

UNIVERSIDADE DO MINHO

ADMINISTRAÇÃO DE BASES DE DADOS  
(4º ANO, 1º SEMESTRE)

---

## Benchmark TPC-C

---

---

### RELATÓRIO DE DESENVOLVIMENTO

---

Mestrado Integrado em Engenharia Informática

Filipa Alves dos Santos, a83631  
Hugo André Coelho Cardoso, a85006  
João da Cunha e Costa, a84775  
Luís Miguel Areira Ramos, a83930  
Válter Ferreira Picas Carvalho, a84464

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Configuração do <i>benchmark</i> TPC-C</b>	<b>4</b>
2.1	Número de <i>warehouses</i> . . . . .	4
2.2	Número de clientes . . . . .	5
2.3	Configuração de referência . . . . .	6
<b>3</b>	<b>Otimização de Interrogações Analíticas</b>	<b>8</b>
3.1	Interrogação analítica 15 . . . . .	8
3.2	Interrogação analítica 12 . . . . .	15
3.3	Interrogação Analítica 10 . . . . .	19
3.4	Interrogação analítica 13 . . . . .	23
3.5	Interrogação analítica 21 . . . . .	26
<b>4</b>	<b>Otimização do desempenho da carga transacional</b>	<b>37</b>
4.1	Settings . . . . .	37
4.1.1	<i>fsync</i> . . . . .	37
4.1.2	<i>synchronous_commit</i> . . . . .	38
4.1.3	<i>WAL_sync_method</i> . . . . .	38
4.1.4	<i>full_page_writes</i> . . . . .	38
4.1.5	<i>wal_buffers</i> . . . . .	39
4.1.6	<i>commit_delay</i> . . . . .	39
4.1.7	<i>commit_siblings</i> . . . . .	39
4.2	Checkpoints . . . . .	40
4.2.1	<i>checkpoint_timeout</i> . . . . .	40
4.2.2	<i>max_wal_size</i> . . . . .	40
4.2.3	<i>min_wal_size</i> . . . . .	40
4.2.4	<i>checkpoint_completion_target</i> . . . . .	41
4.2.5	<i>checkpoint_flush_after</i> . . . . .	41
4.2.6	<i>checkpoint_warning</i> . . . . .	41
4.3	Archiving . . . . .	42
4.3.1	<i>archive_mode</i> . . . . .	42
4.3.2	<i>archive_command</i> . . . . .	42
4.3.3	<i>archive_timeout</i> . . . . .	42
4.4	Combinação dos parâmetros adotados . . . . .	43
4.4.1	Settings . . . . .	43
4.4.2	Checkpoints . . . . .	44
4.4.3	Archiving . . . . .	45
4.4.4	Configuração de referência . . . . .	45
<b>5</b>	<b>Conclusão</b>	<b>47</b>
<b>A</b>	<b>Anexo A</b>	<b>48</b>
<b>B</b>	<b>Anexo B</b>	<b>49</b>
<b>C</b>	<b>Anexo C</b>	<b>50</b>
<b>D</b>	<b>Anexo D</b>	<b>51</b>

## Listas de Figuras

1	Métricas da configuração de referência . . . . .	7
2	Plano de execução da query 15 original . . . . .	9
3	Estatísticas sobre o plano de execução original da query 15 . . . . .	10
4	Plano de execução da query 15 com índices . . . . .	11
5	Plano de execução da query 15 com a primeira versão da vista materializada . . . . .	12
6	Plano de execução da query 15 com a query embebida alterada . . . . .	13
7	Plano de execução da query 15 com a segunda versão da vista materializada . . . . .	14
8	Plano de execução da query 15 com a terceira versão da vista materializada . . . . .	15
9	Plano de execução da query 12 original . . . . .	16
10	Plano de execução da query 12 com o índice sobre a coluna ol_delivery_d . . . . .	16
11	Plano de execução da query 12 com a primeira versão da vista materializada . . . . .	17
12	Plano de execução da query 12 com o índice sobre a vista . . . . .	18
13	Plano de execução da query 12 com a versão final da vista e o índice . . . . .	19
14	Plano de execução da query 10 original . . . . .	20
15	Plano de execução da query 10 com a primeira vista materializada criada . . . . .	21
16	Plano de execução da query 10 com a primeira vista e o respetivo índice . . . . .	22
17	Plano de execução da query 10 com a segunda vista e o respetivo índice . . . . .	23
18	Plano de execução original da query 13 . . . . .	24
19	Plano de execução da query 13 com índice . . . . .	24
20	Plano de execução da query 13 com vista materializada . . . . .	25
21	Plano de execução da query 13 com vista e índice . . . . .	26
22	Plano de execução original da query 21 . . . . .	27
23	Estatísticas do plano de execução original da query 21 . . . . .	28
24	Plano de execução da query 21 com a materialização da tabela <i>supplier</i> . . . . .	29
25	Plano de execução da query 21 com a primeira vista criada e o respetivo índice . . . . .	30
26	Estatísticas do plano de execução da query 21 com uma vista e índice . . . . .	31
27	Plano de execução da query 21 com a segunda vista . . . . .	32
28	Plano de execução da query 21 com o índice na segunda vista . . . . .	33
29	Plano de execução da query 21 com a terceira vista . . . . .	34
30	Plano de execução da query 21 com a terceira vista e respetivo índice . . . . .	35
31	Plano de execução da query 21 com a última vista implementada . . . . .	36
32	Plano de execução da query 21 com a última vista e índice . . . . .	36
33	Métricas da configuração com a nova combinação de <i>settings</i> . . . . .	44
34	Métricas da configuração com a nova combinação de <i>checkpoints</i> . . . . .	45
35	Métricas da configuração com a nova combinação de <i>checkpoints</i> . . . . .	46



## 1 Introdução

O presente trabalho prático foi desenvolvido no âmbito da unidade curricular **Administração de Bases de Dados**, inserida no perfil de mestrado de Engenharia de Aplicações, lecionado no Mestrado Integrado em Engenharia Informática da Universidade do Minho.

O projeto consiste na configuração, otimização e avaliação do *benchmark* **TPC-C** com alguns dados e interrogações adicionais. O TPC-C simula um sistema de bases de dados de uma cadeia de lojas, suportando a operação diária de gestão de vendas e stocks. As interrogações analíticas adicionais, baseadas na adaptação do **TPC-H**, poderão ser corridas ocasionalmente.

A realização deste trabalho passou, numa fase inicial, pela instalação e configuração do *benchmark* TPC-C, recorrendo a inúmeros testes de performance para determinar uma **configuração de referência** óptima em termos de número de *warehouses* e de clientes. De seguida, levou-se a cabo a otimização do **desempenho de interrogações analíticas** tendo em conta, principalmente, os respetivos planos e os mecanismos de redundância em uso e, por fim, o foco mudou para a otimização do **desempenho da carga transacional** através da manipulação dos parâmetros de configuração do PostgreSQL.

O presente relatório procura acompanhar este processo passo a passo, explicando o raciocínio do grupo e exibindo os resultados obtidos.



## 2 Configuração do *benchmark* TPC-C

Para o desenvolvimento deste projeto foi necessário, em primeiro lugar, instalar e configurar o *benchmark* TPC-C, usando uma máquina virtual ”n1-standard-2” da *Google Cloud* para o **servidor PostgreSQL**, que dispõe de 2 CPUs e 7.5 GB de memória. Foi escolhida uma máquina de Iowa (Estados Unidos da América) e série N1, de forma a economizar o saldo disponível para a realização do trabalho, com um disco persistente padrão de 50 GB (para assegurar que a base de dados não ficava sem espaço) e um disco de inicialização Ubuntu 20.04.

O grupo copiou os comandos fornecidos pela *Google Cloud Platform* para criar as máquinas virtuais pela linha de comandos e criou um script com eles, de forma a automatizar o processo para os outros membros e sempre que se pretendesse criar todo o sistema de novo. Este script não está disponível nos anexos do presente relatório porque é muito pouco legível e não envolve lógica implementada pelo grupo.

Além disso, foram também criados *scripts* de configuração das máquinas virtuais (anexos [A](#) e [B](#)) para agilizar o processo.

O primeiro desafio consistiu em descobrir uma configuração de referência em termos de número de *warehouses* e clientes no sistema. O grupo decidiu começar por definir o número de *warehouses*, para perceber a quantidade de dados com que iria trabalhar e, posteriormente, o número de clientes que efetuariam transações sobre esses mesmos dados.

### 2.1 Número de *warehouses*

O principal objetivo desta decisão foi alcançar o **débito máximo** atingível da máquina virtual, no que toca ao desempenho da carga operacional, de maneira a dar o melhor uso possível aos recursos disponíveis, simulando um sistema de maior escala e disponibilizando mais dados para processamento por parte das interrogações analíticas.

Tendo em conta que a máquina virtual possui uma memória RAM de 7.5 GB, o grupo realizou vários testes de maneira a descobrir que número de *warehouses* permitiria obter uma base de dados de tamanho igual ou superior, de forma a possibilitar uma taxa de ocupação de RAM de aproximadamente 100 %. Foi também por este motivo que se decidiu usar um disco com 50 GB de memória, de maneira a dar mais margem aos testes feitos.

O processo consistiu em fazer *load* da base de dados para cada número de *warehouses* considerado, a fim de observar o tamanho da base de dados final, desligando a configuração **fsync** do PostgreSQL, de maneira a tornar o carregamento mais rápido. Os resultados desta experiência são apresentados na seguinte tabela:



Número de <i>warehouses</i>	Tamanho da base de dados
2	256 MB
4	490 MB
8	954 MB
16	1881 MB
64	7116 MB
80	9385 MB

Como é possível observar, a base de dados resultante do uso de 64 *warehouses* ficou muito próxima do tamanho pretendido, pelo que se decidiu aumentar o número de *warehouses* para 80, de maneira a ultrapassar os 7.5 GB. Assim, o grupo optou por este valor para a configuração de referência.

No fim, como foi sugerido pelo docente, foi usado o comando **pg\_dump** para criar um *backup* comprimido dos dados carregados, de maneira a ser possível retomar o estado inicial da base de dados de maneira mais rápida e eficiente, uma vez que o processo de *load* é bastante demorado. Esta decisão permitiu poupar não só tempo ao grupo, como recursos computacionais da máquina da *Cloud*.

## 2.2 Número de clientes

No que toca ao número de clientes, que influenciam diretamente o desempenho das interrogações analíticas, teve-se em consideração, principalmente, o **tempo de resposta** das mesmas, que se pretende minimizar, bem como o *throughput*, que idealmente será o mais elevado possível, e a taxa de aborto, que também se pretende minimizar.

De maneira a obter resultados confiáveis e independentes, o grupo tratou de limpar a base de dados ao fim de cada teste e repor o estado inicial, através do ficheiro *dump* referido anteriormente. Cada teste foi corrido durante 10 minutos e, de maneira a automatizar o processo, foi criado um *script* para a execução da carga transacional, facilitando a especificação do número de clientes (removendo assim a necessidade de editar o ficheiro de configuração com recurso ao Vim entre cada teste). O *script* encontra-se disponível na secção Anexos.



Nº clientes	Throughput (tx/s)	Response time (s)	Abort rate (%)
10	9.247398620786468	2.1469101834282096	0.03213957759412305
20	18.91084349462585	1.0276718724040539	0.04262764434547479
30	20.436619211039478	0.9453241440080564	0.04599495616668668
40	33.7159899125442	0.5750805591122493	0.04659206510681587
50	31.80585673610392	0.6043341570257217	0.03824133088473588
60	30.778887817694482	0.6294932449875226	0.045267827801511666
70	34.52149719145362	0.5612638888888889	0.04529616724738676
80	37.921286275895575	0.5117257814016696	0.04425271360979683
90	34.36667753682265	0.5652729069882376	0.042766581511968965
100	34.47419351500161	0.5611006704693285	0.04683326974437238
110	25.1806684733514	0.7698587443946188	0.04980026631158455

Como é possível observar nos dados apresentados, o desempenho da carga transacional demonstra uma tendência inicial a melhorar com o aumento do número de clientes (com pequenas flutuações), atingindo o seu pico com o valor de 80 clientes, que maximiza o throughput para todos os valores testados, minimiza o *response time* e apresenta uma das taxas de aborto mais baixas obtidas.

A partir desse valor, o aumento do número de clientes torna-se contraprodutivo, levando a um desempenho pior em termos dos parâmetros considerados. Desta forma, o grupo optou por escolher 80 clientes para a configuração de referência.

### 2.3 Configuração de referência

Após os testes realizados, o grupo optou por uma configuração de referência com 80 warehouses e 80 clientes. Eis as métricas obtidas numa execução da carga transacional com este configuração:

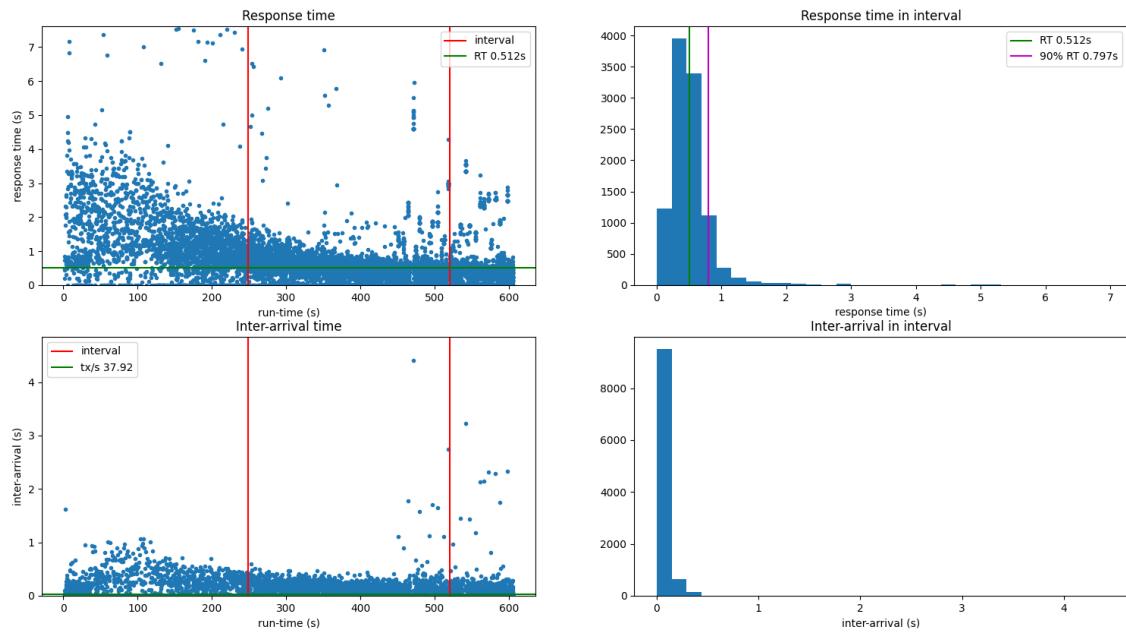


Figura 1: Métricas da configuração de referência



### 3 Otimização de Interrogações Analíticas

O objetivo seguinte do trabalho prático traduzia-se na escolha e adaptação de 5 interrogações analíticas (1 por cada elemento do grupo) do TPC-H, disponibilizadas em <https://db.in.tum.de/research/projects/CHbenCHmark/index.shtml> e na otimização do seu desempenho recorrendo, principalmente, aos respetivos planos e aos mecanismos de redundância em uso.

