



Universidade do Minho
Escola de Engenharia

Benchmark TPC-C

Mestrado Integrado em Engenharia Informática
Administração de Bases de Dados
4.º Ano, 1.º Semestre

A81047 - Catarina Machado A81822 - João Costa
A80987 - Tiago Fontes

Braga, janeiro de 2020

Conteúdo

1	Introdução	7
2	Configuração do <i>benchmark</i> TPC-C	8
2.1	Ambiente de Teste	8
2.2	Análise de Resultados	9
2.2.1	Definir número de <i>Warehouses</i>	9
2.2.2	Definir número de Clientes	10
2.3	Configuração de Referência	13
3	Otimização do PostgreSQL	14
3.1	<i>Settings</i>	14
3.1.1	<i>fsync</i>	14
3.1.2	<i>synchronous commit</i>	15
3.1.3	<i>WAL_sync_method</i>	16
3.1.4	<i>full_pages</i>	17
3.1.5	<i>wal_buffers</i>	17
3.1.6	<i>commit_delay</i>	18
3.1.7	<i>commit_sibling</i>	19
3.2	<i>Checkpoints</i>	20
3.2.1	<i>checkpoint_timeout</i>	20
3.2.2	<i>checkpoint_completion_target</i>	20
3.2.3	<i>checkpoint_flush_after</i>	21
3.2.4	<i>checkpoint_warning</i>	22
3.2.5	<i>max_wal_size</i>	23
3.2.6	<i>min_wal_size</i>	24
3.3	<i>Archiving</i>	25
3.3.1	<i>archive_mode</i>	25
3.3.2	<i>archive_command</i>	26
3.3.3	<i>archive_timeout</i>	26
3.4	Isolamento	27
3.5	Combinações	27
3.5.1	<i>Settings</i>	28
3.5.2	<i>Checkpoints</i>	28
3.5.3	<i>Archiving</i>	29
3.5.4	<i>Checkpoints & Archiving</i>	29
3.5.5	<i>Checkpoints & Settings</i>	30
4	Análise e Otimização de Interrogações Analíticas	33
4.1	Interrogações analíticas (TPC-H)	33
4.1.1	Interrogações analítica A.1 (TPC-H)	34
4.1.1.1	Índices	35
4.1.1.2	Vistas materializadas	35
4.1.1.3	Vistas materializadas e índices	37
4.1.2	Interrogações analítica A.2 (TPC-H)	38

4.1.2.1	Índices	39
4.1.2.2	Vista Materializada 1	39
4.1.2.3	Vista Materializada 2	40
4.1.2.4	Vista Materializada 3	42
4.1.2.5	Vista Materializada 4	43
4.1.3	Interrogações analítica A.3 (TPC-H)	44
4.1.3.1	Índices	47
4.1.3.2	Materialized Views	51
4.1.3.3	Views	55
4.1.3.4	Considerações	59
4.1.4	Interrogações analítica A.4 (TPC-H)	60
4.1.4.1	Vistas Materializadas	60
4.1.4.2	Vista Materializada e Índice	62
5	Replicação e <i>Sharding</i>	63
5.1	Replicação	63
6	Conclusão	65
A	Script de Instalação de Dependências	67
B	Script de Configuração da Base de Dados	68
C	Script de run	69

Lista de Figuras

1	Resp. Time em comparação com Clientes.	11
2	Throughput em comparação com Clientes.	11
3	Resp. Time em comparação com Throughput.	11
4	Métricas Configuração de Referência.	13
5	Métricas com a opção <i>fsync = off</i>	15
6	Gráfico comparativo <i>synchronous commit</i>	16
7	Gráfico comparativo <i>WAL_sync_method</i>	16
8	Métricas com a opção <i>full_pages = off</i>	17
9	Gráfico comparativo <i>wal_buffers</i>	18
10	Gráfico comparativo <i>commit_delay</i>	19
11	Gráfico comparativo <i>commit_siblings</i>	19
12	Gráfico comparativo <i>checkpoint_timeout</i>	20
13	Gráfico comparativo <i>checkpoint_completion_target</i>	21
14	Gráfico comparativo <i>checkpoint_flush_after</i>	22
15	Gráfico comparativo <i>checkpoint_warning</i>	23
16	Gráfico comparativo <i>max_wal_size</i>	24
17	Gráfico comparativo <i>min_wal_size</i>	25
18	Gráfico comparativo <i>archive_mode</i>	25
19	Gráfico comparativo <i>archive_command</i>	26
20	Gráfico comparativo <i>archive_timeout</i>	27
21	Métricas melhor combinação.	31
22	Métricas melhor combinação (<i>rerun</i>).	31
23	Índices presentes no TPC-C.	33
24	<i>Explain Analyze query</i> A.1.	34
25	<i>Explain Analyze query</i> A.1 com <i>seqscan=off</i>	35
26	<i>Explain Analyze</i> da interrogação otimizada com <i>materialized view</i>	36
27	Plano execução com aplicação de <i>materialized view</i> e índices	37
28	<i>Explain Analyze</i> da interrogação analítica A.2.	38
29	<i>Explain Analyze</i> da interrogação analítica A.2 com índices.	39
30	<i>Explain Analyze</i> da interrogação analítica A.2 com a vista materializada 1.	40
31	<i>Explain Analyze</i> da interrogação analítica A.2 com a vista materializada 2.	41
32	<i>Explain Analyze</i> da interrogação analítica A.2 com a vista materializada 3.	42
33	<i>Explain Analyze</i> da interrogação analítica A.2 com a vista materializada 4.	43
34	<i>Explain Analyze</i> da interrogação analítica A.3	45
35	Plano de execução da <i>query</i>	46
36	<i>Stats</i> da interrogação analítica A.3	47
37	<i>Explain Analyze</i> da interrogação analítica A.3, com índices	48
38	Plano de execução da <i>query</i>	49
39	<i>Stats</i> da interrogação analítica A.3, com índices	50

40	<i>Explain Analyze</i> da interrogação analítica A.3, com índices e materialized view	52
41	Plano de execução da <i>query</i>	53
42	<i>Stats</i> da interrogação analítica A.3	54
43	<i>Explain Analyze</i> da interrogação analítica A.3, com índices e materialized view, e views	56
44	Plano de execução da <i>query</i>	57
45	<i>Stats</i> da interrogação analítica A.3	58
46	<i>Explain Analyze</i> da interrogação analítica A.4	60
47	<i>Explain Analyze</i> da interrogação analítica A.4 com <i>materialized view</i>	61
48	<i>Explain Analyze</i> da interrogação analítica A.4 com vista materializada e índice	62
49	Replicação lógica de base de dados	63
50	Resultado replicação base de dados	63
51	Resultado replicação base de dados	64
52	Observação métricas após replicação	64

Lista de Tabelas

1	Tamanho BD para cada n.º de <i>Warehouses</i>	10
2	Métricas para definir número de clientes.	10
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_pages</i>	17
7	Alteração de parâmetros na opção <i>wal_buffers</i>	18
8	Alteração de parâmetros na opção <i>commit_delay</i>	18
9	Alteração de parâmetros na opção <i>commit_siblings</i>	19
10	Alteração de parâmetros na opção <i>checkpoint_timeout</i>	20
11	Alteração de parâmetros na opção <i>checkpoint_completion_target</i>	21
12	Alteração de parâmetros na opção <i>checkpoint_flush_after</i>	21
13	Alteração de parâmetros na opção <i>checkpoint_warning</i>	22
14	Alteração de parâmetros na opção <i>max_wal_size</i>	23
15	Alteração de parâmetros na opção <i>min_wal_size</i>	24
16	Alteração de parâmetros na opção <i>archive_mode</i>	25
17	Alteração de parâmetros na opção <i>archive_command</i>	26
18	Alteração de parâmetros na opção <i>archive_timeout</i>	26
19	Métricas com combinação de settings.	28
20	Métricas com combinação de <i>checkpoints</i> 1	29
21	Métricas com combinação de <i>checkpoints</i> 2.	29
22	Métricas com combinação de <i>checkpoints</i> e <i>archiving</i> 1.	30
23	Métricas com combinação de <i>checkpoints</i> e <i>archiving</i> 2.	30
24	Métricas da melhor combinação.	30
25	Resultados da <i>Query</i>	47
26	Resultados da <i>Query</i> , part(1)	50
27	Resultados da <i>Query</i> , part(2)	54
28	Resultados da <i>Query</i> , part(3)	58

1 Introdução

O presente trabalho prático foi realizado no âmbito da unidade curricular **Administração de Base de Dados**, do perfil Engenharia de Aplicações, presente no Mestrado Integrado em Engenharia Informática da Universidade do Minho

Este projeto consistiu 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 *online* e simula um sistema de bases de dados de uma cadeia de lojas, suportando a operação diária de gestão de vendas e *stocks*. Foram também disponibilizadas interrogações analíticas adicionais, baseadas na adaptação do **TPC-H**.

Desta forma, o primeiro passo do trabalho prático consistiu na instalação e configuração do *benchmark* TPC-C, como forma de obter uma **configuração de referência** (hardware, número de *warehouses* e número de clientes). Em seguida, usando essa configuração, o objetivo foi otimizar o **desempenho da carga transacional** tendo em conta, principalmente, os parâmetros de configuração do PostgreSQL.

Posteriormente, otimizamos o **desempenho das interrogações analíticas** tendo em conta, principalmente, os respetivos planos e os mecanismos de redundância que estão a ser usados.

Por fim, foi testada e proposta uma configuração usando replicação/ processamento distribuído.

2 Configuração do *benchmark* TPC-C

O primeiro passo do presente trabalho prático foi instalar e configurar o *benchmark* TPC-C, de modo a obter uma configuração de referência em termos de *hardware*, número de *warehouses* e número de clientes.

2.1 Ambiente de Teste

A nível de *hardware*, averiguamos a possibilidade da utilização de quatro diferentes máquinas virtuais na *Google Cloud Platform*. Em todas elas, a região escolhida é a Europa (Bélgica), de modo a pouparmos saldo, Série N1 (tecnologia da plataforma CPU *Intel Skylake* ou uma das antecessores), novo disco padrão SSD de 250GB, Disco de Inicialização Ubuntu 18.04 LTS.

Inicialmente, como forma de poupar algum dinheiro, o grupo decidiu utilizar nas máquinas discos de 30GB, isto porque nos pareceu suficiente para as experiências que iam realizar. No entanto, o grupo deparou-se com uma série de problemas neste capítulo, uma vez que ao nível de débito, mesmo com elevado número de clientes e *warehouses*, este era raro ascender acima das 100 transações/segundo. Assim sendo, e após alguma pesquisa e conversas com o docente, o grupo constatou que de facto a capacidade do disco em termos de escritas e leituras era proporcional à capacidade do disco. Assim, como a capacidade escolhida por nós não era muito elevada, a capacidade do disco também era bastante limitada, tal como acabamos por perceber através do *iowait* bastante elevado. Assim sendo, o grupo procedeu à utilização de discos de 250GB em todas as suas máquinas, pelo que este problema foi ultrapassado com sucesso.

O fator diferenciador entre as máquinas seria o número de CPUs e a memória:

- 1 CPU com 4GB de memória;
- 2 CPU com 8GB de memória;
- 4 CPU com 8GB de memória;
- 4 CPU com 15GB de memória.

No entanto, decidimos excluir a opção de 1 CPU com 4GB memória por ser pouco CPU e a opção de 4 CPUs com 15GB memória por ter RAM demasiado elevada. Depois de alguns testes alterando os parâmetros número de clientes e número de *warehouses*, decidimos optar pela máquina de 4 CPUs com 8GB memória, por ser a que permite uma futura maior otimização por possuir melhores características.

Depois de criada a máquina virtual na *Google Cloud Platform*, o passo seguinte foi instalar o *benchmark* TPC-C e a respetiva base de dados.

Como forma de poupar saldo na *Google Cloud Platform* (visto que o *load* da base de dados seria um processo demorado), depois de termos a base de dados

PostgreSQL a correr na nossa máquina local, cada elemento do grupo fez um *backup* da mesma (devido ao nome de utilizador), através do seguinte comando:

```
docker exec postgres pg_dump -U postgres tpcc > backup_${NOME}.sql
```

Desta forma, foi criado um *bucket* que contém o *zip* do TPC-C, cada um dos *backups* dos diferentes elementos e ainda dois *scripts*.

Um desses *scripts*, presente no Anexo A, foi criado com o objetivo de instalar as dependências necessárias do *benchmark* TPC-C, como, por exemplo *Java*, *Maven* e *Python*. Para além disso, o *script* também copia o conteúdo do *bucket* para a diretoria, faz *unzip* do TPC-C e gera uma *build* do trabalho.

O segundo *script*, presente no Anexo B, tem como objetivo configurar a base de dados. Para tal, deve receber como parâmetro o nome do *user* da base de dados.

2.2 Análise de Resultados

Uma vez escolhido o *hardware*, máquina de 4 CPUs com 8GB, restou decidir o número de *warehouses* e o número de clientes necessários.

O objetivo seria encontrar os parâmetros que permitem obter uma taxa de utilização do CPU de cerca de 50%, uma taxa de ocupação da RAM de aproximadamente 100% e uma taxa de *abort rate* no máximo de 2%.

Desta forma, após alguns testes, decidimos seguir a estratégia de começar por encontrar o número de referência de *warehouses*, e, só posteriormente, encontrar o número ideal de clientes.

2.2.1 Definir número de *Warehouses*

Começamos por fazer *load* da base de dados localmente com cada número de *warehouses* necessários e, depois disso, através do *pg_dump* criamos um ficheiro da base de dados compactado, ficheiro esse que foi transferido para a máquina para povoar a base de dados remota, para não gastarmos tantos recursos computacionais na máquina, tornando o processo mais económico. Foi também desligado o *fsync* para tornar o *load* mais rápido e, por sua vez, mais barato.

Após alguns testes, decidimos começar por nos guiar pelo tamanho da base de dados, que deverá ser igual ou superior a 8GB (tal como já mencionado).

Os dados obtidos encontram-se na Tabela 1.

N.º Warehouses	Tamanho da Base Dados (MB)
2	257
4	492
8	961
16	1930
32	3776
64	7292
100	11000

Tabela 1: Tamanho BD para cada n.º de *Warehouses*

Desta forma, o número de *warehouses* escolhido para a configuração de referência foi **100 warehouses**.

2.2.2 Definir número de Clientes

Uma vez definido o número de *warehouses*, resta-nos decidir qual o número de clientes a utilizar.

Para encontrar este parâmetro, tivemos em consideração o *Throughput*, o *Response Time*, o *Abort Rate* e a carga de CPU da máquina.

Para não haver ruído, para cada um dos testes fazíamos tudo de novo a cada teste realizado, eliminava-mos a base de dados e reconstruía-mos a partir ficheiro *dump*. Foi utilizado 10 minutos de *measurement time*. Para facilitar, a o *run*, criamos um script que permite alterar facilmente o **número de clientes** e o **tempo de execução**, esse *script* encontra-se na secção Anexos.

N.º Clientes	Throughput (tx/seg)	Resp. Time (seg)	Abort Rate (%)
10	84,11564654	0,005693521814	0,0430422855
20	174,8992329	0,007019199464	0,002340577979
30	262,0017983	0,00866793889	0,005507415874
40	335,2599397	0,01867113961	0,0259013626
50	371,6773679	0,02705660533	0,06176246937
60	398,2137593	0,04124091592	0,04672307452
70	400,3290247	0,04591810497	0,0441211543
80	403,808691	0,04237566379	0,04992057772
90	407,3265743	0,0429758779	0,05392305362
100	401,0977742	0,04776469718	0,0827326586
110	357,0245751	0,03365851963	0,07702286584
120	416,0256791	0,04330397354	0,04357926671

Tabela 2: Métricas para definir número de clientes.



Figura 1: Resp. Time em comparação com Clientes.

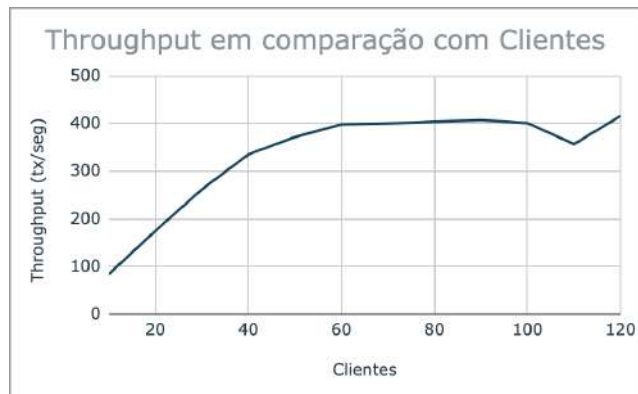


Figura 2: Throughput em comparação com Clientes.

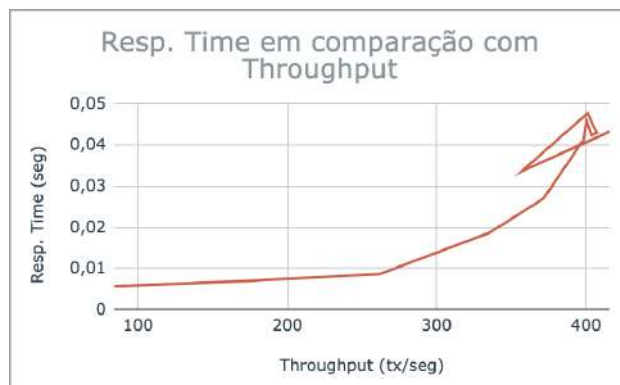


Figura 3: Resp. Time em comparação com Throughput.

Procuramos assim encontrar uma configuração em o *Throughput* seja muito próximo do máximo (perto do joelho do gráfico), que se pode verificar pela Fig 2.

E que tenho uma boa relação (*Throughput - Response Time*, o que em termos práticos é possível ver através do "joelho" do gráfico na Fig 3.

Em termos de CPU, procuramos que a máquina de teste tivesse uma percentagem de CPU na casa dos 50%, para que a máquina não realizasse o trabalho de forma muito fácil, nem ,em caso de CPU acima dos 80%, estivesse com muita dificuldade em realizar as tarefas. Assim, resultou numa número de 70 clientes, que já está na parte de cima do "joelho" da curva, porém tem muita margem de progressão.

2.3 Configuração de Referência

Após todos os testes já apresentados, concluímos que a configuração de referência deverá ter uma máquina de **4 CPUs** e **8GB** de memória, **100 warehouses** e **70 clientes**.

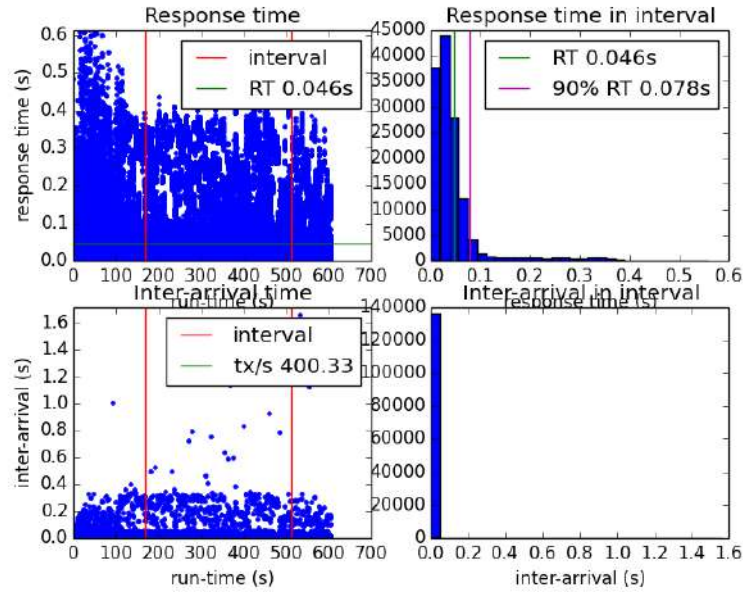


Figura 4: Métricas Configuração de Referência.

3 Otimização do PostgreSQL

Nesta secção, iremos tentar averiguar qual o impacto de algumas configurações do *Postgresql* no desempenho do nosso *Benchmark*.

Inicialmente, após algumas pesquisas na Internet sobre tema, encontramos algumas informações acerca da otimização. Lemos que mudanças no **shared_buffers**, **work_mem** e no **vacuum** trariam diversas vantagens ao nosso *Postgresql Server*. Porém, após algum aconselhamento por parte do docente, decidimos que o nosso foco iria ser apenas as configurações relativas a transações, dado que foram estas que mereceram mais atenção durante as aulas práticas.

Dada a nossa inexperiência no campo e com o foco na aprendizagem, investigamos e testamos **todas** as configurações da secção de *Write Ahead Logs*, com diversas opções em cada parâmetro. Esta “zona” da configuração pode ser dividida em 3 partes, **settings**, **checkpoints**, **archiving**.

Testamos cada umas delas de forma isolada, (i.e, apenas o parâmetro em teste foi alterado, todos os outros foram deixados em modo *default*). E para que o resultado fosse o menos enviesado possível, todos os testes foram realizados na mesma configuração de referência (numa máquina da *Google Cloud*) e a base de dados partiu sempre do mesmo estado.

No final desta etapa de testes de configurações, será feita uma análise, com o objetivo de determinar quais os parâmetros a considerar na otimização do *Postgresql*.

De seguida iremos falar um pouco acerca dos parâmetros que alteramos e sobre a sua influência no desempenho.

3.1 Settings

3.1.1 *fsync*

Quando este parâmetro está no seu valor por defeito, *On*, o *Postgresql* garante que os *updates* estão, de facto, escritos no disco, fazendo chamadas de funções de sincronização, garantindo assim que a base de dados é recuperável em caso de falha.

