um '%p' numa *string*, este é substituído pelo nome do *path* do ficheiro que será arquivado. Já no caso da opção '%f', esta é substituída apenas pelo nome do ficheiro.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'%p'	522.5733	0.0344	0.0317
'%f'	513.6011	0.0367	0.0350

Tabela 15: Alteração de parâmetros na opção *archive command*

Através dos resultados obtidos, o grupo chegou à conclusão que a opção '%f' era aquela que produzia resultados com uma melhor *performance*.

3.3.3 archive_timeout

Por norma, o *archive timeout* só é chamado para segmentos WAL completos. No entanto, se o servidor gerar pouco tráfego WAL, pode existir um atraso entre a finalização da transação e o arquivamento no *archive storage*.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
'10'	498.1931	0.0241	0.0266
'60'	205.7807	0.0343	0.0338
'100'	374.0678	0.0300	0.0316
'120'	538.7799	0.0236	0.0247
'180'	533.9000	0.0290	0.0285

Tabela 16: Alteração de parâmetros na opção *archive_timeout*

Através dos resultados obtidos, o grupo chegou à conclusão que a opção '120' era aquela que produzia resultados com uma melhor *performance*.

3.4 Isolamento

Para além dos parâmetros abordados acima, consideramos relevante estudar os diferentes níveis de isolamento que podem ser considerados no programa, e a influência que cada um tem no sistema e nas transações.

Opção	Throughput (tx/s)	Response time (s)	Abort rate (%)
<i>Read committed</i>	531.723	0.0245	0.021
<i>Repeatable read</i>	536.158	0.0266	0.029
<i>Serializable</i>	539.097	0.028	0.022

Tabela 17: Alteração de parâmetros no isolamento

Em relação à *performance* do programa, pode-se verificar que aumentou, não muito significativamente, consoante o parâmetro de isolamento escolhido, tal como aumentaram o tempo de resposta e o *abort rate*.

Tendo em consideração as características que definem cada parâmetro do isolamento, é possível explicar a obtenção destes valores. O nível *Serializable* é o mais rigoroso, em que a execução concorrente de transações produz o mesmo efeito que a sua execução em série. Já os outros níveis definem-se pela interação existente entre transações concorrentes. O nível *default* de isolamento –*Read committed*–, apenas considera nas *queries* os dados submetidos antes do começo da mesma, e não os não submetidos ou as mudanças durante a execução da *query* por transações concorrentes. O nível *Repeatable read* difere do *Read committed* no sentido em que garante rigorosamente que cada transação vê a base de dados num estado estável, e por isso apresentando maior *abort rate* e tempo de resposta.

Devido ao aumento de desempenho não ser significativo, o grupo decidiu manter o nível de isolamento *default* do *PostgreSQL* – *Read committed*.

3.5 Combinação dos parâmetros

Após testados alguns dos parâmetros das configurações do *PostgreSQL* individualmente e selecionados para cada um deles o valor que apresentava melhores métricas, falta verificar se a sua combinação se traduz também numa melhoria da *performance* da carga transacional.

3.5.1 *Settings*

Com base nos resultados dos testes anteriores, a configuração dos *settings* que vamos utilizar é a seguinte:

Settings

```
fsync = on
synchronous_commit = off
wal_sync_method = fsync
full_page_writes = on
commit_delay = 500
commit_siblings = 10
```

Comparando com a configuração *default* obtemos os seguintes valores:

Configuração	Throughput (tx/s)	Response time (s)	Abort rate (%)
<i>Default</i>	512.7671	0.0337	0.0320
Testada	537.0298	0.027	0.0297

Tabela 18: Comparação das configurações - *settings*

Comparando os valores da tabela acima, podemos verificar que utilizando a configuração que combina os melhores parâmetros dos testes que realizamos anteriormente, conseguimos ter uma melhoria no *throughput* medido durante a execução da carga transacional e por essa razão foi optado por manter esta configuração.

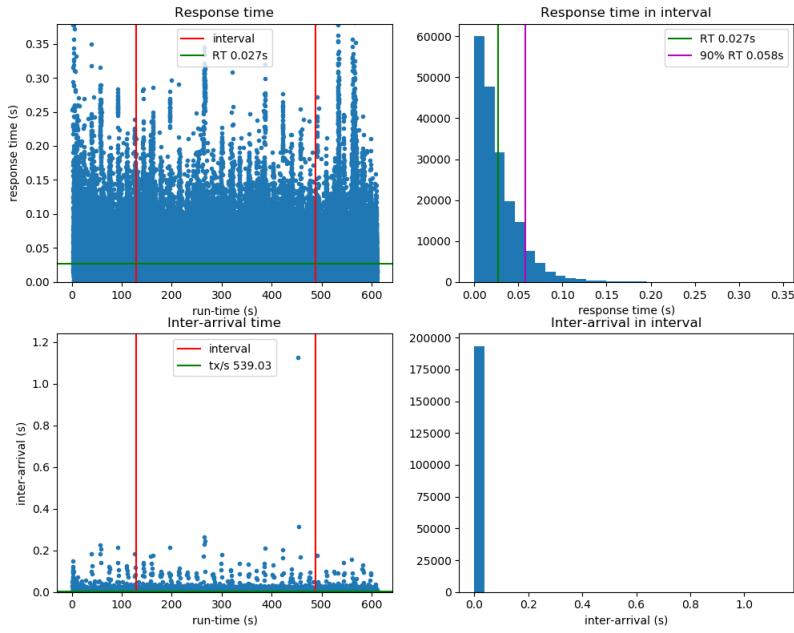


Figura 5: Métricas da configuração de *settings* escolhida

3.5.2 Checkpoints

Em relação às configurações relativas aos *checkpoints*, os parâmetros que obtiveram melhores resultados foram os seguintes:

Checkpoints

```
checkpoint_timeout = 1min
checkpoint_flush_after = 0kb
checkpoint_warning = 60s
max_wal_size = 1GB
min_wal_size = 160MB
```

Comparando com a configuração *default* obtemos os seguintes valores:

Configuração	Throughput (tx/s)	Response time (s)	Abort rate (%)
<i>Default</i>	512.7671	0.0337	0.0320
Testada	529.0361	0.0315	0.0321

Tabela 19: Comparação das configurações - *Checkpoints*

Podemos verificar que, alterando as configurações dos *checkpoints* para aquelas que nos deram melhor desempenho, foi atingida uma *performance* superior à configuração *default*. Sendo assim, decidimos manter a configuração que testamos.

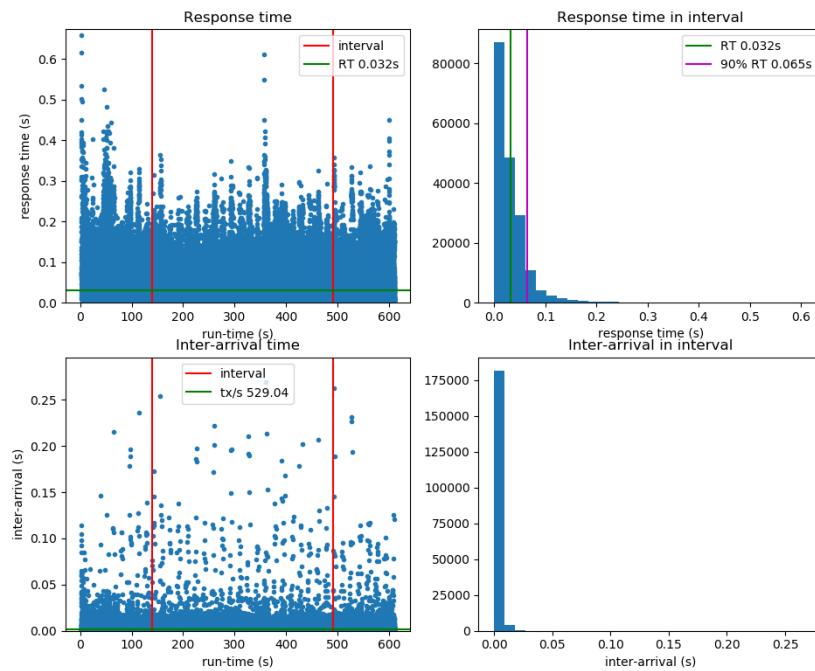


Figura 6: Métricas da configuração de *checkpoints* escolhida

3.5.3 Archiving

Finalmente, passando agora aos parametros do *archiving* chegamos à seguinte configuração:

Archiving

```
archive_mode = off
archive_command = '%f'
archive_timeout = 120
```

Comparando com a configuração *default* obtemos os seguintes valores:

Configuração	Throughput (tx/s)	Response time (s)	Abort rate (%)
<i>Default</i>	512.7671	0.0337	0.0320
Testada	535.8648	0.03037	0.0307

Tabela 20: Comparaçao das configurações - *Archiving*

Novamente, conseguimos verificar uma melhor *performance* com a configuração testada.

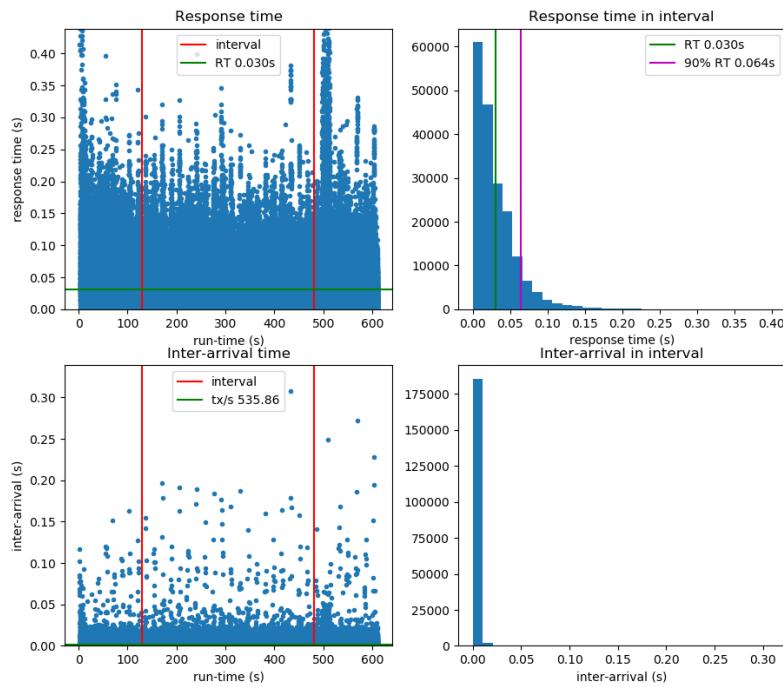


Figura 7: Métricas da configuração de *archiving* escolhida

3.5.4 *Settings* e *Checkpoints*

Os testes anteriores foram realizados de forma independente, ou seja, em relação à configuração *default* apenas foram alterados os parâmetros presentes no bloco de código. Nestes três testes realizados, aquele em que obtemos melhores métricas foi no *settings*. Agora pretendemos testar se a combinação destes três tipos resultará numa configuração com melhor desempenho do que naquela que foi referida anteriormente.

Assim, começamos por combinar as melhores configurações de *Settings* e *Checkpoints* que obtivemos.

Settings

```
fsync = on
synchronous_commit = off
wal_sync_method = fsync
full_page_writes = on
```

```
commit_delay = 500
commit_siblings = 10
```

Checkpoints

```
checkpoint_timeout = 1min
checkpoint_flush_after = 0kb
checkpoint_warning = 60s
max_wal_size = 1GB
min_wal_size = 160MB
```

Com as quais obtemos os seguintes resultados:

Configuração	Throughput (tx/s)	Response time (s)	Abort rate (%)
<i>Settings</i>	537.0298	0.027	0.0297
Testada	521.0464	0.0321	0.0333

Tabela 21: Comparação das configurações - *Settings* e *Checkpoints*

Comparando os valores na tabela acima, podemos verificar que houve uma diminuição do desempenho quando se combinaram as melhores configurações de *Settings* e *Checkpoints* quando comparado com os resultados obtidos quando alteramos apenas os parâmetros do *settings*. Por essa razão, foi decidido não se utilizar esta configuração como configuração de referência.

3.5.5 *Settings* e *Archiving*

Combinando agora as configurações dos *settings* e *archiving*, os resultados foram os seguintes:

Settings

```
fsync = on
synchronous_commit = off
wal_sync_method = fsync
full_page_writes = on
commit_delay = 500
commit_siblings = 10
```

Archiving

```
archive_mode = off
archive_command = '%f'
archive_timeout = 120
```

Configuração	Throughput (tx/s)	Response time (s)	Abort rate (%)
<i>Settings</i>	537.0298	0.027	0.0297
Testada	527.7618	0.0327	0.03429

Tabela 22: Comparaçāo das configurações - *Settings* e *Archiving*

Mais uma vez, podemos verificar que a configuração em que mudamos apenas os parâmetros dos *settings* apresenta um melhor desempenho do que quando tentamos combinar os melhores parâmetros que encontramos para *Settings* e *Archiving*.

3.5.6 *Checkpoints* e *Archiving*

O último par que falta testar é *checkpoints* e *archiving*, cujos resultados são apresentados abaixo:

Checkpoints

```
checkpoint_timeout = 1min
checkpoint_flush_after = 0kb
checkpoint_warning = 60s
max_wal_size = 1GB
min_wal_size = 160MB
```

Archiving

```
archive_mode = off
archive_command = '%f'
archive_timeout = 120
```

Configuração	Throughput (tx/s)	Response time (s)	Abort rate (%)
<i>Settings</i>	537.0298	0.027	0.0297
Testada	519.1044	0.0321	0.0329

Tabela 23: Comparaçāo das configurações - *Checkpoints* e *Archiving*

Novamente, verificamos que apenas alterando os parâmetros dos *settings* se verifica um melhor desempenho da carga transacional.