No que toca às interrogações analíticas, o grupo começou sempre por recorrer ao comando **EXPLAIN ANALYZE** do PostgreSQL para analisar o plano de execução das *queries*, procurando identificar os gargalos e as operações redundantes da aplicação, procedendo então à resolução dos mesmos através da aplicação de índices e vistas materializadas, que levaria a tempos de execução inferiores.

Todas as decisões tomadas foram no sentido de otimizar a *performance* das interrogações ao máximo, tentando também preservar a sua flexibilidade e a liberdade do utilizador, isto é, evitou-se restringir as condições sobre variáveis com valor arbitrário a um único valor (p.e. um que tenha sido considerado pelo grupo durante o seu trabalho) embora isso se traduzisse num grande aumento de desempenho, de maneira a que a *query* pudesse continuar a ser usada em vários cenários, dando ao utilizador a liberdade de escolher os valores que achasse mais interessantes ou adequados.

Houve também sempre o cuidado de verificar que os resultados da interrogação não se alteravam, a cada passo tomado. Houve várias tentativas em que as estratégias implementadas levavam a resultados diferentes, efetivamente mudando a finalidade da interrogação inicial, pelo que foi necessário tomar a precaução de verificar sempre a consistência dos resultados.

Além disso, recorreu-se também ao site <https://explain.depesz.com/> para realizar uma análise mais detalhada e informada das interrogações, visto que esta ferramenta é capaz de fornecer estatísticas úteis sobre a *query* através do respetivo plano de execução como, por exemplo, o tempo total gasto em cada operação do plano e a percentagem correspondente do tempo total de execução. Assim, recebendo estes dados de maneira mais legível, torna-se mais fácil identificar os gargalos da aplicação.

#### 3.1 Interrogação analítica 15

```
with revenue (supplier_no, total_revenue) as (
    select (mod((s_w_id * s_i_id),10000)) as supplier_no,
           sum(ol_amount) as total_revenue
      from order_line, stock
     where ol_i_id = s_i_id and ol_supply_w_id = s_w_id
       and ol_delivery_d >= '2020-12-18 22:15:00.000000'
   group by (mod((s_w_id * s_i_id),10000)))
select su_suppkey, su_name, su_address, su_phone, total_revenue
  from supplier, revenue
```

```

where su_suppkey = supplier_no
      total_revenue = (select max(total_revenue) from revenue)
order by su_suppkey;
  
```

Esta interrogação identifica o(s) fornecedor(es) que mais contribuíram para a receita global em items enviados durante um dado período de tempo. Para começar, como foi referido acima, foi usado o comando **EXPLAIN ANALYZE** para visualizar o plano de execução criado pelo PostgreSQL e aferir o tempo de execução da *query* original, que corresponde a 11088.227 ms, como é possível observar na captura abaixo.

```

Sort  (cost=1361184.91..1361194.91 rows=4000 width=103) (actual time=10966.211..11080.517 rows=1 loops=1)
  Sort Key: supplier.su_suppkey
  Sort Method: quicksort  Memory: 25kB
  CTE revenue
    -> Finalize GroupAggregate  (cost=1086212.19..1324443.58 rows=800000 width=36) (actual time=9818.759..11055.525 rows=10000 loops=1)
        Group Key: (mod((stock.s_w_id * stock.s_i_id), 10000))
        -> Gather Merge  (cost=1086212.19..1298443.58 rows=1600000 width=36) (actual time=9818.707..11016.485 rows=30000 loops=1)
            Workers Planned: 2
            Workers Launched: 2
            -> Partial GroupAggregate  (cost=1085212.17..1112763.86 rows=800000 width=36) (actual time=9763.244..10812.412 rows=10000 loops=3)
                Group Key: (mod((stock.s_w_id * stock.s_i_id), 10000))
                -> Sort  (cost=1085212.17..1089729.40 rows=1806892 width=7) (actual time=9762.538..10245.786 rows=1423504 loops=3)
                    Sort Key: (mod((stock.s_w_id * stock.s_i_id), 10000))
                    Sort Method: external merge Disk: 24600kB
                    Worker 0: Sort Method: external merge Disk: 24392kB
                    Worker 1: Sort Method: external merge Disk: 24792kB
                    -> Parallel Hash Join  (cost=459991.33..872723.71 rows=1806892 width=7) (actual time=6644.582..8404.953 rows=1423504 loops=3)
                        Hash Cond: ((order_line.ol_i_id = stock.s_i_id) AND (order_line.ol_supply_w_id = stock.s_w_id))
                        -> Parallel Seq Scan on order_line  (cost=0.00..364150.59 rows=1766538 width=11) (actual time=1396.362..2061.868 rows=1423504 loops=3)
                            Filter: (ol_delivery_d >= '2020-12-18 22:15:00'::timestamp without time zone)
                            Rows Removed by Filter: 5376598
                        -> Parallel Hash  (cost=396970.33..396970.33 rows=3323333 width=8) (actual time=3807.310..3807.333 rows=2666667 loops=3)
                            Buckets: 131072  Batches: 128  Memory Usage: 3520kB
                            -> Parallel Seq Scan on stock  (cost=0.00..396970.33 rows=3333333 width=8) (actual time=773.196..2365.768 rows=2666667 loops=3)
InitPlan 2 (returns $2)
  -> Aggregate  (cost=18000.00..18000.01 rows=1 width=32) (actual time=1142.848..1142.849 rows=1 loops=1)
      -> CTE Scan on revenue revenue_1  (cost=0.00..16000.00 rows=800000 width=32) (actual time=0.001..1140.067 rows=10000 loops=1)
-> Hash Join  (cost=447.00..18502.00 rows=4000 width=103) (actual time=10965.192..10966.206 rows=1 loops=1)
  Hash Cond: (revenue.supplier_no = supplier.su_suppkey)
  -> CTE Scan on revenue  (cost=0.00..18000.00 rows=4000 width=36) (actual time=10961.858..10962.868 rows=1 loops=1)
      Filter: (total_revenue = $2)
      Rows Removed by Filter: 9999
  -> Hash  (cost=32.00..322.00 rows=10000 width=71) (actual time=3.308..3.309 rows=10000 loops=1)
      Buckets: 16384  Batches: 1  Memory Usage: 1144kB
      -> Seq Scan on supplier  (cost=0.00..322.00 rows=10000 width=71) (actual time=0.019..1.486 rows=10000 loops=1)
Planning Time: 0.472 ms
JIT:
  Functions: 76
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 18.930 ms, Inlining 336.341 ms, Optimization 1228.837 ms, Emission 752.439 ms, Total 2336.547 ms
Execution Time: 11088.227 ms
(41 rows)
  
```

Figura 2: Plano de execução da query 15 original

O grupo reparou que a operação *CTE Scan* executada pelo programa ocupava a vasta maioria do tempo de execução do programa, após introduzir o plano de execução no <https://explain.depesz.com/>.

Per node type stats			
node type	count	sum of times	% of query
Aggregate	1	2.782 ms	0.0 %
CTE Scan	2	12,102.935 ms	109.2 %
Finalize GroupAggregate	1	39.040 ms	0.4 %
Gather Merge	1	204.073 ms	1.8 %
Hash	1	1.823 ms	0.0 %
Hash Join	1	0.029 ms	0.0 %
Parallel Hash	1	1,441.565 ms	13.0 %
Parallel Hash Join	1	2,535.752 ms	22.9 %
Parallel Seq Scan	2	4,427.636 ms	40.0 %
Partial GroupAggregate	1	566.626 ms	5.1 %
Seq Scan	1	1.486 ms	0.0 %
Sort	2	1,840.833 ms	16.6 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
<b>order_line</b>	1	2,061.868 ms	18.6 %
Parallel Seq Scan	1	2,061.868 ms	100.0 %
<b>stock</b>	1	2,365.768 ms	21.4 %
Parallel Seq Scan	1	2,365.768 ms	100.0 %
<b>supplier</b>	1	1.486 ms	0.0 %
Seq Scan	1	1.486 ms	100.0 %

Figura 3: Estatísticas sobre o plano de execução original da query 15

Após alguma pesquisa sobre esta operação, descobriu-se que um *CTE Scan* consiste num scan sequencial de uma tabela intermediária de resultados materializados criada durante a execução da interrogação, neste caso a tabela *revenue*. Como tal, o elevado tempo de execução deve-se, na realidade, à *query* embedida na criação desta tabela intermediária, pois o tempo do *CTE Scan* inclui o tempo de criação da mesma.

Numa primeira abordagem, o grupo reparou que a interrogação filtrava as linhas da tabela *order\_line* cuja data de entrega fosse superior a um determinado valor e, posteriormente, fazia a sua união com a tabela *supplier* pela *su\_suppkey*. Dado que cada uma destas operações selecionaria apenas uma determinada percentagem das linhas da tabela, foram criados os seguintes índices, de maneira a tornar sua identificação mais rápida e reduzir os acessos na *heap* às linhas pretendidas:

```
create index q10.ol_delivery_d on order_line(ol_delivery_d);
create index q10.su_suppkey on supplier(su_suppkey);
```

Correndo a *query* de novo, foi possível observar que o programa decidiu que seria benificial a utilização dos índices, pelo que esta mudança teve um impacto positivo no desempenho da interrogação, embora pouco significativo, diminuindo o tempo de execução para 10177.341 ms.



```
Merge Join  (cost=1146153.24..1146733.38 rows=4000 width=103) (actual time=10055.804..10169.418 rows=1 loops=1)
  Merge Cond: (supplier.su_suppkey = revenue.supplier_no)
  CTE revenue
    --> Finalize GroupAggregate  (cost=871682.24..1109913.03 rows=800000 width=36) (actual time=8522.213..9773.854 rows=10000 loops=1)
      Group Key: (mod(stock.s_w_id * stock.s_i_id), 10000)
    --> Gather Merge  (cost=871682.24..1083913.03 rows=1000000 width=36) (actual time=8522.072..9730.887 rows=30000 loops=1)
        Workers Planned: 2
        Workers Launched: 2
          --> Partial GroupAggregate  (cost=870682.21..898233.90 rows=800000 width=36) (actual time=8443.431..9456.143 rows=10000 loops=3)
            Group Key: (mod(stock.s_w_id * stock.s_i_id), 10000)
            --> Sort  (cost=870682.21..875199.44 rows=1806892 width=7) (actual time=8442.747..8936.843 rows=1423504 loops=3)
              Sort Key: (mod(stock.s_w_id * stock.s_i_id), 10000)
              Sort Method: external merge Disk: 25896kB
              Worker 0: Sort Method: external merge Disk: 23502kB
              Worker 1: Sort Method: external merge Disk: 24304kB
            --> Parallel Hash Join  (cost=459991.89..658193.75 rows=1806892 width=7) (actual time=5287.785..7073.482 rows=1423504 loops=3)
              Hash Cond: ((order_line.ol_i_id = stock.s_i_id) AND (order_line.ol_supply_w_id = stock.s_w_id))
              --> Parallel Index Scan using q10_su_suppkey on supplier  (cost=0.29..495.43 rows=10000 width=7) (actual time=2.061..1203.885 rows=1423504 loops=3)
                Index Cond: (ol_delivery_d >= '2020-12-18 22:15:00'::timestamp without time zone)
              --> Parallel Hash Join  (cost=396970.33..396970.33 rows=3333333 width=8) (actual time=3367.446..3367.448 rows=2666667 loops=3)
                Buckets: 131072 Batches: 128 Memory Usage: 35204kB
                --> Parallel Seq Scan on stock  (cost=0.00..396970.33 rows=3333333 width=8) (actual time=431.517..1965.843 rows=2666667 loops=3)

Init Plan 2 (returns $2)
--> Aggregate  (cost=18000.00..18000.01 rows=1 width=32) (actual time=1146.056..1146.057 rows=1 loops=1)
  --> CTE Scan on revenue revenue_1  (cost=0.00..10000.00 rows=800000 width=32) (actual time=0.001..1143.795 rows=10000 loops=1)
--> Index Scan using q10_su_suppkey on supplier  (cost=0.29..495.43 rows=10000 width=7) (actual time=0.057..0.970 rows=2001 loops=1)
--> Sort  (cost=0.00..0.00 rows=2 width=32) (actual time=9671.487..9671.489 rows=1 loops=1)
  Sort Key: revenue.supplier_no
  Sort Method: quicksort Memory: 25kB
--> CTE Scan on revenue  (cost=0.00..10000.00 rows=4000 width=36) (actual time=9669.398..9671.469 rows=1 loops=1)
  Filter: (total_revenue = $2)
  Rows Removed by Filter: 9999
Planning Time: 0.864 ms
JIT:
  Functions: 74
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 18.879 ms, Inlining 379.780 ms, Optimization 758.037 ms, Emission 537.890 ms, Total 1694.586 ms
Execution Time: 10177.341 ms
(38 rows)
```

Figura 4: Plano de execução da query 15 com índices

De seguida, o grupo virou a sua atenção para as operações realizadas sobre a tabela *stock*. Deu-se conta que se perdia muito tempo no cálculo de  $\text{mod}((s\_w\_id * s\_i\_id), 10000)$  para cada linha da tabela e que, fora isso, eram apenas usadas duas colunas da tabela como condições de união.