Alterando o seu estado para *Off*, o sistema vai dizer que o *update* está feito, mesmo não estando, e escreve no disco logo que possível.

Apesar do apreciável aumento no desempenho, em caso de falha é impossível recuperar as transações que não foram escritas.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'off'	531.82	0.0193	0.0276

Tabela 3: Alteração de parâmetros na opção *fsync*.

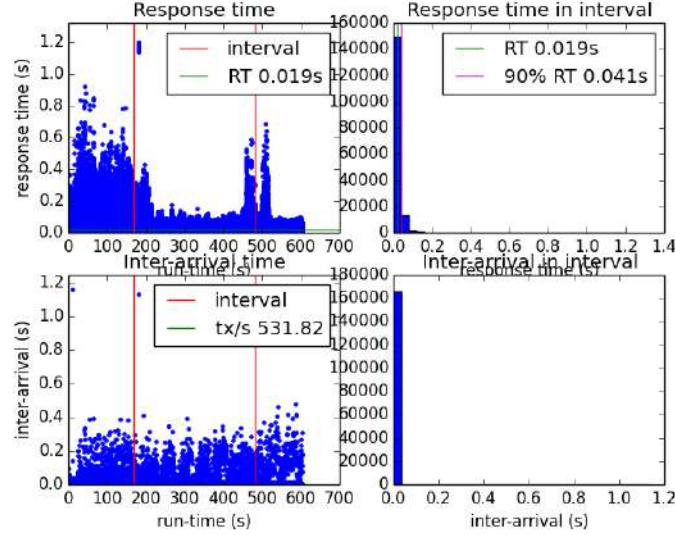


Figura 5: Métricas com a opção *fsync = off*.

Apesar desta opção parecer apetecível, não deve usada, pelo facto de existir a possibilidade de perda de coerência da base de dados. Daí, este teste foi feito apenas para efeitos de aprendizagem.

3.1.2 *synchronous commit*

Este parâmetro especifica se um *commit* espera, ou não, que o *WAL record* seja escrito no disco antes de enviar a informação de “sucesso” para o cliente. Quando esta opção está a *Off*, existe um pequeno *delay* entre a mensagem de “sucesso” enviada ao cliente e o momento em que a transação está de facto garantida.

Ao contrário do *fsync*, colocar este parâmetro a *Off* não cria o risco de inconsistência da Base de Dados. Em caso de *crash* o resultado é a perda de alguns *commits*, sendo o estado da BD igual a se esses *commits* fossem abortados de “forma limpa”.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'off'	165.35	0.0350	0.0344
'local'	397.18	0.0402	0.0800
'remote write'	391.99	0.0402	0.0784
'on'	399.09	0.0426	0.0823

Tabela 4: Alteração de parâmetros na opção *synchronous commit*.

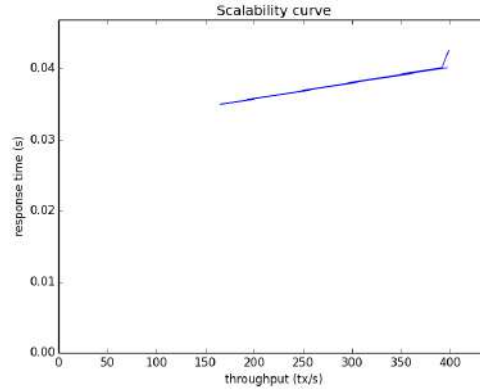


Figura 6: Gráfico comparativo *synchronous commit*.

3.1.3 *WAL_sync_method*

Este parâmetro permite-nos escolher qual o método que é usado para forçar os *WAL updates* para o disco. De seguida é possível ver a comparação entre as diferentes opções que estão disponíveis.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'open_datsync'	392.73	0.0445	0.0830
'fdatsync'	398.43	0.0421	0.0796
'fsync'	415.34	0.0445	0.0397
'open_sync'	364.70	0.0493	0.0906

Tabela 5: Alteração de parâmetros na opção *WAL_sync_method*.

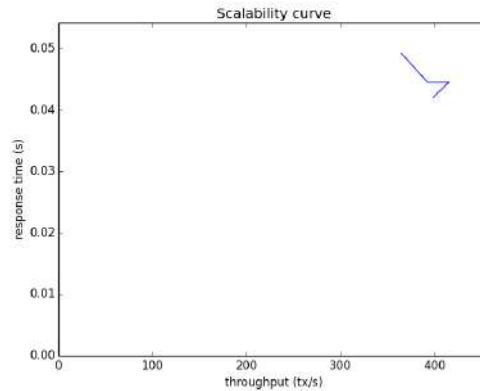


Figura 7: Gráfico comparativo *WAL_sync_method*.

3.1.4 *full_pages*

Quando ativo, o *Postgresql* escreve o conteúdo de cada página do disco no WAL durante a primeira modificação da página depois de um *checkpoint*.

Desativar esta *feature* aumenta o desempenho das operações, porém corremos o risco de chegar a um estado de **dados corrompidos**. O risco é similar a desabilitar o *fsync* mas em menor escala.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'off'	530.91	0.0163	0.0408

Tabela 6: Alteração de parâmetros na opção *full_pages*.

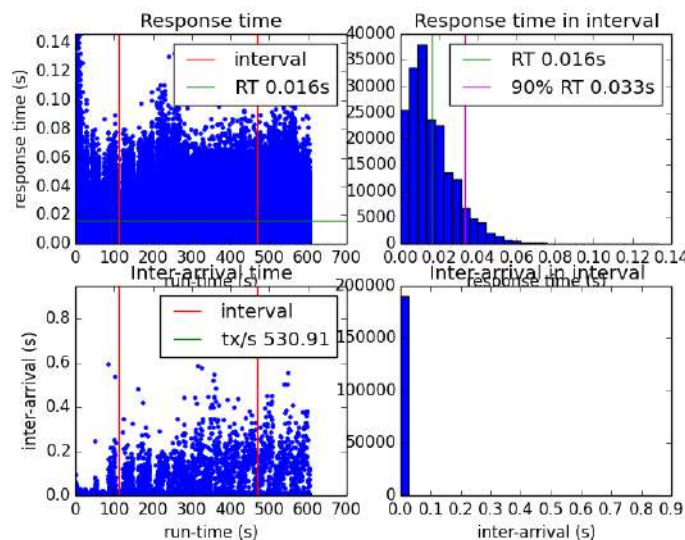


Figura 8: Métricas com a opção *full_pages* = *off*.

Tal como o *fsync*, não vamos utilizar esta opção, porque , apesar de menor, existe o risco de haver dados corrompidos irrecuperáveis.

3.1.5 *wal_buffers*

Este parâmetro diz respeito à quantidade de *shared memory* usada para dados do WAL que ainda não forma escritos em disco.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'2MB'	398.22	0.0440	0.0720
'4MB'	395.53	0.0448	0.0786
'8MB'	406.30	0.0447	0.0474
'16MB'	407.87	0.0428	0.0571
'32MB'	387.54	0.0423	0.0946

Tabela 7: Alteração de parâmetros na opção *wal_buffers*.

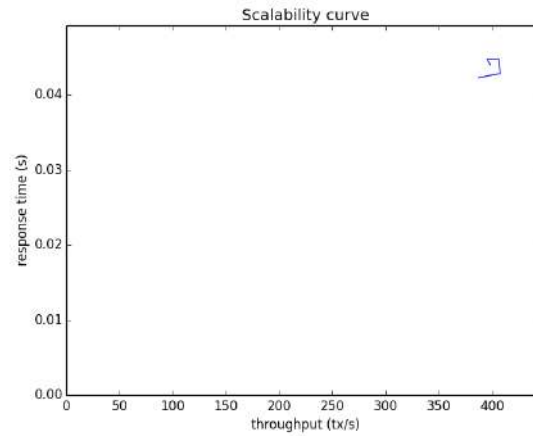


Figura 9: Gráfico comparativo *wal_buffers*.

3.1.6 *commit_delay*

Através desta opção é possível adicionar um atraso, em microsegundos, antes do *flush* do WAL ser iniciado. Isto pode aumentar o *throughput*, permitindo que mais transações façam *commit* num só WAL *flush*.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'100ms'	380.50	0.0405	0.0836
'200ms'	395.12	0.0434	0.0779
'500ms'	410.87	0.0421	0.0428
'1000ms'	413.85	0.0403	0.0439
'2000ms'	398.64	0.0427	0.0461

Tabela 8: Alteração de parâmetros na opção *commit_delay*.

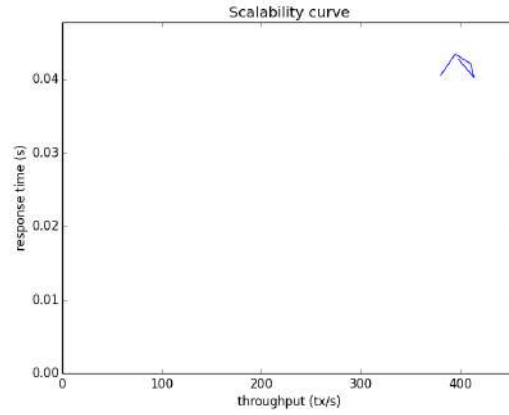


Figura 10: Gráfico comparativo *commit_delay*.

3.1.7 *commit_sibling*

Número mínimo de transações em simultâneo antes de iniciar o *commit_delay*. É de espera que um valor maior, aumenta a probabilidade de uma outra transação ficar preparada para dar *commit* durante o intervalo.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'2'	360.78	0.026	0.0442
'10'	408.63	0.035	0.0428
'20'	388.79	0.045	0.0825

Tabela 9: Alteração de parâmetros na opção *commit_sibling*.

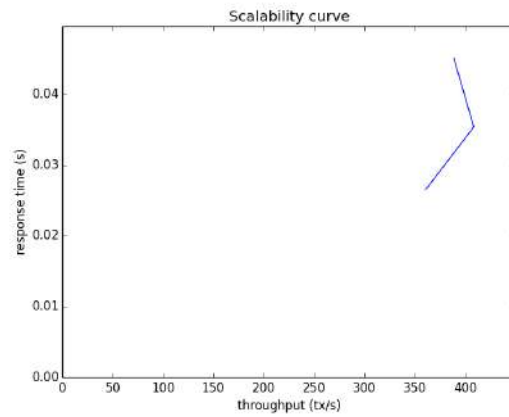


Figura 11: Gráfico comparativo *commit_sibling*.

3.2 Checkpoints

3.2.1 *checkpoint_timeout*

Neste parâmetro é possível definir o tempo máximo, em segundos, entre *WAL Checkpoints* automáticos.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'30s'	388.87	0.0493	0.0834
'1min'	402.56	0.0484	0.0444
'2min'	403.54	0.0475	0.0796
'5min'	373.32	0.0511	0.0945

Tabela 10: Alteração de parâmetros na opção *checkpoint_timeout*.

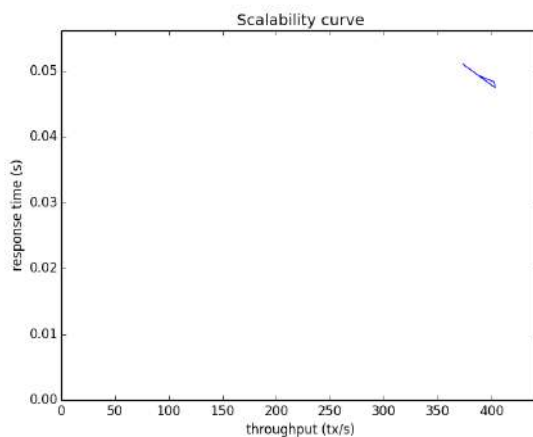


Figura 12: Gráfico comparativo *checkpoint_timeout*.

3.2.2 *checkpoint_completion_target*

Este parâmetro especifica a meta de conclusão do *Checkpoint*, como uma fração do tempo total entre os *Checkpoints*.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'0'	389.07	0.0492	0.0854
'0.2'	398.88	0.0481	0.0812
'0.4'	382.49	0.0498	0.0903
'0.6'	430.97	0.0451	0.0402
'0.8'	345.07	0.0556	0.0908
'1'	365.98	0.0525	0.0866

Tabela 11: Alteração de parâmetros na opção *checkpoint_completion_target*.

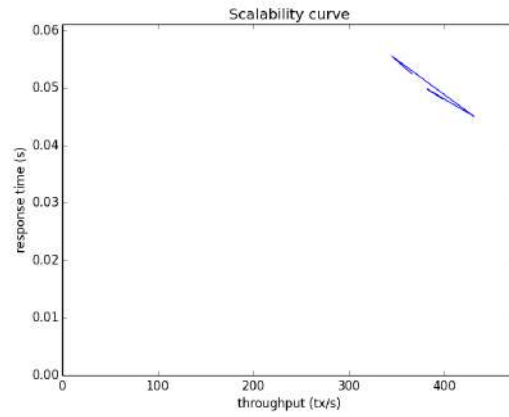


Figura 13: Gráfico comparativo *checkpoint_completion_target*.

3.2.3 *checkpoint_flush_after*

Neste parâmetro é possível determinar o número de *bytes* mínimos para que, enquanto o sistema está a fazer um *checkpoint*, tenta forçar o SO a escrever numa *storage* adjacente.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'0'	408.92	0.0470	0.0753
'512kB'	405.69	0.0472	0.0783
'1024kB'	394.45	0.0485	0.0816

Tabela 12: Alteração de parâmetros na opção *checkpoint_flush_after*.

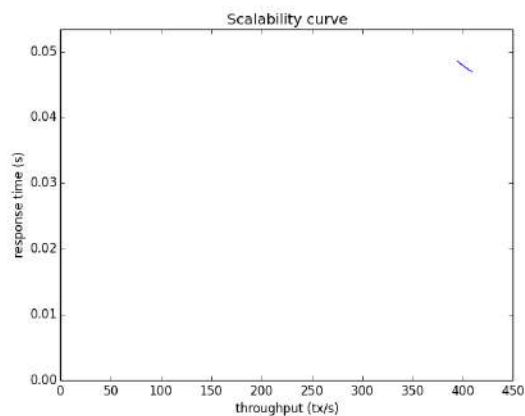


Figura 14: Gráfico comparativo *checkpoint_flush_after*.

3.2.4 *checkpoint_warning*

Este parametro permite que seja escrita uma mensagem no *log* do servidor se os *checkpoints* causados pelo preenchimento de arquivos de segmentos de *checkpoint* acontecerem mais próximos uns dos outros do que estes segundos, isto sugere que o *max_wal_size* deve ser aumentado.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'0'	406.02	0.0470	0.0819
'10s'	440.61	0.0442	0.04
'30s'	411.07	0.0472	0.0501
'60s'	417.13	0.0465	0.0490
'100s'	406.10	0.0477	0.0620

Tabela 13: Alteração de parâmetros na opção *checkpoint_warning*.

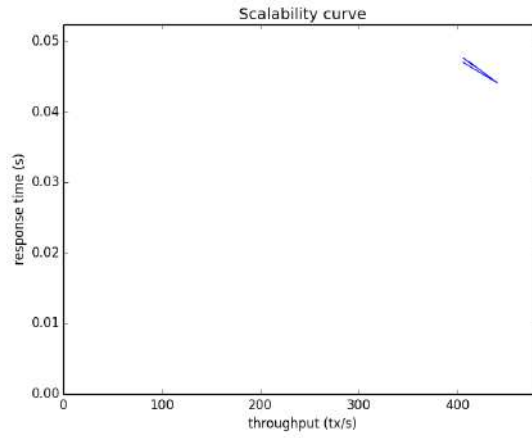


Figura 15: Gráfico comparativo *checkpoint_warning*.

3.2.5 *max_wal_size*

O *max_wal_size* é o parâmetro que nos possibilita escolher o tamanho máximo que o WAL pode crescer entre *Checkpoints* automáticos. Este limite é considerado pela documentação de *soft limit*. É considerado assim pois é possível exceder este valor em situações especiais, como por exemplo, muita carga.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'1GB'	399.12	0.0472	0.0844
'2GB'	485.61	0.0322	0.0785
'4GB'	528.40	0.0278	0.0389

Tabela 14: Alteração de parâmetros na opção *max_wal_size*.

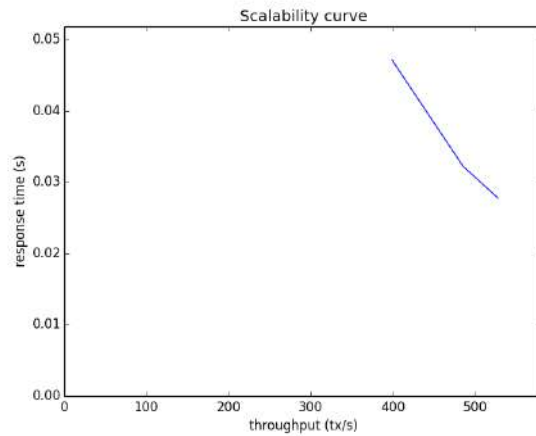


Figura 16: Gráfico comparativo *max_wal_size*.

3.2.6 *min_wal_size*

Enquanto o tamanho do WAL se mantenha abaixo do valor desta opção, os WAL antigos são reciclados para uso futuro, em vez de removidos. Isto pode ser usado para assegurar que o espaço necessário é reservado para lidar com picos no seu uso.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'80MB'	398.13	0.0481	0.0846
'160MB'	396.53	0.0482	0.0853
'320MB'	401.08	0.0478	0.0861
'640MB'	400.82	0.0476	0.0855

Tabela 15: Alteração de parâmetros na opção *min_wal_size*.

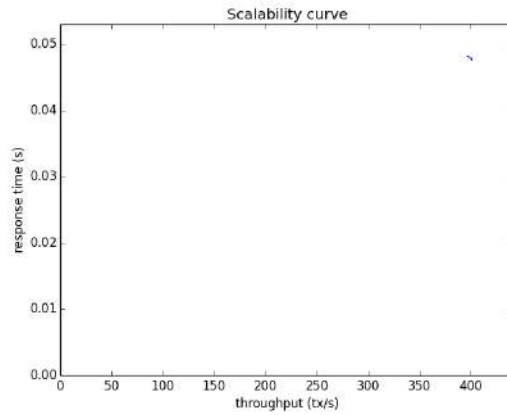


Figura 17: Gráfico comparativo *min_wal_size*.

3.3 Archiving

3.3.1 *archive_mode*

Quando ativo, segmentos WAL completos são enviados para o *archive storage*.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'always'	312.01	0.0616	0.0874
'on'	312.72	0.0613	0.0906
'off'	384.68	0.0498	0.0818

Tabela 16: Alteração de parâmetros na opção *archive_mode*.

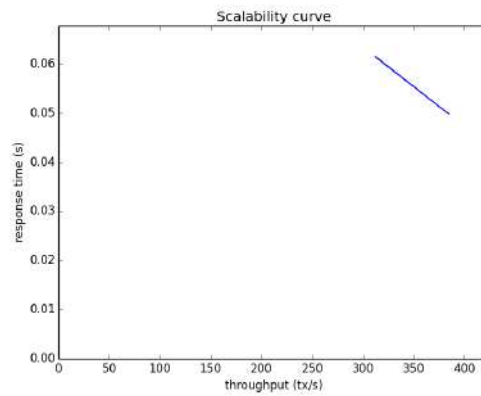


Figura 18: Gráfico comparativo *archive_mode*.

3.3.2 *archive_command*

Este parâmetro indica que comando a *shell* executa para arquivar um ficheiro com segmentos WAL completos.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'%p'	396.80	0.0482	0.0814
'%f'	389.72	0.0491	0.0836

Tabela 17: Alteração de parâmetros na opção *archive_command*.

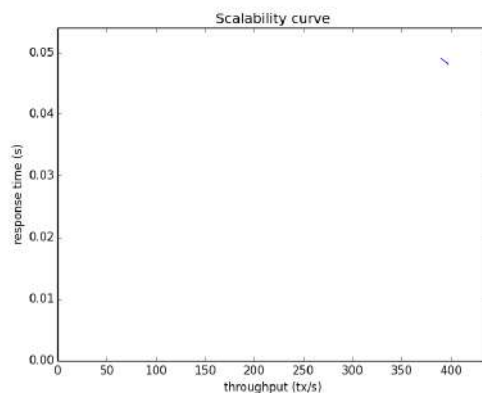


Figura 19: Gráfico comparativo *archive_command*.

3.3.3 *archive_timeout*

Por norma, o *archive_command* só é invocado para segmentos WAL completos. No entanto, se o *server* gerar pouco “tráfego de WAL” pode acontecer que demore muito a completar o segmento, para limitar a tempo de existência de dados não arquivados, é possível definir um *timeout* para forçar a troca para um novo ficheiro de WAL.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
'10'	403.08	0.0475	0.0807
'60'	228.99	0.0816	0.2223
'100'	389.76	0.0491	0.0837
'120'	443.60	0.0439	0.0397
'180'	396.71	0.0484	0.0787
'240'	400.23	0.0486	0.0457

Tabela 18: Alteração de parâmetros na opção *archive_timeout*.

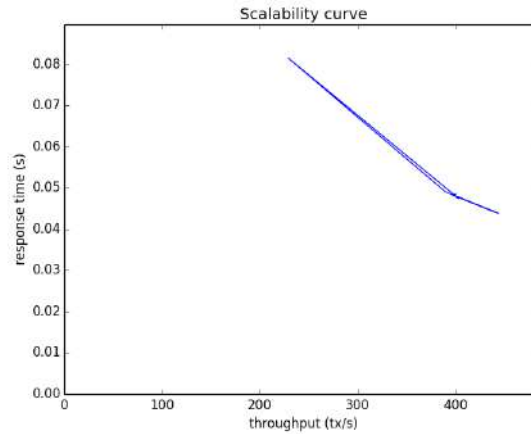


Figura 20: Gráfico comparativo *archive_timeout*.