3.5.7 *Settings, Checkpoints e Archiving*

Finalmente, passamos à testagem dos três tipos de configurações:

Settings

```
fsync = on
synchronous_commit = off
wal_sync_method = fsync
full_page_writes = on
commit_delay = 500
commit_siblings = 10
```

Checkpoints

```
checkpoint_timeout = 1min
checkpoint_flush_after = 0kb
checkpoint_warning = 60s
max_wal_size = 1GB
min_wal_size = 160MB
```

Archiving

```
archive_mode = off
archive_command = '%f'
archive_timeout = 120
```

Configuração	Throughput (tx/s)	Response time (s)	Abort rate (%)
<i>Settings</i>	537.0298	0.027	0.0297
Testada	514.4554	0.0360	0.0296

Tabela 24: Comparaçāo das configurações - *Settings, Checkpoints e Archiving*

Testando a combinação dos três tipos de configurações, verificamos que mais uma vez, foi medido um melhor desempenho quando alteramos apenas os parâmetros dos *settings*.

3.5.8 Configuração de referência

Testadas todas as combinações possíveis dos três tipos de parâmetros testados, concluímos que a melhor configuração seria apenas alterar a configuração dos *settings*.

Sendo assim os valores de configuração para o *PostgreSQL* que otimizam o desempenho da carga transacional são:

Settings

```
fsync = on  
synchronous_commit = off  
wal_sync_method = fsync  
full_page_writes = on  
commit_delay = 500  
commit_siblings = 10
```

Checkpoints

```
checkpoint_timeout = 5min  
checkpoint_flush_after = 256kb  
checkpoint_warning = 30s  
max_wal_size = 1GB  
min_wal_size = 80MB
```

Archiving

```
archive_mode = off  
archive_command = ""  
archive_timeout = 0
```

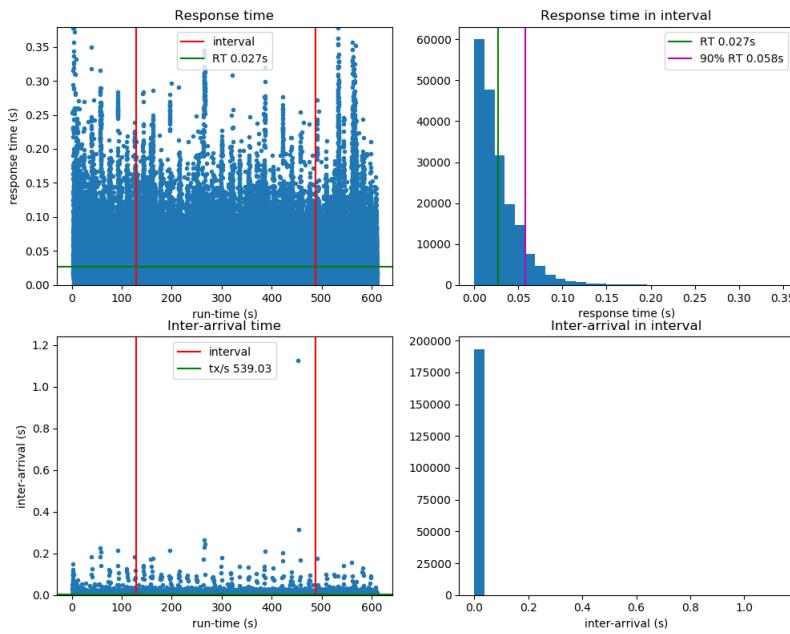


Figura 8: Métricas da configuração de *settings* escolhida

4 Análise e otimização das interrogações analíticas

Depois de termos encontrado uma configuração do *PostgreSQL* que otimizasse o desempenho da carga transacional, passamos à otimização das interrogações analíticas fornecidas no enunciado, baseadas na adaptação TPC-H.

Para se realizar a otimização de cada uma das interrogações, o grupo começou sempre por analisar o *output* do comando **EXPLAIN ANALYSE** do *PostgreSQL*, que devolve o plano de execução das *queries*. Cada um dos planos foi cuidadosamente analisado com o objetivo de encontrar as operações mais "pesadas" e que tinham maior impacto na *performance*. Depois de encontrados os gargalos do desempenho, prosseguimos à otimização das *queries*, utilizando para isso índices e vistas materializadas, que apesar de acrescentarem redundância, permitem aumentar, significativamente, o desempenho de *queries*.

A razão pela qual optamos por utilizar índices deve-se ao facto destes serem utilizados para acelerar a execução de *queries*, visto que permitem realizar *lookup* de registos bastante rápido. Já as vistas materializadas contêm os resultados de uma *query* que está materializada na base de dados, evitando assim que esta seja calculada mais do que uma vez.

No início de cada teste foi deixado o valor *default* do parâmetro '**max_parallel_workers_per_gather**'. Mais tarde, depois de adicionados os índices e vistas convenientes, va-

mos testar se alterar o número de *workers* a executar em paralelo tem algum impacto na *performance* da *query*.

Em seguida, encontram-se listados os índices que já estavam presentes na base de dados:

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

Figura 9: Listagem dos índices presentes na base de dados

Adicionalmente, o grupo também optou por recorrer ao site <https://explain.depesz.com/> para facilitar a leitura do plano de execução e das suas estatísticas como o tempo de execução de cada uma das operações.

Para além disso, recorremos à especificação do *benchmark* disponibilizada [aqui](#), que inclui informações como as colunas de cada uma das tabelas da base de dados, de forma a facilitar a leitura de cada uma das interrogações.

4.1 Interrogacão A1

```

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;

```

Nesta interrogação obtemos informação quanto ao total de gastos de cada cliente a partir de uma dada data, '2022-01-01 00:00:00.000000'. Como *output* é dado um conjunto de informações de cada cliente, como o seu id e último nome, a informação do país em

que reside e o total gasto desde a data referida, "revenue". Os resultados encontram-se ordenados por ordem decrescente em função do total gasto.

Ainda sem alterar nada, executando o comando EXPLAIN ANALYZE para esta interrogação, obtemos o resultado presente na figura abaixo, onde podemos verificar que o tempo de execução inicial era 263511.998ms, o que é um valor muito elevado.

```

QUERY PLAN
Sort  (cost=108117920.13 .. 10833977.13 rows=6462801 width=195) (actual time=262358.724 .. 263134.178 rows=2400000 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC
  Sort Method: external merge Disk: 232976kB
-> Finalize GroupAggregate  (cost=7353758.92 .. 8231217.86 rows=6462801 width=195) (actual time=237415.637 .. 259090.479 rows=2400000 loops=1)
  	Group Key: customer.c_id, customer.c_last, customer.c_city, customer.c_phone, nation.n_name
  	-> Gather Merge  (cost=7353758.92 .. 80956183.66 rows=5385668 width=195) (actual time=237415.537 .. 254779.946 rows=2401199 loops=1)
   	  Workers Planned: 2
   	  Workers Launched: 2
  	-> Partial GroupAggregate  (cost=7352758.80 .. 7433543.92 rows=2692834 width=195) (actual time=237279.162 .. 250141.211 rows=800400 loops=3)
   	  Group Key: customer.c_id, customer.c_last, customer.c_city, customer.c_phone, nation.n_name
  	-> Sort  (cost=7352758.90 .. 7433543.98 rows=2692834 width=168) (actual time=237279.045 .. 244797.750 rows=6800102 loops=3)
    	Sort Key: customer.c_id, customer.c_last, customer.c_city, customer.c_phone, nation.n_name
    	Sort Method: external merge Disk: 648160kB
      Worker 0: Sort Method: external merge Disk: 649280kB
      Worker 1: Sort Method: external merge Disk: 670936kB
  	-> Hash Join  (cost=6047936.79 .. 6623356.88 rows=2692834 width=168) (actual time=184479.824 .. 199288.936 rows=6800102 loops=3)
    	Hash Cond: ((asclilsubstr((customer.c_state)::text, 1, 1)) % nation.nationkey)
    	-> Parallel Hash Join  (cost=6047922.96 .. 6568894.37 rows=3168040 width=67) (actual time=183067.273 .. 188576.883 rows=6800102 loops=3)
      	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=5829837.63 .. 6271608.90 rows=3122301 width=25) (actual time=130573.047 .. 169512.755 rows=6800102 loops=3)
        	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=5343040.22 .. 5399267.24 rows=22490808 width=25) (actual time=121513.871 .. 145528.038 rows=18545702 loops=3)
          	Sort Key: order_line.ol_w_id, order_line.ol_d_id, order_line.ol_o_id
          	Sort Method: external merge Disk: 726040kB
          	Worker 0: Sort Method: external merge Disk: 719904kB
          	Worker 1: Sort Method: external merge Disk: 736704kB
        	-> Parallel Seq Scan on order_line  (cost=0.00 .. 982261.07 rows=22490808 width=25) (actual time=0.849 .. 20405.659 rows=18560071 loops=3)
          	-> Materialize  (cost=6055.37 .. 19981.20 rows=2443159 width=24) (actual time=9655.352 .. 12769.684 rows=8342031 loops=3)
          	-> Sort  (cost=486791.41 .. 492985.31 rows=2443159 width=24) (actual time=9655.341 .. 10664.675 rows=2400000 loops=3)
            	Sort Key: orders.o_w_id, orders.o_d_id, orders.o_id
            	Sort Method: external merge Disk: 79896kB
            	Worker 0: Sort Method: external merge Disk: 79896kB
            	Worker 1: Sort Method: external merge Disk: 79896kB
            	-> Seq Scan on orders  (cost=0.00 .. 127362.38 rows=2443159 width=24) (actual time=0.092 .. 3285.230 rows=2400000 loops=3)
              	Filter: (o_entry_d >= '2022-01-01 00:00:00'::timestamp without time zone)
              	Rows Removed by Filter: 3528030
          	-> Parallel Hash  (cost=188854.12 .. 188854.12 rows=1080412 width=70) (actual time=5338.609 .. 5338.617 rows=800000 loops=3)
            	Buckets: 65536 Batches: 128 Memory Usage: 2464kB
            	-> Parallel Seq Scan on customer  (cost=0.00 .. 188854.12 rows=1080412 width=70) (actual time=0.335 .. 4797.476 rows=800000 loops=3)
              	-> Hash  (cost=11.70 .. 11.70 rows=170 width=108) (actual time=1463.453 .. 1463.455 rows=25 loops=3)
                	Buckets: 1024 Batches: 1 Memory Usage: 10kB
                	-> Seq Scan on nation  (cost=0.00 .. 127362.170 rows=170 width=108) (actual time=1463.407 .. 1463.418 rows=25 loops=3)

Planning Time: 31.749 ms
JIT:
  Functions: 159
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 38.909 ms, Inlining 444.632 ms, Optimization 2451.25 ms, Emission 1489.975 ms, Total 4424.541 ms
Execution Time: 263511.998 ms
(49 rows)

```

Figura 10: EXPLAIN ANALYZE da interrogação A1

Utilizando o *site* que foi mencionado em cima, obtemos o seguinte resultado:

#	exclusive	inclusive	rows x	rows	loops	node
1.	4,043,699	263,134,178	± 2.7	2,400,000	1	→ Sort (cost=10.817,820.13..10.833,977.13 rows=6,462,801 width=195) (actual time=262,358,724..263,134,178 rows=2,400,000 loops=1) Sort Key: (sum(order_line.ol_amount)) DESC Sort Method: external merge Disk: 232,976kB
2.	4,319,533	259,090,479	± 2.7	2,400,000	1	★ → Finalize GroupAggregate (cost=7,353,758.92..8,231,217.86 rows=6,462,801 width=195) (actual time=237,415,637..259,090,479 rows=2,400,000 loops=1) Group Key: customer.c_id, customer.c_last, customer.c_city, customer.c_phone, nation.n_name
3.	4,629,735	254,770,946	± 2.2	2,401,199	1	→ Gather Merge (cost=7,353,758.92..8,056,183.66 rows=5,385,668 width=195) (actual time=237,415,537..254,770,946 rows=2,401,199 loops=1) Workers Planned: 2 Workers Launched: 2
4.	5,343,461	250,141,211	± 3.4	2,401,200	3 / 3	→ Partial GroupAggregate (cost=7,352,758.90..7,433,543.92 rows=2,692,834 width=195) (actual time=237,279,162..250,141,211 rows=800,400 loops=3) Group Key: customer.c_id, customer.c_last, customer.c_city, customer.c_phone, nation.n_name
5.	45,508,814	244,797,750	± 2.5	20,400,306	3 / 3	→ Sort (cost=7,352,758.90..7,359,490.98 rows=2,692,834 width=168) (actual time=237,279,045..244,797,750 rows=6,800,102 loops=3) Sort Key: customer.c_id, customer.c_last, customer.c_city, customer.c_phone, nation.n_name Sort Method: external merge Disk: 648,160kB Worker 0: Sort Method: external merge Disk: 649,280kB Worker 1: Sort Method: external merge Disk: 670,936kB
6.	9,248,598	199,288,936	± 2.5	20,400,306	3 / 3	→ Hash Join (cost=6,047,936.79..6,623,356.88 rows=2,692,834 width=168) (actual time=184,470,824..199,288,936 rows=6,800,102 loops=3) Hash Cond: (ascii(substr(customer.c_state),text,1,1) % 24) = nation.n_nationkey
7.	13,725,511	188,576,883	± 2.1	20,400,306	3 / 3	→ Parallel Hash Join (cost=6,047,922.96..6,568,694.37 rows=3,168,040 width=67) (actual time=183,007,273..188,576,883 rows=6,800,102 loops=3) 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)
8.	11,275,113	169,512,755	± 2.2	20,400,306	3 / 3	→ Merge Join (cost=5,829,837.63..6,271,608.90 rows=3,122,301 width=25) (actual time=130,573,047..169,512,755 rows=6,800,102 loops=3) 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.delivery_d)
9.	125,122,379	145,528,036	± 1.2	55,637,106	3 / 3	→ Sort (cost=5,343,940.22..5,399,267.24 rows=22,490,808 width=25) (actual time=121,513,871..145,528,038 rows=18,545,702 loops=3) Sort Key: order_line.ol_w_id, order_line.ol_d_id, order_line.ol_o_id Sort Method: external merge Disk: 726,040kB Worker 0: Sort Method: external merge Disk: 719,904kB Worker 1: Sort Method: external merge Disk: 730,704kB
10.	20,405,659	20,405,659	± 1.2	55,680,213	3 / 3	→ Parallel Seq Scan on order_line (cost=0..0.982,261.07 rows=22,490,808 width=25) (actual time=0.849..20,405,659 rows=18,560,071 loops=3)
11.	2,044,929	12,709,604	± 3.4	25,026,093	3 / 3	→ Materialize (cost=486,797.41..499,013.20 rows=2,443,159 width=24) (actual time=9,055,352..12,709,604 rows=8,342,031 loops=3)
12.	7,379,445	10,664,675	± 1.0	7,200,000	3 / 3	→ Sort (cost=486,797.41..492,905.31 rows=2,443,159 width=24) (actual time=9,055,341..10,664,675 rows=2,400,000 loops=3) Sort Key: orders.o_w_id, orders.o_d_id, orders.o_id Sort Method: external merge Disk: 79,896kB Worker 0: Sort Method: external merge Disk: 79,896kB Worker 1: Sort Method: external merge Disk: 79,896kB
13.	3,285,230	3,285,230	± 1.0	7,200,000 - 10,584,090	3 / 3	→ Seq Scan on orders (cost=0..0.127,362.39 rows=2,443,159 width=24) (actual time=0.092..3,285,230 rows=2,400,000 loops=3) Filter: (o_entry_d >= 2022-01-01 00:00:00 timestamp without time zone) Row Removal by Filter: 3,528,030
14.	541,141	5,338,617	± 1.3	2,400,000	3 / 3	→ Parallel Hash (cost=188,854.12..188,854.12 rows=1,000,412 width=70) (actual time=5,338,609..5,338,617 rows=800,000 loops=3) Buckets: 65,536 Batches: 128 Memory Usage: 2,464kB
15.	4,797,476	4,797,476	± 1.3	2,400,000	3 / 3	→ Parallel Seq Scan on customer (cost=0..0.188,854.12 rows=1,000,412 width=70) (actual time=0.335..4,797,476 rows=800,000 loops=3)
16.	0.037	1,463,455	± 6.8	75	3 / 3	→ Hash (cost=11,70..11,70 rows=170 width=108) (actual time=1,463,453..1,463,455 rows=25 loops=3) Buckets: 1,024 Batches: 1 Memory Usage: 10kB
17.	1,463,418	1,463,418	± 6.8	75	3 / 3	→ Seq Scan on nation (cost=0..0.11,70 rows=170 width=108) (actual time=1,463,407..1,463,418 rows=25 loops=3)

Figura 11: EXPLAIN ANALYZE da interrogação A1

4.1.1 Primeira versão de vista materializada

Como podemos ver, as operações que são o gargalo da *performance* da interrogação são os *joins*, *aggregate* e o *order by*. Sendo assim, como primeiro passo para melhorar o desempenho desta *query*, o grupo decidiu criar uma vista materializada que contivesse as condições de *join* das tabelas e a projeção das colunas que estão presentes na *query* original. Isso exclui o *order by*, *group by*, o cálculo das somas de '*ol_amount*' e a condição filtra os resultados consoante a data dada como *input*, para que o utilizador continuasse a ter a liberdade de filtrar pela data que desejassem.

Depois de testada, o grupo apercebeu-se que ainda poderia ser feita uma otimização à vista criada, ordenando os registos com base na coluna '*o_entry_d*'. Isto tornaria mais rápida a filtragem por data quando no futuro fosse criado um índice sobre esta mesma coluna. Isto acontece para evitar o problema de dispersão de registos, que podia levar ao carregamento de um número desnecessário de blocos do disco. Desta forma, a vista criada foi a seguinte:

```
create materialized view a1_aux as select c_id, c_last, ol_amount,
c_city, c_phone, n_name, o_entry_d
```

```

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
order by o_entry_d desc;

```

Com a vista materializada, a interrogação passa a ter o seguinte aspeto:

```

select c_id, c_last, sum(ol_amount) as revenue, c_city, c_phone,
n_name
from a1_aux
where o_entry_d >= '2022-01-01 00:00:00.000000'
group by c_id, c_last, c_city, c_phone, n_name
order by revenue desc;

```

Correndo agora o EXPLAIN ANALYZE com esta nova interrogação, obtemos o resultado presente na imagem abaixo, onde vemos que o tempo de execução baixou para 65973.442ms, uma descida de aproximadamente 74.96% :

```

QUERY PLAN
-----
Sort (cost=3832483.54..3837583.58 rows=2040018 width=117) (actual time=65197.283..65756.760 rows=2400000 loops=1)
  Sort Key: (sum(ol_amount)) DESC
  Sort Method: external merge Disk: 232976kB
    -> Finalize GroupAggregate (cost=2625593.24..3367668.11 rows=2040018 width=117) (actual time=40445.740..61642.819 rows=2400000 loops=1)
        Group Key: c_id, c_last, c_city, c_phone, n_name
        -> Gather Merge (cost=2625593.24..3270767.26 rows=4080036 width=117) (actual time=40445.660..57398.632 rows=2401209 loops=1)
            Workers Planned: 2
            Workers Launched: 2
            -> Partial GroupAggregate (cost=2624593.22..2798829.85 rows=2040018 width=117) (actual time=40172.734..52940.479 rows=800403 loops=3)
                Group Key: c_id, c_last, c_city, c_phone, n_name
                -> Sort (cost=2624593.22..2645841.28 rows=8499223 width=88) (actual time=40172.663..47667.449 rows=6800102 loops=3)
                    Sort Key: c_id, c_last, c_city, c_phone, n_name
                    Sort Method: external merge Disk: 664412kB
                    Worker 0: Sort Method: external merge Disk: 656816kB
                    Worker 1: Sort Method: external merge Disk: 647432kB
                    -> Parallel Seq Scan on a1_aux (cost=0.00..426268.92 rows=8499223 width=88) (actual time=560.012..4256.189 rows=6800102 loops=3)
                        Filter: (o_entry_d >= '2022-01-01 00:00:00'::timestamp without time zone)
Planning Time: 0.195 ms
 JIT:
   Functions: 30
   Options: Inlining true, Optimization true, Expressions true, Deforming true
   Timing: Generation 8.761 ms, Inlining 335.340 ms, Optimization 818.572 ms, Emission 524.027 ms, Total 1686.701 ms
Execution Time: 65973.442 ms
(23 rows)

```

Figura 12: EXPLAIN ANALYZE da interrogação A1 com a primeira versão de vista materializada

4.1.2 Índice

Como foi referido anteriormente, o próximo objetivo era adicionar um índice sobre a coluna '*o_entry_d*' da vista criada para que a sua filtragem se tornasse mais rápida. Para

isso, foi criado o seguinte índice:

```
create index a1_aux_index on a1_aux(o_entry_d);
```

Com a adição deste índice, não se verificou um aumento significativo, visto que este não foi utilizado (mesmo depois de desligar o parâmetro '*enable_seqscan*' do ficheiro de configuração do *PostgreSQL*). Mesmo assim, por razões exteriores a nós, podendo estar por exemplo com a carga computacional da máquina, o tempo de execução diminuiu cerca de 8 segundos, passando a 65197.155ms.

```

-----  

QUERY PLAN  

-----  

Sort  (cost=3832741.98..3837842.05 rows=2040031 width=117) (actual time=64431.195..64989.194 rows=2400000 loops=1)
  Sort Key: (sum(ol_amount)) DESC
  Sort Method: external merge Disk: 232976kB
    -> Finalize GroupAggregate (cost=2625830.61..3367925.10 rows=2040031 width=117) (actual time=39833.323..60899.831 rows=2400000 loops=1)
      Group Key: c_id, c_last, c_city, c_phone, n_name
      -> Gather Merge (cost=2625830.61..3271023.63 rows=4080062 width=117) (actual time=39833.270..56661.216 rows=2401125 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        Partial GroupAggregate (cost=2624830.58..2799083.21 rows=2040031 width=117) (actual time=39595.302..52185.015 rows=800375 loops=3)
          Group Key: c_id, c_last, c_city, c_phone, n_name
          -> Sort (cost=2624830.58..2646680.90 rows=8500128 width=88) (actual time=39595.233..47019.789 rows=6800102 loops=3)
            Sort Key: c_id, c_last, c_city, c_phone, n_name
            Sort Method: external merge Disk: 654248kB
            Worker 0: Sort Method: external merge Disk: 641032kB
            Worker 1: Sort Method: external merge Disk: 673104kB
            -> Parallel Seq Scan on a1_aux (cost=0.00..426269.59 rows=8500128 width=88) (actual time=617.153..3622.563 rows=6800102 loops=3)
              Filter: (o_entry_d >= '2022-01-01 00:00:00'::timestamp without time zone)
Planning Time: 0.796 ms
JIT:
  Functions: 30
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 8.652 ms, Inlining 368.091 ms, Optimization 916.008 ms, Emission 557.378 ms, Total 1850.129 ms
Execution Time: 65197.155 ms
(23 rows)

```

Figura 13: EXPLAIN ANALYZE da interrogação A1 com índice

Como tentativa final de otimizar o desempenho da interrogação, utilizando redundância, foi decidido criar uma nova versão da vista materializada que desta vez incluía também o cálculo das somas dos totais de cada cliente, assim como a operação de *group by*, com o objetivo de diminuir a carga computacional que é executada durante a interrogação em si. Novamente, a operação de filtragem pela data não foi incluída na vista, pela mesma razão referida anteriormente.

De forma semelhante, foi criado novamente um índice sobre a coluna '*o_entry_d*' com o intuito de observar se este passaria a ser agora utilizado.

4.1.3 Segunda versão de vista materializada

```
create materialized view a1_aux2 as select c_id, c_last, sum(ol_amount)
as revenue, c_city, c_phone, n_name, o_entry_d
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 c_id, c_last, c_city, c_phone, n_name, o_entry_d  
order by o_entry_d desc;  
  
create index a1_aux2_index on a1_aux2(o_entry_d);
```

A nova interrogação passou a ter este aspetto:

```
    select c_id, c_last, revenue, c_city, c_phone, n_name  
from al_aux2  
where o_entry_d >= '2022-01-01 00:00:00.000000'  
order by revenue desc;
```

```
QUERY PLAN
-----
Gather Merge  (cost=253340.87..486690.49 rows=2000000 width=89) (actual time=2170.733..3767.263 rows=2400000 loops=1)
  Workers Planned: 2
  Workers Launched: 2
-> Sort  (cost=252340.84..254840.84 rows=1000000 width=89) (actual time=2095.291..2586.619 rows=800000 loops=3)
    Sort Key: revenue
    Sort Method: external merge Disk: 69872kB
    Worker 0: Sort Method: external merge Disk: 79536kB
    Worker 1: Sort Method: external merge Disk: 83704kB
-> Parallel Seq Scan on a1_aux2  (cost=0.00..50140.00 rows=1000000 width=89) (actual time=12.484..507.141 rows=800000 loops=3)
      Filter: (o_entry_d >= '2022-01-01 00:00:00':timestamp without time zone)
Planning Time: 0.204 ms
JIT:
  Functions: 12
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 3.871 ms, Inlining 0.000 ms, Optimization 2.236 ms, Emission 33.961 ms, Total 40.068 ms
Execution Time: 3945.224 ms
(16 rows)
```

Figura 14: EXPLAIN ANALYZE da interrogação A1 com a segunda versão de vista materializada

Com esta última vista materializada conseguimos baixar o tempo de execução para 3945.224ms, o que comparando com os 263511.998ms iniciais, se traduz numa diminuição de 98.50%.

Uma nota que gostaríamos de salientar é o facto da vista ser muito específica para a *query* em estudo e que por essa razão provavelmente não teria muito uso fora desta. Outro ponto importante é que a vista foi criada a partir de tabelas que podem ser alteradas com frequência, e por isso deve ser refletido qual a estratégia para a atualização da mesma. Se esta interrogação for realizada com muita frequência e precise de resultados em tempo real, a atualização da vista deve ser feita utilizando *triggers*. No entanto, se a interrogação for corrida esporadicamente, deve-se optar por recalcular a *view* quando for necessário, visto que os *triggers* têm a consequência de acrescentar peso computacional sobre a base de dados.

4.1.4 Paralelismo

Adicionalmente, o grupo ainda experimentou aumentar o número de *workers* para 4, alterando para isso o parâmetro '*max_parallel_workers_per_gather*', no sentido de testar

o paralelismo. Em seguida, voltou-se a correr o EXPLAIN ANALYZE, com o qual averiguamos que o *PostgreSQL* optou por recorrer na mesma apenas a 2 *workers*.

```
QUERY PLAN
-----
Gather Merge (cost=253340.87..486690.49 rows=2000000 width=89) (actual time=2118.996..3719.325 rows=2400000 loops=1)
  Workers Planned: 2
  Workers Launched: 2
    -> Sort (cost=252340.84..254840.84 rows=1000000 width=89) (actual time=2060.563..2533.103 rows=800000 loops=3)
        Sort Key: revenue
        Sort Method: external merge Disk: 76320kB
        Worker 0: Sort Method: external merge Disk: 74016kB
        Worker 1: Sort Method: external merge Disk: 82792kB
          -> Parallel Seq Scan on ai_aux2 (cost=0.00..50140.00 rows=1000000 width=89) (actual time=16.523..477.917 rows=800000 loops=3)
              Filter: (o_entry_d >= '2022-01-01 00:00:00'::timestamp without time zone)
Planning Time: 0.174 ms
JIT:
  Functions: 12
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 2.946 ms, Inlining 0.000 ms, Optimization 1.785 ms, Emission 46.983 ms, Total 51.715 ms
Execution Time: 3888.948 ms
(16 rows)
```

Figura 15: EXPLAIN ANALYZE da interrogação A1 com 4 workers

Sendo assim, a otimização que fizemos permitiu baixar o tempo de execução para 3888.948ms, o que consideramos ser uma otimização significativa.