Analisando a fundo a tabela em questão, foi tirada a conclusão de que a coluna mais importante e mais frequentemente alterada era a *s\_quantity*, que diz respeito à quantidade de stock disponível de cada item, em cada *warehouse*. Além de esta coluna não ser acedida na interrogação, as colunas usadas (*s\_i\_id* e *s\_w\_id*) são, à partida, imutáveis (correspondem ao id do item e do *warehouse*), pelo que se decidiu colocar as mesmas numa vista materializada, bem como o  $\text{mod}((s\_w\_id * s\_i\_id), 10000)$ . Desta maneira, deixaria de ser necessário aceder à tabela *stock* na *heap* em tempo de execução, bem como fazer aquelas operações aritméticas para cada linha. Uma vez que os dados materializados podem ser considerados estáticos, esta vista materializada também precisaria de ser atualizada muito raramente.

Como tal, foi criada a vista mencionada e adaptou-se a interrogação analítica à mesma.

```
create materialized view q10_mv as
select s_i_id, s_w_id, (mod((s_w_id * s_i_id),10000)) as supplier_no
from stock;

with revenue (supplier_no, total_revenue) as (
  select supplier_no, sum(ol_amount) as total_revenue
  from order_line, q10_mv
  where ol_i_id = s_i_id and ol_supply_w_id = s_w_id
    and ol_delivery_d >= '2020-12-18 22:15:00.000000'
  group by supplier_no)
select su_suppkey, su_name, su_address, su_phone, total_revenue
from supplier, revenue
```

```

where su_suppkey = supplier_no
      and total_revenue = (select max(total_revenue) from revenue)
order by su_suppkey;
  
```

Executando a nova *query* com o comando **EXPLAIN ANALYZE**, foi possível observar que o seu desempenho voltou a melhorar, passando de 10177.341 ms para 7918.078 ms.

```

Sort  (cost=554910.39..554910.50 rows=45 width=10) (actual time=7790.097..7912.695 rows=1 loops=1)
  Sort Key: supplier.su_suppkey
  Sort Method: quicksort Memory: 25kB
  CTE revenue
    > Finalize GroupAggregate  (cost=551844.15..554217.62 rows=9099 width=36) (actual time=7745.280..7905.498 rows=10000 loops=1)
      Group Key: q10_mv.supplier_no
      > Other Merge  (cost=551844.15..553967.40 rows=18198 width=36) (actual time=7745.247..7884.207 rows=30000 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        > Sort  (cost=550844.13..550866.88 rows=9099 width=36) (actual time=7538.626..7540.896 rows=10000 loops=3)
          Sort Key: q10_mv.supplier_no
          Sort Method: quicksort Memory: 1166kB
          Worker 0: Sort Method: quicksort Memory: 1166kB
          Worker 1: Sort Method: quicksort Memory: 1166kB
          > Partial HashAggregate  (cost=550132.07..550245.80 rows=9099 width=36) (actual time=7519.555..7534.079 rows=10000 loops=3)
            Group Key: q10_mv.supplier_no
            > Parallel Hash Join  (cost=142853.26..541997.97 rows=18066820 width=7) (actual time=4744.335..6546.529 rows=1423504 loops=3)
              Hash Cond: ((order_line.ol_i_id = q10_mv.s_i_id) AND (order_line.ol_supply_w_id = q10_mv.s_w_id))
              > Parallel Index Scan using q10_mv_delivery_d on order_line  (cost=0.56..149620.64 rows=1766538 width=11) (actual time=1.261..1079.589 rows=1423504 loops=3)
              > Parallel Hash  (cost=76577.08..76577.08 rows=3333388 width=12) (actual time=2932.273..2932.276 rows=2666667 loops=3)
                Buckets: 131072 Batches: 128 Memory Usage: 4000kB
                > Parallel Seq Scan on q10_mv  (cost=0.00..70577.08 rows=3333308 width=12) (actual time=626.451..1458.099 rows=2666667 loops=3)
Planning Time: 0.431 ms
InitPlan 2 (returns $2)
  >> Aggregate  (cost=204.73..2044.74 rows=1 width=2) (actual time=43.238..43.239 rows=1 loops=1)
      >> CTE Scan on revenue revenue_  (cost=0.00..181.98 rows=9099 width=32) (actual time=0.001..41.622 rows=10000 loops=1)
-> Nested Loop  (cost=0.29..480.79 rows=45 width=3) (actual time=7790.157..7790.895 rows=1 loops=1)
  > CTE Scan on revenue revenue_  (cost=0.00..204.73 rows=45 width=30) (actual time=7789.050..7790.785 rows=1 loops=1)
    Filter: (total_revenue = $2)
    Rows Removed by Filter: 9999
  > Index Scan using q10_su_suppkey on supplier  (cost=0.29..6.26 rows=1 width=71) (actual time=0.087..0.089 rows=1 loops=1)
    Index Cond: ($u_suppkey = revenue.supplier_no)
Index Cond: ($u_suppkey = revenue.supplier_no)

Functions: 71
Options: Inlining true, Optimization true, Expressions true, Deforming true
Timing: Generation 20.632 ms, Inlining 406.112 ms, Optimization 858.603 ms, Emission 613.102 ms, Total 1898.449 ms
Execution Time: 7918.078 ms
(38 rows)
  
```

Figura 5: Plano de execução da query 15 com a primeira versão da vista materializada

De seguida, o grupo reparou que a interrogação selecionava apenas os fornecedores que produziram mais receita (*select max(total\_revenue) from revenue*), descartando tudo o resto, pelo que surgiu a ideia de tentar identificar a receita máxima logo na criação da tabela temporária *revenue*, devolvendo apenas uma linha, de maneira a diminuir o trabalho do programa nas uniões posteriores. Como tal, a *query* foi reescrita da seguinte forma:

```

with revenue (supplier_no, total_revenue) as (
  select supplier_no, sum(ol_amount) as total_revenue
  from order_line, q10_mv
  where ol_i_id = s_i_id and ol_supply_w_id = s_w_id
        and ol_delivery_d >= '2020-12-18 22:15:00.000000'
  group by supplier_no order by total_revenue desc limit 1)
select su_suppkey, su_name, su_address, su_phone, total_revenue
from supplier, revenue
where su_suppkey = supplier_no
      and total_revenue = (select max(total_revenue) from revenue)
order by su_suppkey;
  
```

Contudo, esta alteração acabou por não ter nenhum efeito significativo na performance da query, ou até por aumentar o tempo de execução, devido à carga de processamento extra derivada da nova operação **LIMIT**. Como tal, esta alteração foi revertida.

```

Sort (cost=554271.50..554271.50 rows=1 width=103) (actual time=7890.752..8021.710 rows=1 loops=1)
  Sort Key: supplier.su_suppkey
  Sort Method: quicksort Memory: 25kB
  CTE revenue
    -> LHM1 (cost=554269.12..554269.12 rows=1 width=36) (actual time=7890.591..8021.552 rows=1 loops=1)
        Sort Key: (sum(order_line.ol_amount)) DESC
        Sort Method: top-N heapsort Memory: 25kB
        -> Finalize GroupAggregate (cost=551844.15..554217.62 rows=9099 width=36) (actual time=7492.668..7654.218 rows=10000 loops=1)
            Group Key: q10_mv.supplier_no
            -> Gather Merge (cost=551844.15..553967.40 rows=18198 width=36) (actual time=7492.634..7632.379 rows=30000 loops=1)
                Workers Planned: 2
                Workers Actually Used: 2
                -> Sort (cost=550844.13..550866.88 rows=9099 width=36) (actual time=7450.801..7452.952 rows=10000 loops=3)
                    Sort Key: q10_mv.supplier_no
                    Sort Method: quicksort Memory: 1166kB
                    Worker 0: Sort Method: quicksort Memory: 1166kB
                    Worker 1: Sort Method: quicksort Memory: 1166kB
                    -> Partial HashAggregate (cost=550132.07..550245.88 rows=9099 width=36) (actual time=7432.274..7444.908 rows=10000 loops=3)
                        Group Key: q10_mv.supplier_no
                        -> Parallel Hash 2nd (cost=142863.26..541097.97 rows=1066820 width=?) (actual time=4592.993..6447.870 rows=1423504 loops=3)
                            Hash Cond: ((order_line.ol_i_id = q10_mv.s_i_id) AND (order_line.ol_supply_w_id = q10_mv.s_w_id))
                            -> Parallel Index Scan using q10_mv.delivery_d on order_line (cost=0..56..149620.64 rows=1766538 width=11) (actual time=2.324..1190.807 rows=1423504 loops=3)
                                Index Cond: (ol_delivery_d >= '2020-12-18 22:15:00'::timestamp without time zone)
                            -> Parallel Hash (cost=65777.08..76577.08 rows=3333308 width=12) (actual time=2752.027..2752.030 rows=40000B
                                Buckets: 131072 Batches: 128 Memory Usage: 40000B
                                -> Parallel Seq Scan on q10_mv (cost=0..00..76577.08 rows=3333308 width=12) (actual time=397.301..1326.169 rows=2666667 loops=3)
                        Planning Time: 0.597 ms
Planning Time: 0.597 ms
 JIT:
  Functions: 72
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 18.620 ms, Inlining 385.005 ms, Optimization 705.773 ms, Emission 464.373 ms, Total 1573.770 ms
Execution Time: 8025.698 ms
(41 rows)

```

Figura 6: Plano de execução da query 15 com a query embebida alterada

Posteriormente, decidiu-se incluir mais alguma da lógica da interrogação na vista materializada, realizando as uniões com as colunas da tabela *stock* diretamente na mesma (*ol\_i\_id = s\_i\_id and ol\_supply\_w\_id = s\_w\_id*). Desta forma, a interrogação passaria a aceder aos resultados já filtrados dessas operações, não sendo necessário realizá-las em tempo de execução.

Optou-se por deixar fora da vista a condição sobre a data de maneira a conservar alguma flexibilidade da query, dado que o utilizador continua a poder escolher qualquer data que pretenda para esta interrogação, não sendo restringido a uma única como aconteceria se essa condição fosse colocada na vista materializada.

```

create materialized view q10_mv as
select (mod((s_w_id * s_i_id),10000)) as supplier_no, ol_delivery_d, ol_amount
from stock, order_line
where ol_i_id = s_i_id and ol_supply_w_id = s_w_id;

with revenue (supplier_no, total_revenue) as (
  select supplier_no, sum(ol_amount) as total_revenue
  from q10_mv
  where ol_delivery_d >= '2020-12-18 22:15:00.000000'
  group by supplier_no)
select su_suppkey, su_name, su_address, su_phone, total_revenue
from supplier, revenue
where su_suppkey = supplier_no
  and total_revenue = (select max(total_revenue) from revenue)
order by su_suppkey;

```

Esta alteração teve um impacto bastante positivo na performance da interrogação analítica, observando-se uma redução significativa do tempo de execução para 3117.893 ms.

```

Sort (cost=249775.58..249775.69 rows=46 width=103) (actual time=3111.143..3114.358 rows=1 loops=1)
  Sort Key: supplier.su_suppkey
  Sort Method: quicksort Memory: 25kB
  CTE revenue
    -> Finalize GroupAggregate (cost=246693.96..249073.43 rows=9122 width=36) (actual time=3073.057..3106.650 rows=10000 loops=1)
        Group Key: q10_mv.supplier_no
      -> Gather Merge (cost=246693.96..248822.58 rows=18244 width=36) (actual time=3073.026..3084.488 rows=30000 loops=1)
          Workers Planned: 2
          Workers Launched: 2
            -> Sort (cost=245693.94..245716.74 rows=9122 width=36) (actual time=3035.993..3038.043 rows=10000 loops=3)
                Sort Key: q10_mv.supplier_no
                Sort Method: quicksort Memory: 1166kB
                Worker 0: Sort Method: quicksort Memory: 1166kB
                Worker 1: Sort Method: quicksort Memory: 1166kB
            -> Partial HashAggregate (cost=244979.91..245093.93 rows=9122 width=36) (actual time=3016.312..3031.511 rows=10000 loops=3)
                Group Key: q10_mv.supplier_no
              -> Parallel Seq Scan on q10_mv (cost=0.00..236188.56 rows=1758269 width=7) (actual time=38.841..1866.544 rows=1423504 loops=3)
                  Filter: (ol_delivery_d >='2020-12-18 22:15:00':timestamp without time zone)
                  Rows Removed by Filter: 5376598
Planning Plan 2 (returns $2)
  -> Aggregate (cost=205.25..205.25 rows=1 width=32) (actual time=35.204..35.205 rows=1 loops=1)
      -> CTE Scan on revenue revenue_1 (cost=0.00..182.44 rows=9122 width=32) (actual time=0.001..33.525 rows=10000 loops=1)
  -> Nested Loop (cost=0.29..495.62 rows=46 width=103) (actual time=3108.883..3111.130 rows=1 loops=1)
      -> CTE Scan on revenue (cost=0.00..205.25 rows=46 width=36) (actual time=3108.841..3111.085 rows=1 loops=1)
          Filter: (total_revenue = $2)
          Rows Removed by Filter: 9999
      -> Index Scan using q10_su_suppkey on supplier (cost=0.29..6.30 rows=1 width=71) (actual time=0.027..0.028 rows=1 loops=1)
          Index Cond: (su_suppkey = revenue.supplier_no)
Planning Time: 0.225 ms
JIT:
  Functions: 41
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 6.951 ms, Inlining 0.000 ms, Optimization 8.557 ms, Emission 86.877 ms, Total 102.384 ms
Execution Time: 3117.893 ms
(34 rows)

```

Figura 7: Plano de execução da query 15 com a segunda versão da vista materializada

Por fim, procurou-se ainda criar um índice na coluna *ol\_delivery\_d* da vista materializada, de maneira a tornar mais rápida a verificação da condição sobre a mesma.

```
create index q15_mv_date on q15_mv(ol_delivery_d);
```