3.4 Isolamento

Um outro aspeto que consideramos importante na análise dos resultados é o nível de isolamento que está a ser levado em conta. Assim, e tendo em conta a natureza da aplicação TPC-C, o grupo considerou que faria sentido analisar a influência das várias opções de isolamento no comportamento das transações e consequentemente no sistema.

Nível de Isolamento	Throughput (tx/s)	Response Time (s)	Abort rate (%)
Read Committed (nível base)	400.3290247	0.04591810497	0.0441211543
Repeatable Read	403.13571964251395	0.047509875689216316	0.0795739735681
Serializable	409.8341336947235	0.04748727625620014	0.0450267387021

Assim sendo, conseguimos analisar que houve uma subida mas muito ligeira em termos de débito (nesta ordem de grandeza de valores, não será muito significativo), bem como um aumento subtil em termos de tempo de resposta e *abort rate*.

Isto poderá ser explicado tendo em conta as características dos vários níveis de isolamento. Como sabemos, o nível *Serializable* fornece um isolamento mais rigoroso. Por outro lado, o nível *Repeatable Read* também promove maior cautela na forma como trata as transações (quando comparado com o nível base), exibindo assim maior *abort rate* e também um pequeno aumento no tempo de resposta.

3.5 Combinações

Nesta secção vamos apresentar o resultado da análise que fizemos dos diferentes parâmetros e da utilidade de cada um.

Como forma de abranger mais possibilidades para uma maior probabilidade de melhorarmos o desempenho da carga transacional, combinamos as opções com melhores resultados de cada uma das três secções, Secção 3.1, 3.2 e 3.3. O objetivo foi tentar encontrar a melhor configuração possível do *postgres* para o nosso problema.

3.5.1 *Settings*

Começamos por combinar as melhores opções da secção de *settings*, alterando desta forma os seguintes parâmetros no ficheiro *postgresql.conf*:

```
wal_sync_method = fsync
synchronous_commit = off
wal_buffers = 16MB
commit_delay = 1000
commit_siblings = 10
```

Os resultados obtidos encontram-se na tabela seguinte:

	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
1. ^a	449.32	0.0374	0.0286
2. ^a	492.99	0.0370	0.0294

Tabela 19: Métricas com combinação de settings.

Corremos várias vezes esta configuração uma vez que o resultado pode variar um pouco devido aos números aleatórios produzidos em *runtime* mas os resultados mantêm-se dentro da mesma gama de valores.

Os resultados obtidos são melhores comparativamente aos da configuração de referência.

3.5.2 *Checkpoints*

No caso dos *checkpoints*, Secção 3.2, para cada um dos diferentes parâmetros possíveis escolhemos também o melhor de cada grupo (tendo em consideração que essa melhor opção tem que ter um *throughput* superior ao da configuração de referência (400,33 tx/seg)).

Desta forma, no ficheiro *postgresql.conf* foram alteradas as opções para as mencionadas em seguida:

```
checkpoint_timeout = 2min
checkpoint_completion_target = 0.6
checkpoint_flush_after = 0
checkpoint_warning = 10s
max_wal_size = 4GB
min_wal_size = 320MB
```

Os resultados obtidos com estas alterações encontram-se na tabela seguinte.

CPU (%)	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
59.25	507.73	0.0308	0.0581

Tabela 20: Métricas com combinação de *checkpoints* 1 .

Apesar dos resultados serem mais satisfatórios do que os obtidos na configuração de referência, optamos por tentar reduzir o número de parâmetros alterados como forma de obtermos resultados ainda superiores.

Desta forma, numa nova tentativa alteramos somente as seguintes opções:

```
checkpoint_completion_target = 0.6
checkpoint_warning = 10s
max_wal_size = 4GB
```

Estas alterações de facto levaram-nos a resultados melhores tanto a nível de *throughput* (aumentou), como a nível de *response time* e *abort rate* (que diminuíram).

CPU (%)	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
61.72	520.36	0.0292	0.0477

Tabela 21: Métricas com combinação de *checkpoints* 2.

3.5.3 Archiving

No caso das opções de *archiving*, presentes na Secção 3.3, todas elas apresentam resultados de *throughput* inferiores ao da configuração de referência, o que nos levou a não realizar testes isolados com opções combinadas de *archiving*.

3.5.4 Checkpoints & Archiving

Tentamos perceber se combinando a melhor opção de *archiving* (ainda que inferior à de referência) com as melhores opções do *checkpoints* conseguíamos obter resultados superiores aos mencionados anteriormente (presentes na Tabela 21).

No entanto, tanto com todos os parâmetros de *checkpoint* como só com apenas os 3 melhores, os resultados obtidos foram menos satisfatórios, tal como se pode ver na Tabela 22 e na Tabela 23.

```
checkpoint_timeout = 2min
checkpoint_completion_target = 0.6
checkpoint_flush_after = 0
checkpoint_warning = 10s
max_wal_size = 4GB
```

```
min_wal_size = 320MB
archive_timeout = 120
```

CPU (%)	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
59.82	503.86	0.0280	0.0539

Tabela 22: Métricas com combinação de *checkpoints* e *archiving* 1.

```
checkpoint_completion_target = 0.6
checkpoint_warning = 10s
max_wal_size = 4GB
archive_timeout = 120
```

CPU (%)	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
61.05	517.90	0.0295	0.0491

Tabela 23: Métricas com combinação de *checkpoints* e *archiving* 2.

3.5.5 Checkpoints & Settings

Por fim, tentamos então combinar as melhores opções de *settings* com as de *checkpoints*. As opções a alterar no ficheiro de configuração do *postgres* são os seguintes:

```
wal_sync_method = fsync
synchronous_commit = off
wal_buffers = 16MB
commit_delay = 1000
commit_siblings = 10
checkpoint_completion_target = 0.6
checkpoint_warning = 10s
max_wal_size = 4GB
```

Executamos 2 vezes, e, em seguida, encontram-se os resultado dessas execuções.

Tal como se pode ver, os resultados obtidos são bastante superiores a todos os já obtidos até então.

	CPU (%)	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
1. ^a	64.55	532.49	0.0268	0.0339
2. ^a	64.55	595.60	0.0163	0.0237

Tabela 24: Métricas da melhor combinação.

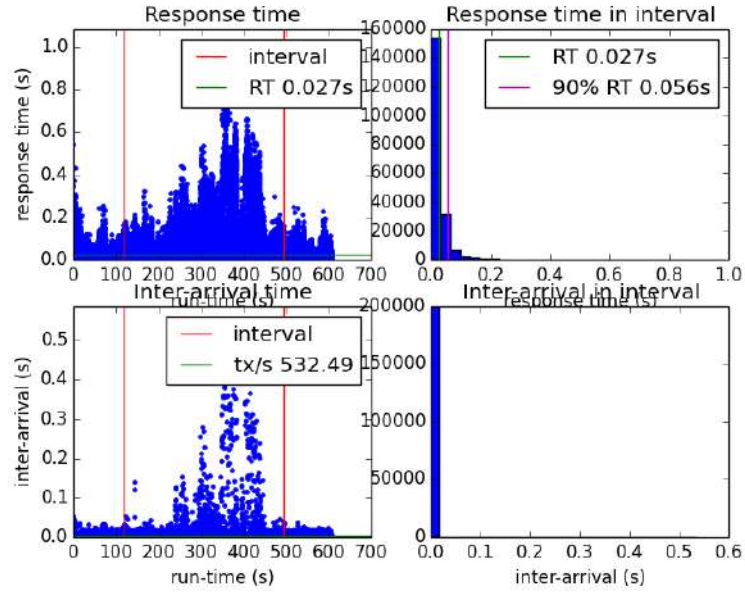


Figura 21: Métricas melhor combinação.

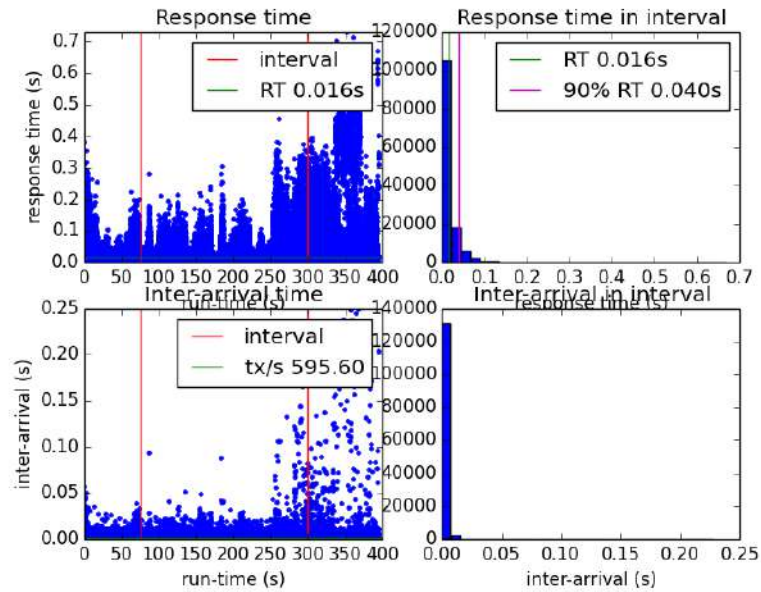


Figura 22: Métricas melhor combinação (*rerun*).

Desta forma, comparando com a configuração de referência, Tabela 1, é possível observar que conseguimos aumentar o valor de *throughput* de 400.33 para 595.60 *tx/s*, diminuir o valor de *response time* de 0.0459 para 0.0163 s. A taxa de *abort rate* diminuiu ligeiramente, de 0.0441 para 0.0237. O objetivo de otimizar o *Postgresql*, foi realizando com sucesso.

4 Análise e Otimização de Interrogações Analíticas

Após conseguirmos otimizar a carga transacional através da alteração de parâmetros de configuração do *postgresql*, o objetivo foi otimizar o desempenho das interrogações analíticas.

Desta forma, procedemos à análise das interrogações analíticas adaptadas do TPC-H e tentamos otimizá-las através dos métodos que iremos explicar em seguida.

Em seguida, apresentam-se os índices que já se encontram contidos na base de dados:

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

Figura 23: Índices presentes no TPC-C.

4.1 Interrogações analíticas (TPC-H)

Relativamente à otimização das quatro interrogações analíticas baseadas na adaptação do TPC-H, estudou-se, para cada uma delas, qual o seu tempo de execução através do comando *EXPLAIN ANALYZE*. Sendo que de seguida, tentou-se, através do uso correto de índices e vistas materializadas, obter tempo de execução inferiores. A escolha da utilização de índices deve-se ao facto de estes melhorarem a performance da base de dados através de uma identificação mais rápida dos dados que se procura. As vistas materializadas permitem guardar consultas em disco, fazendo com que seja mais rápido aceder aos resultados já que não é necessário estar sempre a calcular os mesmos.

4.1.1 Interrogações analítica A.1 (TPC-H)

```
select su_name, su_address
from supplier, nation
where su_suppkey in
(select mod(s_i_id * s_w_id, 10000)
from stock, order_line
where s_i_id in
(select i_id
from item
where i_data like 'c%')
and ol_i_id=s_i_id
and extract(second from ol_delivery_d) > 50
group by s_i_id, s_w_id, s_quantity
having 2*s_quantity > sum(ol_quantity))
and su_nationkey = n_nationkey
and n_name = 'GERMANY'
order by su_name;
```

Com o intuito de otimizar a *query*, o grupo decidiu então aplicar o comando *EXPLAIN ANALYZE* para assim poder aferir quais os passos que o *postgresql* estava a utilizar para dar resposta. Assim sendo, o grupo deparou-se com um tempo de execução 24800.385ms o que é de facto muito tempo. Como tal, iniciou-se então o processo de otimização.

```
Sort (cost=2385358.74..2385358.81 rows=38 width=51) (actual time=24769.265..24769.272 rows=147 loops=1)
  Sort Key: supplier.su_name
  Sort Method: quicksort Memory: 42kB
  -> Hash Join (cost=2385355.24..2385358.00 rows=38 width=51) (actual time=24767.779..24768.779 rows=147 loops=1)
    Hash Cond: ((mod((stock.s_i_id * stock.s_w_id), 10000)) = supplier.su_suppkey)
    -> HashAggregate (cost=2384982.28..2384984.28 rows=280 width=4) (actual time=24754.553..24755.177 rows=3662 loops=1)
      Group Key: mod((stock.s_i_id * stock.s_w_id), 10000)
      -> Finalize GroupAggregate (cost=2033514.27..2372482.28 rows=1000000 width=15) (actual time=23266.218..24751.242 rows=5323 loops=1)
        Group Key: stock.s_i_id, stock.s_w_id
        Filter: ((2 * stock.s_quantity) > sum(order_line.ol_quantity))
        Rows Removed by Filter: 144977
        -> Gather Merge (cost=2833514.27..2342482.28 rows=2000000 width=20) (actual time=23256.072..24678.825 rows=458000 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          -> Partial GroupAggregate (cost=2432514.25..2118632.63 rows=1000000 width=20) (actual time=22964.888..24057.948 rows=158380 loops=3)
            Group Key: stock.s_i_id, stock.s_w_id
            -> Sort (cost=2032514.25..2849543.04 rows=6811838 width=16) (actual time=22964.795..23504.293 rows=2155500 loops=3)
              Sort Key: stock.s_i_id, stock.s_w_id
              Sort Method: external merge Disk: 59152kB
              -> Hash Join (cost=19602.60..1020556.83 rows=6811838 width=16) (actual time=6866.350..21299.395 rows=2155500 loops=3)
                Hash Cond: (order_line.ol_i_id = stock.s_i_id)
                -> Parallel Seq Scan on order_line (cost=0.00..483543.28 rows=3553473 width=8) (actual time=13.733..13475.298 rows=1431855 loops=3)
                  Filter: (date_part('second',::text, ol_delivery_d) > '50'::double precision)
                  Rows Removed by Filter: 7899868
                -> Hash (cost=192098.69..192098.69 rows=202000 width=16) (actual time=6699.824..6699.824 rows=158380 loops=3)
                  Buckets: 131072 Batches: 4 Memory Usage: 2774kB
                  -> Nested Loop (cost=0.43..192098.69 rows=202000 width=16) (actual time=2.554..6640.914 rows=158380 loops=3)
                    -> Seq Scan on item (cost=0.00..2789.00 rows=2020 width=4) (actual time=0.064..40.494 rows=1503 loops=3)
                      Filter: (i_data ~ 'c% '::text)
                      Rows Removed by Filter: 98497
                    -> Index Scan using ix_stock on stock (cost=8.43..98.24 rows=347 width=12) (actual time=1.690..4.364 rows=100 loops=4589)
                      Index Cond: (s_i_id = item.i_id)
            -> Hash (cost=372.23..372.23 rows=59 width=55) (actual time=13.218..13.218 rows=396 loops=1)
              Buckets: 1024 Batches: 1 Memory Usage: 43kB
              -> Hash Join (cost=12.14..372.23 rows=59 width=55) (actual time=1.376..13.080 rows=396 loops=1)
                Hash Cond: (supplier.su_nationkey = nation.n_nationkey)
                -> Seq Scan on supplier (cost=0.00..322.00 rows=10000 width=59) (actual time=0.864..11.210 rows=10000 loops=1)
                -> Hash (cost=12.12..12.12 rows=1 width=4) (actual time=0.688..0.688 rows=1 loops=1)
                  Buckets: 1024 Batches: 1 Memory Usage: 9kB
                  -> Seq Scan on nation (cost=0.00..12.12 rows=1 width=4) (actual time=0.674..0.677 rows=1 loops=1)
                    Filter: (n_name = 'GERMANY'::bpchar)
                    Rows Removed by Filter: 24
Planning time: 78.189 ms
Execution time: 24800.385 ms
```

Figura 24: *Explain Analyze query A.1.*

4.1.1.1 Índices

Como primeira tentativa, o grupo decidiu investir algum tempo na criação de índices. Assim sendo, e tal como já foi mostrado na Figura 23, o TPCC já possui vários índices criados. Assim sendo, e atendendo à estrutura da *query*, o grupo decidiu proceder à criação de mais um índice, **ind_i.data**, por forma a facilitar a procura em “i.data LIKE ‘b’%”. Para além disso, foi alterado no ficheiro *postgresql.conf* o uso de *seqscan* para *off*, para assim forçar o motor a utilizar índices.

```
Sort (cost=300507.70..3005500.28 rows=200 width=51) (actual time=10886.090..10886.105 rows=147 loops=1)
  Sort Key: supplier.su_name
  Sort Method: quicksort Memory: 42kB
  => Hash Join (cost=3005252.96..3005500.13 rows=200 width=51) (actual time=10884.675..10885.969 rows=147 loops=1)
    Hash Cond: (supplier.su_suppkey = "ANY_subquery".nod)
    => Nested Loop (cost=11.52..255.42 rows=400 width=55) (actual time=0.121..1.382 rows=396 loops=1)
      => Index Scan using ind_n_name on nation (cost=0.14..0.15 rows=1 width=4) (actual time=0.026..0.029 rows=1 loops=1)
        Index Cond: (n_name = 'GERMANY'::bpchar)
      => Bitmap Heap Scan on supplier (cost=11.39..243.27 rows=400 width=59) (actual time=0.093..1.288 rows=396 loops=1)
        Recheck Cond: (su_nationkey = nation.n_nationkey)
        Heap Blocks: exact=191
        => Bitmap Index Scan on ind_su_nationkey (cost=0.00..11.29 rows=400 width=8) (actual time=0.058..0.059 rows=396 loops=1)
          Index Cond: (su_nationkey = nation.n_nationkey)
    => Hash (cost=300338.94..3003238.94 rows=200 width=4) (actual time=10884.478..10884.478 rows=3662 loops=1)
      Buckets: 4096 (originally 1024) Batches: 1 (originally 1) Memory Usage: 161kB
      => HashAggregate (cost=3005236.94..3005238.94 rows=200 width=4) (actual time=10883.216..10883.921 rows=3662 loops=1)
        Group Key: "ANY_subquery".nod
        => Subquery Scan on "ANY_subquery" (cost=2653768.93..3002736.94 rows=1000000 width=4) (actual time=9970.185..10880.492 rows=5323 loops=1)
          => Finalize GroupAggregate (cost=2653768.93..2992736.94 rows=1000000 width=16) (actual time=9970.184..10879.848 rows=5323 loops=1)
            Group Key: stock.s_i_id, stock.s_w_id
            Filter: ((i2 = stock.s_quantity) > sum(order_line.ol_quantity))
            Rows Removed by Filter: 144977
            => Gather Merge (cost=2653768.93..2992736.94 rows=2000000 width=20) (actual time=9959.632..10844.658 rows=150300 loops=1)
              Workers Planned: 2
              Workers Launched: 2
              => Partial GroupAggregate (cost=2652768.91..2730807.29 rows=1000000 width=20) (actual time=9959.997..10625.558 rows=50100 loops=3)
                Group Key: stock.s_i_id, stock.s_w_id
                => Sort (cost=2652768.91..2669798.90 rows=6811838 width=16) (actual time=9899.980..10169.705 rows=2155500 loops=3)
                  Sort Key: stock.s_i_id, stock.s_w_id
                  Sort Method: external sort Disk: 74792kB
                  => Merge Join (cost=25.39..1646811.40 rows=6811838 width=16) (actual time=36.232..6983.490 rows=2155500 loops=3)
                    Merge Cond: (order_line.ol_i_id = stock.s_i_id)
                    => Nested Loop (cost=4.56..477479.37 rows=69932 width=12) (actual time=0.295..1040.746 rows=21555 loops=3)
                      => Parallel Index Scan using pk_item on item (cost=0.29..3667.13 rows=642 width=4) (actual time=0.067..15.026 rows=501 loops=3)
                        Filter: (i.data ~ 'cs'::text)
                        Rows Removed by Filter: 32032
                      => Bitmap Heap Scan on order_line (cost=6.27..561.89 rows=83 width=8) (actual time=0.136..2.031 rows=43 loops=1503)
                        Recheck Cond: (ol_i_id = item.i_id)
                        Filter: (date_part('second':text, ol_delivery_d) > '50'::double precision)
                        Rows Removed by Filter: 213
                        Heap Blocks: exact=173006
                        => Bitmap Index Scan on ix_order_line (cost=0.00..6.25 rows=240 width=8) (actual time=0.051..0.051 rows=256 loops=1503)
                          Index Cond: (ol_i_id = item.i_id)
                      => Index Scan using ix_stock on stock (cost=0.43..1076066.09 rows=10000000 width=12) (actual time=0.019..4184.410 rows=12084068 loops=3)
Planning time: 1.993 ms
Execution time: 10915.482 ms
```

Figura 25: *Explain Analyze query A.1 com seqscan=off.*

Como se pode ver, já foi conseguido uma melhoria de tempo e também se provocou a utilização dos índices que já existiam no TPCC. Os índices desenvolvidos pelo grupo não foram utilizados pelo PostgreSQL, mas foram utilizados os índices já existentes (pk.item, por exemplo). Com isto, também se verifica a diminuição de tempo de sensivelmente 24800ms para 10915ms.

4.1.1.2 Vistas materializadas

Após terem sido realizados alguns testes com esta *query*, percebeu-se que o custo maioritário estava associado à *query* aninhada que se inicia por “select mod(s.i_id * s.w_id, 10000)”. Com isto, o grupo entendeu que seria relevante proceder à criação de uma vista materializada para que o custo da execução da *query* total fosse reduzido em grande parte.