4.2 Interrogacão A2

```
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;
```

Nesta interrogação obtemos informação quanto aos 100 fornecedores que mais contribuíram para a receita global em itens enviados durante um dado período de tempo (a partir do dia 1 de janeiro de 2022).

Ainda sem alterar nada, executando o comando EXPLAIN ANALYZE para esta interrogação obtemos o resultado presente na figura abaixo, onde podemos verificar que o tempo de execução inicial era 92068.149ms.

```

Merge Join  (cost=5644485.21..5651536.40 rows=400068 width=103) (actual time=91715.229..92008.445 rows=99 loops=1)
  Merge Cond: (supplier.su_suppkey = revenue.supplier_no)
    CTE revenue
      > Finalize GroupAggregate  (cost=5313475.85..5526191.97 rows=800137 width=36) (actual time=91295.635..91630.468 rows=10000 loops=1)
        Group Key: (mod(stock.s_w_id * stock.s_i_id), 10000)
        > Gather Merge  (cost=5313475.85..5500187.52 rows=1600274 width=36) (actual time=91295.569..91609.901 rows=30000 loops=1)
          Workers Planned: 2
          Workers Launched: 2
            > Sort  (cost=5312475.83..5314476.17 rows=800137 width=36) (actual time=91189.254..91191.623 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=4781835.71..5212140.84 rows=800137 width=36) (actual time=90410.507..91184.087 rows=10000 loops=3)
              Group Key: mod((stock.s_w_id * stock.s_i_id), 10000)
              Planned Partitions: 64  Batches: 65  Memory Usage: 4497kB  Disk Usage: 57864kB
              Worker 0:  Batches: 65  Memory Usage: 4497kB  Disk Usage: 53768kB
              Worker 1:  Batches: 65  Memory Usage: 4497kB  Disk Usage: 57864kB
                > Parallel Hash Join  (cost=515579.57..2050889.77 rows=42629400 width=9) (actual time=65323.664..78579.592 rows=14898327 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..1838488.09 rows=18024580 width=1) (actual time=1.861..28446.237 rows=14898327 loops=3)
                    Filter: (ol_delivery_d >='2022-01-01 00:00:00'::timestamp without time zone)
                    Rows Removed by Filter: 3661744
                  > Parallel Hash  (cost=452547.03..452547.03 rows=3333903 width=8) (actual time=17376.135..17376.137 rows=2666667 loops=3)
                    Buckets: 131072  Batches: 128  Memory Usage: 3520kB
                      > Parallel Seq Scan on stock  (cost=0.00..452547.03 rows=3333903 width=8) (actual time=245.101..15108.332 rows=2666667 loops=3)

      > Sort  (cost=986.39..1011.39 rows=10000 width=71) (actual time=365.324..366.317 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=357.158..359.269 rows=10000 loops=1)

-> Materialize  (cost=117306.86..119307.20 rows=400068 width=36) (actual time=91349.833..91349.872 rows=100 loops=1)
  > Sort  (cost=117306.86..118307.03 rows=400068 width=36) (actual time=91349.825..91349.843 rows=100 loops=1)
    Sort Key: revenue.supplier_no
    Sort Method: quicksort Memory: 29kB
    > Hash Semi Join  (cost=46585.90..69139.76 rows=400068 width=36) (actual time=91345.893..91349.789 rows=100 loops=1)
      Hash Cond: (revenue.total_revenue = revenue_.total_revenue)
      > CTE Scan on revenue  (cost=0.00..16002.74 rows=800137 width=36) (actual time=91295.639..91297.168 rows=10000 loops=1)
      > Hash  (cost=46584.65..46584.65 rows=100 width=32) (actual time=50.223..50.227 rows=100 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 13kB
        > Limit  (cost=46583.40..46583.74 rows=800137 width=32) (actual time=50.142..50.165 rows=100 loops=1)
          > Sort  (cost=46583.40..48583.74 rows=800137 width=32) (actual time=50.134..50.145 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..16002.74 rows=800137 width=32) (actual time=0.006..46.613 rows=10000 loops=1)

Planning Time: 2.508 ms
JIT:
  Functions: 99
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 16.753 ms, Inlining 435.978 ms, Optimization 602.002 ms, Emission 333.970 ms, Total 1388.703 ms
Execution Time: 92086.149 ms
(50 rows)

```

Figura 16: EXPLAIN ANALYZE da interrogação A2

Utilizando o *site* que foi mencionado em cima, obtemos o seguinte resultado:

#	exclusive	inclusive	rows_X	rows	loops	node
1.	292.256	92.008.445	+ 4.041.1		99	1 → <u>Merge Join</u> (cost=5,644,485.21..5,651,536.40 rows=400,068 width=103) (actual time=91,715,229..92,008,445 rows=99 loops=1) Merge Cond: (supplier.su_suppkey = revenue.supplier_no)
2.						CTE revenue
3.	20.567	91.630.468	+ 80.0	10,000	1	→ <u>Finalize GroupAggregate</u> (cost=5,313,475.85..5,526,191.97 rows=800,137 width=36) (actual time=91,295,635..91,630,468 rows=10,000 loops=1) Group Key: (mod((stock.s_w_id * stock.s_i_id), 10000))
4.	418.278	91.609.901	+ 53.3	30,000	1	→ <u>Gather Merge</u> (cost=5,313,475.85..5,500,187.52 rows=1,600,274 width=36) (actual time=91,295,569..91,609,901 rows=30,000 loops=1) Workers Planned: 2 Workers Launched: 2
5.	7.536	91.191.623	+ 80.0	30,000	3 / 3	→ <u>Sort</u> (cost=5,312,475.83..5,314,476.17 rows=800,137 width=36) (actual time=91,189,254..91,191,623 rows=10,000 loops=3) Sort Key: (mod((stock.s_w_id * stock.s_i_id), 10000)) Sort Method: quicksort Memory: 1,791kB Worker 0: Sort Method: quicksort Memory: 1,791kB Worker 1: Sort Method: quicksort Memory: 1,791kB
6.	12,604.495	91.184.087	+ 80.0	30,000	3 / 3	→ <u>Partial HashAggregate</u> (cost=4,781,835.71..5,212,140.84 rows=800,137 width=36) (actual time=90,410,507..91,184,087 rows=10,000 loops=3) Group Key: mod((stock.s_w_id * stock.s_i_id), 10000) Planned Partitions: 64 Batches: 65 Memory Usage: 4,497kB Disk Usage: 57,864kB Worker 0: Batches: 65 Memory Usage: 4,497kB Disk Usage: 53,768kB Worker 1: Batches: 65 Memory Usage: 4,497kB Disk Usage: 57,864kB
7.	32,757.218	78.579.592	+ 2.9	44,694,981	3 / 3	→ <u>Parallel Hash Join</u> (cost=515,579.57..2,050,889.77 rows=42,629,400 width=9) (actual time=65,323,664..78,579,592 rows=14,898,327 loops=3) Hash Cond: ((order_line.ol_i_id = stock.s_i_id) AND (order_line.ol_supply_w_id = stock.s_w_id))
8.	28,446.237	28,446.237	+ 1.2	44,694,981 - 10,985,232	3 / 3	→ <u>Parallel Seq Scan</u> on order_line (cost=0.00..1,038,488.09 rows=18,024,580 width=13) (actual time=1,861..28,446,237 rows=14,898,327 loops=3) Filter: (ol_delivery_d >= 2022-01-01 00:00:00 timestamp without time zone) Rows Removed by Filter: 3,661,744
9.	2,267.805	17,376.137	+ 1.3	8,000,001	3 / 3	→ <u>Parallel Hash</u> (cost=452,547.03..452,547.03 rows=3,333,903 width=8) (actual time=17,376,135..17,376,137 rows=2,666,667 loops=3) Buckets: 131,072 Batches: 128 Memory Usage: 3,520kB
10.	15,108.332	15,108.332	+ 1.3	8,000,001	3 / 3	→ <u>Parallel Seq Scan</u> on stock (cost=0.00..452,547.03 rows=3,333,903 width=8) (actual time=245,101..15,108,332 rows=2,666,667 loops=3)
11.	7,048	366,317	+ 1.0	9,921	1	→ <u>Sort</u> (cost=986.39..1,011,39 rows=10,000 width=11) (actual time=365,324..366,317 rows=9,921 loops=1) Sort Key: supplier.su_suppkey Sort Method: quicksort Memory: 1,791kB
13.	0.029	91,349,872	+ 4,000.7	100	1	→ <u>Materialize</u> (cost=117,306.66..119,307.20 rows=400,068 width=36) (actual time=91,349,833..91,349,872 rows=100 loops=1)
14.	0.054	91,349,843	+ 4,000.7	100	1	→ <u>Sort</u> (cost=117,306.86..118,307.03 rows=400,068 width=36) (actual time=91,349,825..91,349,843 rows=100 loops=1) Sort Key: revenue.supplier_no Sort Method: quicksort Memory: 29kB
15.	2.394	91,349,789	+ 4,000.7	100	1	→ <u>Hash Semi Join</u> (cost=46,585.90..69,139.76 rows=400,068 width=36) (actual time=91,345,893..91,349,789 rows=100 loops=1) Hash Cond: (revenue.total_revenue = revenue_1.total_revenue)
16.	0.000	91,297,166	+ 80.0	10,000	1	→ <u>CTE Scan</u> on revenue (cost=0.00..16,002.74 rows=800,137 width=36) (actual time=91,295,639..91,297,168 rows=10,000 loops=1)
17.	0.062	50,227	+ 1.0	100	1	→ <u>Hash</u> (cost=46,584.65..46,584.65 rows=100 width=32) (actual time=50,223..50,227 rows=100 loops=1) Buckets: 1,024 Batches: 1 Memory Usage: 13kB
18.	0.020	50,165	+ 1.0	100	1	→ <u>Limit</u> (cost=46,583.40..46,583.65 rows=100 width=32) (actual time=50,142..50,165 rows=100 loops=1)
19.	3.532	50,145	+ 8,001.4	100	1	→ <u>Sort</u> (cost=46,583.40..48,583.74 rows=800,137 width=32) (actual time=50,134..50,145 rows=100 loops=1) Sort Key: revenue_1.total_revenue DESC Sort Method: top-N heapsort Memory: 32kB
20.	0.000	46,613	+ 80.0	10,000	1	→ <u>CTE Scan</u> on revenue revenue_1 (cost=0.00..16,002.74 rows=800,137 width=32) (actual time=0.006..46,613 rows=10,000 loops=1)

Figura 17: EXPLAIN ANALYZE da interrogação A2

4.2.1 Primeira versão com índices

Como podemos ver, as operações que são o gargalo da *performance* da interrogação são os *aggregate*, *sort* e o *join*. Como primeiro passo, reparamos que a interrogação trata de filtrar a tabela *order_line* de acordo com a data de entrega no espaço de tempo definido (a partir do dia 1 de janeiro de 2022) e, posteriormente, faz *join* desta com a tabela *supplier* pela coluna *su_suppkey*. Para tornar a filtragem mais rápida, foram criados os seguintes índices:

```
create index a2.ol_delivery_d_index on order_line(ol_delivery_d);
create index a2.su_suppkey_index on supplier(su_suppkey);
```