Contudo, após correr a *query* de novo, o grupo reparou que o programa optou por não usar este índice. Acontece que o PostgreSQL guarda a informação em memória por blocos de linhas, em vez de guardar todas as linhas individuais, de maneira a caber tudo em memória. Como a informação da vista não estava ordenada, acabava por haver linhas que faziam *match* com a condição na maioria dos blocos, pelo que o programa teria de fazer muitos acessos à *heap* através do índice, o que não compensaria. Tendo isto em conta, o grupo refez a vista materializada anterior, ordenando-a segundo a data, de maneira a concentrar as linhas que interessavam nos mesmos blocos de memória, reduzindo o número de acessos à *heap* pelo índice. Depois, criou-se o índice de novo e voltou-se a correr a interrogação.

```

create materialized view q10_mv as
select (mod((s_w_id * s_i_id),10000)) as supplier_no, ol_delivery_d, ol_amount
from stock, order_line
where ol_i_id = s_i_id and ol_supply_w_id = s_w_id
order by ol_delivery_d desc;

```

```

Sort (cost=141755.97..141756.09 rows=46 width=103) (actual time=2211.718..2211.817 rows=1 loops=1)
  Sort Key: supplier.su.supkey
  Sort Method: quicksort  Memory: 25kB
  CTE revenue
    -> Finalize GroupAggregate (cost=138658.76..141051.53 rows=9173 width=36) (actual time=2167.768..2205.500 rows=10000 loops=1)
        Group Key: q10_mv.supplier_no
        -> Gather Merge (cost=138658.76..140799.27 rows=18346 width=36) (actual time=2167.738..2183.577 rows=30000 loops=1)
            Workers Planned: 2
            Workers Launched: 2
            -> Sort (cost=137658.73..137681.66 rows=9173 width=36) (actual time=1957.710..1959.721 rows=10000 loops=3)
                Sort Key: q10_mv.supplier_no
                Sort Method: quicksort  Memory: 1166kB
                Worker 0: Sort Method: quicksort  Memory: 1166kB
                Worker 1: Sort Method: quicksort  Memory: 1166kB
                -> Partial HashAggregate (cost=136940.34..137655.00 rows=9173 width=36) (actual time=1939.594..1952.416 rows=10000 loops=3)
                    Group Key: q10_mv.supplier_no
                    -> Parallel Index Scan using q10_mv_date on q10_mv  (cost=0.56..127767.19 rows=1834630 width=7) (actual time=0.073..925.214 rows=1423504 loops=3)
                        Index Cond: (ol_delivery_d >= '2020-12-18 22:15:00'::timestamp without time zone)
Planning Time: 0.255 ms
JIT:
  Functions: 41
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 6.887 ms, Inlining 0.000 ms, Optimization 2.882 ms, Emission 48.322 ms, Total 58.092 ms
Execution Time: 2214.901 ms
(33 rows)

```

Figura 8: Plano de execução da query 15 com a terceira versão da vista materializada

Verificou-se que o programa passou a usar o índice após esta alteração, diminuindo ainda mais o tempo de execução da interrogação analítica, que começou em 11088.227 ms e acabou em 2214.901 ms.

### 3.2 Interrogação analítica 12

```

select o.ol_cnt,
       sum(case when o_carrier_id = 1 or o_carrier_id = 2 then 1 else 0 end)
             as high_line_count,
       sum(case when o_carrier_id <> 1 and o_carrier_id <> 2 then 1 else 0 end)
             as low_line_count
  from orders, order_line
 where ol_w_id = o_w_id
   and ol_d_id = o_d_id
   and ol_o_id = o_id
   and o_entry_d <= ol_delivery_d
   and ol_delivery_d < '2020-12-18 21:30:00.000000'
 group by o.ol_cnt
 order by o.ol_cnt;

```

Esta interrogação analítica conta a quantidade de encomendas agrupadas pelo número de linhas de pedido em cada encomenda, tendo em conta o número de encomendas que são enviadas conforme a sua prioridade. O tempo de execução inicial é de 15089.288 ms.

```

Finalize GroupAggregate (cost=1097640.98..1097643.56 rows=10 width=20) (actual time=15035.280..15062.716 rows=10 loops=1)
  Group Key: orders.o.ol_cnt
    > Gather Merge (cost=1097640.98..1097643.31 rows=20 width=20) (actual time=15035.253..15062.684 rows=30 loops=1)
      Workers Planned: 2
      Workers Launched: 2
        > Sort (cost=1096640.95..1096640.98 rows=10 width=20) (actual time=14797.902..14797.907 rows=10 loops=3)
          Sort Key: orders.o.ol_cnt
          Sort Method: quicksort Memory: 25kB
          Worker 0: Sort Method: quicksort Memory: 25kB
          Worker 1: Sort Method: quicksort Memory: 25kB
            > Partial HashAggregate (cost=1096640.69..1096640.79 rows=10 width=20) (actual time=14797.857..14797.865 rows=10 loops=3)
              Group Key: orders.o.ol_cnt
              > Merge Join (cost=1019585.41..1086953.42 rows=533558 width=8) (actual time=9650.640..14216.043 rows=1343740 loops=3)
                Merge Cond: ((order_line.ol_o_id = orders.o_id) AND (order_line.ol_w_id = orders.o_w_id) AND (order_line.ol_d_id = orders.o_d_id))
                Join Filter: (orders.o.entry_d <= order_line.ol_delivery_d)
                > Sort (cost=603968.79..608122.53 rows=1661496 width=20) (actual time=4523.825..5159.897 rows=1343740 loops=3)
                  Sort Key: order_line.ol_o_id, order_line.ol_w_id, order_line.ol_d_id
                  Sort Method: external merge Disk: 43832kB
                  Worker 0: Sort Method: external merge Disk: 43848kB
                  Worker 1: Sort Method: external merge Disk: 46520kB
                  > Sort (cost=415614.24..427614.24 rows=2400000 width=28) (actual time=5126.755..7201.091 rows=3577229 loops=3)
                    Filter: (ol_delivery_d < '2020-12-18 21:30:00':timestamp without time zone)
                    Rows Removed by Filter: 5456362
                    > Materialize (cost=415614.24..427614.24 rows=2400000 width=28) (actual time=5126.748..6194.379 rows=2399356 loops=3)
                      Sort Key: orders.o_id, orders.o_w_id, orders.o_d_id
                      Sort Method: external merge Disk: 98720kB
                      Worker 0: Sort Method: external merge Disk: 98720kB
                      Worker 1: Sort Method: external merge Disk: 98720kB
                      > Seq Scan on orders (cost=0.00..46430.00 rows=2400000 width=28) (actual time=5.783..1334.850 rows=2400000 loops=3)

Planning Time: 0.401 ms
JIT:
  Functions: 87
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 22.229 ms, Inlining 306.233 ms, Optimization 1493.639 ms, Emission 941.996 ms, Total 2824.097 ms
Execution Time: 15089.288 ms
(36 rows)

```

Figura 9: Plano de execução da query 12 original

A primeira tentativa de otimização consistiu na implementação de um índice sobre a coluna *ol\_delivery\_d* da tabela *order\_line*, de maneira a tornar mais rápida a filtragem segundo a condição da data.

```
create index q12.ol_delivery_d on order_line(ol_delivery_d);
```

Esta alteração ajudou a otimizar o desempenho da *query* em cerca de 2 segundos, baixando o tempo de execução para 13550.733 ms.

```

Finalize GroupAggregate (cost=874214.84..874217.42 rows=10 width=20) (actual time=13494.524..13524.969 rows=10 loops=1)
  Group Key: orders.o.ol_cnt
    > Gather Merge (cost=874214.84..874217.17 rows=20 width=20) (actual time=13494.501..13524.939 rows=30 loops=1)
      Workers Planned: 2
      Workers Launched: 2
        > Sort (cost=873214.82..873214.84 rows=10 width=20) (actual time=13286.944..13286.950 rows=10 loops=3)
          Sort Key: orders.o.ol_cnt
          Sort Method: quicksort Memory: 25kB
          Worker 0: Sort Method: quicksort Memory: 25kB
          Worker 1: Sort Method: quicksort Memory: 25kB
            > Partial HashAggregate (cost=873214.55..873214.65 rows=10 width=20) (actual time=13286.912..13286.918 rows=10 loops=3)
              Group Key: orders.o.ol_cnt
              > Merge Join (cost=796159.28..863527.28 rows=533558 width=8) (actual time=8184.011..12738.093 rows=1343740 loops=3)
                Merge Cond: ((order_line.ol_o_id = orders.o_id) AND (order_line.ol_w_id = orders.o_w_id) AND (order_line.ol_d_id = orders.o_d_id))
                Join Filter: (orders.o.entry_d <= order_line.ol_delivery_d)
                > Sort (cost=415614.24..427614.24 rows=2400000 width=20) (actual time=3413.203..4030.989 rows=1343740 loops=3)
                  Sort Key: order_line.ol_o_id, order_line.ol_w_id, order_line.ol_d_id
                  Sort Method: external merge Disk: 41280kB
                  Worker 0: Sort Method: external merge Disk: 49008kB
                  Worker 1: Sort Method: external merge Disk: 43920kB
                  > Parallel Index Scan using q12.ol_delivery_d on order_line (cost=0.56..140724.46 rows=1661496 width=20) (actual time=885.385..1827.962 rows=1343740 loops=3)
                    Index Cond: (ol_delivery_d < '2020-12-18 21:30:00':timestamp without time zone)
                    > Materialize (cost=415614.24..427614.24 rows=2400000 width=20) (actual time=4770.731..6838.006 rows=3583148 loops=3)
                      Sort Key: orders.o_id, orders.o_w_id, orders.o_d_id
                      Sort Method: external merge Disk: 98720kB
                      Worker 0: Sort Method: external merge Disk: 98720kB
                      Worker 1: Sort Method: external merge Disk: 98720kB
                      > Seq Scan on orders (cost=0.00..46430.00 rows=2400000 width=28) (actual time=0.052..799.993 rows=2400000 loops=3)

Planning Time: 0.338 ms
JIT:
  Functions: 87
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 21.410 ms, Inlining 350.006 ms, Optimization 1416.428 ms, Emission 886.930 ms, Total 2674.774 ms
Execution Time: 13550.733 ms
(35 rows)

```

Figura 10: Plano de execução da query 12 com o índice sobre a coluna *ol\_delivery\_d*

De seguida, o grupo decidiu criar uma vista materializada para guardar os resultados das condições de união entre as tabelas *orders* e *order\_line*, deixando de fora apenas a condição onde é especificado o valor da data, de maneira a preservar grande parte da versatilidade da interrogação. Após adaptar a *query* à nova vista, mediu-se o seu tempo de execução

de novo e verificou-se que diminuiu bastante, para 2818.943 ms, uma redução do tempo de aproximadamente **4,81x**.

```

create materialized view q12_mv as
select o.ol_cnt, o.carrier_id, o.entry_d, ol.delivery_d
from orders, order_line
where ol.w_id = o.w_id
    and ol.d_id = o.d_id
    and ol.o_id = o.id
    and o.entry_d <= ol.delivery_d
order by ol.delivery_d;

select o.ol_cnt,
       sum(case when o.carrier_id = 1 or o.carrier_id = 2 then 1 else 0 end)
             as high_line_count,
       sum(case when o.carrier_id <> 1 and o.carrier_id <> 2 then 1 else 0 end)
             as low_line_count
  from q12_mv
 where ol.delivery_d < '2020-12-18 21:30:00.000000'
 group by o.ol_cnt
 order by o.ol_cnt;
  
```

```

-----
Finalize GroupAggregate (cost=266836.31..266838.89 rows=10 width=20) (actual time=2817.099..2817.186 rows=10 loops=1)
  Group Key: o.ol_cnt
-> Gather Merge (cost=266836.31..266838.64 rows=20 width=20) (actual time=2817.052..2817.136 rows=30 loops=1)
    Workers Planned: 2
    Workers Launched: 2
      -> Sort (cost=265836.28..265836.31 rows=10 width=20) (actual time=2785.751..2785.754 rows=10 loops=3)
          Sort Key: o.ol_cnt
          Sort Method: quicksort Memory: 25kB
          Worker 0: Sort Method: quicksort Memory: 25kB
          Worker 1: Sort Method: quicksort Memory: 25kB
            -> Partial HashAggregate (cost=265836.02..265836.12 rows=10 width=20) (actual time=2785.707..2785.712 rows=10 loops=3)
                Group Key: o.ol_cnt
                  -> Parallel Seq Scan on q12_mv (cost=0.00..236191.21 rows=1693989 width=8) (actual time=22.285..2084.587 rows=1343740 loops=3)
                      Filter: (ol.delivery_d < '2020-12-18 21:30:00'::timestamp without time zone)
                      Rows Removed by Filter: 5456362
Planning Time: 0.125 ms
JIT:
  Functions: 30
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 13.459 ms, Inlining 0.000 ms, Optimization 2.342 ms, Emission 59.241 ms, Total 75.041 ms
Execution Time: 2818.943 ms
(21 rows)
  
```

Figura 11: Plano de execução da query 12 com a primeira versão da vista materializada

Além disso, é possível observar no excerto de código acima correspondente à criação da vista materializada que a mesma foi ordenada pela data. Isto foi implementado já a pensar no próximo passo, a criação de um índice sobre essa coluna da vista, que vai desempenhar o mesmo papel que o primeiro índice. A ordenação da vista segundo a data era importante pelo mesmo motivo que na interrogação 15, de maneira a condensar todas as linhas de interesse nos mesmos blocos de memória para o uso do índice se tornar mais proveitoso.

```
create index q12_mv_index on q12_mv(ol_delivery_d);
```

Executando a *query* de novo, é possível observar que o programa o usa, de facto, passando a realizar um *Parallel Index Scan* em vez do anterior *Parallel Seq Scan*, o que leva o tempo de execução a 1683.087 ms.

```

Finalize GroupAggregate (cost=148619.89..148622.47 rows=10 width=20) (actual time=1680.674..1681.407 rows=10 loops=1)
  Group Key: o.ol_cnt
-> Gather Merge (cost=148619.89..148622.22 rows=20 width=20) (actual time=1680.662..1681.388 rows=30 loops=1)
    Workers Planned: 2
    Workers Launched: 2
      -> Sort (cost=147619.87..147619.89 rows=10 width=20) (actual time=1650.356..1650.358 rows=10 loops=3)
          Sort Key: o.ol_cnt
          Sort Method: quicksort Memory: 25kB
          Worker 0: Sort Method: quicksort Memory: 25kB
          Worker 1: Sort Method: quicksort Memory: 25kB
        -> Partial HashAggregate (cost=147619.60..147619.70 rows=10 width=20) (actual time=1650.302..1650.306 rows=10 loops=3)
            Group Key: o.ol_cnt
            -> Parallel Index Scan using q12_mv_index on q12_mv (cost=0.56..117974.97 rows=1693979 width=8) (actual time=0.072..903.003 rows=1343740 loops=3)
                Index Cond: (ol_delivery_d < '2020-12-18 21:30:00'::timestamp without time zone)
Planning Time: 0.169 ms
JIT:
  Functions: 30
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 5.307 ms, Inlining 0.000 ms, Optimization 2.261 ms, Emission 53.292 ms, Total 60.861 ms
Execution Time: 1683.087 ms
(20 rows)

```

Figura 12: Plano de execução da query 12 com o índice sobre a vista

Por fim, o grupo reparou que a interrogação fazia duas operações *case* sobre a coluna *o\_carrier\_id* nas linhas filtradas das condições de união, fazendo a soma desses valores para todas as linhas de maneira a obter os dados *high\_line\_count* e *low\_line\_count*.