Desta forma, com a criação de uma vista materializada, é possível obter de uma forma simples e rápida o resultado da sub *query* que mencionamos acima. Assim sendo, mantemos na *query* principal a restrição relativa à nação, permitindo assim flexibilidade na execução das interrogações. Entendemos por isso que esta deveria ser a estrutura a utilizar para dar resposta ao problema.

```
CREATE MATERIALIZED VIEW a1_vista1 AS
(select mod(s_i_id * s_w_id, 10000)
from stock, order_line
where s_i_id in
(select i_id
from item
where i_data like 'c%')
and ol_i_id=s_i_id
and extract(second from ol_delivery_d) > 50
group by s_i_id, s_w_id, s_quantity
having 2*s_quantity > sum(ol_quantity));
```

Com isto, a *query* que inicialmente possuímos traduz-se na seguinte interrogação:

```
SELECT su_name, su_address
from supplier, nation
where su_suppkey in
(select * from a1_vista1)
and su_nationkey = n_nationkey
and n_name = 'GERMANY'
order by su_name;
```

O resultado após a criação da vista materializada é bastante satisfatório, com uma redução muito grande relativamente ao *execution time* da *query* sem estar otimizada.

```
QUERY PLAN
-----
Sort (cost=517.05..517.11 rows=22 width=51) (actual time=3.495..3.502 rows=147 loops=1)
  Sort Key: supplier.su_name
  Sort Method: quicksort  Memory: 42kB
  -> Hash Semi Join (cost=155.91..516.56 rows=22 width=51) (actual time=1.085..3.396 rows=147 loops=1)
    Hash Cond: (supplier.su_suppkey = a1_vista1.mod)
    -> Hash Join (cost=12.14..372.23 rows=59 width=51) (actual time=0.031..2.297 rows=396 loops=1)
      Hash Cond: (supplier.su_nationkey = nation.n_nationkey)
      -> Seq Scan on supplier (cost=0.00..322.00 rows=10000 width=59) (actual time=0.009..1.185 rows=10000 loops=1)
      -> Hash (cost=12.12..12.12 rows=1 width=4) (actual time=0.010..0.010 rows=1 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 9kB
        -> Seq Scan on nation (cost=0.00..12.12 rows=1 width=4) (actual time=0.007..0.009 rows=1 loops=1)
          Filter: (n_name = 'GERMANY'::bpchar)
          Rows Removed by Filter: 24
    -> Hash (cost=77.23..77.23 rows=5323 width=4) (actual time=1.016..1.016 rows=5323 loops=1)
      Buckets: 8192  Batches: 1  Memory Usage: 252kB
      -> Seq Scan on a1_vista1 (cost=0.00..77.23 rows=5323 width=4) (actual time=0.007..0.444 rows=5323 loops=1)
Planning time: 0.155 ms
Execution time: 3.544 ms
(18 rows)

tpcc=#
```

Figura 26: *Explain Analyze* da interrogação otimizada com *materialized view*.

De notar a redução gigantesca de tempo de execução, passando dos 24800.385ms iniciais para os atuais 3.544ms. Como se pode também constatar pela análise do plano, de facto a vista materializada criada foi bastante benéfica e foi fundamental para uma redução brusca no tempo que a interrogação é executada.

4.1.1.3 Vistas materializadas e índices

Apesar de já se ter conseguido um resultado bastante satisfatório no que toca à otimização desta *query*, o grupo entendeu que poderia conseguir ainda melhores resultados através da criação de índices nas tabelas que assim o necessitem. Deste modo, decidimos proceder à criação do índice *ind_su_nationkey* na tabela *supplier*, de forma a ser mais rápido a procura no filtro aplicado na interrogação. Por outro lado, foi criado também o índice *ind_n_name* na tabela *nation*. Os resultados obtidos são os seguintes:

```
Sort (cost=401.36..401.73 rows=146 width=51) (actual time=1.555..1.563 rows=147 loops=1)
  Sort Key: supplier.su_name
  Sort Method: quicksort Memory: 42kB
  -> Hash Semi Join (cost=155.15..396.12 rows=146 width=51) (actual time=1.104..1.455 rows=147 loops=1)
    Hash Cond: (supplier.su_suppkey = a1_vista1.mod)
    -> Nested Loop (cost=11.38..248.50 rows=400 width=55) (actual time=0.073..0.373 rows=396 loops=1)
      -> Seq Scan on nation (cost=0.00..1.31 rows=1 width=4) (actual time=0.010..0.014 rows=1 loops=1)
        Filter: (n_name = 'GERMANY'::bpchar)
        Rows Removed by Filter: 24
      -> Bitmap Heap Scan on supplier (cost=11.38..243.27 rows=400 width=50) (actual time=0.060..0.306 rows=396 loops=1)
        Recheck Cond: (su_nationkey = nation.n_nationkey)
        Heap Blocks: exact=191
        -> Bitmap Index Scan on ind_su_nationkey (cost=0.00..11.29 rows=400 width=0) (actual time=0.035..0.035 rows=396 loops=1)
          Index Cond: (su_nationkey = nation.n_nationkey)
    -> Hash (cost=77.23..77.23 rows=5323 width=4) (actual time=1.007..1.007 rows=5323 loops=1)
      Buckets: 8192 Batches: 1 Memory Usage: 252kB
      -> Seq Scan on a1_vista1 (cost=0.00..77.23 rows=5323 width=4) (actual time=0.006..0.446 rows=5323 loops=1)
Planning time: 0.220 ms
Execution time: 1.608 ms
```

Figura 27: Plano execução com aplicação de *materialized view* e índices

De facto ainda foi possível baixar mais este valor, pelo que o grupo entendeu que o resultado obtido era muito satisfatório, tendo sido reduzido de cerca de 24800ms para os exibidos 1.688ms. De facto, a criação da vista materializada foi um fator preponderante na otimização desta interrogação.

No entanto, e como a *materialized view* irá ficar estática no tempo, é necessário atualizar o seu conteúdo. Assim, uma possibilidade seria a utilização de *triggers* quando existir alguma alteração nas tabelas que são utilizadas pela interrogação, ou então através do comando *REFRESH MATERIALIZED VIEW a1_vista1*. Deste modo, será feita a atualização dos dados, uma vez que se fizermos novos *run*, serão gerados novos dados. Ainda que o processo de atualização da vista seja custoso, no nosso entender é mais vantajoso que seja feito em intervalos de tempo, ao invés de serem utilizados os *triggers*.

```
tpcc=# refresh materialized view a1_vista1 ;
REFRESH MATERIALIZED VIEW
Time: 12729.516 ms (00:12.730)
```

Assim, o grupo procurou soluções para a atualização das vistas materializadas, não tendo contudo aprofundado este tópico. Deste modo, a utilização do comando acima apresentado pareceu-nos razoável, dada a natureza das *queries*.

4.1.2 Interrogações analítica A.2 (TPC-H)

Esta *query* indica quantos fornecedores são capazes de fornecer itens com determinados atributos classificados por ordem decrescente. O resultado é agrupado pelo identificador do item.

```
SELECT i_name,
       substr(i_data, 1, 3) as brand,
       i_price,
       count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
FROM stock, item
WHERE i_id = s_i_id
      and i_data not like 'z%'
      and (mod((s_w_id * s_i_id),10000) not in
      (SELECT su_suppkey
       FROM supplier
       WHERE su_comment like '%bean%'))
GROUP BY i_name, substr(i_data, 1, 3), i_price
ORDER BY supplier_cnt DESC;
```

Através do comando *Explain Analyze* começamos por tentar perceber o tempo de execução da *query*, assim como a contagem das linhas e outras informações que poderão ser úteis quando a tentarmos otimizar. Tal como se pode ver na figura seguinte, o tempo de execução da interrogação analítica é de 33.416571 segundos.

```
QUERY PLAN
-----
Sort (cost=1770323.80..1770571.27 rows=98990 width=71) (actual time=33177.754..33190.065 rows=98522 loops=1)
  Sort Key: (count(DISTINCT mod((stock.s_w_id * stock.s_i_id), 10000))) DESC
  Sort Method: external merge  Disk: 5600kB
  -> GroupAggregate (cost=1669945.40..1758046.61 rows=98990 width=71) (actual time=25987.242..33100.598 rows=98522 loops=1)
    Group Key: item.i_name, (substr((item.i_data)::text, 1, 3)), item.i_price
    -> Sort (cost=1669945.40..1682319.17 rows=4949506 width=71) (actual time=25987.139..29377.276 rows=8834353 loops=1)
      Sort Key: item.i_name, (substr((item.i_data)::text, 1, 3)), item.i_price
      Sort Method: external merge  Disk: 466832kB
      -> Hash Join (cost=5730.40..713567.72 rows=4949506 width=71) (actual time=49.121..11352.488 rows=8834353 loops=1)
        Hash Cond: (stock.s_i_id = item.i_id)
        -> Seq Scan on stock (cost=350.03..629896.24 rows=5000006 width=8) (actual time=3.025..3680.044 rows=8966900 loops=1)
          Filter: (NOT (hashed SubPlan 1))
          Rows Removed by Filter: 1033100
          SubPlan 1
            -> Seq Scan on supplier (cost=0.00..347.00 rows=1212 width=4) (actual time=0.011..2.795 rows=1071 loops=1)
              Filter: ((su_comment)::text ~ '%bean% '::text)
              Rows Removed by Filter: 8929
        -> Hash (cost=2789.00..2789.00 rows=98990 width=86) (actual time=45.932..45.932 rows=98522 loops=1)
          Buckets: 32768  Batches: 4  Memory Usage: 3175kB
          -> Seq Scan on item (cost=0.00..2789.00 rows=98990 width=86) (actual time=0.009..20.393 rows=98522 loops=1)
            Filter: (i_data !~ 'z% '::text)
            Rows Removed by Filter: 1478
Planning time: 0.351 ms
Execution time: 33416.571 ms
(24 rows)
```

Figura 28: *Explain Analyze* da interrogação analítica A.2.

Como forma de tentar minimizar este tempo de execução, recorreremos a índices e vistas materializadas, que iremos apresentar em seguida.

4.1.2.1 Índices

Para a utilização dos índices já contidos no *dump* (Figura 23) é necessário desativar a opção *enable_seqscan* do ficheiro *postgresql.conf*. Assim, voltando a correr a *query*, tal como se pode ver na figura seguinte, consegue perceber-se que o *postgres* recorre ao índice *ix_stock* (da tabela *stock* para a coluna *s_i_id*) e ao índice *pk_item* (da tabela *item* para a coluna *i_id*), ou seja, das colunas da primeira cláusula da condição do *WHERE*.

No entanto, a utilização destes índices nesta interrogação acaba até por prejudicar o seu tempo de execução, como se pode comprovar através da figura seguinte (passou de aproximadamente 33 para aproximadamente 35 segundos).

```
QUERY PLAN
-----
Sort (cost=18001976262.63..18001976510.11 rows=98990 width=71) (actual time=35217.289..35229.029 rows=98522 loops=1)
  Sort Key: (count(DISTINCT mod((stock.s_w_id * stock.s_i_id), 10000))) DESC
  Sort Method: external merge  Disk: 5600kB
-> GroupAggregate (cost=10001875884.24..10001963985.44 rows=98990 width=71) (actual time=27879.017..35141.515 rows=98522 loops=1)
  Group Key: item.i_name, (substr((item.i_data)::text, 1, 3)), item.i_price
  -> Sort (cost=10001875884.24..1000188258.00 rows=4949506 width=71) (actual time=27878.898..31367.701 rows=8834353 loops=1)
    Sort Key: item.i_name, (substr((item.i_data)::text, 1, 3)), item.i_price
    Sort Method: external merge  Disk: 466832kB
    -> Hash Join (cost=10000007338.13..10000919506.55 rows=4949506 width=71) (actual time=66.006..13073.614 rows=8834353 loops=1)
      Hash Cond: (stock.s_i_id = item.i_id)
      -> Index Scan using ix_stock on stock (cost=10000000350.47..10000834227.78 rows=5000006 width=8) (actual time=3.394..5337.314 rows=8966900 loops=1)
        Filter: (NOT (hashed SubPlan 1))
        Rows Removed by Filter: 1033100
        SubPlan 1
        -> Seq Scan on supplier (cost=10000000000.00..10000000347.00 rows=1212 width=4) (actual time=0.013..3.114 rows=1071 loops=1)
          Filter: ((su_comment)::text ~ 'bean%')::text
          Rows Removed by Filter: 8929
      -> Hash (cost=4396.29..4396.29 rows=98990 width=86) (actual time=62.147..62.147 rows=98522 loops=1)
        Buckets: 32768  Batches: 4  Memory Usage: 3175kB
        -> Index Scan using pk_item on item (cost=0.29..4396.29 rows=98990 width=86) (actual time=0.011..33.201 rows=98522 loops=1)
          Filter: (i_data !~ 'ZM')::text
          Rows Removed by Filter: 1478
Planning time: 0.411 ms
Execution time: 35291.465 ms
(24 rows)
```

Figura 29: *Explain Analyze* da interrogação analítica A.2 com índices.

O grupo tentou criar novos índices, como o que se encontra em seguida, mas sem sucesso visto que o *postgres* não recorre a eles quando se corre a *query*.

```
CREATE INDEX item_i_data ON
item(i_data);
```

4.1.2.2 Vista Materializada 1

Em seguida, recorreremos a vistas materializadas para a optimização da *query*.

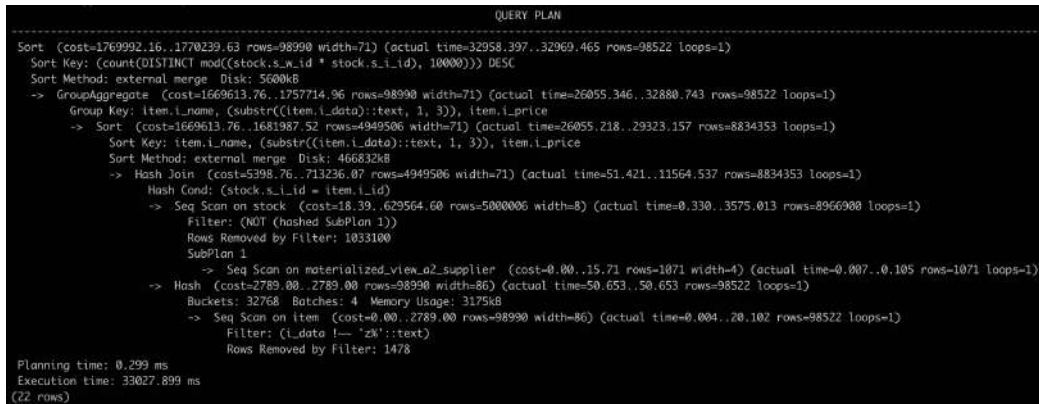
Desta forma, observando a *query* em questão é possível perceber que esta possui uma outra *query* embebida, que pode ser tornada numa vista materializada:

```
CREATE MATERIALIZED VIEW materialized_view_A2_supplier AS
SELECT su_suppkey
FROM supplier
WHERE su_comment like '%bean%';
```


Desta forma, o código para a execução da interrogação passa a ser o seguinte:

```
SELECT i_name,
       substr(i_data, 1, 3) as brand,
       i_price,
       count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
FROM stock, item
WHERE i_id = s_i_id
      and i_data not like 'z%'
      and (mod((s_w_id * s_i_id),10000) not in
           (SELECT * FROM materialized_view_A2_supplier))
GROUP BY i_name, substr(i_data, 1, 3), i_price
ORDER BY supplier_cnt DESC;
```

No entanto, após a execução da *query* é possível perceber que não houve um impacto significativo no tempo de execução, diminuindo somente de 33.416571 segundos para 33.027899 segundos, o que se encontra muito aquém do desejado.



```
QUERY PLAN
Sort (cost=1769992.16..1770239.63 rows=98990 width=71) (actual time=32958.397..32969.465 rows=98522 loops=1)
  Sort Key: (count(DISTINCT mod((stock.s_w_id * stock.s_i_id), 10000))) DESC
  Sort Method: external merge  Disk: 5600kB
  -> GroupAggregate (cost=1669613.76..1757714.96 rows=98990 width=71) (actual time=26055.346..32880.743 rows=98522 loops=1)
    Group Key: item.i_name, (substr((item.i_data)::text, 1, 3)), item.i_price
    -> Sort (cost=1669613.76..1681987.52 rows=4949506 width=71) (actual time=26055.218..29323.157 rows=8834353 loops=1)
      Sort Key: item.i_name, (substr((item.i_data)::text, 1, 3)), item.i_price
      Sort Method: external merge  Disk: 466832kB
      -> Hash Join (cost=5398.76..713236.07 rows=4949506 width=71) (actual time=51.421..11564.537 rows=8834353 loops=1)
        Hash Cond: (stock.s_i_id = item.i_id)
        -> Seq Scan on stock (cost=18.39..629564.60 rows=5000006 width=8) (actual time=0.330..3575.013 rows=8966900 loops=1)
          Filter: (NOT (hashed SubPlan 1))
          Rows Removed by Filter: 1033100
          SubPlan 1
            -> Seq Scan on materialized_view_a2_supplier (cost=0.00..15.71 rows=1071 width=4) (actual time=0.007..0.105 rows=1071 loops=1)
        -> Hash (cost=2789.00..2789.00 rows=98990 width=86) (actual time=50.653..50.653 rows=98522 loops=1)
          Buckets: 32768  Batches: 4  Memory Usage: 3175kB
          -> Seq Scan on item (cost=0.00..2789.00 rows=98990 width=86) (actual time=0.004..20.102 rows=98522 loops=1)
            Filter: (i_data != 'z% '::text)
            Rows Removed by Filter: 1478
Planning time: 0.299 ms
Execution time: 33027.899 ms
(22 rows)
```

Figura 30: *Explain Analyze* da interrogação analítica A.2 com a vista materializada 1.

4.1.2.3 Vista Materializada 2

Após uma análise dos resultados da Figura 28 percebemos que operação “*count(distinct (mod((s_w_id * s_i_id),10000)))*” consome bastante tempo.

Desta forma, decidimos criar uma segunda vista materializada que efetua o cálculo do “*mod((s_w_id * s_i_id),10000)*”, que pode assim ser utilizado em dois momentos na *query*.


```
CREATE MATERIALIZED VIEW materialized_view_A2_mod_stock_optimized AS
SELECT s_i_id, s_w_id
, mod((s_w_id * s_i_id),10000) as calculo
FROM stock;
```

Culminando assim na seguinte interrogação:

```
SELECT i_name,
       substr(i_data, 1, 3) as brand,
       i_price,
       count(distinct calculo) as supplier_cnt
FROM materialized_view_A2_mod_stock_optimized, item
WHERE i_id = s_i_id
      and i_data not like 'z%'
      and calculo not in
(select su_suppkey
from supplier
where su_comment like '%bean%')
GROUP BY i_name, substr(i_data, 1, 3), i_price
ORDER BY supplier_cnt DESC;
```

No entanto, os resultados obtidos também ficaram aquém do expectável, visto que o tempo de execução da *query* apenas diminui de 33.416571 segundos para 31.5184245 segundos.

```
QUERY PLAN
-----
Sort (cost=295063.92..1295311.39 rows=98990 width=71) (actual time=31491.317..31503.369 rows=98522 loops=1)
  Sort Key: (count(distinct materialized_view_a2_mod_stock_optimized.calculo)) Desc
  Sort Method: external merge  Disk: 5680K
  -> GroupAggregate (cost=1219433.99..1282786.72 rows=98990 width=71) (actual time=24511.289..31400.821 rows=98522 loops=1)
    Group Key: item.i_name, (substr((item.i_data)::text, 1, 3)), item.i_price
    -> Sort (cost=1219433.99..1231807.56 rows=4949431 width=67) (actual time=24511.141..27935.324 rows=8834353 loops=1)
      Sort Key: item.i_name, (substr((item.i_data)::text, 1, 3)), item.i_price
      Sort Method: external merge  Disk: 452240K
      -> Hash Join (cost=5730.40..263072.18 rows=4949431 width=67) (actual time=49.942..9994.837 rows=8834353 loops=1)
        Hash Cond: (materialized_view_a2_mod_stock_optimized.s_i_id = item.i_id)
        -> Seq Scan on materialized_view_a2_mod_stock_optimized (cost=350.03..179403.28 rows=4999930 width=8) (actual time=2.868..2388.817 rows=8966200 loops=1)
          Filter: (NOT (hashed SubPlan 1))
          Rows Removed by Filter: 1033100
          SubPlan 1
            -> Seq Scan on supplier (cost=0.00..347.00 rows=1212 width=4) (actual time=0.008..2.592 rows=1071 loops=1)
              Filter: ((su_comment)::text = 'beans'::text)
              Rows Removed by Filter: 8929
        -> Hash (cost=2789.00..2789.00 rows=98990 width=86) (actual time=46.758..46.758 rows=98522 loops=1)
          Buckets: 32768  Batches: 4  Memory Usage: 3176K
          -> Seq Scan on item (cost=0.00..2789.00 rows=98990 width=86) (actual time=0.009..28.788 rows=98522 loops=1)
            Filter: (i_data !~ 'z'::text)
            Rows Removed by Filter: 1478
Planning time: 0.479 ms
Execution time: 31584.245 ms
(24 rows)
```

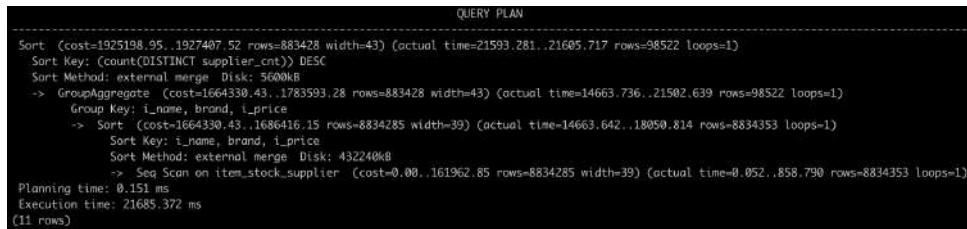
Figura 31: *Explain Analyze* da interrogação analítica A.2 com a vista materializada 2.