Correndo a *query* de novo, podemos ver que só o índice da tabela *supplier* foi utilizado e que a sua utilização diminuiu significativamente o tempo de execução para 72568.990ms.

```

Merge Join  (cost=5729967.65..5737488.98 rows=400068 width=103) (actual time=72341.941..72516.791 rows=99 loops=1)
  Merge Cond: (supplier.su_suppkey = revenue.supplier_no)
    CTE revenue
      -> Finalize GroupAggregate  (cost=5399944.38..5612668.51 rows=800137 width=36) (actual time=71935.886..72140.326 rows=10000 loops=1)
          Group Key: (mod((stock.s_w_id * stock.s_i_id), 10000))
          -> Gather Merge  (cost=5399944.38..5586656.05 rows=1600274 width=36) (actual time=71935.845..72120.691 rows=30000 loops=1)
              Workers Planned: 2
              Workers Launched: 2
              -> Sort  (cost=5398944.36..5400944.70 rows=800137 width=36) (actual time=71836.631..71838.837 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=4859407.06..5298609.38 rows=800137 width=36) (actual time=71029.232..71819.463 rows=10000 loops=3)
                      Group Key: mod((stock.s_w_id * stock.s_i_id), 10000)
                      Planned Partitions: 64 Batches: 65 Memory Usage: 4497kB Disk Usage: 56328kB
                      Worker 0: Batches: 65 Memory Usage: 4497kB Disk Usage: 53768kB
                      Worker 1: Batches: 65 Memory Usage: 4497kB Disk Usage: 57352kB
                      -> Parallel Hash Join  (cost=515579.57..2070095.57 rows=43540472 width=9) (actual time=34768.373..53703.558 rows=14898327 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..1047354.12 rows=18409880 width=13) (actual time=0.113..12985.476 rows=14898327 loops=3)
                              Filter: (ol_delivery_d >= '2022-01-01 00:00:00'::timestamp without time zone)
                              Rows Removed by Filter: 3661744
                          -> Parallel Hash  (cost=452547.03..452547.03 rows=3333903 width=8) (actual time=13730.518..13730.520 rows=2666667 loops=3)
                              Buckets: 131072 Batches: 128 Memory Usage: 3520kB
                              -> Parallel Seq Scan on stock  (cost=0.00..452547.03 rows=3333903 width=8) (actual time=235.344..12692.567 rows=2666667 loops=3)
--> Index Scan using a2_su_suppkey_index on supplier  (cost=0.00..495.43 rows=10000 width=36) (actual time=71986.096..71986.109 rows=100 loops=1)
--> Materialize  (cost=117306.86..119307.20 rows=400068 width=36) (actual time=71986.081..71986.109 rows=100 loops=1)
    -> Sort  (cost=117306.86..118307.03 rows=400068 width=36) (actual time=71986.081..71986.109 rows=100 loops=1)
        Sort Key: revenue.supplier_no
        Sort Method: quicksort Memory: 29kB
        -> Hash Semi Join  (cost=46585.90..69139.76 rows=400068 width=36) (actual time=71982.222..71986.047 rows=100 loops=1)
            Hash Cond: (revenue.total_revenue = revenue_1.total_revenue)
            -> CTE Scan on revenue  (cost=0.00..16002.65 rows=800137 width=36) (actual time=71935.891..71937.397 rows=10000 loops=1)
            -> Hash  (cost=46584.65..46584.65 rows=100 width=32) (actual time=40.300..46.304 rows=100 loops=1)
                Buckets: 1024 Batches: 1 Memory Usage: 1kB
                -> Limit  (cost=46583.40..46583.65 rows=100 width=32) (actual time=46.209..46.239 rows=100 loops=1)
                    -> Sort  (cost=46583.40..48583.74 rows=800137 width=32) (actual time=46.204..46.220 rows=100 loops=1)
                        Sort Key: revenue_1.total_revenue DESC
                        Sort Method: top-N heapsort Memory: 32kB
                        -> CTE Scan on revenue revenue_1  (cost=0.00..16002.74 rows=800137 width=32) (actual time=0.003..42.810 rows=10000 loops=1)

Planning Time: 2.036 ms
 JIT:
  Functions: 88
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 21.058 ms, Inlining 467.788 ms, Optimization 571.691 ms, Emission 314.169 ms, Total 1374.706 ms
 Execution Time: 72568.990 ms
 (47 rows)

```

Figura 18: EXPLAIN ANALYZE da interrogação A2 com a utilização do índice

4.2.2 Vistas materializadas

De seguida, reparamos que a operação $mod((s_w_id * s_i_id), 10000)$ gastava muito tempo ao ser realizada para cada linha e que, fora isso, eram apenas usadas duas colunas da tabela como condições de união. Antes de mais, decidimos alterar a cláusula *group by* ao não repetir o cálculo do *mod* e a substituí-lo pela variável gerada *supplier_no*, o que ajudou no tempo de execução, que desceu para 63864.672 ms.

```

-----  

QUERY PLAN  

-----  

Merge Join  (cost=5729967.65..5737488.98 rows=400068 width=103) (actual time=63605.182..63848.032 rows=99 loops=1)
  Merge Cond: (supplier.su_suppkey = revenue.supplier_no)
    CTE revenue
      -> Finalize GroupAggregate  (cost=5399944.38..5612660.51 rows=800137 width=36) (actual time=63246.098..63525.958 rows=10000 loops=1)
          Group Key: (mod(stock.s_w_id * stock.s_i_id), 10000)
          -> Gather Merge  (cost=5399944.38..5586656.05 rows=1600274 width=36) (actual time=63246.061..63506.713 rows=30000 loops=1)
              Workers Planned: 2
              Workers Launched: 2
              -> Sort  (cost=5398944.36..5400944.70 rows=800137 width=36) (actual time=63178.800..63180.885 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=4859407.06..5298609.38 rows=800137 width=36) (actual time=62359.926..63173.448 rows=10000 loops=3)
                      Group Key: mod((stock.s_w_id * stock.s_i_id), 10000)
                      Planned Partitions: 64 Batches: 65 Memory Usage: 4497kB Disk Usage: 55312kB
                      Worker 0: Batches: 65 Memory Usage: 4497kB Disk Usage: 55320kB
                      Worker 1: Batches: 65 Memory Usage: 4497kB Disk Usage: 56328kB
                      -> Parallel Hash Join  (cost=515579.57..207095.57 rows=43540472 width=9) (actual time=37587.385..50724.396 rows=14898327 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..1047354.12 rows=18499880 width=13) (actual time=2.772..15683.548 rows=14898327 loops=3)
                              Filter: (ol_delivery_d >= '2022-01-01 00:00:00'::timestamp without time zone)
                              Rows Removed by Filter: 3661744
                          -> Parallel Hash  (cost=452547.03..452547.03 rows=3333903 width=8) (actual time=13899.080..13899.082 rows=2666667 loops=3)
                              Buckets: 131072 Batches: 128 Memory Usage: 3520kB
                              -> Parallel Seq Scan on stock  (cost=0.00..452547.03 rows=3333903 width=8) (actual time=259.089..12766.904 rows=2666667 loops=3)
                                  -> Parallel Seq Scan on supplier  (cost=0.00..495.43 rows=10000 width=71) (actual time=0.026..3.397 rows=9921 loops=1)
                                  -> Materialize  (cost=17306.86..119307.20 rows=400068 width=36) (actual time=63297.667..63297.676 rows=100 loops=1)
                                      Sort Key: revenue.supplier_no
                                      Sort Method: quicksort Memory: 29kB
                                      -> Hash Semi Join  (cost=46585.96..69139.76 rows=400068 width=36) (actual time=63293.963..63297.623 rows=100 loops=1)
                                          Hash Cond: (revenue.total_revenue = revenue_1.total_revenue)
                                          -> CTE Scan on revenue  (cost=0.00..16002.74 rows=800137 width=36) (actual time=63246.102..63247.527 rows=10000 loops=1)
                                          -> Hash  (cost=46584.65..46584.65 rows=100 width=32) (actual time=47.828..47.831 rows=100 loops=1)
                                              Buckets: 1024 Batches: 1 Memory Usage: 13kB
                                              -> Limit  (cost=46583.40..46583.40 rows=100 width=32) (actual time=47.751..47.774 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..16002.74 rows=800137 width=32) (actual time=0.003..44.466 rows=10000 loops=1)
Planning Time: 1.522 ms
JIT:
  Functions: 88
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 16.745 ms, Inlining 428.973 ms, Optimization 575.204 ms, Emission 338.128 ms, Total 1359.050 ms
Execution Time: 63864.672 ms
(47 rows)

```

Figura 19: EXPLAIN ANALYZE da interrogação A2 com a alteração da cláusula *group by*

Reparamos ainda que a coluna mais alterada era a *s_quantity*, que representa a quantidade de *stock* disponível de cada item, em cada *warehouse*. No entanto, esta coluna nunca é utilizada na interrogação, por isso decidimos colocar uma vista materializada nas colunas *s_i_id* e *s_w_id* que são, à partida, imutáveis (sendo que representam os ids do item e do *warehouse*), e outra no cálculo $\text{mod}((s_w_id * s_i_id), 10000)$. Como os dados materializados são estáticos, esta vista não terá de ser atualizada.

Como tal, encontram-se abaixo as vistas materializadas utilizadas e o novo código adaptado às mesmas.

```

create materialized view a2_aux as
select s_i_id, s_w_id,
(mod((s_w_id * s_i_id),10000)) as supplier_no
from stock;

WITH revenue (supplier_no, total_revenue) AS ( SELECT supplier_no,
sum(ol_amount) AS total_revenue
FROM order_line, a2_aux
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;

```

Correndo agora o EXPLAIN ANALYZE com esta nova interrogação, obtemos o resultado presente na imagem abaixo, onde vemos que o tempo de execução baixou para 58911.011 ms:

```

----- QUERY PLAN -----
Sort  (cost=3729758.81..3729770.23 rows=4569 width=103) (actual time=58701.395..58901.505 rows=99 loops=1)
  Sort Key: supplier.su_suppkey
  Sort Method: quicksort Memory: 38kB
  CTE revenue
    -> Finalize GroupAggregate (cost=3725795.51..3728179.16 rows=9138 width=36) (actual time=58336.092..58577.188 rows=10000 loops=1)
      Group Key: a2_aux.supplier_no
      -> Gather Merge (cost=3725795.51..3727927.86 rows=18276 width=36) (actual time=58335.961..58557.971 rows=30000 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Sort  (cost=3724795.49..3724818.34 rows=9138 width=36) (actual time=58273.009..58275.127 rows=10000 loops=3)
          Sort Key: a2_aux.supplier_no
          Sort Method: quicksort Memory: 1791kB
          Worker 0: Sort Method: quicksort Memory: 1791kB
          Worker 1: Sort Method: quicksort Memory: 1791kB
          -> Partial HashAggregate (cost=3724080.09..3724194.32 rows=9138 width=36) (actual time=57849.731..58260.732 rows=10000 loops=3)
            Group Key: a2_aux.supplier_no
            Batches: 5 Memory Usage: 4529kB Disk Usage: 15808kB
            Worker 0: Batches: 5 Memory Usage: 4529kB Disk Usage: 17312kB
            Worker 1: Batches: 5 Memory Usage: 4529kB Disk Usage: 16440kB
            -> Parallel Hash Join  (cost=142855.80..36083708.00 rows=24074418 width=9) (actual time=25079.224..42888.104 rows=14898327 loops=3)
              Hash Cond: ((order_line.ol_i_id = a2_aux.s_i_id) AND (order_line.ol_supply_w_id = a2_aux.s_w_id))
              -> Parallel Seq Scan on order_line  (cost=0.00..1047354.12 rows=18409800 width=13) (actual time=0.109..13931.123 rows=14898327 loops=3)
                Filter: (ol_delivery_d >= '2022-01-01 00:00:00'::timestamp without time zone)
                Rows Removed by Filter: 3661744
              -> Parallel Hash  (cost=76577.92..76577.92 rows=3333392 width=12) (actual time=2736.910..2736.913 rows=2666667 loops=3)
                Buckets: 131072 Batches: 128 Memory Usage: 4000kB
                -> Parallel Seq Scan on a2_aux  (cost=0.00..76577.92 rows=3333392 width=12) (actual time=311.854..1126.357 rows=2666667 loops=3)
                  Buckets: 131072 Batches: 128 Memory Usage: 4000kB
                  -> Parallel Seq Scan on a2_aux  (cost=0.00..76577.92 rows=3333392 width=12) (actual time=311.854..1126.357 rows=2666667 loops=3)
                    Buckets: 131072 Batches: 128 Memory Usage: 4000kB
                    -> Parallel Hash Join  (cost=981.51..1301.91 rows=4569 width=103) (actual time=58697.023..58701.321 rows=99 loops=1)
                      Hash Cond: (revenue.supplier_no = supplier.su_suppkey)
                      -> Hash Semi Join  (cost=534.51..792.09 rows=4569 width=36) (actual time=58383.831..58387.758 rows=100 loops=1)
                        Hash Cond: (revenue.total_revenue = revenue_1.total_revenue)
                        -> CTE Scan on revenue  (cost=0.00..182.76 rows=9138 width=36) (actual time=58336.099..58337.672 rows=10000 loops=1)
                        -> Hash  (cost=533.26..533.26 rows=100 width=32) (actual time=47.622..47.626 rows=100 loops=1)
                          Buckets: 1024 Batches: 1 Memory Usage: 13kB
                          -> Limit  (cost=532.01..532.26 rows=100 width=32) (actual time=47.493..47.516 rows=100 loops=1)
                            -> Sort  (cost=532.01..554.85 rows=9138 width=32) (actual time=47.481..47.490 rows=100 loops=1)
                              Sort Key: revenue_1.total_revenue DESC
                              Sort Method: top-N heapsort Memory: 32kB
                              -> CTE Scan on revenue revenue_1  (cost=0.00..182.76 rows=9138 width=32) (actual time=0.003..44.201 rows=10000 loops=1)
                            -> Hash  (cost=322.00..322.00 rows=10000 width=71) (actual time=313.086..313.088 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=308.454..310.790 rows=10000 loops=1)
Planning Time: 0.732 ms
 JIT:
  Functions: 89
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 35.115 ms, Inlining 441.945 ms, Optimization 626.230 ms, Emission 485.732 ms, Total 1589.023 ms
Execution Time: 58911.011 ms
(48 rows)

```

Figura 20: EXPLAIN ANALYZE da interrogação A2 com a vista materializada

Mais tarde, decidimos incluir na vista materializada a união das colunas das tabelas *stock* e *order_line* ($l_i_id = s_i_id$ and $ol_supply_w_id = s_w_id$), para, dessa forma, a interrogação aceder aos resultados já filtrados sem precisar de os filtrar. Para além disso, decidimos ordenar a vista materializada de maneira a que os blocos de memória acedidos sejam contíguos.