Logo, deduziu-se que não era necessário guardar o valor da coluna *o\_carrier\_id*, podendo guardar-se antes os resultados dos *cases* diretamente na vista materializada, o que pouparia algum tempo de execução à *query*. Como tal, a vista foi refeita desta maneira e voltou a criar-se o índice sobre a coluna *ol\_delivery\_d* da mesma, adaptando-se depois a *query* à nova lógica.

```

create materialized view q12_mv as
select o.ol_cnt,
       (case when o_carrier_id = 1 or o_carrier_id = 2 then 1 else 0 end)
             as high_line,
       (case when o_carrier_id <> 1 and o_carrier_id <> 2 then 1 else 0 end)
             as low_line,
       o_entry_d, ol_delivery_d
from orders, order_line
where ol_w_id = o_w_id
      and ol_d_id = o_d_id
      and ol_o_id = o_id
      and o_entry_d <= ol_delivery_d
order by ol_delivery_d;

create index q12_mv_date on q12_mv(ol_delivery_d);

select o.ol_cnt,
       sum(high_line) as high_line_count,
       sum(low_line) as low_line_count

```

```
from q12_mv
where ol_delivery_d < '2020-12-18 21:30:00.000000'
group by o.ol_cnt
order by o.ol_cnt;
```

Embora não tenha tido um impacto significativo, esta última mudança ajudou a baixar mais um bocado o tempo de execução da interrogação analítica, acabando em 1562.789 ms.

```
Finalize GroupAggregate (cost=135585.63..135588.21 rows=10 width=20) (actual time=1552.895..1561.131 rows=10 loops=1)
  Group Key: o.ol_cnt
-> Gather Merge (cost=135585.63..135587.96 rows=20 width=20) (actual time=1552.853..1561.103 rows=30 loops=1)
    Workers Planned: 2
    Workers Launched: 2
      -> Sort (cost=134585.61..134585.63 rows=10 width=20) (actual time=1525.630..1525.632 rows=10 loops=3)
          Sort Key: o.ol_cnt
          Sort Method: quicksort Memory: 25kB
          Worker 0: Sort Method: quicksort Memory: 25kB
          Worker 1: Sort Method: quicksort Memory: 25kB
            -> Parallel HashAggregate (cost=134585.34..134585.44 rows=10 width=20) (actual time=1525.567..1525.571 rows=10 loops=3)
                Group Key: o.ol_cnt
                  -> Parallel Index Scan using q12_mv_date on q12_mv (cost=0.56..121889.05 rows=1692839 width=12) (actual time=0.077..851.402 rows=1343740 loops=3)
                    Index Cond: (ol_delivery_d < '2020-12-18 21:30:00'::timestamp without time zone)
Planning Time: 0.104 ms
JIT:
  Functions: 30
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 9.041 ms, Inlining 0.000 ms, Optimization 2.216 ms, Emission 69.606 ms, Total 80.863 ms
Execution Time: 1562.789 ms
(20 rows)
```

Figura 13: Plano de execução da query 12 com a versão final da vista e o índice

### 3.3 Interrogação Analítica 10

```
select c_id, c_last, sum(ol_amount) as revenue, c_city, c_phone, n_name
from customer, orders, order_line, nation
where c_id = o_c_id
  and c_w_id = o_w_id
  and c_d_id = o_d_id
  and ol_w_id = o_w_id
  and ol_d_id = o_d_id
  and ol_o_id = o_id
  and o_entry_d >= '2020-12-18 20:38:30.000000'
  and o_entry_d <= ol_delivery_d
  and n_nationkey = ascii(substr(c_state,1,1))
group by c_id, c_last, c_city, c_phone, n_name
order by revenue desc;
```

Esta interrogação analisa os custos de todos os clientes, listando o seu país de residência, alguns dados pessoais e a quantidade de dinheiro que gastaram nas suas encomendas desde uma data específica, e apresenta a lista de clientes por ordem da quantidade de encomendas.

Contudo, esta *query* tem um erro que a impede de funcionar. A condição *n\_nationkey* = *ascii(substr(c\_state,1,1))* nunca se verifica, uma vez que o valor de *n\_nationkey* vai de 0 a 24, enquanto que os valores de *ascii(substr(c\_state,1,1))* se encontram nos intervalos 49 a 57 e 97 a 122. Como tal, foi preciso corrigir esta condição, pelo que a interrogação ficou com o seguinte aspeto:

```

select c_id, c_last, sum(ol_amount) as revenue, c_city, c_phone, n_name
from customer, orders, order_line, nation
where c_id = o_c_id
  and c_w_id = o_w_id
  and c_d_id = o_d_id
  and ol_w_id = o_w_id
  and ol_d_id = o_d_id
  and ol_o_id = o_id
  and o_entry_d >= '2020-12-18 20:38:30.000000'
  and o_entry_d <= ol_delivery_d
  and n_nationkey = ascii(substr(c_state,1,1)) -
    case when substr(c_state,1,1) >= '1' and substr(c_state,1,1) <= '9'
      then ascii('1') else ascii('a') end;
group by c_id, c_last, c_city, c_phone, n_name
order by revenue desc;
  
```

Através do comando **EXPLAIN ANALYZE**, verificou-se que o tempo de execução inicial era 51078.533 ms, o que é bastante elevado.

```

Sort {cost=593536.61..594637.17 rows=440227 width=195} (actual time=50648.493..50963.016 rows=1225334 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC
  Sort Method: external merge Disk: 119800kB
-> Finalize GroupAggregate {cost=402324.03..468003.97 rows=440227 width=195} (actual time=37831.039..48904.061 rows=1225334 loops=1)
  Group Key: customer.c_last, customer.c_city, customer.c_phone, nation.n_name
    -> Gather Merge {cost=402324.03..456881.16 rows=166856 width=195} (actual time=37831.016..46586.392 rows=1225334 loops=1)
      Workers Planned: 2
      Workers Launched: 2
        -> Partial GroupAggregate {cost=407234.01..412736.85 rows=183428 width=195} (actual time=37633.936..44464.390 rows=408445 loops=3)
          Group Key: customer.c_last, customer.c_city, customer.c_phone, nation.n_name
          -> Sort {cost=407234.01..407692.58 rows=183428 width=195} (actual time=37633.856..41601.082 rows=3470435 loops=3)
            Sort Key: customer.c_id, customer.c_last, customer.c_city, customer.c_phone, nation.n_name
            Sort Method: external merge Disk: 330728kB
          Worker 0: Sort Method: external merge Disk: 344880kB
          Worker 1: Sort Method: external merge Disk: 329816kB
          -> Nested Loop {cost=407234.01..412736.85 rows=183428 width=166} (actual time=6460.799..20901.600 rows=2470435 loops=3)
            Join Filter: ((customer.c_id = order_line.ol_w_id) AND (customer.c_id = order_line.ol_d_id))
              -> Nested Loop {cost=1.99..252225.31 rows=64667 width=191} (actual time=6459.389..12499.709 rows=408445 loops=3)
                -> Hash Join {cost=1.56..187581.56 rows=125080 width=171} (actual time=1789.021..6573.663 rows=788266 loops=3)
                  Hash Cond: ((ascii(substr((customer.c_state)::text, 1, 1)) >= '1')::text) AND (substr((customer.c_state)::text, 1, 1) <= '9')::text)
                  THEN 49 ELSE 97 END = nation.n_nationkey
                  -> Parallel Scan on nation {cost=0.08..1788.00 rows=1000000 width=108} (actual time=0.040..1799.602 rows=800000 loops=3)
                    -> Hash {cost=1.25..1.25 rows=25 width=108} (actual time=1788.898..1788.900 rows=25 loops=3)
                      Buckets: 1024 Batches: 1 Memory Usage: 10kB
                      -> Seq Scan on nation {cost=0.00..1.25 rows=25 width=108} (actual time=1788.853..1788.864 rows=25 loops=3)
-> Index Scan using tx_orders on orders {cost=0.43..0.51 rows=1 width=24} (actual time=0.007..0.007 rows=1 loops=2364799)
  Filter: ((o_w_id = customer.c_w_id) AND (o_d_id = customer.c_d_id) AND (o_c_id = customer.c_id))
  Filters: (customer.c_last >= '2020-12-18 20:38:30'::timestamp without time zone)
  Rows Removed by Filter: 0
-> Index Scan using pk_order_line on order_line {cost=0.56..1.87 rows=3 width=23} (actual time=0.010..0.018 rows=8 loops=1225334)
  Index Cond: ((ol_w_id = orders.o_w_id) AND (ol_d_id = orders.o_d_id) AND (ol_o_id = orders.o_id))
  Filter: (orders.o_entry_d <= ol_delivery_d)
Planning Time: 1.263 ms
JIT:
  Functions: 111
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 31.321 ns, Inlining 414.890 ns, Optimization 2986.668 ns, Emission 1960.459 ns, Total 5393.338 ms
Execution Time: 51078.533 ms
(37 rows)
  
```

Figura 14: Plano de execução da query 10 original

O primeiro passo de otimização consistiu na criação de uma vista materializada que incorporasse as condições de união entre as tabelas para selecionar apenas a informação relevante, de forma a aliviar a carga computacional a realizar em tempo de execução. Ficou fora apenas a condição onde é especificado o valor da data de entrada da encomenda, de maneira a preservar a liberdade do utilizador de correr a interrogação para a data que entender. Todos os restantes parâmetros das condições incluídas são estáticos no contexto da *query*, pelo que a sua materialização não está a infringir a liberdade de escolha do utilizador. Optou-se também por deixar fora da vista o cálculo das somas de *ol\_amount* e o *GROUP BY*, de forma a manter alguma da estrutura inicial da interrogação.

Além disso, dado que a interrogação ia percorrer a vista em busca das linhas cuja data fosse superior ou igual a um determinado valor, ordenou-se ainda a mesma segundo esta coluna, de maneira a permitir a implementação de um índice para tornar a identificação e acesso à informação relevante mais rápida. Desta maneira, a vista criada e a interrogação adaptada à mesma são as seguintes:

```

create materialized view q10_mv as
select c_id, c_last, ol_amount, c_city, c_phone, n_name, o_entry_d
from customer, orders, order_line, nation
where c_id = o_c_id
    and c_w_id = o_w_id
    and c_d_id = o_d_id
    and ol_w_id = o_w_id
    and ol_d_id = o_d_id
    and ol_o_id = o_id
    and o_entry_d <= ol_delivery_d
    and n_nationkey = ascii(substr(c_state,1,1)) -
        case when substr(c_state,1,1) >= '1' and substr(c_state,1,1) <= '9'
            then ascii('1') else ascii('a') end
order by o_entry_d desc;

select c_id, c_last, sum(ol_amount) as revenue, c_city, c_phone, n_name
from q10_mv
where o_entry_d >= '2020-12-18 20:38:30.000000'
group by c_id, c_last, c_city, c_phone, n_name
order by revenue desc;
  
```

O tempo de execução com esta vista materializada baixou para 33918.964 ms, pelo que é possível concluir que a alteração foi bastante eficaz.

```

Sort (cost=2436795.46..2441817.55 rows=2008833 width=117) (actual time=33489.229..33806.106 rows=1225334 loops=1)
  Sort Key: (sum(ol_amount)) DESC
  Sort Method: external merge Disk: 119000kB
-> Finalize GroupAggregate (cost=1318767.36..1979307.48 rows=2008833 width=117) (actual time=20388.675..31653.935 rows=1225334 loops=1)
    Group Key: c_id, c_last, c_city, c_phone, n_name
    -> Gather Merge (cost=1318767.36..1883887.91 rows=4017666 width=117) (actual time=20388.645..29286.249 rows=1302957 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial GroupAggregate (cost=1317767.34..1419149.54 rows=2008833 width=117) (actual time=20134.676..26991.253 rows=434319 loops=3)
            Group Key: c_id, c_last, c_city, c_phone, n_name
            -> Sort (cost=1317767.34..1328663.31 rows=4358388 width=88) (actual time=20134.604..24110.569 rows=3470435 loops=3)
                Sort Key: c_id, c_last, c_city, c_phone, n_name
                Sort Method: external merge Disk: 322448kB
                Worker 0: Sort Method: external merge Disk: 343744kB
                Worker 1: Sort Method: external merge Disk: 338440kB
                -> Parallel Seq Scan on q10_mv (cost=0.00..420022.19 rows=4358388 width=88) (actual time=656.962..3977.018 rows=3470435 loops=3)
                    Filter: (o_entry_d >= '2020-12-18 20:38:30'::timestamp without time zone)
                    Rows Removed by Filter: 3229993
Planning Time: 0.351 ms
JIT:
  Functions: 38
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 10.146 ms, Inlining 370.104 ms, Optimization 970.473 ms, Emission 628.294 ms, Total 1979.018 ms
Execution Time: 33918.964 ms
(24 rows)
  
```

Figura 15: Plano de execução da query 10 com a primeira vista materializada criada

De seguida, foi criado o índice mencionado anteriormente sobre a coluna da data de entrada das encomendas, na vista:

```
create index q10_mv_date on q10_mv(o_entry_d);
```