4.1.2.4 Vista Materializada 3

Numa terceira tentativa, tentamos colocar um pouco mais de lógica na vista materializada, culminando assim nas seguintes interrogações:

```
CREATE MATERIALIZED VIEW item_stock_supplier AS
SELECT i_name,
       substr(i_data, 1, 3) as brand,
       i_price,
       (mod((s_w_id * s_i_id),10000)) as supplier_cnt
FROM stock, item
WHERE i_id = s_i_id
     and i_data not like 'z%'
     and (mod((s_w_id * s_i_id),10000) not in
        (SELECT su_suppkey
         FROM supplier
         WHERE su_comment like '%bean%'));

SELECT i_name, brand, i_price,
       count(distinct supplier_cnt) as supplier_cnt
FROM item_stock_supplier
GROUP BY 1,2,3
ORDER BY 4 DESC;
```



```
QUERY PLAN
-----
Sort (cost=1925198.95..1927407.52 rows=883428 width=43) (actual time=21593.281..21605.717 rows=98522 loops=1)
  Sort Key: (count(DISTINCT supplier_cnt)) DESC
  Sort Method: external merge  Disk: 5600kB
-> GroupAggregate (cost=1664330.43..1783593.28 rows=883428 width=43) (actual time=14663.736..21502.639 rows=98522 loops=1)
  Group Key: i_name, brand, i_price
  -> Sort (cost=1664330.43..1686416.15 rows=8834285 width=39) (actual time=14663.642..18050.814 rows=8834353 loops=1)
    Sort Key: i_name, brand, i_price
    Sort Method: external merge  Disk: 432240kB
    -> Seq Scan on item_stock_supplier (cost=0.00..161962.85 rows=8834285 width=39) (actual time=0.052..858.790 rows=8834353 loops=1)
Planning time: 0.151 ms
Execution time: 21685.372 ms
(11 rows)
```

Figura 32: *Explain Analyze* da interrogação analítica A.2 com a vista materializada 3.

De facto, o tempo de execução reduziu apenas para aproximadamente 21.6 segundos, o que significa que o *count distinct* é realmente o grande gargalo desta *query*. No entanto, para otimizar este comando, devido aos JOINS existentes seria necessário colocar praticamente toda a *query* na vista materializada.

4.1.2.5 Vista Materializada 4

Numa quarta e última tentativa, o grupo colocou então grande parte da *query* na vista materializada, deixando somente os comandos *GROUP BY* e *ORDER BY* de fora. No entanto, apesar do tempo de execução reduzir bastante, de 33.416571 segundos para 0.075688 segundos, como se pode ver através da figura seguinte, não acreditamos que esta seja uma boa opção visto que tira praticamente toda a flexibilidade à *query*.

```
CREATE MATERIALIZED VIEW materialized_view_A2 AS
SELECT i_name,
       substr(i_data, 1, 3) as brand,
       i_price,
       count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
FROM stock, item
WHERE i_id = s_i_id
     and i_data not like 'z%'
     and (mod((s_w_id * s_i_id),10000) not in
        (SELECT su_suppkey
         FROM supplier
         WHERE su_comment like '%bean%'))
GROUP BY i_name, substr(i_data, 1, 3), i_price;

SELECT * FROM materialized_view_A2
ORDER BY supplier_cnt DESC;
```

```
QUERY PLAN
-----
Sort  (cost=7632.56..7736.17 rows=41445 width=152) (actual time=57.510..69.745 rows=98522 loops=1)
  Sort Key: supplier_cnt DESC
  Sort Method: external merge  Disk: 5600kB
  -> Seq Scan on materialized_view_a2  (cost=0.00..1335.45 rows=41445 width=152) (actual time=0.008..11.040 rows=98522 loops=1)
Planning time: 0.083 ms
Execution time: 74.688 ms
(6 rows)
```

Figura 33: *Explain Analyze* da interrogação analítica A.2 com a vista materializada 4.

4.1.3 Interrogações analítica A.3 (TPC-H)

Esta interrogação ordena os países europeus, por ordem decrescente, consoante o valor da soma das suas *order_line*.

```
select n_name,  
sum(ol_amount) as revenue  
from customer, orders, order_line, stock, supplier, nation, region  
where c_id = o_c_id  
and c_w_id = o_w_id  
and c_d_id = o_d_id  
and 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 ascii(substr(c_state,1,1))-ascii('a') = su_nationkey  
and su_nationkey = n_nationkey  
and n_regionkey = r_regionkey  
and r_name = 'EUROPE'  
group by n_name  
order by revenue desc;
```

Iniciamos a interpretação da *Query* do mesmo modo que todas as anteriores, executando-a com as opções *EXPLAIN ANALYZE* de maneira a ser possível observar o plano de execução, bem como o seu tempo de execução. Assim, verificamos que a interrogação demora cerca de 101677.317ms (101s), que são 1min e 41s, a terminar a execução. Devido ao seu elevado tempo de execução, é de extrema importância reduzir ao máximo este tempo.

```

QUERY PLAN
-----
Sort (cost=775118.84..775118.00 rows=25 width=136) (actual time=101665.735..101665.736 rows=5 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC
  Sort Method: quicksort  Memory: 25kB
-> Finalize GroupAggregate (cost=775095.43..775110.26 rows=25 width=136) (actual time=101656.064..101665.721 rows=5 loops=1)
  Group Key: nation.n_name
  -> Gather Merge (cost=775095.43..775117.57 rows=96 width=136) (actual time=101624.100..101675.102 rows=15 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    -> Partial GroupAggregate (cost=774095.41..774111.70 rows=25 width=136) (actual time=101623.110..101632.485 rows=5 loops=3)
      Group Key: nation.n_name
      -> Sort (cost=774016.41..774100.70 rows=2141 width=197) (actual time=101620.735..101629.238 rows=25077 loops=3)
        Sort Key: nation.n_name
        Sort Method: quicksort  Memory: 2742kB
        -> Hash Join (cost=475.07..773976.97 rows=2141 width=197) (actual time=44.622..101574.465 rows=25077 loops=3)
          Hash Cond: ((mod((stock.s.w_id * stock.s.i_id), 10000) = supplier.su_suppkey) AND (nation.n.nationkey = supplier.su_nationkey))
          -> Nested Loop (cost=3.07..772849.40 rows=53510 width=122) (actual time=0.391..101125.710 rows=62655 loops=2)
            Join Filter: (customer.c.w_id = stock.s.w_id)
            -> Nested Loop (cost=3.43..324830.53 rows=53323 width=130) (actual time=6.227..10296.466 rows=62655 loops=3)
              Join Filter: ((customer.c.w_id = order_line.ol.w_id) AND (customer.c.d_id = order_line.ol.d_id))
              -> Nested Loop (cost=2.87..258667.77 rows=6270 width=131) (actual time=4.374..4180.914 rows=79761 loops=3)
                -> Hash Join (cost=2.44..223458.44 rows=6250 width=123) (actual time=2.802..3657.400 rows=73520 loops=3)
                  Hash Cond: ((ascii(substr(customer.c.state), text, 1, 1)) - 97) = nation.n.nationkey)
                  -> Parallel Seq Scan on customer (cost=0.80..212466.00 rows=125000 width=15) (actual time=0.597..2436.886 rows=100000 loops=3)
                  -> Hash (cost=2.45..2.42 rows=1 width=100) (actual time=0.607..0.607 rows=5 loops=3)
                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                    -> Hash Join (cost=1.87..2.42 rows=1 width=100) (actual time=0.596..0.602 rows=5 loops=2)
                      Hash Cond: (nation.n.regionkey = region.r.regionkey)
                      -> Seq Scan on nation (cost=0.00..1.25 rows=25 width=112) (actual time=0.270..0.272 rows=25 loops=3)
                      -> Hash (cost=1.00..1.00 rows=1 width=4) (actual time=0.305..0.306 rows=1 loops=3)
                        Buckets: 1024  Batches: 1  Memory Usage: 9kB
                        -> Seq Scan on region (cost=0.00..1.00 rows=1 width=4) (actual time=0.300..0.308 rows=1 loops=3)
                          Filter: (r_name = 'EUROPE')::bpchar)
                          Rows Removed by Filter: 4
                -> Index Scan using ix_orders on orders (cost=0.43..4.34 rows=1 width=16) (actual time=0.013..0.014 rows=1 loops=220560)
                  Index Cond: ((o.w_id = customer.c.w_id) AND (o.d_id = customer.c.d_id) AND (o.c_id = customer.c_id))
              -> Index Scan using pk_order_line on order_line (cost=0.16..11.64 rows=8 width=19) (actual time=0.105..0.201 rows=8 loops=221204)
                Index Cond: ((ol.w_id = orders.o.w_id) AND (ol.d_id = orders.o.d_id) AND (ol.c_id = orders.o.c_id))
            -> Index Only Scan using pk_stock on stock (cost=0.43..5.37 rows=3 width=0) (actual time=0.130..0.130 rows=1 loops=107056)
              Index Cond: ((s.w_id = order_line.ol.w_id) AND (s.i_id = order_line.ol.i_id))
              Heap Fetches: 626147
          -> Hash (cost=322.08..322.08 rows=10000 width=0) (actual time=7.294..7.294 rows=10000 loops=3)
            Buckets: 10384  Batches: 1  Memory Usage: 519kB
            -> Seq Scan on supplier (cost=0.80..322.80 rows=10000 width=0) (actual time=0.277..4.903 rows=10000 loops=3)

```

Planning time: 10.353 ms
Execution time: 101677.317 ms

Figura 34: *Explain Analyze* da interrogação analítica A.3

De maneira a perceber melhor o impacto deste plano de execução, utilizamos uma ferramenta que nos foi aconselhada, utilizamos o *website* <https://explain.depesz.com/> que nos dá, de uma forma mais intuitiva, o plano de execução.

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.015	101.665.736	+5.0	5	1	→ Sort (cost=775.118.84.775.118.90 rows=25 width=136) (actual time=101.665.735.101.665.736 rows=5 loops=1) Sort Key: (sum(order_line.ol_amount)) DESC Sort Method: quicksort Memory: 25kB
2.	0.000	101.665.721	+5.0	5	1	→ Finalize GroupAggregate (cost=775.095.43.775.118.26 rows=25 width=136) (actual time=101.656.864.101.665.721 rows=5 loops=1) Group Key: nation.n_name
3.	0.000	101.675.182	+3.3	15	1	→ Gather Merge (cost=775.095.43.775.117.57 rows=50 width=136) (actual time=101.654.186.101.675.182 rows=15 loops=1) Workers Planned: 2 Workers Launched: 2
4.	27.741	304.897.455	+5.0	5	3	→ Partial GroupAggregate (cost=774.095.41.774.111.78 rows=25 width=136) (actual time=101.823.110.101.832.485 rows=5 loops=3) Group Key: nation.n_name
5.	146.319	304.869.714	+11.7	25,077	3	→ Sort (cost=774.095.41.774.180.76 rows=25 width=107) (actual time=101.620.735.101.623.238 rows=25,077 loops=3) Sort Key: nation.n_name Sort Method: quicksort Memory: 2742kB
6.	1.324.383	304.723.595	+11.7	25,077	3	→ Hash Join (cost=475.87.773.975.97 rows=2,141 width=107) (actual time=64.622.101.574.465 rows=25,077 loops=3) Hash Cond: ((mod((stocks_w_id * stocks_i_id), 10000) = supplier.su_supply) AND (nation.n_nationkey = supplier.su_nationkey)) Join Filter: (customer.c_w_id = order_line.ol_w_id) AND (customer.c_d_id = order_line.ol_d_id)
7.	1.062.152	303.877.130	+11.7	826.855	3	→ Nested Loop (cost=3.87.772.048.40 rows=53.516 width=122) (actual time=8.381.101.125.710 rows=826.855 loops=3) Join Filter: (customer.c_w_id = stocks_s_w_id)
8.	868.572	57.889.398	+11.8	626.650	3	→ Nested Loop (cost=3.43.324.404.53 rows=53.323 width=130) (actual time=6.227.19.296.466 rows=626.650 loops=3) Join Filter: ((customer.c_w_id = order_line.ol_w_id) AND (customer.c_d_id = order_line.ol_d_id))
9.	282.702	12.542.742	+11.8	73.781	3	→ Nested Loop (cost=2.87.250.687.77 rows=6.270 width=111) (actual time=4.174.4.180.814 rows=73.781 loops=3)
10.	1.859.721	9.172.200	+11.8	73.520	3	→ Hash Join (cost=2.44.223.408.44 rows=6.290 width=128) (actual time=2.802.3.057.400 rows=73.520 loops=3) Hash Cond: ((ascii(substr((customer.c_state)::text, 1, 1)) - 97) = nation.n_nationkey) Join Filter: ((customer.c_w_id = order_line.ol_w_id) AND (customer.c_d_id = order_line.ol_d_id))
11.	7.310.659	7.310.659	+1.2	1,000,000	3	→ Parallel Seq Scan on customer (cost=0.00.212.466.00 rows=1,250,000 width=15) (actual time=0.597.2.426.886 rows=1,000,000 loops=3)
12.	0.015	1.821	+5.0	5	3	→ Hash (cost=2.43.2.43 rows=1 width=108) (actual time=0.607.0.807 rows=5 loops=3) Buckets: 1624 Batches: 1 Memory Usage: 9kB
13.	0.072	1.806	+5.0	5	3	→ Hash Join (cost=1.07.2.43 rows=1 width=108) (actual time=0.596.0.602 rows=5 loops=3) Hash Cond: (nation.n_regionkey = region.r_regionkey)
14.	0.816	0.816	+1.0	25	3	→ Seq Scan on nation (cost=0.00.1.25 rows=25 width=112) (actual time=0.270.0.272 rows=25 loops=3)
15.	0.018	0.918	+1.0	1	3	→ Hash (cost=1.06.1.06 rows=1 width=4) (actual time=0.305.0.306 rows=1 loops=3) Buckets: 1024 Batches: 1 Memory Usage: 9kB
16.	0.900	0.900	+1.0	1	3	→ Seq Scan on region (cost=0.00.1.06 rows=1 width=4) (actual time=0.300.0.300 rows=1 loops=3) Filter: (r_name = 'EUROPE')::bpchar Rows Removed by Filter: 4
17.	3.087.840	3.087.840	+1.0	1	220,560	→ Index Scan using ix_orders on orders (cost=0.43.4.34 rows=1 width=16) (actual time=0.013.0.014 rows=1 loops=220,560) Index Cond: ((o_w_id = customer.c_w_id) AND (o_d_id = customer.c_d_id) AND (o_c_id = customer.c_id))
18.	44.478.084	44.478.084	+1.0	8	221,294	→ Index Scan using pk_order_line on order_line (cost=0.56.11.64 rows=8 width=19) (actual time=0.185.0.201 rows=8 loops=221,294) Index Cond: ((ol_w_id = orders.o_w_id) AND (ol_d_id = orders.o_d_id) AND (ol_c_id = orders.o_id))
19.	244.395.580	244.395.580	+3.0	1	1,879,966	→ Index Only Scan using pk_stock on stock (cost=0.43.8.37 rows=3 width=8) (actual time=0.130.0.130 rows=1 loops=1,879,966) Index Cond: ((s_w_id = order_line.ol_w_id) AND (s_i_id = order_line.ol_i_id)) Heap Fetches: 826,147
20.	6.933	21.882	+1.0	10,000	3	→ Hash (cost=322.00.322.00 rows=10,000 width=8) (actual time=7.294.7.294 rows=10,000 loops=3) Buckets: 16384 Batches: 1 Memory Usage: 519kB
21.	14.949	14.949	+1.0	10,000	3	→ Seq Scan on supplier (cost=0.00.322.00 rows=10,000 width=8) (actual time=0.277.4.983 rows=10,000 loops=3)
Planning time : 19.353 ms						
Execution time : 101.677.317 ms						

Figura 35: Plano de execução da *query*

HTML

SOURCE

STATS

Per node type stats

node type	count	sum of times	% of query
Finalize GroupAggregate	1	0.000 ms	0.0 %
Gather Merge	1	0.000 ms	0.0 %
Hash	3	6.966 ms	0.0 %
Hash Join	3	3,184.176 ms	3.1 %
Index Only Scan	1	244,395.580 ms	240.4 %
Index Scan	2	47,565.924 ms	46.8 %
Nested Loop	3	2,243.426 ms	2.2 %
Parallel Seq Scan	1	7,310.658 ms	7.2 %
Partial GroupAggregate	1	27.741 ms	0.0 %
Seq Scan	3	16.665 ms	0.0 %
Sort	2	146.334 ms	0.1 %

Per table stats

Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
customer	1	7,310.658 ms	7.2 %
Parallel Seq Scan	1	7,310.658 ms	100.0 %
nation	1	0.816 ms	0.0 %
Seq Scan	1	0.816 ms	100.0 %
order_line	1	44,478.084 ms	43.7 %
Index Scan	1	44,478.084 ms	100.0 %
orders	1	3,087.840 ms	3.0 %
Index Scan	1	3,087.840 ms	100.0 %
region	1	0.900 ms	0.0 %
Seq Scan	1	0.900 ms	100.0 %
stock	1	244,395.580 ms	240.4 %
Index Only Scan	1	244,395.580 ms	100.0 %
supplier	1	14.949 ms	0.0 %
Seq Scan	1	14.949 ms	100.0 %

Figura 36: *Stats* da interrogação analítica A.3

Como é possível ver na Fig 35, existem muitas muitas zonas onde a execução está a ser problemática. É nessa zona que vamos tentar atuar. Na Fig 36, é possível ver em termos percentuais os pontos críticos da *query*.

O resultado de executar a *query* é o seguinte.

<i>n_name</i>	<i>revenue</i>
FRANCE	23519391.36
ROMANIA	23453148.62
RUSSIA	22811911.78
GERMANY	21675945.57
UNITED KINGDOM	21556976.32

Tabela 25: Resultados da *Query*

4.1.3.1 Índices

A nossa abordagem inicial foi tentar otimizar o desempenho através da criação de índices que pudessem ser úteis e que não existem desde já (Algu-

mas tabelas já possuíam índices nas suas chaves primárias). Assim, tentamos perceber que índices poderiam ser úteis na *query*, como por exemplo o índice sobre o *n_name* que é usado para fazer *GROUP BY*. Os índices que decidimos introduzir foram os seguintes.

```
CREATE INDEX q3_supplier ON public.supplier USING btree (su_suppkey);
```

```
CREATE INDEX q3_su_nationk ON public.supplier USING btree (su_nationkey);
```

```
CREATE INDEX q3_n_name ON public.nation USING btree (n_name);
```

```
CREATE INDEX pk_nation ON public.nation USING btree (n_nationkey, n_name);
```

```
CREATE INDEX q3_ol ON public.order_line USING btree (ol_o_id, ol_w_id, ol_d_id, ol_w_id, ol_o_id);
```

```
CREATE INDEX pk_orders ON public.orders USING btree (o_w_id, o_d_id, o_id);
```

Após a criação destes índices, voltamos a ver qual foi o tempo de execução da *query*, dado que é o nosso objetivo, e o resultado foi o seguinte.

```
Sort (cost=775118.84..775118.88 rows=25 width=136) (actual time=78974.341..78974.342 rows=3 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC
  Sort Method: quicksort Memory: 23kB
  -> Finalize GroupAggregate (cost=775895.43..775118.26 rows=25 width=136) (actual time=78965.444..78974.323 rows=3 loops=1)
    Group Key: nation.n_name
    -> Gather Merge (cost=775895.43..775117.57 rows=50 width=136) (actual time=78962.535..78968.128 rows=15 loops=1)
      Workers Planned: 2
      Workers Launched: 2
      -> Partial GroupAggregate (cost=774095.41..774111.78 rows=25 width=136) (actual time=78955.831..78964.631 rows=3 loops=3)
        Group Key: nation.n_name
        -> Sort (cost=774095.41..774108.78 rows=2141 width=107) (actual time=78954.395..78956.581 rows=23877 loops=3)
          Sort Key: nation.n_name
          Sort Method: quicksort Memory: 2732kB
          -> Hash Join (cost=475.87..773976.97 rows=2141 width=107) (actual time=14.823..78917.790 rows=25877 loops=3)
            Hash Cond: ((mod((stock.s_w_id * stock.s_i_id), 100000) = supplier.su_suppkey) AND (nation.n_nationkey = supplier.su_nationkey))
            -> Nested Loop (cost=3.87..772948.40 rows=52219 width=122) (actual time=8.760..78318.867 rows=42653 loops=3)
              Join Filter: (customer.c_w_id = stock.s_w_id)
              -> Nested Loop (cost=3.43..324484.53 rows=53323 width=130) (actual time=0.263..13980.250 rows=42655 loops=3)
                Join Filter: ((customer.c_w_id = order_line.ol_w_id) AND (customer.c_d_id = order_line.ol_d_id))
                -> Nested Loop (cost=1.87..230897.77 rows=8276 width=111) (actual time=0.265..2886.512 rows=7761 loops=3)
                  -> Hash Join (cost=2.44..223468.44 rows=4329 width=123) (actual time=0.358..1992.125 rows=73538 loops=3)
                    Hash Cond: ((ascii(substr(customer.c_state),text,1,11) - 97) = nation.n_nationkey)
                    -> Parallel Seq Scan on customer (cost=0.00..212466.00 rows=1250000 width=13) (actual time=0.031..1410.666 rows=1088000 loops=3)
                    -> Hash (cost=2.43..2.43 rows=1 width=108) (actual time=0.853..0.853 rows=5 loops=3)
                      Buckets: 1024 Batches: 1 Memory Usage: 96B
                  -> Hash Join (cost=1.97..2.41 rows=1 width=108) (actual time=0.643..0.648 rows=3 loops=3)
                    Hash Cond: (nation.n_regionkey = region.r_regionkey)
                    -> Seq Scan on nation (cost=0.00..1.25 rows=25 width=112) (actual time=0.814..0.817 rows=25 loops=3)
                    -> Hash (cost=1.96..1.96 rows=1 width=4) (actual time=0.815..0.815 rows=1 loops=3)
                      Buckets: 1024 Batches: 1 Memory Usage: 96B
                  -> Seq Scan on region (cost=0.88..1.86 rows=1 width=4) (actual time=0.818..0.811 rows=1 loops=3)
                    Filter: (r_name = 'EUROPE'::bpchar)
                    Rows Removed by Filter: 4
                -> Index Scan using ix_orders on orders (cost=0.43..4.34 rows=1 width=6) (actual time=0.889..0.818 rows=1 loops=228580)
                  Index Cond: ((o_w_id = customer.c_w_id) AND (o_d_id = customer.c_d_id) AND (o_r_id = customer.c_id))
              -> Index Scan using pk_order_line on order_line (cost=0.56..11.64 rows=8 width=19) (actual time=0.110..0.121 rows=8 loops=221284)
                Index Cond: ((ol_w_id = orders.o_w_id) AND (ol_d_id = orders.o_d_id) AND (ol_o_id = orders.o_id))
            -> Index Only Scan using pk_stock on stock (cost=0.43..0.37 rows=3 width=8) (actual time=0.185..0.185 rows=1 loops=1879986)
              Index Cond: ((s_w_id = order_line.ol_w_id) AND (s_i_id = order_line.ol_i_id))
              Heap Fetches: 832478
          -> Hash (cost=322.80..322.80 rows=10880 width=8) (actual time=4.919..4.919 rows=10880 loops=3)
            Buckets: 16384 Batches: 1 Memory Usage: 519kB
            -> Seq Scan on supplier (cost=0.00..322.00 rows=10880 width=8) (actual time=0.824..2.915 rows=10880 loops=3)
```

Planning time: 15.841 ms
Execution time: 78980.371 ms

Figura 37: *Explain Analyze* da interrogação analítica A.3, com índices

É possível verificar que o tempo de execução melhor substancialmente, executando agora a *query* em 78980.371ms (78.9s), 1min:19s. De seguida para perceber a nossa margem de melhoramento verificamos as estatísticas da nossa interrogação. Obtivemos os seguintes resultados.