O código fica da seguinte forma:

```

create materialized view a2_aux 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
order by ol_delivery_d desc;

    WITH revenue (supplier_no, total_revenue) AS ( SELECT supplier_no,
sum(ol_amount) AS total_revenue
FROM a2_aux
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;

```

Correndo agora o EXPLAIN ANALYZE com esta nova interrogação, obtemos o resultado presente na imagem abaixo, onde vemos que o tempo de execução baixou para 17145.151 ms:

```

QUERY PLAN
-----
Sort (cost=742568.55..742579.15 rows=4240 width=103) (actual time=17136.632..17138.267 rows=99 loops=1)
  Sort Key: supplier.su_suppkey
  Sort Method: quicksort Memory: 38kB
  CTE revenue
    -> Finalize GroupAggregate (cost=738860.90..741072.64 rows=8479 width=36) (actual time=16830.325..16868.577 rows=10000 loops=1)
      Group Key: a2_aux.supplier_no
      -> Gather Merge (cost=738860.90..740839.47 rows=16958 width=36) (actual time=16830.186..16847.393 rows=30000 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Sort (cost=737860.87..737882.07 rows=8479 width=36) (actual time=16759.528..16761.527 rows=10000 loops=3)
          Sort Key: a2_aux.supplier_no
          Sort Method: quicksort Memory: 1791kB
          Worker 0: Sort Method: quicksort Memory: 1791kB
          Worker 1: Sort Method: quicksort Memory: 1791kB
          -> Partial HashAggregate (cost=737201.65..737307.63 rows=8479 width=36) (actual time=16364.902..16753.335 rows=10000 loops=3)
            Group Key: a2_aux.supplier_no
            Batches: 5 Memory Usage: 4273kB Disk Usage: 17256kB
            Worker 0: Batches: 5 Memory Usage: 4273kB Disk Usage: 16224kB
            Worker 1: Batches: 5 Memory Usage: 4273kB Disk Usage: 15608kB
            -> Parallel Seq Scan on a2_aux (cost=0.00..644653.90 rows=18509550 width=9) (actual time=225.606..5302.275 rows=14898327 loops=3)
              Filter: (ol_delivery_d >= '2022-01-01 00:00:00'::timestamp without time zone)
              Rows Removed by Filter: 3661744
    -> Hash Join (cost=943.14..1240.45 rows=4240 width=103) (actual time=17134.305..17136.539 rows=99 loops=1)
      Hash Cond: (revenue.supplier_no = supplier.su_suppkey)
      -> Hash Semi Join (cost=496.14..735.15 rows=4240 width=36) (actual time=16873.680..16875.880 rows=100 loops=1)
        Hash Cond: (revenue.total_revenue = revenue_1.total_revenue)
        -> CTE Scan on revenue (cost=0.00..169.58 rows=8479 width=36) (actual time=16830.330..16831.123 rows=10000 loops=1)
        -> Hash (cost=494.89..494.89 rows=100 width=32) (actual time=43.300..43.302 rows=100 loops=1)
          Buckets: 1024 Batches: 1 Memory Usage: 13kB
          -> Limit (cost=493.64..493.89 rows=100 width=32) (actual time=43.198..43.212 rows=100 loops=1)
            -> Sort (cost=493.64..514.84 rows=8479 width=32) (actual time=43.193..43.199 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..169.58 rows=8479 width=32) (actual time=0.003..39.645 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=249.461..258.188 rows=10000 loops=1)
Planning Time: 1.965 ms
JIT:
  Functions: 62
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 10.566 ms, Inlining 431.641 ms, Optimization 571.436 ms, Emission 346.942 ms, Total 1360.584 ms
Execution Time: 17145.151 ms
(43 rows)

```

Figura 21: EXPLAIN ANALYZE da interrogação A2 com a vista materializada atualizada

4.2.3 Novo índice

Posteriormente, decidimos criar um último índice na coluna `ol_delivery_d` da vista materializada, de forma a tornar mais rápida a verificação da condição sobre a mesma.

```
create index a2.ol_delivery_d on a2_aux(ol_delivery_d);
```

No entanto, o programa optou por não usar o índice, tal como se pode ver na figura abaixo:

```
QUERY PLAN
Sort (cost=741941.60..741952.14 rows=4213 width=103) (actual time=16164.946..16171.799 rows=99 loops=1)
  Sort Key: supplier.su_suppkey
  Sort Method: quicksort Memory: 38kB
CTE revenue
-> Finalize GroupAggregate (cost=738254.56..740452.48 rows=8426 width=36) (actual time=15894.184..15930.575 rows=10000 loops=1)
  Group Key: a2_aux.supplier_no
-> Gather Merge (cost=738254.56..740220.76 rows=16852 width=36) (actual time=15894.149..15909.569 rows=30000 loops=1)
  Workers Planned: 2
  Workers Launched: 2
-> Sort (cost=737254.53..737275.60 rows=8426 width=36) (actual time=15848.927..15850.817 rows=10000 loops=3)
  Sort Key: a2_aux.supplier_no
  Sort Method: quicksort Memory: 1791kB
  Worker 0: Sort Method: quicksort Memory: 1791kB
  Worker 1: Sort Method: quicksort Memory: 1791kB
-> Partial HashAggregate (cost=736599.81..736705.13 rows=8426 width=36) (actual time=15435.265..15837.151 rows=10000 loops=3)
  Group Key: a2_aux.supplier_no
  Batches: 5 Memory Usage: 4273kB Disk Usage: 17280kB
  Worker 0: Batches: 5 Memory Usage: 4273kB Disk Usage: 16336kB
  Worker 1: Batches: 5 Memory Usage: 4273kB Disk Usage: 16328kB
-> Parallel Seq Scan on a2_aux (cost=0.00..644653.90 rows=18389182 width=9) (actual time=207.592..4982.657 rows=14898327 loops=3)
  Filter: (ol_delivery_d > '2022-01-01 00:00:00'::timestamp without time zone)
  Rows Removed by Filter: 3661744
-> Hash Join (cost=940.06..1235.49 rows=4213 width=103) (actual time=16161.122..16164.904 rows=99 loops=1)
  Hash Cond: (revenue.supplier_no = supplier.su_suppkey)
-> Hash Semi Join (cost=493.66..738.56 rows=4213 width=36) (actual time=15930.266..15933.978 rows=100 loops=1)
  Hash Cond: (revenue.total_revenue = revenue_1.total_revenue)
-> CTE Scan on revenue (cost=0.00..168.52 rows=8426 width=36) (actual time=15894.187..15895.640 rows=10000 loops=1)
-> Hash (cost=491.81..491.81 rows=100 width=32) (actual time=36.041..36.043 rows=100 loops=1)
  Buckets: 1024 Batches: 1 Memory Usage: 13kB
-> Limit (cost=490.56..490.81 rows=100 width=32) (actual time=35.960..35.984 rows=100 loops=1)
  -> Sort (cost=498.56..511.62 rows=8426 width=32) (actual time=35.955..35.964 rows=100 loops=1)
    Sort Key: revenue_1.total_revenue DESC
    Sort Method: top-N heapsort Memory: 32kB
    -> CTE Scan on revenue revenue_1 (cost=0.00..168.52 rows=8426 width=32) (actual time=0.003..32.551 rows=10000 loops=1)
-> Hash (cost=322.00..322.00 rows=10000 width=71) (actual time=230.784..230.785 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=227.087..228.769 rows=10000 loops=1)
  Execution Time: 16177.787 ms
(43 rows)

Planning Time: 0.312 ms
JIT:
  Functions: 62
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 25.784 ms, Inlining 363.916 ms, Optimization 546.859 ms, Emission 330.716 ms, Total 1267.276 ms
Execution Time: 16177.787 ms
```

Figura 22: EXPLAIN ANALYZE da interrogação A2 com a tentativa de novo índice

node type	count	sum of times	% of query
CTE Scan	2	0.000 ms	0.0 %
Finalize GroupAggregate	1	21.006 ms	0.1 %
Gather Merge	1	58.752 ms	0.4 %
Hash	2	2.075 ms	0.0 %
Hash Join	1	0.141 ms	0.0 %
Hash Semi Join	1	2.295 ms	0.0 %
Limit	1	0.020 ms	0.0 %
Parallel Seq Scan	1	4,982.657 ms	30.8 %
Partial HashAggregate	1	10,854.494 ms	67.1 %
Seq Scan	1	228.769 ms	1.4 %
Sort	3	23.974 ms	0.1 %

Figura 23: EXPLAIN ANALYZE da interrogação A2 com a tentativa de novo índice

Tal como não utiliza este último índice, o programa deixou também de utilizar o índice anteriormente criado, aquando da utilização da nova vista materializada. Reparamos que o maior gargalo de desempenho neste ponto encontra-se no *HashAggregate*. Apesar de termos tentado que o programa utilizasse os índices, através da desativação do *enable_seqscan* e da diminuição da *work_mem*, não tivemos sucesso e o programa continuou sem escolher utilizar os índices.

4.2.4 Paralelismo

Para esta interrogação, o grupo tentou novamente verificar se aumentando o valor do parâmetro '*max_parallel_workers_per_gather*' para 4 resultaria alguma melhoria, mas acabou por se demonstrar inútil, visto que o número de trabalhadores permaneceu a ser 2, como se pode ver abaixo:

```
QUERY PLAN
Sort  (cost=741941.60..741952.14 rows=4213 width=103) (actual time=16470.319..16476.815 rows=99 loops=1)
  Sort Key: supplier.su.supkey
  Sort Method: quicksort Memory: 38kB
  CTE revenue
    -> Finalize GroupAggregate  (cost=738254.56..740452.48 rows=8426 width=36) (actual time=16193.109..16232.716 rows=10000 loops=1)
        Group Key: a2_aux.supplier_no
        -> Gather Merge  (cost=738254.56..740220.76 rows=16852 width=36) (actual time=16193.054..16211.421 rows=30000 loops=1)
            Workers Planned: 2
            Workers Launched: 2
              -> Sort  (cost=737254.53..737275.60 rows=8426 width=36) (actual time=16147.247..16149.194 rows=10000 loops=3)
                  Sort Key: a2_aux.supplier_no
                  Sort Method: quicksort Memory: 1791kB
                  Worker 0: Sort Method: quicksort Memory: 1791kB
                  Worker 1: Sort Method: quicksort Memory: 1791kB
                  -> Partial HashAggregate  (cost=736599.81..736705.13 rows=8426 width=36) (actual time=15733.276..16138.713 rows=10000 loops=3)
                      Group Key: a2_aux.supplier_no
                      Batches: 5 Memory Usage: 4273kB Disk Usage: 17224kB
                      Worker 0: Batches: 5 Memory Usage: 4273kB Disk Usage: 15608kB
                      Worker 1: Batches: 5 Memory Usage: 4273kB Disk Usage: 17192kB
                      -> Parallel Seq Scan on a2_aux  (cost=0.00..644653.90 rows=18389182 width=9) (actual time=187.866..5271.875 rows=14898327 loops=3)
                          Filter: (ol_delivery_d >= '2022-01-01 00:00:00'::timestamp without time zone)
                          Rows Removed by Filter: 3661744
-> Hash Join  (cost=940.06..1235.49 rows=4213 width=103) (actual time=16466.209..16470.251 rows=99 loops=1)
  Hash Cond: (revenue.supplier_no = supplier.su.supkey)
    -> Hash Semi Join  (cost=493.06..730.56 rows=4213 width=36) (actual time=16233.275..16237.253 rows=100 loops=1)
        Hash Cond: (revenue.total_revenue = revenue_1.total_revenue)
        -> CTE Scan on revenue  (cost=0.00..168.52 rows=8426 width=36) (actual time=16193.112..16194.650 rows=10000 loops=1)
        -> Hash  (cost=491.81..491.81 rows=100 width=32) (actual time=40.140..40.143 rows=100 loops=1)
            Buckets: 1024 Batches: 1 Memory Usage: 13kB
            -> Limit  (cost=490.56..490.81 rows=100 width=32) (actual time=40.065..40.088 rows=100 loops=1)
                -> Sort  (cost=490.56..511.62 rows=8426 width=32) (actual time=40.068..40.071 rows=100 loops=1)
                    Sort Key: revenue_1.total_revenue DESC
                    Sort Method: top-N heapsort Memory: 32kB
                    -> CTE Scan on revenue revenue_1  (cost=0.00..168.52 rows=8426 width=32) (actual time=0.004..36.569 rows=10000 loops=1)
-> Hash  (cost=322.00..322.00 rows=10000 width=71) (actual time=232.859..232.860 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=228.882..230.664 rows=10000 loops=1)
Planning Time: 0.396 ms
 JIT:
  Functions: 62
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 18.046 ms, Inlining 338.624 ms, Optimization 558.023 ms, Emission 333.579 ms, Total 1248.273 ms
Execution Time: 16483.382 ms
(43 rows)
```

Figura 24: EXPLAIN ANALYZE da interrogação A2 com paralelismo

Para terminar, todas as otimizações feitas ao programa fizeram com que o tempo de execução da interrogação analítica baixasse de 92068.149ms para 16177.787 ms. Idealmente, num trabalho futuro iríamos tentar trabalhar mais a fundo o problema dos índices, de modo a diminuir ainda mais o tempo de execução.

4.3 Interrogacão A3

```

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;

```

Passando agora à otimização do desempenho da interrogação A3, começamos por perceber o que se pretendia realizar com a *query*.

Como se pode ver, existem dois *select* aninhados. Comecemos então por analisar o interno.

```

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

```

Nesta interrogação (interrogação interna), está a ser calculada a quantidade média vendida de cada item presente na tabela '*order_line*', cujo campo '*i_data*' termine em 'b'.

Com esta informação passa a ser mais fácil entender a interrogação A3, que calcula a média anual da faturaçao (visto que na base de dados só existem 2 anos distintos na coluna '*ol_delivery_d*' – 1899 e 2022) onde, para essa média, apenas são incluídos os registos cuja quantidade vendida é superior à quantidade média vendida do produto em questão.