Apesar de não ter tido um efeito muito significativo no desempenho, é possível verificar que o PostgreSQL optou por o usar e o tempo de execução diminuiu cerca de 2.5 segundos, para 31690.197 ms.

```
Sort  (cost=2417787.69..2422809.77 rows=2008833 width=117) (actual time=31263.595..31580.525 rows=1225334 loops=1)
  Sort Key: (sum(ol_amount)) DESC
  Sort Method: external merge Disk: 119800kB
-> Finalize GroupAggregate  (cost=1299759.59..1960299.70 rows=2008833 width=117) (actual time=18443.936..29569.404 rows=1225334 loops=1)
    Group Key: c_id, c_last, c_city, c_phone, n_name
    -> Gather Merge  (cost=1299759.59..1864880.14 rows=4017660 width=117) (actual time=18443.813..27247.137 rows=1241967 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial GroupAggregate  (cost=1298759.56..1400141.77 rows=2008833 width=117) (actual time=18132.765..24955.461 rows=413989 loops=3)
            Group Key: c_id, c_last, c_city, c_phone, n_name
            -> Sort  (cost=1298759.56..1309655.53 rows=4358388 width=88) (actual time=18132.688..22111.930 rows=3470435 loops=3)
                Sort Method: external merge Disk: 326520kB
                Worker 0: Sort Method: external merge Disk: 336864kB
                Worker 1: Sort Method: external merge Disk: 340864kB
                -> Parallel Index Scan using q10_mv_date on q10_mv  (cost=0.56..401014.41 rows=4358388 width=88) (actual time=685.514..3567.721 rows=3470435 loops=3)
                    Index Cond: (o_entry_d >= '2020-12-18 20:38:30'::timestamp without time zone)

Planning Time: 0.215 ms
JIT:
  Functions: 38
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 7.093 ms, Inlining 309.893 ms, Optimization 1035.153 ms, Emission 627.002 ms, Total 1979.141 ms
Execution Time: 31690.197 ms
(23 rows)
```

Figura 16: Plano de execução da query 10 com a primeira vista e o respetivo índice

Por fim, o grupo decidiu adicionar mais alguma lógica na vista, sacrificando parte da estrutura original restante da interrogação, de maneira a ter ainda mais informação pré-processada, diminuindo a carga computacional da *query*. Como tal, a soma das quantias de cada cliente e o *GROUP BY* passaram também para a vista. De seguida, foi criado outra vez o mesmo índice sobre esta vista também, repetindo o raciocínio anterior.

```
create materialized view q10_mv2 as
select c_id, c_last, sum(ol_amount) as revenue, c_city, c_phone, n_name, o_entry_d
from customer, orders, order_line, nation
where c_id = o_c_id
  and c_w_id = o_w_id
  and c_d_id = o_d_id
  and ol_w_id = o_w_id
  and ol_d_id = o_d_id
  and ol_o_id = o_id
  and o_entry_d <= ol_delivery_d
  and n_nationkey = ascii(substr(c_state,1,1)) -
      case when substr(c_state,1,1) >= '1' and substr(c_state,1,1) <= '9'
          then ascii('1') else ascii('a') end
group by c_id, c_last, c_city, c_phone, n_name, o_entry_d
order by o_entry_d desc;

create index q10_mv2_date on q10_mv2(o_entry_d);
```

```

select c_id, c_last, revenue, c_city, c_phone, n_name
from q10_mv2
where o_entry_d >= '2020-12-18 20:38:30.000000'
order by revenue desc;
  
```

A nova interrogação é executada em apenas 2075.094 ms, o que constitui uma melhoria tremenda em relação aos 51078.533 ms iniciais. Contudo, o grupo não considera que esta última vista criada seja uma boa solução, pois perde-se quase toda a integridade da *query* original, deixando-a dependente sobretudo de resultados pré-processados e tornando-a muito menos fléxivel, por consequência.

Além disso, como a vista é criada através de tabelas que não são imutáveis - *customer*, *orders* e *order\_line* - e calcula dados muitos específicos ao contexto da *query*, executando várias combinações de chaves e até um *GROUP BY*, acaba por ser uma estrutura de dados muito pouco versátil, visto que rapidamente ficaria desatualizada com nova atividade na base de dados. Logo, seria necessário voltar a calcular a vista muito frequentemente, para garantir a viabilidade dos seus dados, pelo que acaba por ser uma solução pouco recomendável.

```

Gather Merge (cost=122979.68..242515.36 rows=1024520 width=89) (actual time=1071.846..1981.153 rows=1225334 loops=1)
  Workers Planned: 2
  Workers Launched: 2
    -> Sort (cost=121979.66..123260.31 rows=512260 width=89) (actual time=1026.842..1292.351 rows=408445 loops=3)
        Sort Key: ol_amount sum DESC
        Sort Method: external merge Disk: 35176kB
        Worker 0: Sort Method: external merge Disk: 41656kB
        Worker 1: Sort Method: external merge Disk: 42272kB
        -> Parallel Index Scan using q10_mv2_date on q10_mv2 (cost=0.43..47136.72 rows=512260 width=89) (actual time=17.357..445.807 rows=408445 loops=3)
            Index Cond: (o_entry_d >= '2020-12-18 20:38:30'::timestamp without time zone)
Planning Time: 0.258 ms
JIT:
  Functions: 12
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 3.161 ms, Inlining 0.000 ms, Optimization 11.132 ms, Emission 31.931 ms, Total 46.223 ms
Execution Time: 2075.094 ms
(16 rows)
  
```

Figura 17: Plano de execução da query 10 com a segunda vista e o respetivo índice

### 3.4 Interrogação analítica 13

```

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

Este interrogação lista o número de clientes agrupado e ordenado pela dimensão das suas encomendas. O conjunto de dados resultante da relação entre clientes e a dimensão das suas

encomendas é ordenado, por sua vez, segundo a dimensão das encomendas e faz-se a contagem do número de clientes apresentam resultados iguais.

```

Sort  (cost=320184.85..320185.35 rows=200 width=16) (actual time=2612.031..2612.039 rows=75 loops=1)
  Sort Key: (count(*)) DESC, c_orders.c_count DESC
  Sort Method: quicksort  Memory: 28KB
-> GroupAggregate  (cost=320152.70..320177.20 rows=200 width=16) (actual time=2611.544..2611.999 rows=75 loops=1)
    Group Key: c_orders.c_count
->  Sort  (cost=320152.70..320160.20 rows=3000 width=8) (actual time=2611.533..2611.737 rows=3000 loops=1)
      Sort Key: c_orders.c_count DESC
      Sort Method: quicksort  Memory: 237kB
        -> Subquery Scan on c_orders  (cost=319919.44..319979.44 rows=3000 width=8) (actual time=2610.396..2611.089 rows=3000 loops=1)
          -> HashAggregate  (cost=319919.44..319949.44 rows=3000 width=12) (actual time=2610.394..2610.807 rows=3000 loops=1)
            Group Key: customer.c_id
            -> Hash Right Join  (cost=237719.06..387919.44 rows=2400000 width=8) (actual time=1232.064..2227.398 rows=2400000 loops=1)
              Hash Cond: ((orders.o_w_id = customer.c_w_id) AND (orders.o_d_id = customer.c_d_id) AND (orders.o_c_id = customer.c_id))
              -> Seq Scan on orders  (cost=0.00..52430.00 rows=343040 width=16) (actual time=0.055..313.227 rows=336334 loops=1)
                Filter: (o_carrier_id > 8)
                Rows Removed by Filter: 2063666
              -> Hash  (cost=184000.00..184000.00 rows=2400000 width=12) (actual time=1230.649..1230.650 rows=2400000 loops=1)
                Buckets: 131072  Batches: 64  Memory Usage: 2639kB
                -> Seq Scan on customer  (cost=0.00..184000.00 rows=2400000 width=12) (actual time=12.711..726.612 rows=2400000 loops=1)
Planning Time: 0.292 ms
JIT:
  Functions: 26
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 2.318 ms, Inlining 0.000 ms, Optimization 0.587 ms, Emission 11.757 ms, Total 14.662 ms
Execution Time: 2614.510 ms
(25 rows)

```

Figura 18: Plano de execução original da query 13

Como é possível observar na imagem acima, esta *query* tem um plano de execução relativamente simples, sendo realizada em 2614.510 ms. Contudo, é possível diminuir ainda mais esse tempo.

O grupo reparou que uma das primeiras operações que o PostgreSQL fazia era um scan sequencial da tabela *orders*, que é bastante grande, em busca das linhas cujo valor da coluna *o\_carrier\_id* fosse superior a um determinado número. Como tal, decidiu-se implementar um índice sobre essa coluna:

```
create index q13_orders on orders(o_carrier_id);
```

Correndo a *query* de novo com o comando **EXPLAIN ANALYZE**, foi possível verificar que o programa passou a realizar um *Bitmap Index Scan* seguido de um *Bitmap Heap Scan*, deixando de percorrer a tabela *orders* na sua íntegra e reduzindo assim o número de acessos à *heap*. Não se observou uma melhoria muito significativa, mas ainda assim o tempo de execução diminui para os 2438.336 ms.

```

Sort  (cost=300903.84..300904.34 rows=200 width=16) (actual time=2435.714..2435.723 rows=75 loops=1)
  Sort Key: (count(*)) DESC, c_orders.c_count DESC
  Sort Method: quicksort  Memory: 28KB
-> GroupAggregate  (cost=300871.69..300896.19 rows=200 width=16) (actual time=2435.283..2435.696 rows=75 loops=1)
    Group Key: c_orders.c_count
->  Sort  (cost=300871.69..300879.19 rows=3000 width=8) (actual time=2435.275..2435.448 rows=3000 loops=1)
      Sort Key: c_orders.c_count DESC
      Sort Method: quicksort  Memory: 237kB
        -> Subquery Scan on c_orders  (cost=300638.43..300698.43 rows=3000 width=8) (actual time=2434.146..2434.834 rows=3000 loops=1)
          -> HashAggregate  (cost=300638.43..300668.43 rows=3000 width=12) (actual time=2434.143..2434.554 rows=3000 loops=1)
            Group Key: customer.c_id
            -> Hash Right Join  (cost=244149.99..288638.43 rows=2400000 width=8) (actual time=1224.065..2038.577 rows=2400000 loops=1)
              Hash Cond: ((orders.o_w_id = customer.c_w_id) AND (orders.o_d_id = customer.c_d_id) AND (orders.o_c_id = customer.c_id))
              -> Bitmap Heap Scan on orders  (cost=6430.99..33148.99 rows=343040 width=16) (actual time=20.829..162.951 rows=336334 loops=1)
                Recheck Cond: (o_carrier_id > 8)
                Heap Blocks: exact=16430
                -> Bitmap Index Scan on q13_orders  (cost=0.00..6345.23 rows=343040 width=0) (actual time=17.985..17.986 rows=336334 loops=1)
                  Index Cond: (o_carrier_id > 8)
                -> Hash  (cost=184000.00..184000.00 rows=2400000 width=12) (actual time=1202.136..1202.137 rows=2400000 loops=1)
                  Buckets: 131072  Batches: 64  Memory Usage: 2639kB
                  -> Seq Scan on customer  (cost=0.00..184000.00 rows=2400000 width=12) (actual time=12.488..693.168 rows=2400000 loops=1)
Planning Time: 0.297 ms
JIT:
  Functions: 26
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 2.452 ms, Inlining 0.000 ms, Optimization 0.553 ms, Emission 11.600 ms, Total 14.604 ms
Execution Time: 2438.336 ms
(27 rows)

```

Figura 19: Plano de execução da query 13 com índice

De seguida, o grupo virou a sua atenção para o scan sequencial da tabela *customers*, através da qual se realizavam as combinações de chaves de maneira a extrair os ids dos clientes que verificassem a condição na coluna *o\_carrier\_id*. Tendo isto em conta, o grupo procurou incorporar parte da *query* embbebida, nomeadamente as seleções e as combinações de chaves, numa vista materializada, a fim de ter a informação relevante já pré-processada, evitando assim ter de aceder e percorrer a tabela *customer* em tempo de execução.

A condição sobre a coluna *o\_carrier\_id* ficou de fora da vista por questões de flexibilidade, permitindo ao utilizador correr a *query* com o valor que quisesse.

```
create materialized view q13_mv as
select o_carrier_id, c_id, count(o_id) as o_id_count
from customer left outer join orders on (
c_w_id = o_w_id
    and c_d_id = o_d_id
    and c_id = o_c_id)
group by o_carrier_id, c_id
order by o_carrier_id desc;

select c_count, count(*) as custdist
from (select c_id, sum(o_id_count) from q13_mv
      where o_carrier_id > 8
      group by c_id
      order by c_id) as c_orders (c_id, c_count)
group by c_count
order by custdist desc, c_count desc;
```

```
Sort (cost=626.97..627.47 rows=200 width=40) (actual time=4.581..4.586 rows=74 loops=1)
  Sort Key: (count(*)) DESC, (sum(q13_mv.o_id_count)) DESC
  Sort Method: quicksort Memory: 28kB
-> HashAggregate (cost=617.32..619.32 rows=200 width=40) (actual time=4.543..4.556 rows=74 loops=1)
    Group Key: (sum(q13_mv.o_id_count))
    -> Sort (cost=575.92..581.83 rows=2366 width=36) (actual time=3.839..3.977 rows=2100 loops=1)
        Sort Key: q13_mv.c_id
        Sort Method: quicksort Memory: 195kB
        -> HashAggregate (cost=413.75..443.33 rows=2366 width=36) (actual time=2.744..3.363 rows=2100 loops=1)
            Group Key: q13_mv.c_id
            -> Seq Scan on q13_mv  (cost=0.00..392.75 rows=4200 width=12) (actual time=0.015..1.735 rows=4200 loops=1)
                Filter: (o_carrier_id > 8)
                Rows Removed by Filter: 17700
Planning Time: 0.086 ms
Execution Time: 4.647 ms
(15 rows)
```

Figura 20: Plano de execução da query 13 com vista materializada

Esta mudança teve um impacto enorme no desempenho da interrogação, baixando o tempo de execução de 2438.336 ms para 4.647 ms. Contudo, o grupo tinha ainda mais uma otimização em mente: colocar um índice na coluna *o\_carrier\_id* da vista criada, semelhante ao índice inicialmente criado, que já não era usado uma vez que a *query* deixara de aceder à tabela *orders* em tempo de execução. Por estes motivos, os resultados da vista foram agrupados

e ordenados segundo essa coluna, permitindo uma organização dos dados em memória mais propícia ao uso de dito índice:

```
create index q13_mv_i on q13_mv(o_carrier_id);
```

```
Sort  (cost=397.86..398.36 rows=200 width=40) (actual time=3.590..3.596 rows=74 loops=1)
  Sort Key: (count(*)) DESC, (sum(q13_mv.o_id_count)) DESC
  Sort Method: quicksort  Memory: 28kB
-> HashAggregate  (cost=388.21..390.21 rows=200 width=40) (actual time=3.551..3.564 rows=74 loops=1)
    Group Key: (sum(q13_mv.o_id_count))
    -> Sort  (cost=346.81..352.72 rows=2366 width=36) (actual time=2.879..2.994 rows=2100 loops=1)
        Sort Key: q13_mv.c_id
        Sort Method: quicksort  Memory: 195kB
        -> HashAggregate  (cost=184.64..214.22 rows=2366 width=36) (actual time=1.928..2.403 rows=2100 loops=1)
            Group Key: q13_mv.c_id
            -> Index Scan using q13_mv_i on q13_mv  (cost=0.29..163.64 rows=4200 width=12) (actual time=0.022..0.872 rows=4200 loops=1)
                Index Cond: (o_carrier_id > 8)
Planning Time: 0.101 ms
Execution Time: 3.661 ms
(14 rows)
```

Figura 21: Plano de execução da query 13 com vista e índice

Assim, o tempo de execução acabou por descer ainda mais, para 3.661 ms. O grupo ficou satisfeito com as otimizações implementadas, visto que foi possível diminuir bastante o tempo de execução, preservando grande parte da estrutura e flexibilidade da interrogação original.

### 3.5 Interrogação analítica 21

```
select su_name, count(*) as numwait
from supplier, order_line l1, orders, stock, nation
where ol_o_id = o_id
  and ol_w_id = o_w_id
  and ol_d_id = o_d_id
  and ol_w_id = s_w_id
  and ol_i_id = s_i_id
  and mod((s_w_id * s_i_id),10000) = su_suppkey
  and l1.ol_delivery_d > o_entry_d
  and not exists (
    select * from order_line l2
    where l2.ol_o_id = l1.ol_o_id
      and l2.ol_w_id = l1.ol_w_id
      and l2.ol_d_id = l1.ol_d_id
      and l2.ol_delivery_d > l1.ol_delivery_d)
  and su_nationkey = n_nationkey
  and n_name = 'GERMANY'
group by su_name
order by numwait desc, su_name;
```

Esta *query* determina os fornecedores que enviaram determinados items de uma encomenda de forma não cronológica, para um determinado país. Através do comando **EX-**



**PLAIN ANALYZE**, é possível observar que o seu plano de execução é bastante complexo, sendo que demora 70582.875 ms a ser realizado, isto é, cerca de 1m10s.

```
Sort (cost=2304234.59..2304259.58 rows=10000 width=34) (actual time=69779.030..70560.639 rows=396 loops=1)
  Sort Key: (count(*)) DESC, supplier.su_name
  Sort Method: quicksort Memory: 55kB
-> Finalize GroupAggregate (cost=2300382.36..2303579.29 rows=10000 width=34) (actual time=69695.938..70560.238 rows=396 loops=1)
    Group Key: supplier.su_name
-> Gather Merge (cost=2300382.36..2303370.20 rows=20000 width=34) (actual time=69695.648..70559.658 rows=1188 loops=1)
  Workers Planned: 2
  Workers Launched: 2
    -> Partial GroupAggregate (cost=2299382.34..2300061.68 rows=10000 width=34) (actual time=69594.117..69647.613 rows=396 loops=3)
      Group Key: supplier.su_name
      -> Sort (cost=2299382.34..2299575.45 rows=77245 width=26) (actual time=69593.802..69626.880 rows=88980 loops=3)
        Sort Key: supplier.su_name
        Sort Method: external merge Disk: 3112kB
        Worker 0: Sort Method: external merge Disk: 3184kB
        Worker 1: Sort Method: external merge Disk: 3152kB
        -> Parallel Hash Anti Join (cost=2011452.52..2299259.64 rows=77245 width=26) (actual time=45197.392..69126.860 rows=88980 loops=3)
          Hash Cond: ((l1.ol_o_id = l2.ol_o_id) AND (l1.ol_w_id = l2.ol_w_id) AND (l1.ol_d_id = l2.ol_d_id))
          Join Filter: (l2.ol_delivery_d > l1.ol_delivery_d)
          Rows Removed by Join Filter: 780831
          -> Merge Join (cost=1469994.01..1507811.35 rows=115867 width=46) (actual time=32232.587..35556.795 rows=268143 loops=3)
            Merge Cond: ((l1.ol_delivery_d = orders.o_id) AND (l1.ol_w_id = orders.o_w_id) AND (l1.ol_d_id = orders.o_d_id))
            Join Filter: (l1.ol_delivery_d > orders.o_entry_d)
            -> Sort (cost=1070785.30..1071054.81 rows=347772 width=50) (actual time=27664.728..27950.137 rows=268143 loops=3)
              Sort Key: (l1.ol_delivery_d, l1.ol_w_id, l1.ol_d_id)
              Sort Method: external merge Disk: 1732kB
              Worker 0: Sort Method: external merge Disk: 1723kB
              Worker 1: Sort Method: external merge Disk: 1769kB
              -> Parallel Hash Join (cost=528578.02..1026887.32 rows=347772 width=50) (actual time=25166.267..27893.385 rows=268143 loops=3)
                Hash Cond: ((stocks.s_w_id = l1.ol_w_id) AND (stocks.s_i_id = l1.ol_i_id))
                -> Hash Join (cost=369.82..423673.49 rows=133333 width=34) (actual time=8.453..16825.557 rows=105260 loops=3)
                  Hash Cond: ((mod(stocks.s_w_id * stocks.s_i_id, 10000) = supplier.su_suppkey)
                  -> Parallel Seq Scan on stock (cost=0..396970.33 rows=3333333 width=30) (actual time=4.965..16081.935 rows=2666667 loops=3)
                  -> Hash (cost=364.02..364.82 rows=400 width=30) (actual time=3.353..3.358 rows=396 loops=3)
                    Buckets: 1024 Batches: 1 Memory Usage: 33kB
                    -> Hash Join (cost=1..32..364.82 rows=400 width=30) (actual time=0.098..3.242 rows=396 loops=3)
                      Hash Cond: (supplier.s_nationkey = nation.nationkey)
                      -> Seq Scan on nation (cost=0..322.00 rows=10000 width=8) (actual time=0.022..1.766 rows=10000 loops=3)
                      -> Hash (cost=1..31..31 rows=1 width=8) (actual time=0.053..0.054 rows=1 loops=3)
                        Buckets: 1024 Batches: 1 Memory Usage: 9kB
                        -> Seq Scan on nation (cost=0..0.1..1.31 rows=1 width=4) (actual time=0.046..0.049 rows=1 loops=3)
                          Filter: (n.name = 'GERMANY'::bpchar)
                          Rows Removed by Filter: 24
                    -> Parallel Hash (cost=342900.28..342900.28 rows=8500128 width=24) (actual time=8078.550..8078.550 rows=6800102 loops=3)
                      Buckets: 65536 Batches: 512 Memory Usage: 2784kB
                      -> Parallel Seq Scan on order_line l1 (cost=0..0.342900.28 rows=8500128 width=24) (actual time=0.070..2745.156 rows=6800102 loops=3)
-> Materialize (cost=399206.24..411286.24 rows=2400000 width=20) (actual time=4627.793..6502.091 rows=2414572 loops=3)
-> Sort (cost=399206.24..405206.24 rows=2400000 width=20) (actual time=4627.784..5744.171 rows=2399983 loops=3)
  Sort Key: orders.o_id, orders.o_w_id, orders.o_d_id
  Sort Method: external merge Disk: 79896kB
  Worker 0: Sort Method: external merge Disk: 79896kB
  Worker 1: Sort Method: external merge Disk: 79896kB
  -> Seq Scan on orders (cost=0..0.46436.00 rows=2400000 width=20) (actual time=0.080..1038.349 rows=2400000 loops=3)
-> Parallel Hash (cost=342900.28..342900.28 rows=8500128 width=20) (actual time=9083.028..9083.029 rows=6800102 loops=3)
  Buckets: 65536 Batches: 512 Memory Usage: 2816kB
-> Parallel Seq Scan on order_line l2 (cost=0..0.342900.28 rows=8500128 width=20) (actual time=1744.143..4422.496 rows=6800102 loops=3)
```

Figura 22: Plano de execução original da query 21

Introduzindo este plano na ferramenta <https://explain.depesz.com/>, salta à vista que a query realiza scans sequenciais em 5 tabelas diferentes, o que é muito pouco desejável, visto que isso implica percorrer essas tabelas na sua totalidade. Desta maneira, o grupo tentou colmatar esses falhas de desempenho.



Per node type stats			
node type	count	sum of times	% of query
Finalize GroupAggregate	1	0.461 ms	0.0 %
Gather Merge	1	629.947 ms	1.0 %
Hash	2	0.158 ms	0.0 %
Hash Join	2	780.439 ms	1.2 %
Materialize	1	806.559 ms	1.3 %
Merge Join	1	1,080.873 ms	1.7 %
Parallel Hash	2	10,950.638 ms	17.4 %
Parallel Hash Anti Join	1	27,573.155 ms	43.9 %
Parallel Hash Join	1	2,316.968 ms	3.7 %
Parallel Seq Scan	3	10,940.838 ms	17.4 %
Partial GroupAggregate	1	26.165 ms	0.0 %
Seq Scan	3	1,373.898 ms	2.2 %
Sort	4	6,350.890 ms	10.1 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
<b>nation</b>	1	0.033 ms	0.0 %
Seq Scan	1	0.033 ms	100.0 %
<b>order_line</b>	2	8,907.366 ms	14.2 %
Parallel Seq Scan	2	8,907.366 ms	100.0 %
<b>orders</b>	1	1,371.727 ms	2.2 %
Seq Scan	1	1,371.727 ms	100.0 %
<b>stock</b>	1	2,033.472 ms	3.2 %
Parallel Seq Scan	1	2,033.472 ms	100.0 %
<b>supplier</b>	1	2.138 ms	0.0 %
Seq Scan	1	2.138 ms	100.0 %

Figura 23: Estatísticas do plano de execução original da query 21

Para começar, o grupo analisou as tabelas em questão, procurando identificar para quais seria mais benificial fazer otimizações. A tabela *nation* é onde é realizado logo o primeiro scan sequencial, contudo esta tabela é estática e muito curta, com apenas 25 linhas, pelo que não valia a pena usar índices ou vistas materializadas sobre a mesma.

De seguida, o grupo virou-se para a tabela *supplier*, na qual só se accede ao nome e chave do fornecedor, e à chave da sua nação. À partida, estes dados são imutáveis, pelo que surgiu a ideia de os carregar para uma vista materializada, acabando assim com a necessidade de aceder à tabela *supplier* na *heap* em tempo de execução, o que tornaria a *query* mais rápida. A vista criada e a nova interrogação são apresentadas de seguida:

```
create materialized view q21_mv1 as
select su_suppkey, su_nationkey, su_name
from supplier
order by su_nationkey, su_suppkey;

select su_name, count(*) as numwait
from q21_mv1, order_line l1, orders, stock, nation
where ol_o_id = o_id
      and ol_w_id = o_w_id
      and ol_d_id = o_d_id
      and ol_w_id = s_w_id
      and ol_i_id = s_i_id
```

```

and mod((s_w_id * s_i_id),10000) = su_suppkey
and l1.ol_delivery_d > o_entry_d
and not exists (
    select * from order_line l2
    where l2.ol_o_id = l1.ol_o_id
        and l2.ol_w_id = l1.ol_w_id
        and l2.ol_d_id = l1.ol_d_id
        and l2.ol_delivery_d > l1.ol_delivery_d)
and su_nationkey = n_nationkey
and n_name = 'GERMANY'
group by su_name
order by numwait desc, su_name;
  
```

Correndo de novo a query, é possível observar que o PostgreSQL deixou de aceder à tabela na *heap*, como se pensava, diminuindo o tempo de execução para 61574.095 ms.