#	exclusive	inclusive	rows_x	rows	loops	node
1.	0.019	78,974.342	↗ 5.0	5	1	→ Sort (cost=775.118.84..775.118.90 rows=25 width=136) (actual time=78.974.341..78.974.342 rows=5 loops=1) Sort Key: (sum(order_line.amount)) DESC Sort Method: quicksort Memory: 25kB
2.	0.000	78,974.323	↗ 5.0	5	1	→ Finalize GroupAggregate (cost=775.095.43..775.118.26 rows=25 width=136) (actual time=78.965.444..78.974.323 rows=5 loops=1) Group Key: nation.n_name
3.	0.000	78,980.128	↗ 3.3	15	1	→ Gather Merge (cost=775.095.43..775.117.57 rows=50 width=136) (actual time=78.962.535..78.980.128 rows=15 loops=1) Workers Planned: 2 Workers Launched: 2
4.	24.390	236,893.893	↗ 5.0	5	3	→ Partial GroupAggregate (cost=774.095.41..774.111.78 rows=25 width=136) (actual time=78.955.911..78.964.631 rows=5 loops=3) Group Key: nation.n_name
5.	118.133	236,899.503	↗ 11.7	25,077	3	→ Sort (cost=774.095.41..774.100.76 rows=2,141 width=107) (actual time=78.954.395..78.956.501 rows=25,077 loops=3) Sort Key: nation.n_name Sort Method: quicksort Memory: 2732kB
6.	1,182.012	236,753.376	↗ 11.7	25,077	3	→ Hash Join (cost=475.87..773.976.67 rows=2,141 width=107) (actual time=14.023..78.917.790 rows=25,077 loops=3) Hash Cond: ((mod((stock.s_w_id * stock.s_l_id), 10000)) = supplier.su_suppkey) AND (nation.n_nationkey = supplier.su_nationkey))
7.	2,201.421	235,556.601	↗ 11.7	626,655	3	→ Nested Loop (cost=3.87..772.948.40 rows=53,516 width=122) (actual time=0.760..78.518.667 rows=26,655 loops=3) Join Filter: (customer.c_w_id = stock.s_w_id)
8.	763.850	35,856.750	↗ 11.8	626,655	3	→ Nested Loop (cost=3.43..324.404.53 rows=53,323 width=136) (actual time=0.263..11.886.250 rows=26,655 loops=3) Join Filter: ((customer.c_w_id = order_line.ol_w_id) AND (customer.c_d_id = order_line.ol_d_id))
9.	237.561	6,419.536	↗ 11.8	73,761	3	→ Nested Loop (cost=2.87..250.667.77 rows=6,270 width=133) (actual time=0.205..2.806.512 rows=73,761 loops=3)
10.	1,744.218	5,976.375	↗ 11.8	73,520	3	→ Hash Join (cost=2.44..223.468.44 rows=6,250 width=123) (actual time=0.150..1.902.125 rows=73,520 loops=3) Hash Cond: ((ascii(substr(customer.c_state, 1, 1)) - 97) = nation.n_nationkey)
11.	4,231.998	4,231.998	↗ 1.2	1,000,000	3	→ Parallel Seq Scan on customer (cost=0.00..212.466.00 rows=1,250,000 width=15) (actual time=0.031..1.410.666 rows=1,000,000 loops=3)
12.	0.012	0.159	↗ 5.0	5	3	→ Hash (cost=2.43..2.43 rows=1 width=108) (actual time=0.053..0.053 rows=5 loops=3) Buckets: 1024 Batches: 1 Memory Usage: 9kB
13.	0.051	0.147	↗ 5.0	5	3	→ Hash Join (cost=1.07..2.43 rows=1 width=108) (actual time=0.043..0.049 rows=5 loops=3) Hash Cond: (nation.n_regionkey = region.r_regionkey)
14.	0.051	0.051	↗ 1.0	25	3	→ Seq Scan on nation (cost=0.00..1.25 rows=25 width=122) (actual time=0.014..0.017 rows=25 loops=3)
15.	0.012	0.045	↗ 1.0	1	3	→ Hash (cost=1.06..1.06 rows=1 width=4) (actual time=0.015..0.015 rows=1 loops=3) Buckets: 1024 Batches: 1 Memory Usage: 9kB
16.	0.033	0.033	↗ 1.0	1	3	→ Seq Scan on region (cost=0.00..1.06 rows=1 width=4) (actual time=0.018..0.011 rows=1 loops=3) Filter: (r_name = 'EUROPE') bpcpar Rows Removed by Filter: 4
17.	2,205.600	2,205.600	↗ 1.0	1	220,560	→ Index Scan using ix_orders on orders (cost=0.43..4.34 rows=1 width=16) (actual time=0.008..0.010 rows=1 loops=220,560) Index Cond: ((o_w_id = customer.c_w_id) AND (o_d_id = customer.c_d_id) AND (o_c_id = customer.c_id))
18.	26,775.364	26,775.364	↗ 1.0	8	221,284	→ Index Scan using pk_order_line on order_line (cost=0.56..11.64 rows=8 width=19) (actual time=0.110..0.121 rows=8 loops=221,284) Index Cond: ((ol_w_id = orders.o_w_id) AND (ol_d_id = orders.o_d_id) AND (ol_o_id = orders.o_id))
19.	197,396.430	197,396.430	↗ 3.0	1	1,879,966	→ Index Only Scan using pk_stock on stock (cost=0.43..8.37 rows=3 width=8) (actual time=0.105..0.105 rows=1 loops=1,879,966) Index Cond: ((s_w_id = order_line.ol_w_id) AND (s_i_id = order_line.ol_i_id)) Heap Fetches: 632478
20.	6.012	14,757	↗ 1.0	10,000	3	→ Hash (cost=322.00..322.00 rows=10,000 width=8) (actual time=4.919..4.919 rows=10,000 loops=3) Buckets: 16384 Batches: 1 Memory Usage: 519kB
21.	8.745	8,745	↗ 1.0	10,000	3	→ Seq Scan on supplier (cost=0.00..322.00 rows=10,000 width=8) (actual time=0.024..2.915 rows=10,000 loops=3)
Planning time : 15.841 ms						
Execution time : 78,980.371 ms						

Figura 38: Plano de execução da *query*

HTML	SOURCE	STATS
------	--------	-------

Per node type stats			
node type	count	sum of times	% of query
Finalize GroupAggregate	1	0.000 ms	0.0 %
Gather Merge	1	0.000 ms	0.0 %
Hash	3	6.036 ms	0.0 %
Hash Join	3	2.926.281 ms	3.7 %
Index Only Scan	1	197,396.430 ms	250.0 %
Index Scan	2	28,980.964 ms	36.7 %
Nested Loop	3	3.202.832 ms	4.1 %
Parallel Seq Scan	1	4,231.998 ms	5.4 %
Partial GroupAggregate	1	24.390 ms	0.0 %
Seq Scan	3	8.829 ms	0.0 %
Sort	2	116.152 ms	0.1 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
customer	1	4,231.998 ms	5.4 %
Parallel Seq Scan	1	4,231.998 ms	100.0 %
nation	1	0.051 ms	0.0 %
Seq Scan	1	0.051 ms	100.0 %
order_line	1	26,775.364 ms	33.9 %
Index Scan	1	26,775.364 ms	100.0 %
orders	1	2,205.600 ms	2.8 %
Index Scan	1	2,205.600 ms	100.0 %
region	1	0.033 ms	0.0 %
Seq Scan	1	0.033 ms	100.0 %
stock	1	197,396.430 ms	250.0 %
Index Only Scan	1	197,396.430 ms	100.0 %
supplier	1	8.745 ms	0.0 %
Seq Scan	1	8.745 ms	100.0 %

Figura 39: *Stats* da interrogação analítica A.3, com índices

Analisando as Figuras Fig 38 e Fig 39, é possível verificar que existem muitas secções que ainda precisam de melhoramentos. Enquanto analisávamos a tabelas das estatísticas derparamos com um valor percentual muito alto no *Index Only Scan* na tabela *stock*. Decidimos começar outra etapa de otimização exatamente por aí. O resultado de executar a *query* é o seguinte.

<i>n_name</i>	<i>revenue</i>
FRANCE	23519391.36
ROMANIA	23453148.62
RUSSIA	22811911.78
GERMANY	21675945.57
UNITED KINGDOM	21556976.32

Tabela 26: Resultados da *Query*, part(1)

4.1.3.2 Materialized Views

A partir do valor de tempo gasto no *Scan* da tabela *stock*, iniciamos uma análise da *query* e da Base de Dados em busca de justificações para tais números. Acontece que na tabela *stock* o *Scan* é feito 2M de vezes. Da análise da *query* retiramos que a tabela, não era utilizada para ver ou alterar um dos seus atributos mais importantes que é o número de artigos disponíveis, mas era apenas utilizada como condições de união das tabelas verificando os **items**, **warehouses**, e **suppliers**. E estes são valores que são alterados um número muito reduzido de vezes, permanecendo-se na maior parte do tempo inalterados. E daí surge a ideia de materializar a tabela *stock*, agrupando-a pelo que é necessário. De maneira a ter uma forma mais rápida de aceder aos conteúdos, que podem ser considerados "estáticos", da tabela. Criamos assim a seguinte *materialized view*.

```
CREATE MATERIALIZED VIEW q3_pls AS (  
    SELECT s_w_id,s_i_id FROM stock  
    GROUP BY s_w_id,s_i_id  
);
```

A *Query* a executar é a seguinte.

```
SELECT n_name, SUM(ol_amount) as revenue  
FROM customer, orders, order_line, supplier, nation, region, (SELECT * FROM q3_pls) AS s  
WHERE c_id = o_c_id  
AND c_w_id = o_w_id  
AND c_d_id = o_d_id  
AND ol_o_id = o_id  
AND ol_w_id = o_w_id  
AND ol_d_id= o_d_id  
AND ol_w_id = s.s_w_id  
AND ol_i_id = s.s_i_id  
AND mod((s.s_w_id * s.s_i_id),10000) = su_suppkey  
AND ascii(substr(c_state,1,1))-ascii('a') = su_nationkey  
AND su_nationkey = n_nationkey  
AND n_regionkey = r_regionkey  
AND r_name = 'EUROPE'  
GROUP BY n_nationkey,n_name  
GROUP BY revenue desc;
```

Convictos que o resultado iria melhorar, executamos a interrogação que nos mostrou o seguinte resultado.

```

Sort (cost=781919.68..781919.74 rows=25 width=140) (actual time=23959.842..23959.843 rows=5 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC
  Sort Method: quicksort Memory: 258K
-> Finalize GroupAggregate (cost=781890.90..781919.10 rows=25 width=140) (actual time=23955.018..23959.828 rows=5 loops=1)
  Group Key: nation.n_nationkey, nation.n_name
  -> Gather Merge (cost=-781899.90..781918.20 rows=50 width=140) (actual time=23951.854..23962.021 rows=10 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    -> Partial GroupAggregate (cost=780890.88..780912.49 rows=25 width=140) (actual time=22876.307..22884.521 rows=3 loops=3)
      Group Key: nation.n_nationkey, nation.n_name
      -> Sort (cost=780890.88..780890.20 rows=130 width=111) (actual time=22874.276..22877.270 rows=25077 loops=3)
        Sort Key: nation.n_nationkey, nation.n_name
        Sort Method: quicksort Memory: 21594K
        -> Hash Join (cost=419135.67..740773.12 rows=2130 width=111) (actual time=20220.998..22850.217 rows=25077 loops=3)
          Hash Cond: ((mod((q3.pls.s.w_id * q3.pls.s.i_id), 10000) = supplier.su_suppkey) AND (nation.n_nationkey = supplier.su_nationkey))
          -> Hash Join (cost=418663.67..779747.42 rows=55240 width=121) (actual time=24089.350..22597.056 rows=62655 loops=3)
            Hash Cond: ((q3.pls.s.w_id = customer.c.w_id) AND (q3.pls.s.i_id = order_line.ol.i_id))
            -> Parallel Seq Scan on q3.pls (cost=0.86..85914.57 rows=410657 width=8) (actual time=0.830..128.278 rows=233333 loops=1)
            -> Hash (cost=414244.05..414244.05 rows=127975 width=130) (actual time=20088.198..20888.198 rows=1879966 loops=3)
              Buckets: 32768 (originally 32768) Matches: 64 (originally 8) Memory Usage: 3841K
              -> Nested Loop (cost=1.13..414244.05 rows=127975 width=130) (actual time=0.256..18062.032 rows=1879966 loops=3)
                Join Filter: ((customer.c.w_id = order_line.ol.w_id) AND (customer.c.i_id = order_line.ol.i_id))
                -> Nested Loop (cost=0.57..212419.27 rows=15049 width=111) (actual time=0.248..14474.336 rows=21284 loops=3)
                  -> Nested Loop (cost=0.14..207470.95 rows=15000 width=123) (actual time=0.202..12512.324 rows=228560 loops=3)
                    Join Filter: (nation.n_nationkey = (ascii(substr(customer.c.state)::text, 1, 1)) - 97))
                    Rows Removed by Join Filter: 14779440
                    -> Nested Loop (cost=0.14..13.95 rows=1 width=108) (actual time=0.065..0.106 rows=3 loops=3)
                      Join Filter: (nation.n_regionkey = region.r_regionkey)
                      Rows Removed by Join Filter: 29
                      -> Index Scan using pk_nation on nation (cost=0.14..12.31 rows=25 width=112) (actual time=0.037..0.052 rows=25 loops=3)
                      -> Materialize (cost=0.00..1.07 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=751)
                        -> Seq Scan on region (cost=0.00..1.06 rows=1 width=4) (actual time=0.013..0.014 rows=1 loops=3)
                          Filter: (r_name = 'Europe')
                          Rows Removed by Filter: 4
                        -> Seq Scan on customer (cost=0.00..229966.00 rows=3000000 width=15) (actual time=0.007..1187.973 rows=3000000 loops=15)
                      -> Index Scan using ix_orders on orders (cost=0.41..1.50 rows=1 width=16) (actual time=0.000..0.000 rows=1 loops=661000)
                        Index Cond: ((o.w_id = customer.c.w_id) AND (o.d_id = customer.c.d_id) AND (o.c_id = customer.c.id))
                      -> Index Scan using pk_order_line on order_line (cost=0.56..0.05 rows=8 width=19) (actual time=0.014..0.017 rows=8 loops=663852)
                        Index Cond: ((ol.w_id = orders.o.w_id) AND (ol.d_id = orders.o.d_id) AND (ol.o_id = orders.o.id))
                    -> Hash (cost=322.08..322.08 rows=10000 width=8) (actual time=0.047..6.048 rows=10000 loops=3)
                      Buckets: 10284 Batches: 1 Memory Usage: 319K
                      -> Seq Scan on supplier (cost=0.00..322.00 rows=10000 width=8) (actual time=0.021..3.154 rows=10000 loops=3)
Planning time: 9.281 ms
Execution time: 23962.275 ms
(44 rows)

```

Figura 40: *Explain Analyze* da interrogação analítica A.3, com índices e materialized view

A criação desta materialized view, permitiu um melhoramento de cerca de **4.24x**, passando agora a *query* a ser executada em 23962.275ms (23.9s). Consideramos que obtivemos um razoável resultado face ao tempo inicial da interrogação. Passamos então para a análise estatística da execução que nos mostra os seguintes dados.

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.015	23.959.843	15.0	5	1	→ Sort (cost=761.919.06..781.919.74 rows=25 width=140) (actual time=23.959.842..23.959.843 rows=5 loops=1) Sort Key: (sum(order_line ol_amount)) DESC Sort Method: quicksort Memory: 25kB
2.	0.000	23.959.828	15.0	5	1	→ Finalize GroupAggregate (cost=781.890.90..781.919.10 rows=25 width=140) (actual time=23.953.818..23.959.828 rows=5 loops=1) Group Key: nation_n_nationkey, nation_n_name
3.	0.000	23.962.021	15.0	10	1	→ Gather Merge (cost=781.890.90..781.918.28 rows=90 width=140) (actual time=23.951.054..23.962.021 rows=10 loops=1) Workers Planned: 2 Workers Launched: 2
4.	21.753	66.653.563	18.3	3	3	→ Partial GroupAggregate (cost=780.890.88..780.912.49 rows=25 width=140) (actual time=22.876.307..22.884.521 rows=3 loops=3) Group Key: nation_n_nationkey, nation_n_name
5.	81.159	66.631.810	111.8	25.077	3	→ Sort (cost=780.890.88..780.890.20 rows=2.130 width=111) (actual time=22.874.276..22.877.270 rows=25.077 loops=3) Sort Key: nation_n_nationkey, nation_n_name Sort Method: quicksort Memory: 2159kB
6.	441.330	66.550.651	111.8	25.077	3	→ Hash Join (cost=419.135.67..780.773.12 rows=2.130 width=111) (actual time=20.220.996..22.850.217 rows=25.077 loops=3) Hash Cond: ((mod((q3_pls_s_w_id * q3_pls_s_l_id), 10000) = supplier_su_supplierkey) AND (nation_n_nationkey = supplier_su_nationkey))
7.	6.205.746	66.091.168	111.8	626.050	3	→ Hash Join (cost=418.663.67..779.747.42 rows=93.240 width=122) (actual time=20.089.338..22.097.056 rows=626.050 loops=3) Hash Cond: ((q3_pls_s_w_id = customer_c_w_id) AND (q3_pls_s_l_id = order_line ol_l_id))
8.	1.560.828	1.560.828	11.2	3.333.333	3	→ Parallel Seq Scan on q3_pls (cost=0.00..85.914.57 rows=4.166.657 width=8) (actual time=0.030..520.276 rows=3.333.333 loops=3)
9.	3.378.499	60.264.594	14.7	1.879.966	3	→ Hash (cost=414.244.05..414.244.05 rows=127.975 width=130) (actual time=20.066.198..20.086.198 rows=1.879.966 loops=3) Buckets: 32768 (originally 32768) Batches: 64 (originally 0) Memory Usage: 3841kB
10.	2.297.604	56.688.096	14.7	1.879.966	3	→ Nested Loop (cost=1.13..414.244.05 rows=127.975 width=130) (actual time=0.296..16.962.032 rows=1.879.966 loops=3) Join Filter: ((customer_c_w_id = order_line ol_w_id) AND (customer_c_d_id = order_line ol_d_id))
11.	472.596	43.303.008	14.7	221.284	3	→ Nested Loop (cost=0.57..321.419.27 rows=15.049 width=131) (actual time=0.249..14.434.336 rows=221.284 loops=3)
12.	19.717.028	37.536.972	14.7	220.560	3	→ Nested Loop (cost=0.14..297.479.95 rows=15.000 width=123) (actual time=0.202..12.512.324 rows=220.560 loops=3) Join Filter: (nation_n_nationkey = (ascii(substr((customer_c_state)::text, 1, 1)) - 97)) Rows Removed by Join Filter: 14779440
13.	0.087	0.318	15.0	5	3	→ Nested Loop (cost=0.14..13.95 rows=1 width=108) (actual time=0.045..0.106 rows=5 loops=3) Join Filter: (nation_n_regionkey = region_r_regionkey) Rows Removed by Join Filter: 20
14.	0.156	0.156	11.0	25	3	→ Index Scan using pk_nation on nation (cost=0.14..12.51 rows=25 width=112) (actual time=0.037..0.052 rows=25 loops=3)
15.	0.093	0.075	11.0	1	75	→ Materialize (cost=0.00..1.07 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=75)
16.	0.042	0.042	11.0	1	3	→ Seq Scan on region (cost=0.00..1.06 rows=1 width=4) (actual time=0.013..0.014 rows=1 loops=3) Filter: (r_name = 'EUROPE')::bpchar Rows Removed by Filter: 4
17.	17.819.866	17.818.965	11.0	3,000,000	15	→ Seq Scan on customer (cost=0.06..225.966.00 rows=3,000,000 width=15) (actual time=0.007..1.187.973 rows=3,000,000 loops=15)
18.	5.293.440	5.293.440	11.0	1 661,680		→ Index Scan using ix_orders on orders (cost=0.43..1.59 rows=1 width=16) (actual time=0.005..0.008 rows=1 loops=661,680) Index Cond: ((o_w_id = customer_c_w_id) AND (o_d_id = customer_c_d_id) AND (o_c_id = customer_c_id))
19.	11.285.494	11.285.494	11.0	8 663,852		→ Index Scan using pk_order_line on order_line (cost=0.56..6.05 rows=8 width=19) (actual time=0.014..0.017 rows=8 loops=663,852) Index Cond: ((o_w_id = orders_o_w_id) AND (o_d_id = orders_o_d_id) AND (o_l_id = orders_o_l_id))
20.	8.662	18.144	11.0	10,000	3	→ Hash (cost=322.00..322.00 rows=10,000 width=8) (actual time=8.047..8.048 rows=18,000 loops=3) Buckets: 16384 Batches: 1 Memory Usage: 519kB
21.	9.462	9.462	11.0	10,000	3	→ Seq Scan on supplier (cost=0.00..322.00 rows=10,000 width=8) (actual time=0.021..3.154 rows=10,000 loops=3)

Figura 41: Plano de execução da *query*

A figura acima mostra que a melhoria é efetiva, o nosso objetivo foi cumprido. Não só os tempos, como os "custos" da componente de *Scan* de *stock* baixou consideravelmente.