Posto isto, começamos por correr o comando EXPLAIN ANALYZE com a interrogação no sentido de analisar o tempo de execução assim como o plano de execução da *query*.

```

-----  

QUERY PLAN  

Aggregate (cost=2566631.06..2566631.06 rows=1 width=32) (actual time=25562.819..25563.353 rows=1 loops=1)
  > Hash Join (cost=1084368.07..2544690.87 rows=8776070 width=5) (actual time=12373.009..25436.579 rows=844305 loops=1)
    Hash Cond: (order_line.ol_i_id = item.i_id)
    Join Filter: (order_line.quantity)::numeric < (avg(order_line.ol_quantity))
    Rows Removed by Join Filter: 2547
    > Seq Scan on order_line (cost=0.00..1314155.16 rows=55680216 width=13) (actual time=0.021..6811.332 rows=55680214 loops=1)
    > Hash (cost=1084118.07..1084118.07 rows=20000 width=36) (actual time=12373.804..12373.333 rows=2547 loops=1)
      Buckets: 32768 Batches: 1 Memory Usage: 377KB
      > Finalize GroupAggregate (cost=1078801.08..1083918.07 rows=20000 width=36) (actual time=12359.119..12372.300 rows=2547 loops=1)
        Group Key: item.i_id
        > Gather Merge (cost=1078801.08..1083468.07 rows=40000 width=36) (actual time=12359.068..12369.148 rows=7641 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          > Sort (cost=1077801.05..1077851.05 rows=20000 width=36) (actual time=12322.070..12322.495 rows=2547 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=1076172.28..1076372.28 rows=20000 width=36) (actual time=12318.961..12320.467 rows=2547 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..1052972.19 rows=4640018 width=8) (actual time=800.518..11887.645 rows=448127 loops=3)
                Hash Cond: (order_line.ol_i_id = item.i_id)
                > Parallel Seq Scan on order_line order_line_1 (cost=0.00..989353.90 rows=23200000 width=8) (actual time=0.282..5855.604 rows=18560071 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=627.819..695.781 rows=849 loops=3)
                    Filter: (TRIM(BOTH FROM i_data) ~~ 'kb'::text)
                    Rows Removed by Filter: 32484
Planning Time: 0.446 ms
JIT:
  Functions: 70
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 10.661 ms, Inlining 302.415 ms, Optimization 1118.508 ms, Emission 452.593 ms, Total 1884.237 ms
Execution Time: 25567.457 ms
(37 rows)

```

Figura 25: EXPLAIN ANALYZE da interrogação A3

Analizando o *output* do comando vemos que o tempo de execução desta interrogação é 25567.457ms.

4.3.1 Índices

Como primeira iteração para melhorar o desempenho da interrogação, verificamos se existiam índices que pudéssemos criar. Analisando a interrogação novamente, vimos que as colunas que poderiam beneficiar da utilização de índices são '*i_data*' e '*i_id*' da tabela 'item', assim como a coluna '*ol_i_id*' da tabela 'order_line'. Visto que já existiam índices nas colunas '*i_id*' e '*ol_i_id*', só nos restava criar um índice para a coluna '*i_data*' da tabela 'item'.

A construção deste índice não foi direta, e após discutir com o professor, percebemos que teríamos de fazer um *reverse* do '*trim(i_data)*' para que realizasse a pesquisa pelo prefixo em vez do sufixo. A função '*trim*' também tem de ser incluída no índice, caso contrário poderiam haver registos cuja última letra fosse 'b', mas devido a existir um espaço, não estariam bem ordenados. Adicionalmente, o operador LIKE só usa índices que tenham sido criados com a opção '*text_pattern_ops*'.

Para que o *PostgreSQL* recorresse aos índices tivemos de desligar o parâmetro '*enable_seqscan*'.

O índice criado e a adaptação da interrogação encontram-se a seguir:

```

create index a3_item_aux on item(reverse(trim(i_data))
text_pattern_ops);

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 REVERSE(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;

```

Com a adição do índice e ao desligar o *scan* sequencial, o resultado que obtivemos do EXPLAIN ANALYZE foi o seguinte:

```

QUERY PLAN
Aggregate  (cost=2231319.04 .. 2231319.06 rows=1 width=32) (actual time=20057.415..20057.530 rows=1 loops=1)
  -> Hash Join  (cost=770120.32 .. 2230759.40 rows=223856 width=5) (actual time=7151.972..19953.517 rows=844305 loops=1)
    Hash Cond: (order_line.ol_i_id = item.i_id)
    Join Filter: ((order_line.ol_quantity)::numeric < (avg(order_line_1.ol_quantity)))
    Rows Removed by Join Filter: 500076
    -> Seq Scan on order_line  (cost=0.00 .. 1314405.56 rows=55705256 width=13) (actual time=0.017..6439.598 rows=55680214 loops=1)
    -> Hash  (cost=770114.07 .. 770114.07 rows=500 width=36) (actual time=7151.845..7151.956 rows=2547 loops=1)
      Buckets: 4096 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 153kB
      -> Finalize GroupAggregate  (cost=1019.11..770109.07 rows=500 width=36) (actual time=788.385..7149.032 rows=2547 loops=1)
        Group Key: item.i_id
        -> Gather Merge  (cost=1019.11..770097.82 rows=1000 width=36) (actual time=784.836..7141.098 rows=2547 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          -> Partial GroupAggregate  (cost=19.08..768982.37 rows=500 width=36) (actual time=591.752..6105.467 rows=849 loops=3)
            Group Key: item.i_id
            -> Nested Loop  (cost=19.08 .. 768397.11 rows=116052 width=8) (actual time=585.656..5973.755 rows=448127 loops=3)
              -> Parallel Index Scan using pk_item on item  (cost=0.29 .. 3979.63 rows=208 width=4) (actual time=585.487..684.700 rows=849 loops=3)
                Filter: (reverse(TRIM(BOTH FROM i_data)) ~~ 'b%'::text)
                Rows Removed by Filter: 32484
              -> Bitmap Heap Scan on order_line order_line_1  (cost=18.79..3661.65 rows=1343 width=8) (actual time=0.276..6.056 rows=528 loops=2547)
                Recheck Cond: (ol_i_id = item.i_id)
                Heap Blocks: exact=223414
                -> Bitmap Index Scan on ix_order_line  (cost=0.00..18.46 rows=1343 width=0) (actual time=0.136..0.136 rows=528 loops=2547)
                  Index Cond: (ol_i_id = item.i_id)

Planning Time: 0.832 ms
JIT:
  Functions: 49
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 9.221 ms, Inlining 311.443 ms, Optimization 968.523 ms, Emission 475.003 ms, Total 1764.191 ms
Execution Time: 20061.930 ms
(30 rows)

```

Figura 26: EXPLAIN ANALYZE da interrogação A3 com índices

Podemos observar que com a utilização de índices, o tempo de execução baixou para 20061.930 ms.

4.3.2 Vista materializada

Em seguida, o grupo decidiu passar o resultado da interrogação interna para uma vista materializada, no sentido desta não ter de estar sempre a ser calculada. Procedeu-se então à criação dessa vista:

```

create materialized view a3_aux 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);

```

Em seguida, a interrogação teve de ser transformada de forma a conter a vista, da seguinte forma:

```
select sum(ol_amount) / 2.0 as avg_yearly
from order_line, a3_aux t
where ol_i_id = t.i_id and ol_quantity < t.a;
```

Voltando a executar o EXPLAIN ANALYZE, chegamos ao seguinte resultado, onde obtivemos o tempo de execução 11366.789 ms, melhor do que a utilização de apenas índices:

```
Finalize Aggregate (cost=1100228.25..1100228.26 rows=1 width=32) (actual time=11359.639..11364.773 rows=1 loops=1)
  -> Gather (cost=1100228.03..1100228.24 rows=2 width=32) (actual time=11359.619..11364.755 rows=3 loops=1)
    Workers Planned: 2
    Workers Launched: 2
      -> Partial Aggregate (cost=1099228.03..1099228.04 rows=1 width=32) (actual time=11325.928..11325.931 rows=1 loops=3)
        -> Hash Join (cost=74.31..1098012.93 rows=486037 width=5) (actual time=441.850..11211.776 rows=281435 loops=3)
          Hash Cond: (order_line.ol_i_id = t.i_id)
          Join Filter: ((order_line.ol_quantity)::numeric < t.a)
          Rows Removed by Join Filter: 166692
          -> Parallel Seq Scan on order_line (cost=0.00..989144.98 rows=23179198 width=13) (actual time=0.071..5144.707 rows=18560071 loops=3)
          -> Hash (cost=42.47..42.47 rows=2547 width=16) (actual time=441.582..441.583 rows=2547 loops=3)
            Buckets: 4096 Batches: 1 Memory Usage: 153kB
              -> Seq Scan on a3_aux t (cost=0.00..42.47 rows=2547 width=16) (actual time=0.013..0.379 rows=2547 loops=3)
Planning Time: 4.400 ms
JIT:
  Functions: 44
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 7.304 ms, Inlining 248.346 ms, Optimization 690.069 ms, Emission 382.413 ms, Total 1328.132 ms
Execution Time: 11366.789 ms
(19 rows)
```

Figura 27: EXPLAIN ANALYZE da interrogação A3 com vista materializada

node type	count	sum of times	% of query
Finalize Aggregate	1	0.034 ms	0.0 %
Gather	1	37.820 ms	0.3 %
Hash	1	455.753 ms	4.0 %
Hash Join	1	5,573.451 ms	49.1 %
Parallel Seq Scan	1	5,160.517 ms	45.5 %
Partial Aggregate	1	115.155 ms	1.0 %
Seq Scan	1	0.348 ms	0.0 %

Figura 28: EXPLAIN ANALYZE da interrogação A3 com vista materializada

Depois de correr o EXPLAIN, reparamos que o índice da tabela 'order_line' deixou de ser utilizado. Analisando o resultado, conseguimos perceber que o maior gargalo de desempenho neste ponto se encontrava no *Hash Join* devido a este realizar um *scan* sequencial sobre a tabela *order_line*. O grupo fez várias tentativas para tentar resolver este problema, acrescentando novos índices e alterando alguns parâmetros como o *work_mem*, no ficheiro de configuração. No entanto, vimo-nos incapazes de o resolver.

4.3.3 Paralelismo

Para esta interrogação o grupo tentou novamente verificar se aumentando o valor do parâmetro '*max_parallel_workers_per_gather*' para 4 resultaria alguma melhoria, mas acabou por demonstrar inútil, visto que o número de trabalhadores permaneceu 2.

```

        QUERY PLAN
Finalize Aggregate (cost=1100228.25..1100228.26 rows=1 width=32) (actual time=11416.074..11416.152 rows=1 loops=1)
  -> Gather (cost=1100228.03..1100228.24 rows=2 width=32) (actual time=11416.055..11416.136 rows=3 loops=1)
    Workers Planned: 2
    Workers Launched: 2
      -> Partial Aggregate (cost=1099228.03..1099228.04 rows=1 width=32) (actual time=11377.182..11377.185 rows=1 loops=3)
        -> Hash Join (cost=74.31..1098012.93 rows=486037 width=5) (actual time=501.713..11278.229 rows=281435 loops=3)
          Hash Cond: (order_line.ol_i_id = t.i_id)
          Join Filter: ((order_line.ol_quantity)::numeric < t.a)
          Rows Removed by Join Filter: 166692
        -> Parallel Seq Scan on order_line (cost=0.00..989144.98 rows=23179198 width=13) (actual time=0.069..5183.400 rows=18560071 loops=3)
          Buckets: 4096 Batches: 1 Memory Usage: 153kB
        -> Hash (cost=42.47..42.47 rows=2547 width=16) (actual time=501.432..501.433 rows=2547 loops=3)
          -> Seq Scan on a3_aux t (cost=0.00..42.47 rows=2547 width=16) (actual time=0.013..0.371 rows=2547 loops=3)
Planning Time: 0.325 ms
 JIT:
  Functions: 44
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 7.301 ms, Inlining 285.072 ms, Optimization 768.671 ms, Emission 445.715 ms, Total 1506.759 ms
Execution Time: 11418.372 ms
(19 rows)

```

Figura 29: EXPLAIN ANALYZE da interrogação A3 com paralelismo

Com isto, no final da otimização desta interrogação obtivemos um tempo de execução de 11366.789 ms. Idealmente, num trabalho futuro tentaríamos voltar a trabalhar nesta *query* de forma a procurar como tratar o problema que foi descrito acima.