De maneira a comprovar essa redução de carga, realizamos uma verificação à tabela das estatísticas. A tabela é a seguinte.

HTML	SOURCE	STATS
------	--------	-------

Per node type stats			
node type	count	sum of times	% of query
Finalize GroupAggregate	1	0.000 ms	0.0 %
Gather Merge	1	0.000 ms	0.0 %
Hash	2	3,387.180 ms	14.1 %
Hash Join	2	6,707.085 ms	28.0 %
Index Scan	3	16,579.080 ms	69.2 %
Materialize	1	0.033 ms	0.0 %
Nested Loop	4	22,487.346 ms	93.9 %
Parallel Seq Scan	1	1,560.828 ms	6.5 %
Partial GroupAggregate	1	21.753 ms	0.1 %
Seq Scan	3	17,829.099 ms	74.4 %
Sort	2	81.174 ms	0.3 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
customer	1	17,819.595 ms	74.4 %
Seq Scan	1	17,819.595 ms	100.0 %
nation	1	0.156 ms	0.0 %
Index Scan	1	0.156 ms	100.0 %
order_line	1	11,285.484 ms	47.1 %
Index Scan	1	11,285.484 ms	100.0 %
orders	1	5,293.440 ms	22.1 %
Index Scan	1	5,293.440 ms	100.0 %
q3_pls	1	1,560.828 ms	6.5 %
Parallel Seq Scan	1	1,560.828 ms	100.0 %
region	1	0.042 ms	0.0 %
Seq Scan	1	0.042 ms	100.0 %
supplier	1	9.462 ms	0.0 %
Seq Scan	1	9.462 ms	100.0 %

Figura 42: *Stats* da interrogação analítica A.3

Os resultados da tabela confirmam as nossas apreciações, a carga de fazer *Scan* na tabela diminui de 250% para 6,5% da *query*.

Não estando contentes com a parte superior que aparece no *Analyze*. Decidimos continuar em busca de resultados melhores. O resultado de executar a *query* é o seguinte.

<i>n_name</i>	<i>revenue</i>
FRANCE	23519391.36
ROMANIA	23453148.62
RUSSIA	22811911.78
GERMANY	21675945.57
UNITED KINGDOM	21556976.32

Tabela 27: Resultados da *Query*, part(2)

4.1.3.3 Views

Agora e com vista a melhor o custo da *query*, decidimos introduzir diferentes *Views*, algumas partes da Interrogação. Criamos então as seguintes vistas.

```
CREATE VIEW q3_1 AS (  
  SELECT o_id, o_c_id,o_w_id,o_d_id, c_state  
    FROM customer,orders  
   WHERE c_id = o_c_id  
         and c_w_id = o_w_id  
         and c_d_id = o_d_id  
   GROUP BY o_id, o_c_id,o_w_id,o_d_id,c_state  
);
```

```
CREATE VIEW q3_2_teste AS (  
  SELECT n_nationkey,n_name, ol_w_id, ol_o_id,ol_i_id,ol_d_id, ol_amount, su_nationkey  
    FROM (SELECT * FROM q3_pls) AS s,order_line, supplier, nation, region  
   WHERE mod((s.s_w_id * s.s_i_id),10000) = su_suppkey  
         and su_nationkey = n_nationkey  
         and n_regionkey = r_regionkey  
         and r_name = 'EUROPE'  
         and ol_w_id = s.s_w_id  
         and ol_i_id = s.s_i_id  
);
```

Depois da criação destas vistas, a interrogação adaptada foi a seguinte.

```
SELECT n_name, SUM(ol_amount) AS revenue  
  FROM (SELECT * FROM q3_2_teste) AS d,(SELECT * FROM q3_1) AS u  
 WHERE d.ol_w_id = u.o_w_id  
       AND d.ol_d_id= u.o_d_id  
       AND d.ol_o_id = u.o_id  
       AND ascii(substr(u.c_state,1,1))-ascii('a') = d.su_nationkey  
 GROUP BY d.n_nationkey,d.n_name  
 ORDER BY revenue desc;
```

O resultado de realizar o *EXPLAIN ANALYZE* da *query* acima é o seguinte.

```

Sort (cost=1165209.13..1165209.19 rows=25 width=148) (actual time=21212.913..21212.914 rows=5 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC
  Sort Method: quicksort Memory: 25kB
  -> GroupAggregate (cost=1165208.07..1165208.55 rows=25 width=148) (actual time=21199.066..21212.894 rows=5 loops=1)
    Group Key: nation.n_nationkey, nation.n_name
    -> Sort (cost=1165206.97..1165207.29 rows=127 width=111) (actual time=21184.588..21194.159 rows=75231 loops=1)
      Sort Key: nation.n_nationkey, nation.n_name
      Sort Method: external merge Disk: 3368kB
      -> Hash Join (cost=945889.84..1165192.83 rows=127 width=111) (actual time=16881.246..21186.584 rows=75231 loops=1)
        Hash Cond: ((n3.pls.s.w_id * n3.pls.s.i_id) < 10000) = supplier.su_supplier AND (nation.n_nationkey = supplier.su_nationkey)
        -> Hash Join (cost=945417.94..1164697.49 rows=1106 width=122) (actual time=16022.875..20737.616 rows=187996 loops=1)
          Hash Cond: ((n3.pls.s.w_id = order_line.ol_w_id) AND (n3.pls.s.i_id = order_line.ol_i_id))
          -> Seq Scan on n3_pls (cost=0.00..144247.77 rows=999997 width=8) (actual time=0.812..954.271 rows=1000000 loops=1)
          -> Hash (cost=94370.90..943370.63 rows=2104 width=126) (actual time=16022.697..18922.698 rows=187996 loops=1)
            Buckets: 65536 (originally 4896) Batches: 64 (originally 1) Memory Usage: 3585kB
            -> Nested Loop (cost=528307.38..945370.63 rows=3154 width=126) (actual time=7516.321..13268.922 rows=187996 loops=1)
              -> Hash Join (cost=528381.81..924548.24 rows=15048 width=123) (actual time=7516.234..11717.347 rows=221284 loops=1)
                Hash Cond: ((ascii(substr((customer.c_state)::text, 1, 1)) < 97) = nation.n_nationkey)
                -> Group (cost=528299.77..887962.41 rows=8889753 width=10) (actual time=7516.181..18741.927 rows=3010933 loops=1)
                  Group Key: orders.o_id, orders.o_c_id, orders.o_w_id, orders.o_d_id, customer.c_state
                  -> Gather Merge (cost=528299.37..836610.78 rows=2588130 width=19) (actual time=7516.178..9738.975 rows=3010933 loops=1)
                    Workers Planned: 2
                    Workers Launched: 2
                    -> Group (cost=527299.35..846118.32 rows=1254065 width=19) (actual time=5112.317..8780.075 rows=1883344 loops=1)
                      Group Key: orders.o_id, orders.o_c_id, orders.o_w_id, orders.o_d_id, customer.c_state
                      -> Sort (cost=527299.35..538434.51 rows=1254065 width=19) (actual time=5112.314..5425.242 rows=1883344 loops=1)
                        Sort Key: orders.o_id, orders.o_c_id, orders.o_w_id, orders.o_d_id, customer.c_state
                        Sort Method: quicksort Memory: 2067kB
                        -> Hash Join (cost=207115.00..374572.47 rows=1254065 width=10) (actual time=2926.453..4159.159 rows=1883344 loops=1)
                          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))
                          -> Parallel Seq Scan on orders (cost=0.00..40663.88 rows=1254108 width=16) (actual time=6.028..175.190 rows=1883344 loops=1)
                          -> Hash (cost=229305.89..229165.08 rows=1000000 width=15) (actual time=2924.585..2924.586 rows=3888600 loops=1)
                            Buckets: 11112 Batches: 64 Memory Usage: 3227kB
                            -> Seq Scan on customer (cost=0.00..229965.88 rows=1000000 width=15) (actual time=6.026..1718.798 rows=3888600 loops=1)
                          -> Hash (cost=2.43..2.43 rows=1 width=188) (actual time=0.835..0.835 rows=6 loops=1)
                            Buckets: 1024 Batches: 1 Memory Usage: 9kB
                            -> Hash Join (cost=1.87..2.43 rows=1 width=188) (actual time=0.028..0.032 rows=5 loops=1)
                              Hash Cond: (nation.o_regionkey = region.r_regionkey)
                              -> Seq Scan on nation (cost=0.00..1.25 rows=25 width=112) (actual time=0.000..0.011 rows=25 loops=1)
                              -> Hash (cost=1.86..1.86 rows=1 width=4) (actual time=0.812..0.812 rows=1 loops=1)
                                Buckets: 1024 Batches: 1 Memory Usage: 9kB
                                -> Seq Scan on region (cost=0.00..1.86 rows=1 width=4) (actual time=0.810..0.811 rows=1 loops=1)
                                  Filter: (r_name = 'EUROPE') bpbchar
                                  Rows Removed by Filter: 4
                            -> Index Scan using pk_order_line on order_line (cost=0.56..1.38 rows=8 width=10) (actual time=0.812..0.814 rows=8 loops=221284)
                              Index Cond: ((ol_w_id = orders.o_w_id) AND (ol_d_id = orders.o_d_id) AND (ol_o_id = orders.o_id))
                            -> Hash (cost=322.86..322.86 rows=10000 width=8) (actual time=4.034..4.034 rows=18100 loops=1)
                              Buckets: 16384 Batches: 1 Memory Usage: 510kB
                              -> Seq Scan on supplier (cost=0.00..322.80 rows=18888 width=8) (actual time=0.018..2.779 rows=18888 loops=1)
        Planning time: 8.628 ms
        Execution time: 21214.115 ms

```

Figura 43: *Explain Analyze* da interrogação analítica A.3, com índices e materialized view, e views

O tempo de execução desta *query* foi 21214.115ms (21s), nota-se alguma melhoria em termo de tempo face à alternativa, o que nos mostra que o grupo fez uma boa aposta ao criar estas vistas. Para perceber os efeitos das nossas alterações no plano, analisamos as seguintes figuras.

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.020	21,212.914	1 5.0	5	1	→ Sort (cost=1.165,209.13..1.165,209.19 rows=25 width=140) (actual time=21,212.913..21,212.914 rows=5 loops=1) Sort Key: (sum(order_line_of_amount)) DESC Sort Method: quicksort Memory: 25kB
2.	18.744	21,212.894	1 5.0	5	1	→ GroupAggregate (cost=1.165,206.97..1.165,206.55 rows=25 width=140) (actual time=21,190.966..21,212.894 rows=5 loops=1) Group Key: nation_n_nationkey, nation_n_name
3.	87.556	21,194.150	1 880.4	75,231	1	→ Sort (cost=1.165,206.97..1.165,207.29 rows=127 width=111) (actual time=21,184.580..21,194.150 rows=75,231 loops=1) Sort Key: nation_n_nationkey, nation_n_name Sort Method: external merge Disk: 3360kB
4.	364.290	21,106.594	1 880.4	75,231	1	→ Hash Join (cost=945,889.94..1.165,202.53 rows=127 width=111) (actual time=16,033.286..21,106.594 rows=75,231 loops=1) Hash Cond: ((mod((q3_pls_s_w_id * q3_pls_s_i_id), 10000) = supplier_su_supplierkey) AND (nation_n_nationkey = supplier_su_nationkey))
5.	3,761.241	20,737.610	1 580.1	1,879,966	1	→ Hash Join (cost=945,417.94..1.164,697.40 rows=3,186 width=122) (actual time=16,022.875..20,737.610 rows=1,879,966 loops=1) Hash Cond: ((q3_pls_s_w_id = order_line_of_w_id) AND (q3_pls_s_i_id = order_line_of_i_id))
6.	954.271	854.271	1 1.0	10,000,000	1	→ Seq Scan on q3_pls (cost=0.00..144,247.77 rows=9,999,977 width=8) (actual time=0.012..954.271 rows=10,000,000 loops=1)
7.	761.176	16,022.998	1 588.6	1,879,966	1	→ Hash (cost=945,370.03..945,370.03 rows=3,194 width=126) (actual time=16,022.097..16,022.098 rows=1,879,966 loops=1) Buckets: 65536 (originally 4096) Batches: 64 (originally 1) Memory Usage: 9595kB
8.	445.999	15,260.922	1 588.6	1,879,966	1	→ Heated Loop (cost=528,302.38..945,370.03 rows=3,194 width=126) (actual time=7,516.321..15,260.922 rows=1,879,966 loops=1)
9.	975.365	11,717.347	1 14.7	221,284	1	→ Hash Join (cost=528,301.61..924,548.24 rows=15,049 width=123) (actual time=7,516.234..11,717.347 rows=221,284 loops=1) Hash Cond: ((asc(substr(customer_c_gstate), text, 1, 1)) = nation_n_nationkey)
10.	1,002.952	10,741.627	1 1.0	3,010,033	1	→ Group (cost=528,299.37..987,962.41 rows=3,009,755 width=19) (actual time=7,516.181..10,741.627 rows=3,010,033 loops=1) Group Key: orders_o_id, orders_o_c_id, orders_o_w_id, orders_o_d_id, customer_c_state
11.	0.000	9,738.975	1 1.2	3,010,033	1	→ Gather Merge (cost=528,299.37..936,610.78 rows=2,508,130 width=19) (actual time=7,516.178..9,738.975 rows=3,010,033 loops=1) Workers Planned: 2 Workers Launched: 2
12.	914.499	17,190.225	1 1.2	1,003,344	3	→ Group (cost=527,299.35..546,110.32 rows=1,254,065 width=19) (actual time=5,112.317..5,730.875 rows=1,003,344 loops=3) Group Key: orders_o_id, orders_o_c_id, orders_o_w_id, orders_o_d_id, customer_c_state
13.	3,825.249	16,275.726	1 1.2	1,003,344	3	→ Sort (cost=527,299.35..530,434.51 rows=1,254,065 width=19) (actual time=5,112.314..5,425.242 rows=1,003,344 loops=3) Sort Key: orders_o_id, orders_o_c_id, orders_o_w_id, orders_o_d_id, customer_c_state Sort Method: quicksort Memory: 2867kB
14.	3,149.943	12,450.477	1 1.2	1,003,344	3	→ Hash Join (cost=297,115.00..374,552.47 rows=1,254,065 width=19) (actual time=2,926.453..4,156.159 rows=1,003,344 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))
15.	525.570	525.570	1 1.2	1,003,344	3	→ Parallel Seq Scan on orders (cost=0.00..40,663.80 rows=1,254,195 width=16) (actual time=3.028..179.189 rows=1,003,344 loops=3)
16.	3,642.094	8,774.564	1 1.0	3,000,000	3	→ Hash (cost=229,966.00..229,966.00 rows=3,000,000 width=15) (actual time=2,304.988..2,924.988 rows=3,000,000 loops=3) Buckets: 131072 Batches: 64 Memory Usage: 3227kB
17.	5,132.370	5,132.370	1 1.0	3,000,000	3	→ Seq Scan on customer (cost=0.00..229,966.00 rows=3,000,000 width=15) (actual time=0.026..1,710.790 rows=3,000,000 loops=3)
18.	0.003	0.035	1 5.0	5	1	→ Hash (cost=2.43..2.43 rows=1 width=108) (actual time=0.035..0.035 rows=5 loops=1) Buckets: 1024 Batches: 1 Memory Usage: 9kB
19.	0.009	0.032	1 5.0	5	1	→ Hash Join (cost=1.07..2.43 rows=1 width=108) (actual time=0.028..0.032 rows=5 loops=1) Hash Cond: (nation_n_regionkey = region_r_regionkey)
20.	0.011	0.011	1 1.0	25	1	→ Seq Scan on nation (cost=0.00..1.25 rows=25 width=112) (actual time=0.009..0.011 rows=25 loops=1)
21.	0.001	0.012	1 1.0	1	1	→ Hash (cost=1.06..1.06 rows=1 width=4) (actual time=0.012..0.012 rows=1 loops=1) Buckets: 1024 Batches: 1 Memory Usage: 9kB
22.	0.011	0.011	1 1.0	1	1	→ Seq Scan on region (cost=0.00..1.06 rows=1 width=4) (actual time=0.010..0.011 rows=1 loops=1) Filter: (r_name = 'EUROPE'::bpchar) Rows Removed by Filter: 4
23.	3,097.976	3,097.976	1 1.0	8 221,284	1	→ Index Scan using pk_order_line on order_line (cost=0.56..1.38 rows=8 width=18) (actual time=0.012..0.014 rows=8 loops=221,284) Index Cond: ((ol_w_id = orders_o_w_id) AND (ol_d_id = orders_o_d_id) AND (ol_o_id = orders_o_id))
24.	1.915	4.694	1 1.0	10,000	1	→ Hash (cost=322.00..322.00 rows=10,000 width=8) (actual time=4.694..4.694 rows=10,000 loops=1) Buckets: 10384 Batches: 1 Memory Usage: 519kB
25.	2.779	2.779	1 1.0	10,000	1	→ Seq Scan on supplier (cost=0.00..322.00 rows=10,000 width=8) (actual time=0.018..2.779 rows=10,000 loops=1)
						Planning time : 3.026 ms
						Execution time : 21,214.115 ms

Figura 44: Plano de execução da *query*

HTMLSOURCESTATS

Per node type stats

node type	count	sum of times	% of query
Gather Merge	1	0.000 ms	0.0 %
Group	2	1,917.451 ms	9.0 %
GroupAggregate	1	18.744 ms	0.1 %
Hash	5	4,405.689 ms	20.8 %
Hash Join	5	8,250.868 ms	38.9 %
Index Scan	1	3,097.976 ms	14.6 %
Nested Loop	1	445.599 ms	2.1 %
Parallel Seq Scan	1	525.570 ms	2.5 %
Seq Scan	5	6,089.442 ms	28.7 %
Sort	3	3,912.825 ms	18.4 %

Per table stats

Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
customer	1	5,132.370 ms	24.2 %
Seq Scan	1	5,132.370 ms	100.0 %
nation	1	0.011 ms	0.0 %
Seq Scan	1	0.011 ms	100.0 %
order_line	1	3,097.976 ms	14.6 %
Index Scan	1	3,097.976 ms	100.0 %
orders	1	525.570 ms	2.5 %
Parallel Seq Scan	1	525.570 ms	100.0 %
q3_pls	1	954.271 ms	4.5 %
Seq Scan	1	954.271 ms	100.0 %
region	1	0.011 ms	0.0 %
Seq Scan	1	0.011 ms	100.0 %
supplier	1	2.779 ms	0.0 %
Seq Scan	1	2.779 ms	100.0 %

Figura 45: *Stats* da interrogação analítica A.3

Começando análise pelas estatísticas, vemos que as percentagens estão muito atrativas. Sendo que a que apresenta maior percentagem é o *Scan* na tabela *customer*, e este apresenta uma percentagem de 24.2% da *query*. O caminho a seguir poderia passar por tentar melhorar esse scan na tabela *customer*. Analisando agora o plano de execução (Fig 44), vemos que a melhoria é de facto notável face aos anteriores. O resultado de executar a *query* é o seguinte.

<i>n_name</i>	<i>revenue</i>
FRANCE	23519391.36
ROMANIA	23453148.62
RUSSIA	22811911.78
GERMANY	21675945.57
UNITED KINGDOM	21556976.32

Tabela 28: Resultados da *Query*, part(3)

4.1.3.4 Considerações

Continuamos a tentar otimizar a *query*, através de manipulação da query, tentar isolar tabelas para materializar, porém não obtivemos melhorias de desempenho. Optamos por não materializar mais nenhuma componente da Interrogação como a tabela *order_line*, por exemplo, por não serem tabelas imutáveis. e caso não existisse o *refresh* frequente da *view* os dados materializados ficam facilmente desatualizados, perdendo assim "força" à interrogação analítica. Por fim, achamos que a query pode evoluir, porém estamos satisfeitos com o resultado obtido.

4.1.4 Interrogações analítica A.4 (TPC-H)

```
select c_last, c_id o_id, o_entry_d, o_ol_cnt, sum(ol_amount)
from customer, orders, order_line
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
group by o_id, o_w_id, o_d_id, c_id, c_last, o_entry_d, o_ol_cnt
having sum(ol_amount) > 200
order by sum(ol_amount) desc, o_entry_d;
```

Tal como já havíamos feito nas restantes *queries*, procedemos desta forma à análise do comando *EXPLAIN ANALYZE* para assim poder aferir quais as metodologias estavam a ser utilizadas pelo *POSTGRESQL* durante a execução desta interrogação. Assim sendo, o grupo deparou-se com um tempo de execução bastante elevado, cerca de 83910.388ms. Como tal, era imperativo que um valor tão alto fosse reduzido para o mínimo possível.

```
Sort (cost=15537720.11..15620945.74 rows=25486653 width=77) (actual time=83179.982..83681.670 rows=9000000 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC, orders.o_entry_d
  Sort Method: external merge  Disk: 5818448
  -> Finalize GroupAggregate (cost=5235014.27..9024557.08 rows=25486653 width=77) (actual time=79224.632..82191.122 rows=9000000 loops=1)
    Group Key: orders.o_id, orders.o_w_id, orders.o_d_id, customer.c_id, customer.c_last, orders.o_entry_d, orders.o_ol_cnt
    Filter: (sum(order_line.ol_amount) > '200'::numeric)
    Rows Removed by Filter: 2110033
    -> Gather Merge (cost=5235014.27..8058188.14 rows=21238878 width=77) (actual time=72340.782..79334.642 rows=3197291 loops=1)
      Workers Planned: 2
      Workers Launched: 2
      -> Partial GroupAggregate (cost=5234014.24..5605694.61 rows=18619439 width=77) (actual time=65422.648..72400.550 rows=1865764 loops=3)
        Group Key: orders.o_id, orders.o_w_id, orders.o_d_id, customer.c_id, customer.c_last, orders.o_entry_d, orders.o_ol_cnt
        -> Sort (cost=5234014.24..5260562.84 rows=18619439 width=48) (actual time=65422.616..67119.888 rows=8531723 loops=3)
          Sort Key: orders.o_id, orders.o_w_id, orders.o_d_id, customer.c_id, customer.c_last, orders.o_entry_d, orders.o_ol_cnt
          Sort Method: external sort  Disk: 33341600
          -> Merge Join (cost=3814865.66..3341369.87 rows=18619439 width=48) (actual time=31013.902..41839.495 rows=8531723 loops=3)
            Merge Cond: ((order_line.ol_o_id = orders.o_id) AND (order_line.ol_w_id = customer.c_w_id) AND (order_line.ol_d_id = customer.c_d_id))
            -> Sort (cost=2038990.02..2065641.86 rows=18660418 width=15) (actual time=16237.004..19320.637 rows=8531723 loops=3)
              Sort Key: order_line.ol_o_id, order_line.ol_w_id, order_line.ol_d_id
              Sort Method: external merge  Disk: 23568848
              -> Parallel Seq Scan on order_line (cost=0.00..430241.18 rows=18660418 width=15) (actual time=0.022..5380.043 rows=8531723 loops=3)
                -> Materialize (cost=975870.44..990918.45 rows=3000003 width=53) (actual time=14776.884..17866.212 rows=18475978 loops=3)
                  -> Sort (cost=975870.44..983390.45 rows=3000003 width=53) (actual time=14776.878..15915.689 rows=3010023 loops=3)
                    Sort Key: orders.o_id, customer.c_w_id, customer.c_d_id
                    Sort Method: external merge  Disk: 19443248
                    -> Hash Join (cost=302903.33..446571.68 rows=3000003 width=53) (actual time=5569.127..9377.052 rows=3010033 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))
                      -> Seq Scan on orders (cost=0.00..58222.33 rows=3010033 width=28) (actual time=0.047..0.965 rows=3010033 loops=3)
                      -> Hash (cost=229975.12..229975.12 rows=3000012 width=29) (actual time=5367.953..5567.953 rows=3000000 loops=3)
                        Buckets: 65536  Batches: 64  Memory Usage: 3444kB
                        -> Seq Scan on customer (cost=0.00..229975.12 rows=3000012 width=29) (actual time=0.328..4396.429 rows=3000000 loops=3)
Planning time: 1.359 ms
Execution time: 83910.388 ms
```

Figura 46: *Explain Analyze* da interrogação analítica A.4

4.1.4.1 Vistas Materializadas

Como forma de reduzir o elevado tempo de execução da *query*, e após analisar a quantidade de registos que a interrogação envolvia, achou-se por bem utilizar uma *materialized view* como forma de suportar parte do peso computacional da *query*.

Assim sendo, decidimos criar uma vista materializada que albergasse parte da interrogação inicial, mais concretamente a parte onde são feitas as combinações das chaves e seleção da informação pretendida.

Deste modo, a vista criada é a seguinte:

```
create materialized view a4_view1 as
select c_last, c_id,o_id, o_entry_d, o_ol_cnt, ol_amount
from customer, orders, order_line
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;

SELECT c_last, c_id,o_id, o_entry_d, o_ol_cnt, sum(ol_amount)
from a4_view1 as t
group by t.o_id,t.c_id, t.c_last, t.o_entry_d, t.o_ol_cnt
having sum(ol_amount) > 200
order by sum(ol_amount) desc, t.o_entry_d;
```

Decidimos por isso deixar o peso da operação SUM() para a interrogação final (colocando fora da vista materializada). A estruturação da vista materializada desta forma, permite tal como nas interrogações anteriores, maior liberdade e flexibilidade. Deste modo, o grupo considerou que faria bastante mais sentido a criação desta vista neste formato, ao invés de criar uma vista materializada com a totalidade da *query*. Assim, por forma a procurar baixar o *execution time* mas também manter alguma estrutura na interrogação analítica, a estrutura adotada foi a apresentada. Como alternativa, o grupo considerou o caso em que a *query* era colocada na totalidade numa vista materializada. No entanto, consideramos esta opção como não sendo de todo a melhor (em termos de flexibilidade), apesar do menor tempo de execução comparando com outras abordagens.

O resultado da aplicação desta técnica é o que se segue.

```
Sort (cost=3654673.11..3661118.59 rows=2578189 width=69) (actual time=48852.801..49364.981 rows=900000 loops=1)
  Sort Key: (sum(ol_amount)) DESC, o_entry_d
  Sort Method: external merge  Disk: 53760kB
-> Finalize GroupAggregate (cost=2198539.14..3168624.71 rows=2578189 width=69) (actual time=44800.852..47878.071 rows=900000 loops=1)
  Group Key: o_id, c_id, c_last, o_entry_d, o_ol_cnt
  Filter: (sum(ol_amount) > '200'::numeric)
  Rows Removed by Filter: 2129341
-> Gather Merge (cost=2198539.14..3013933.37 rows=5156378 width=69) (actual time=36444.620..44833.364 rows=3228143 loops=1)
  Workers Planned: 2
  Workers Launched: 2
-> Partial GroupAggregate (cost=2197539.11..2417759.39 rows=2578189 width=69) (actual time=36432.488..42862.527 rows=1076048 loops=3)
  Group Key: o_id, c_id, c_last, o_entry_d, o_ol_cnt
-> Sort (cost=2197539.11..2224395.24 rows=10742452 width=40) (actual time=36432.462..38001.502 rows=8595985 loops=3)
  Sort Key: o_id, c_id, c_last, o_entry_d, o_ol_cnt
  Sort Method: external sort  Disk: 387736kB
-> Parallel Seq Scan on a4_view1 t (cost=0.00..355589.53 rows=10742452 width=40) (actual time=0.037..1740.252 rows=8595985 loops=3)

Planning time: 0.220 ms
Execution time: 49464.687 ms
```

Figura 47: *Explain Analyze* da interrogação analítica A.4 com *materialized view*

Como se pode ver pela imagem, o tempo já é consideravelmente mais reduzido. No entanto, após uma análise e pesquisa, percebemos que esta vista poderia ser melhorada de forma acentuada com a criação de índices.

4.1.4.2 Vista Materializada e Índice

Através da análise do plano de execução da interrogação com a vista materializada, percebeu-se que o GROUP BY tinha um papel muito relevante no que toca ao tempo de execução. Assim sendo, o grupo procurou criar um índice nesta vista que fosse ao encontro do passo GROUP KEY. Assim, o índice criado foi:

```
CREATE INDEX ind_grou_key_a4_mv
ON a4_view1(o_id, c_id, c_last, o_entry_d, o_ol_cnt);
```

Após a aplicação do índice, o tempo e plano de execução ficaram bastante mais estreitos, verificando-se assim uma redução muito significativa do tempo de execução quando comparado com a execução sem qualquer otimização.

```
Sort (cost=2789915.55..2796362.53 rows=2578795 width=69) (actual time=15864.280..16374.701 rows=900000 loops=1)
  Sort Key: (sum(o_l_amount)) DESC, o_entry_d
  Sort Method: external merge  Disk: 53760kB
-> GroupAggregate (cost=0.56..2303749.24 rows=2578795 width=69) (actual time=10151.918..14865.656 rows=900000 loops=1)
  Group Key: o_id, c_id, c_last, o_entry_d, o_ol_cnt
  Filter: (sum(o_l_amount) > '200'::numeric)
  Rows Removed by Filter: 2129341
-> Index Scan using ind_group_key_a4 on a4_view1 t (cost=0.56..1813778.12 rows=25787954 width=40) (actual time=0.853..6671.563 rows=25787954 loops=1)
Planning time: 0.205 ms
Execution time: 16420.646 ms
```

Figura 48: *Explain Analyze* da interrogação analítica A.4 com vista materializada e índice

5 Replicação e *Sharding*

Com vista a melhorar a distribuição de carga da base de dados, tendo em conta as várias operações de escrita e leitura, estudou-se nesta secção a aplicação de uma técnica que é bastante importante na manutenção da acessibilidade e principalmente na disponibilidade dos SGBD.

Um mecanismo de replicação de dados permite a criação de várias cópias idênticas de dados em vários servidores. O mecanismo de *sharding* consiste numa partição horizontal de dados, sendo que cada partição é designada por *shard*. Cada um destes *shard* é armazenado num servidor de base de dados, de forma a distribuir a carga. Sendo que apenas era necessário a implementação de um destes mecanismos, optamos por testar a implementação da replicação. De facto, tivemos em atenção que o *sharding* apenas seria benéfico na existência de queries otimizadas para operações paralelas.

5.1 Replicação

Tal como mencionado acima, a técnica com a qual o grupo decidiu trabalhar no trabalho prático é a replicação. De facto, existem duas formas de efetuar a replicação: replicação lógica e replicação física. Nesse sentido, optou-se apenas por explorar a replicação lógica. A replicação lógica permite, através de mecanismos de sincronização, replicar modificações ao nível da base de dados.

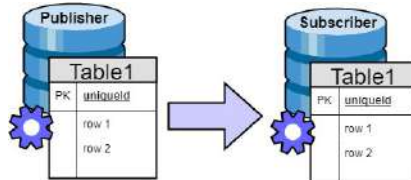


Figura 49: Replicação lógica de base de dados

Assim sendo, o grupo procedeu então à criação da **publicação** e da *subscrição* respetivas, alterando no postgres os parâmetros necessários para fazer a replicação lógica.

```
2020-01-03 15:19:25.375 UTC [3720] STATEMENT: CREATE SUBSCRIPTION subscricao CONNECTION 'dbname=tpcc host=localhost port=5433 user=tiagofontes' PUBLICATION replicacao;
2020-01-03 15:24:42.485 UTC [4057] LOG: logical replication apply worker for subscription "subscricao" has started
2020-01-03 15:24:42.492 UTC [4059] LOG: logical replication table synchronization worker for subscription "subscricao", table "customer" has started
2020-01-03 15:24:42.503 UTC [4061] LOG: logical replication table synchronization worker for subscription "subscricao", table "district" has started
2020-01-03 15:24:42.532 UTC [4061] LOG: logical replication table synchronization worker for subscription "subscricao", table "district" has finished
2020-01-03 15:24:42.538 UTC [4063] LOG: logical replication table synchronization worker for subscription "subscricao", table "history" has started
```

Figura 50: Resultado replicação base de dados


```

2020-01-03 15:44:57.035 UTC [4611] LOG: logical replication apply worker for subscription "subscricao" has started
2020-01-03 15:44:57.065 UTC [4613] LOG: logical replication table synchronization worker for subscription "subscricao", table "order_line" has started
2020-01-03 15:45:07.196 UTC [3688] LOG: checkpoints are occurring too frequently (6 seconds apart)
2020-01-03 15:45:07.196 UTC [3688] HINT: Consider increasing the configuration parameter "max_wal_size".
2020-01-03 15:45:12.639 UTC [3688] LOG: checkpoints are occurring too frequently (5 seconds apart)
2020-01-03 15:45:12.639 UTC [3688] HINT: Consider increasing the configuration parameter "max_wal_size".
2020-01-03 15:45:22.389 UTC [3688] LOG: checkpoints are occurring too frequently (10 seconds apart)

```

Figura 51: Resultado replicação base de dados

O grupo, como forma de análise, decidiu aplicar os parâmetros ótimos de configuração do PostgreSQL também na replicação, para assim poder aferir qual o impacto desta técnica nas métricas que temos vindo a analisar.

```

name throughput(tx/s) response_time(s) abort_rate
TPCC-2020-01-03_16_59-TPC-C-time-10-clients-7000-frag-1-think-true.dat 480.72324776837803 0.0366364558382426 0.0394104752577

```

Figura 52: Observação métricas após replicação

Como podemos ver, existiu uma diminuição a nível de débito com a aplicação da replicação lógica. Por outro lado, o tempo de resposta também subiu, pelo que é possível dizer que a replicação veio prejudicar o desempenho.

De facto, estes foram resultados esperados, uma vez que, a replicação trata-se sobretudo de um mecanismo que serve para aumentar a disponibilidade e não o desempenho. Aliás, a replicação seria bastante útil caso o PostgreSQL usasse algum tipo de mecanismo de Load Balancing como PGPOOL-II, de forma a aumentar o paralelismo e consequentemente o desempenho. No entanto, este facto não se verificou, pelo que o desempenho piorou, muito provavelmente devido às operações necessárias para que a replicação ocorra. Para além disso, o TPC-C evidencia bastantes transações de escritas (contrastando com as transações de leitura), pelo que é mais um fator que não abona a favor da *performance* na implementação da replicação.

6 Conclusão

O primeiro passo do presente trabalho prático consistiu na instalação e configuração do *benchmark* TPC-C. Desta forma, o primeiro desafio prendeu-se na escolha do *hardware* da máquina onde o serviço iria correr.

O grupo debruçou-se bastante sobre este objetivo, uma vez que as possibilidades de CPU, memória, entre outros constituintes, eram infinitas. Tendo sempre também em vista a poupança de recursos monetários, foram efetuados vários testes e instalações do *benchmark* para de certa forma se testar os tempos (de *load*, de *run*,...) da base de dados, alterando também o número de *warehouses* e clientes. No entanto, a opção final do *hardware* acabou por ser tomada mais por bom senso e por uma tentativa de arranjar um equilíbrio entre uma máquina não muito fraca, pois reduziria as possibilidades de otimização, nem uma máquina muito boa, que consumiria demasiado tempo e, consequentemente, recursos monetários.

O objetivo seguinte foi a escolha do número ideal de *warehouses* e de clientes de modo a conseguir encontrar uma configuração de referência. O grupo sentiu algumas dificuldades no decorrer desta busca, mais uma vez devido ao facto da enorme quantidade de combinações existentes, acabando mais tarde por optar definir primeiro o número de *warehouses* e, só depois, o número de clientes. Para encontrar o número de *warehouses*, a estratégia utilizada foi atingir um tamanho da base de dados igual ou superior à RAM existente e encontrar o número de clientes tendo em consideração a maximização do *throughput*. Estas estratégias foram definidas após vários testes com alterações cruzadas de *warehouses* e clientes.

Uma vez encontrada a configuração de referência, o passo seguinte foi tentar otimizar o desempenho da carga transacional tendo em conta, principalmente, os parâmetros de configuração do PostgreSQL. Desta forma, tendo em consideração o que o professor mencionou no decorrer das aulas teórico-práticas, o grupo optou por alterar todas as opções contidas na secção de *Settings*, *Checkpoints* e *Archiving*, presentes no ficheiro de configuração do PostgreSQL. Este processo foi bastante demorado tendo em consideração que são vários parâmetros e, para a maioria dos casos, são bastantes as alternativas de valores existentes. Também o facto de povoarmos sempre novamente a base de dados antes de cada *run* contribuiu para que este procedimento fosse ainda mais demorado. No entanto, esta decisão foi tomada de modo a que não existisse enviesamentos nos resultados e que corrêsemos sempre o programa partindo do mesmo estado.

Com o intuito de maximizar o débito, após testarmos com cada opção individualmente, foi feita uma análise e escolha dos melhores valores para, posteriormente, testarmos com várias opções de diferente carácter, acabando por chegar a uma configuração com um *throughput* e tempo de resposta bastante satisfatórios.

Em seguida, o alvo foi a otimização do desempenho de interrogações analíticas. Neste âmbito, a métrica considerada foi o tempo de resposta de cada *query* e o objetivo prendeu-se em tentar minimizar o mais possível este tempo. Para tal, o grupo tirou proveito do comando *EXPLAIN ANALYZE* para tentar perceber a

estratégia do *postgresql* na execução da *query* e, numa fase posterior, recorreu a índices e a vistas materializadas para tentar otimizar os respetivos desempenhos. Os resultados obtidos foram, do ponto de vista do grupo, bastante satisfatórios visto que foi possível reduzir consideravelmente o tempo de execução de todas as interrogações analíticas.

Por fim, o grupo decidiu enveredar pela implementação de um mecanismo de replicação, tendo recorrido à replicação lógica. Tal como havíamos visto durante as aulas práticas, foi necessário ter dois servidores, funcionado um como *publisher* e o outro como *subscriber*. Tal como vimos, os resultados em termos de métricas são um pouco abaixo dos resultados obtidos com a melhor configuração, reforçando assim as principais vantagens e desvantagens da replicação e circunstâncias onde deverá ser aplicada. Deste modo, conseguimos concluir que este mecanismo é bastante útil no que diz respeito à disponibilidade da base de dados e não ao desempenho em si.

Em suma, a realização deste trabalho permitiu ao grupo alargar os seus conhecimentos relativamente a conceitos expostos nas aulas. Assim, consideramos o resultado do trabalho satisfatório.

A Script de Instalação de Dependências

```
#!/bin/bash

sudo apt install default-jre

sudo apt-get install software-properties-common

sudo apt-add-repository universe

sudo apt-get update

sudo apt-get install maven

sudo apt update

sudo apt install software-properties-common

sudo add-apt-repository ppa:deadsnakes/ppa

sudo apt install python3.7

sudo apt-get install python-numpy python-scipy python-matplotlib
                        ipython ipython-notebook python-pandas
                        python-sympy python-nose

sudo apt-get install python-pip

sudo pip install numpy scipy

sudo apt install postgresql postgresql-contrib

sudo apt-get install unzip

export JAVA_HOME="/usr/"

gsutil cp gs://abd1920/tpcc_100.sql

gsutil cp gs://abd1920/EscadaTPC-C-master.zip .

unzip EscadaTPC-C-master

cd EscadaTPC-C-master

mvn package
```

B Script de Configuração da Base de Dados

```
#!/bin/bash

NAME=$1

sudo -u postgres psql -U postgres -d postgres -c
    "CREATE ROLE $NAME;"

sudo -u postgres psql -U postgres -d postgres -c
    "ALTER USER $NAME WITH SUPERUSER;"

sudo -u postgres psql -U postgres -d postgres -c
    "ALTER USER $NAME WITH CREATEROLE;"

sudo -u postgres psql -U postgres -d postgres -c
    "ALTER USER $NAME WITH CREATEDB;"

sudo -u postgres psql -U postgres -c
    "ALTER USER $NAME WITH LOGIN;"

createdb -U $NAME tpcc;

cd ~/EscadaTPC-C-master/tpc-c-0.1-SNAPSHOT/etc/sql/postgresql

sudo -u postgres psql -U postgres -d tpcc -a -f createtable.sql
sudo -u postgres psql -U postgres -d tpcc -a -f sequence.sql
sudo -u postgres psql -U postgres -d tpcc -a -f createindex.sql
sudo -u postgres psql -U postgres -d tpcc -a -f delivery01
sudo -u postgres psql -U postgres -d tpcc -a -f neworder01
sudo -u postgres psql -U postgres -d tpcc -a -f orderstatus01
sudo -u postgres psql -U postgres -d tpcc -a -f payment01
sudo -u postgres psql -U postgres -d tpcc -a -f stocklevel01
```

C Script de run

```
#!/bin/bash

CL=$1
T=$2

cd EscadaTPC-C-master/tpc-c-0.1-SNAPSHOT/

sudo sed -i "" "s/tpcc.clients = [0-9]*/tpcc.clients = $CL/g" \
etc/workload-config.properties

sudo sed -i "" "s/tpcc.measument=[0-9]*/tpcc.measument=$T/g" \
etc/workload-config.properties
```