4.4 Interrogação A4

A última interrogação analítica fornecida no enunciado é a seguinte:

```

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 12.ol_delivery_d > 11.ol_delivery_d)
  AND su_nationkey = n_nationkey
  AND n_name = 'GERMANY'
GROUP BY su_name
ORDER BY numwait DESC, su_name;

```

De forma a ser possível analisar a *query*, executamos com EXPLAIN ANALYZE e o resultado obtido foi o seguinte:

```

QUERY PLAN
Sort  (cost=1041935.20..1041906.20 rows=10000 width=34) (actual time=34888.579..35281.599 rows=390 loops=1)
  Sort Key: (count())
  Sort Method: quicksort  Memory: 55KB
->  Finalize GroupAggregate  (cost=1038576.95..1041270.88 rows=10000 width=34) (actual time=34793.533..35201.804 rows=390 loops=1)
      Group Key: supplier.su.name
      ->  Gather Merge  (cost=1038576.95..1041070.88 rows=20000 width=34) (actual time=34793.170..35200.518 rows=1188 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          ->  Parallel GroupAggregate  (cost=1037576.93..1037762.30 rows=10000 width=34) (actual time=34088.870..34799.130 rows=390 loops=3)
              Group Key: supplier.su.name
              ->  Sort  (cost=1037576.93..1037605.40 rows=1191 width=20) (actual time=34679.962..34763.828 rows=15305 loops=3)
                  Sort Key: supplier.su.name
                  Sort Method: external merge  Disk: 5424KB
              Worker 1:  Sort  (cost=1037576.93..1037605.40 rows=1191 width=20) (actual time=34679.962..34763.828 rows=15305 loops=3)
                  Sort Method: external merge  Disk: 5359KB
              Worker 2:  Sort  (cost=1037576.93..1037605.40 rows=1191 width=20) (actual time=34679.962..34763.828 rows=15305 loops=3)
                  Sort Method: external merge  Disk: 5359KB
              ->  Nested Loop Anti Join  (cost=208994.15..1030809.42 rows=1191 width=20) (actual time=14590.063..33430.617 rows=15305 loops=3)
                  ->  Parallel Hash Join  (cost=208994.15..1030809.42 rows=17880 width=48) (actual time=14580.131..15450.173 rows=257514 loops=3)
                      Hash Cond: ((l1.ol_o_id = orders.o_id) AND (l1.ol_w_id = orders.o_w_id) AND (l1.ol_d_id = orders.o_d_id))
                      Join Filter: ((l1.ol_delivery_d = orders.o_entry_d))
                      Row Removed by Filter: 0
                  ->  Parallel Hash Join  (cost=182779.02..708081.78 rows=12000 width=50) (actual time=8136.014..12077.704 rows=259731 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 l1  (cost=0.00..340258.92 rows=5581492 width=24) (actual time=0.042..2030.551 rows=803734 loops=3)
                      ->  Parallel Hash Join  (cost=182484.98..182484.98 rows=19088 width=34) (actual time=1828.980..1828.994 rows=108880 loops=3)
                          Buckets: 65536  (originally 53398)  Batches: 8 (orinality 1)  Memory Usage: 1024kB
                          ->  Hash  (cost=182484.98..182484.98 rows=19088 width=34) (actual time=1828.980..1828.994 rows=108880 loops=3)
                              Hash Cond: (mod(stock.s_w_id * stock.s_i_id), 10000) = supplier.su.supkey
                              ->  Parallel Index Only Scan using pk_stock on stock  (cost=0.43..101881.89 rows=3333339 width=8) (actual time=0.071..1132.053 rows=2066067 loops=3)
                                  Heap Fetches: 0
                          ->  Hash  (cost=3.23..3.23 rows=59 width=30) (actual time=0.055..0.000 rows=390 loops=3)
                          Bucketed: 324  Batches: 1  Memory Usage: 39kB
                          ->  Hash Join  (cost=12.14..372.23 rows=59 width=30) (actual time=2.130..3.919 rows=390 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=0.010..1.454 rows=10000 loops=3)
                              ->  Hash  (cost=12.12..12.12 rows=1 width=4) (actual time=2.086..2.088 rows=1 loops=3)
                                  Bucketed: 1  Batches: 1  Memory Usage: 9kB
                              ->  Seq Scan on nation  (cost=0.00..11.12 rows=1 width=4) (actual time=2.077..2.088 rows=1 loops=3)
                                  Filter: (n.name = 'GERMANY'::varchar)
                                  Row Removed by Filter: 24
                  ->  Parallel Hash  (cost=32669.47..32669.47 rows=1067948 width=20) (actual time=152.723..2152.725 rows=886358 loops=3)
                      Buckets: 65536  (originally 53398)  Batches: 8 (orinality 1)  Memory Usage: 2024kB
                      ->  Parallel Seq Scan on orders  (cost=0.00..32669.47 rows=1067948 width=20) (actual time=1215.410..1001.988 rows=886358 loops=3)
                      ->  Index Scan using pk_order_line on order_line l1  (cost=0.20..18.00 rows=3 width=28) (actual time=0.009..0.069 rows=0 loops=77243)
                          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: 0
Planning Time: 20.413 ms
 JIT:
  Functions: 108
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 21.432 ms, Inlining 499.347 ms, Optimization 1911.617 ms, Emission 1223.922 ms, Total 3856.318 ms
Execution Time: 35247.849 ms
[51 rows]

```

Figura 30: EXPLAIN ANALYZE da interrogação A4

A partir da imagem, é possível observar que a *query* utiliza índices já existentes para otimizar a procura dos elementos da *order_line*. Isto é, o índice '*pk_order_line*', por exemplo, é utilizado automaticamente para melhorar o desempenho dos *joins* associados aos elementos *l2.ol_o_id*, *l1.ol_o_id*, *l2.ol_w_id*, *l1.ol_w_id*, *l2.ol_d_id*, *l1.ol_d_id*.

4.4.1 Índice

Como a execução da *query* foi relativamente demorada, com a adição de mais índices julgamos ser possível que esse tempo melhore.

Foi possível observar que não existem índices definidos para a tabela *supplier*, o que quer dizer que são feitos *seq scans* para percorrer os dados da tabela. Sendo que estas operações apresentam algum custo de tempo, consideramos uma forma de melhorar esse aspecto – criar indexação nos valores de *su_nationkey*, da junção das tabelas *supplier* e *nation*:

```
Create index a4_supp_index on supplier(su_nationkey);
```

Executando de novo a *query*, foi obtido o seguinte resultado:

Figura 31: EXPLAIN ANALYZE da interrogação A4 com índice

O tempo de execução não melhorou significativamente, mas o índice criado foi reconhecido pelo *PostgreSQL* como uma possível melhoria e utilizado na execução da *query*. Apesar de isso significar que foi uma boa estratégia a criação desse índice, o tempo de execução apenas melhorou perto de 1 segundo, o que não é de todo satisfatório. Assim, procedemos a tentativas de otimizar mais esta interrogação.

4.4.2 Vista materializada

Posteriormente, procedemos a tentar seguir as otimizações relacionadas com *materialized views*, na esperança de termos mais sucesso em relação à indexação.

Conseguimos notar que a operação mais custosa da *query* era o '*not exists*' do *select* aninhado. Para contornar este problema mas ainda apresentar uma solução que mantivesse a lógica da interrogação original, decidimos substituir o *select* à tabela toda de *order_line* por uma *materialized view* que já tivesse previamente calculado os valores a serem excluídos pela cláusula '*not exists*'. Esta ideia deu origem à seguinte *materialized view*:

```
Create materialized view a4_aux as (
  select ll.ol_o_id, ll.ol_w_id, ll.ol_d_id,
         ll.ol_i_id, ll.ol_delivery_d
    from order_line ll
   where 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));

```

Sendo que agora o trabalho do '*not exists*' já está todo feito na tabela '*a4_aux*', reescrevemos a *query* para ficar de acordo com as novas alterações mas fiel à original:

```

select su_name, count(*) as numwait
from supplier, a4_aux, 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 ol_delivery_d > o_entry_d
and su_nationkey = n_nationkey
and n_name = 'GERMANY'
group by su_name
order by numwait desc, su_name;

```

Finalmente, pudemos proceder à análise do novo tempo de execução:

```

-----  

QUERY PLAN  

-----  

Sort  (cost=438386.37..438411.37 rows=10000 width=34) (actual time=12534.189..12648.947 rows=396 loops=1)  

  Sort Key: (count(*)) DESC, supplier.su_name  

  Sort Method: quicksort Memory: 55kB  

->  Finalize GroupAggregate (cost=435036.62..437721.98 rows=10000 width=34) (actual time=12445.701..12648.518 rows=396 loops=1)  

    Group Key: supplier.su_name  

    ->  Gather Merge (cost=435036.62..437521.98 rows=20000 width=34) (actual time=12445.329..12648.063 rows=1188 loops=1)  

      Workers Planned: 2  

      Workers Launched: 2  

        ->  Partial GroupAggregate (cost=434036.59..434213.46 rows=10000 width=34) (actual time=12289.783..12393.036 rows=396 loops=3)  

          Group Key: supplier.su_name  

          ->  Sort  (cost=434036.59..434062.22 rows=10249 width=26) (actual time=12289.669..12352.666 rows=155365 loops=3)  

            Sort Key: supplier.su_name  

            Sort Method: external merge Disk: 5416kB  

            Worker 0: Sort Method: external merge Disk: 5504kB  

            Worker 1: Sort Method: external merge Disk: 5552kB  

            ->  Nested Loop (cost=182662.61..433353.85 rows=10249 width=26) (actual time=6002.059..10856.716 rows=155365 loops=3)  

              ->  Parallel Hash Join (cost=182662.18..433353.28 rows=108655 width=50) (actual time=6002.760..8333.428 rows=157581 loops=3)  

                Hash Cond: (a4.aux_o1_w_id = stock.s_i_id AND (a4.aux_o1_j_id = stock.s_l_id))  

                ->  Parallel Seq Scan on a4 aux_o1 (cost=0..08..131720.17 rows=5209018 width=24) (actual time=0.048..1036.510 rows=167270 loops=3)  

                ->  Parallel Hash (cost=82368.06..82368.06 rows=19608 width=34) (actual time=2060.920..2060.928 rows=108000 loops=3)  

                  Buckets: 65536 (originally 65536) Batches: 8 (originally 1) Memory Usage: 3232kB  

                  Hash Join (cost=256.56..182368.06 rows=19608 width=34) (actual time=51.033..1904.843 rows=108000 loops=3)  

                    Hash Cond: (mod(stock.s_w_id * stock.s_l_id), 10800) = supplier.su_suppkey  

                    ->  Parallel Index Only Scan using pk_stock on stock (cost=0..43..161081.89 rows=3333339 width=8) (actual time=0.110..1232.808 rows=2666667 loops=3)  

                      Heap Fetches: 0  

                    ->  Hash (cost=255.39..255.39 rows=59 width=30) (actual time=50.830..50.834 rows=396 loops=3)  

                      Buckets: 1024 Batches: 1 Memory Usage: 33kB  

                      ->  Nested Loop (cost=7..38..255.39 rows=59 width=30) (actual time=49.299..50.670 rows=396 loops=3)  

                        ->  Seq Scan on nation (cost=0..08..12.12 rows=1 width=4) (actual time=49.038..49.046 rows=1 loops=3)  

                          Filter: (n.name = 'GERMANY'::bpchar)  

                          Rows Removed by Filter: 24  

                        ->  Bitmap Heap Scan on supplier (cost=7..38..239.27 rows=400 width=34) (actual time=0.227..1.514 rows=396 loops=3)  

                          Recheck Cond: (su.nationkey = nation.n.nationkey)  

                          Heap Blocks: exact=191  

                        ->  Bitmap Index Scan on a4 supp_index (cost=0..00..7.29 rows=400 width=0) (actual time=0.156..0.157 rows=396 loops=3)  

                          Index Cond: (su.nationkey = nation.n.nationkey)  

                          Index Cond: ((o_w_id = a4.aux_o1_w_id) AND (o_d_id = a4.aux_o1_d_id) AND (o_id = a4.aux_o1_o_id))  

                          Rows Removed by Filter: 0  

Planning Time: 1.455 ms
JIT:
  Functions: 114
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 31.974 ms, Inlining 0.000 ms, Optimization 14.804 ms, Emission 130.138 ms, Total 176.916 ms
Execution Time: 12653.992 ms
(46 rows)

```

Figura 32: EXPLAIN ANALYZE da interrogação A4 materialized

Como foi previsto anteriormente, é possível de facto notar-se uma melhoria bastante significativa no tempo de execução da interrogação, o que demonstra que um grande entrave no desempenho era o *anti join*. Ainda assim, o tempo de execução não é ótimo, pois ainda ronda os 12 segundos, onde o peso dos restantes *joins* apresenta um grande impacto.

Seguindo a mesma ideia apresentada das *materialized views*, a *query* poderia ser ainda mais otimizada caso toda ela fosse colocada numa *materialized view*. No entanto, a estrutura lógica da *query* deixaria de ser respeitada.

É importante referir que para manter a *materialized view* constantemente atualizada, era possível recorrer a *triggers*. Caso essa atualização fosse muito custosa, era possível programá-la para fazer *refresh* apenas num determinado tempo, calculado num intervalo fixo, com a consciência de que a informação da *query* poderia encontrar-se desatualizada.

5 Conclusão

O desenvolvimento deste trabalho prático permitiu aos elementos do grupo consolidar os conceitos que foram dados ao longo do semestre na UC de Administração de bases de dados. Este projeto permitiu-nos adquirir mais conhecimento do funcionamento interno do *PostgreSQL*, relativamente aos seus parâmetros de configuração como aos seus impactos na carga transacional. Adicionalmente, também nos permitiu aprofundar a nossa aprendizagem quanto à otimização de interrogações, através, por exemplo, da utilização de vistas materializadas e de índices.

Considera-se que todos os objetivos do trabalho prático foram atingidos. No entanto, como trabalho futuro, o foco seria uma melhor otimização das *queries*, de forma a tentar baixar mais o tempo de execução de cada uma delas.

6 Anexos

6.1 Anexo A - *Script* das dependências

```
#!/bin/bash

sudo apt install maven

sudo apt install unzip

sudo apt install docker.io

export JAVA_HOME="/usr/"

sudo apt install python3-pip
```

Listing 1: dependencias.sh

6.2 Anexo B - *Script* do *setup* das VMs

```
#!/bin/bash

#instalar postgres
chmod +x install_postgres.sh
./install_postgres.sh

# instalar dependencias
chmod +x dependencias.sh
./dependencias.sh

#unzip dump
gunzip tpcc_bd.tar.gz

#create tpcc database
PGPASSWORD=postgres psql -U postgres -c 'create database
tpcc;'

#dump databse backup
PGPASSWORD=postgres pg_restore -U postgres -F c -d tpcc <
tpcc_bd.tar

chmod +x tpc-c-0.1-SNAPSHOT/run.sh
```

Listing 2: setup_vm.sh

Referências

- [1] <https://www.postgresql.org/docs/current/runtime-config-wal.html>
- [2] <https://www.postgresql.org/docs/current/transaction-iso.html>
- [3] <https://www.2ndquadrant.com/en/blog/evolution-fault-tolerance-postgresql-synchronous-commit/>
- [4] https://postgresqlco.nf/doc/en/param/full_page_writes/
- [5] https://postgresqlco.nf/doc/en/param/synchronous_commit/
- [6] https://postgresqlco.nf/doc/en/param/commit_siblings/