```

Sort (cost=2304096.58..2304121.58 rows=10000 width=34) (actual time=61060.009..61549.947 rows=396 loops=1)
  Sort Key: (count(*)) DESC, q21.mv1.su_name
  Sort Method: quicksort Memory: 55KB
-> Finalize GroupAggregate (cost=2306244.36..2303432.20 rows=10000 width=34) (actual time=60975.546..61549.439 rows=396 loops=1)
  Group Key: q21.mv1.su_name
-> Gather Merge (cost=2306244.36..2303232.20 rows=20000 width=34) (actual time=60975.321..61548.959 rows=1188 loops=1)
  Workers Planned: 2
  Workers Launched: 2
-> Partial GroupAggregate (cost=2299244.34..2299233.68 rows=10000 width=34) (actual time=60915.914..60968.788 rows=396 loops=3)
  Group Key: q21.mv1.su_name
-> Sort (cost=2299244.34..2299437.45 rows=77245 width=26) (actual time=60915.604..60949.443 rows=88980 loops=3)
  Sort Key: q21.mv1.su_name
  Sort Method: external merge Disk: 3056KB
  Worker 0: Sort Method: external merge Disk: 3240KB
  Worker 1: Sort Method: external merge Disk: 3144KB
-> Parallel Hash Anti Join (cost=2911314.52..2291121.64 rows=77245 width=26) (actual time=35154.759..60586.611 rows=88980 loops=3)
  Hash Cond: ((l1.ol_o_id = l2.ol_o_id) AND (l1.ol_w_id = l2.ol_w_id) AND (l1.ol_d_id = l2.ol_d_id))
  Join Filter: (l2.ol_delivery_d > l1.ol_delivery_d)
  Rows Removed by Join Filter: 781395
-> Merge Join (cost=1469856.01..1507673.35 rows=115867 width=46) (actual time=22227.084..25515.625 rows=268143 loops=3)
  Merge Cond: (l1.o_id = orders.o_id) AND (l1.o_w_id = orders.o_w_id) AND (l1.o_d_id = orders.o_d_id)
  Join Filter: (l1.ol_delivery_d > l1.o_entry_d)
-> Sort (cost=1870647.38..1871515.81 rows=347772 width=50) (actual time=14741.517..15087.624 rows=268143 loops=3)
  Sort Key: l1.ol_o_id, stock.s_w_id, l1.ol_d_id
  Sort Method: external merge Disk: 17240KB
  Worker 0: Sort Method: external merge Disk: 15912KB
  Worker 1: Sort Method: external merge Disk: 19016KB
-> Parallel Hash Join (cost=520440.02..1026749.32 rows=347772 width=50) (actual time=12381.987..14272.307 rows=268143 loops=3)
  Hash Cond: ((stock.s_w_id = l1.ol_w_id) AND (stock.s_i_id = l1.ol_i_id))
  Hash Join (cost=231.82..423353.49 rows=133333 width=34) (actual time=75.654..4179.525 rows=105260 loops=3)
  Hash Cond: ((mod(stock.s_w_id * stock.s_i_id, 10000) = q21.mv1.su_suppkey)
-> Parallel Seq Scan on stock (cost=0..0 rows=133333 width=34) (actual time=48.394..3439.814 rows=2666667 loops=3)
  Hash (cost=226.82..226.82 rows=400 width=34) (actual time=2.033..2.039 rows=396 loops=3)
  Buckets: 1024 Batches: 1 Memory Usage: 9KB
-> Hash Join (cost=1..32..226.82 rows=10000 width=30) (actual time=0.615..1.972 rows=396 loops=3)
  Hash Cond: (q21.mv1.su_nationkey = nation.nationkey)
-> Seq Scan on q21.mv1.su_nationkey (cost=0..0 rows=10000 loops=3)
-> Hash (cost=1.31..1.31 rows=1 width=4) (actual time=0.077..0.079 rows=1 loops=3)
  Buckets: 1024 Batches: 1 Memory Usage: 9KB
-> Seq Scan on nation (cost=0..0 rows=1 width=4) (actual time=0.071..0.073 rows=1 loops=3)
  Filter: (n_name = 'GERMANY'::bpchar)
  Rows Removed by Filter: 24
-> Parallel Hash (cost=342900.28..342900.28 rows=8506128 width=24) (actual time=7911.525..7911.526 rows=6800102 loops=3)
  Buckets: 65536 Batches: 512 Memory Usage: 2784KB
-> Parallel Seq Scan on order_line l2 (cost=0..0..342900.28 rows=8506128 width=24) (actual time=0.091..2721.552 rows=6800102 loops=3)
-> Materialize (cost=399206.24..399206.24 rows=240000 width=20) (actual time=7405.494..3921.137 rows=240000 loops=3)
  Sort Key: orders.o_id, orders.o_w_id, orders.o_d_id
  Sort Method: external merge Disk: 79896KB
  Worker 0: Sort Method: external merge Disk: 79896KB
  Worker 1: Sort Method: external merge Disk: 79896KB
-> Parallel Hash (cost=342900.28..342900.28 rows=2400000 width=20) (actual time=107.913..3921.802 rows=2400000 loops=3)
  Buckets: 65536 Batches: 512 Memory Usage: 2816KB
-> Parallel Seq Scan on order_line l2 (cost=0..0..342900.28 rows=8506128 width=20) (actual time=1794.351..4419.216 rows=6800102 loops=3)
  
```

Planning Time: 2.426 ms  
JIT:  
 Functions: 204  
 Options: Inlining true, Optimization true, Expressions true, Deforming true  
 Timing: Generation 41.146 ms, Inlining 537.827 ms, Optimization 2827.287 ms, Emission 2013.947 ms, Total 5420.207 ms  
Execution Time: 61574.095 ms  
(91 rows)

Figura 24: Plano de execução da query 21 com a materialização da tabela *supplier*

Visto que o programa depois filtra as linhas da vista cuja *su\_nationkey* seja a chave da Alemanha, o grupo ordenou a vista por essa coluna, de maneira a que as linhas de interesse ficassem todas concentradas na mesma zona da tabela, aumentando a viabilidade de usar um índice. Além disso, a vista ordena ainda as linhas por *su\_suppkey*, para cada *su\_nationkey*,

na perspetiva de tornar mais previsível o comportamento da igualdade seguinte ( $mod((s\_w\_id * s\_i\_id), 10000) = su\_suppkey)$ .

Tendo isto em conta, o próximo passo consistiu na criação do índice mencionado sobre a coluna *su\_nationkey* da vista:

```
create index q21_mv1_i on q21_mv1(su_nationkey);
```

Com este índice, o PostgreSQL passou a executar um *Index Scan* na vista, o que permitiu reduzir o tempo de execução em cerca de 4 segundos, para 57833.222 ms.

```

Sort (cost=2303897.36..2303922.36 rows=10000 width=34) (actual_time=57340.957..57889.532 rows=396 loops=1)
  Sort Key: (count(*)) DESC, q21_mv1.su_name
  Sort Method: quicksort Memory: 55KB
-> Finalize GroupAggregate (cost=2300045.14..2303232.97 rows=10000 width=34) (actual_time=57247.656..57889.066 rows=396 loops=1)
    Group Key: q21_mv1.su_name
      -> Gather Merge (cost=2300045.14..2303032.97 rows=20000 width=34) (actual_time=57247.555..57888.605 rows=1188 loops=1)
        Workers Planned: 2
        Workers Actually Used: 2
          -> Partial GroupAggregate (cost=2299045.11..2299274.45 rows=10000 width=34) (actual_time=57164.546..57224.056 rows=396 loops=3)
            Group Key: q21_mv1.su_name
              -> Sort (cost=2299045.11..2299288.22 rows=77245 width=26) (actual_time=57164.194..57205.709 rows=88980 loops=3)
                Sort Key: q21_mv1.su_name
                Sort Method: external merge Disk: 3200KB
                Worker 0: Sort Method: external merge Disk: 3104KB
                Worker 1: Sort Method: external merge Disk: 3136KB
                -> Parallel Hash Anti Join (cost=201115.29..2299224.42 rows=77245 width=26) (actual_time=37211.426..56778.311 rows=88980 loops=3)
                  Hash Cond: ((l1.ol_o_id = l2.ol_o_id) AND (l1.ol_w_id = l2.ol_w_id) AND (l1.ol_d_id = l2.ol_d_id))
                  Filter: (l2.ol_delivery_d > l1.ol_delivery_d)
                  Rows Removed by Filter: 60949
                  -> Merge Join (cost=146954.79..1507474.12 rows=115867 width=46) (actual_time=24247.150..27555.935 rows=268143 loops=3)
                    Merge Cond: ((l1.ol_o_id = orders.o_id) AND (l1.ol_w_id = orders.o_w_id) AND (l1.ol_d_id = orders.o_d_id))
                    Filter: (l1.ol_delivery_d > orders.o_entry_d)
                    -> Sort (cost=1076448.15..1671317.58 rows=347772 width=50) (actual_time=19291.776..19641.266 rows=268143 loops=3)
                      Sort Key: l1.ol_o_id, stock_s.i_id, l1.ol_l_id
                      Sort Method: external merge Disk: 17224KB
                      Worker 0: Sort Method: external merge Disk: 16648KB
                      Worker 1: Sort Method: external merge Disk: 18288KB
                      -> Parallel Hash Join (cost=528240.79..1026550.10 rows=347772 width=50) (actual_time=16995.584..18780.696 rows=268143 loops=3)
                        Hash Cond: ((stock_s.w_id = l1.ol_w_id) AND (stock_s.i_id = l1.ol_l_id))
                        -> Hash Join (cost=32.68..423356.26 rows=133333 width=34) (actual_time=0.550..8722.926 rows=105260 loops=3)
                          Hash Cond: ((stock_s.w_id = l1.ol_w_id) AND (stock_s.i_id = l1.ol_l_id))
                          -> Parallel Seq Scan on stock (cost=0.00..8722.926 rows=133333 width=8) (actual_time=0.032..7921.952 rows=2666667 loops=3)
                          -> Hash (cost=27.69..27.69 rows=400 width=30) (actual_time=0.362..0.365 rows=1 loops=3)
                            Buckets: 1024 Batches: 1 Memory Usage: 39KB
                            -> Nested Loop (cost=0.29..27.69 rows=400 width=30) (actual_time=0.082..0.266 rows=396 loops=3)
                              -> Seq Scan on nation (cost=0.00..1.31 rows=1 width=4) (actual_time=0.052..0.055 rows=1 loops=3)
                                Filter: (n_name = 'GERMANY'::bpchar)
                                Rows Removed by Filter: 24
                              -> Index Scan using q21_mv1_i on q21_mv1 (cost=0.29..22.29 rows=400 width=34) (actual_time=0.023..0.133 rows=396 loops=3)
                                Index Cond: (su_nationkey = nation.nationkey)
                              -> Parallel Hash (cost=342900.28..342900.28 rows=8500128 width=24) (actual_time=8025.201..8025.202 rows=6800102 loops=3)
                                Buckets: 65536 Batches: 512 Memory Usage: 2784KB
                                -> Parallel Seq Scan on order_line (cost=0.00..342900.28 rows=8500128 width=24) (actual_time=0.079..2651.467 rows=6800102 loops=3)
                                  -> Materialize (cost=399286.24..411200.24 rows=2400000 width=20) (actual_time=4955.363..6810.051 rows=2414633 loops=3)
                                    -> Sort (cost=399286.24..405286.24 rows=2400000 width=20) (actual_time=4955.257..6016.412 rows=2399983 loops=3)
                                      Sort Key: orders.o_id, orders.o_w_id, orders.o_d_id
                                      Sort Method: external merge Disk: 79896KB
                                      Worker 0: Sort Method: external merge Disk: 79896KB
                                      Worker 1: Sort Method: external merge Disk: 79896KB
                                      -> Seek Scan on orders (cost=0.00..46438.00 rows=2400000 width=20) (actual_time=0.043..1649.839 rows=2400000 loops=3)
                                      Buckets: 65536 Batches: 512 Memory Usage: 2816KB
                                      -> Parallel Seq Scan on order_line_l2 (cost=0.00..342900.28 rows=8500128 width=20) (actual_time=1769.975..4396.712 rows=6800102 loops=3)
                                        -> Parallel Seq Scan on order_line_l1 (cost=0.00..342900.28 rows=8500128 width=20) (actual_time=1769.975..4396.712 rows=6800102 loops=3)

Planning Time: 2.471 ms
Planning Time: 2.471 ms
Functions: 189
Options: Inlining true, Optimization true, Expressions true, Deforming true
Timing: Generation 46.618 ms, Inlining 769.909 ms, Optimization 2614.959 ms, Emission 1919.758 ms, Total 5351.243 ms
Execution Time: 57833.222 ms
(59 rows)

```

Figura 25: Plano de execução da query 21 com a primeira vista criada e o respetivo índice

De seguida, o grupo voltou a recorrer ao <https://explain.depesz.com/> para analisar o efeito das mudanças no panorama geral da query e determinar onde devia atuar a seguir.

Per node type stats			
node type	count	sum of times	% of query
Finalize GroupAggregate	1	0.461 ms	0.0 %
Gather Merge	1	584.549 ms	1.0 %
Hash	1	0.105 ms	0.0 %
Hash Join	1	800.609 ms	1.4 %
Index Scan	1	0.133 ms	0.0 %
Materialize	1	793.639 ms	1.4 %
Merge Join	1	1,104.618 ms	1.9 %
Nested Loop	1	0.072 ms	0.0 %
Parallel Hash	2	10,071.303 ms	17.4 %
Parallel Hash Anti Join	1	20,128.096 ms	34.8 %
Parallel Hash Join	1	2,032.568 ms	3.5 %
Parallel Seq Scan	3	14,970.131 ms	25.9 %
Partial GroupAggregate	1	18.347 ms	0.0 %
Seq Scan	2	1,649.894 ms	2.9 %
Sort	4	5,655.007 ms	9.8 %

Per table stats			
Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
<b>nation</b>	1	0.055 ms	0.0 %
Seq Scan	1	0.055 ms	100.0 %
<b>order_line</b>	2	7,048.179 ms	12.2 %
Parallel Seq Scan	2	7,048.179 ms	100.0 %
<b>orders</b>	1	1,649.839 ms	2.9 %
Seq Scan	1	1,649.839 ms	100.0 %
<b>q21_mv1</b>	1	0.133 ms	0.0 %
Index Scan	1	0.133 ms	100.0 %
<b>stock</b>	1	7,921.952 ms	13.7 %
Parallel Seq Scan	1	7,921.952 ms	100.0 %

Figura 26: Estatísticas do plano de execução da query 21 com uma vista e índice

Como se pretendia, o scan sequencial da tabela *stock* foi substituído por um scan do índice da vista, diminuindo a carga da operação *Parallel Hash Anti Join* na *query*, onde estava incluído dito scan. Neste ponto, o scan mais pesado a ser realizado era na tabela *stock* que, tal como na [interrogação 15](#), servia apenas para fazer combinações de chaves através das coluna *s\_w\_id* e *s\_i\_id* (estáticas), bem como calcular o  $mod((s_w_id * s_i_id), 10000)$  para unir com a tabela *supplier*. Assim, pelos mesmos motivos que foram explicados na *query 15*, decidiu-se carregar esses dados para uma vista materializada.

```

create materialized view q21_mv2 as
select s_i_id, s_w_id, mod((s_w_id * s_i_id),10000) as supplier_no
from stock
order by supplier_no, s_i_id, s_w_id;

select su_name, count(*) as numwait
from q21_mv1, order_line l1, orders, q21_mv2, nation
where ol_o_id = o_id
      and ol_w_id = o_w_id
      and ol_d_id = o_d_id
      and ol_w_id = s_w_id
      and ol_i_id = s_i_id
      and supplier_no = su_suppkey
  
```

```

and l1.ol_delivery_d > o_entry_d
and not exists (select *
  from order_line l2
  where l2.ol_o_id = l1.ol_o_id
    and l2.ol_w_id = l1.ol_w_id
    and l2.ol_d_id = l1.ol_d_id
    and l2.ol_delivery_d > l1.ol_delivery_d)
and su_nationkey = n_nationkey
and n_name = 'GERMANY'
group by su_name
order by numwait desc, su_name;
  
```

Correndo a *query* de novo, é possível verificar que o programa deixou de aceder à tabela *stock* na *heap*, o que teve um impacto tremendo no tempo de execução, que desceu para 26932.615 ms, diminuindo em mais de metade. O plano de execução é agora muito mais compacto do que originalmente era.

```

Sort  (cost=1551617.20..1551642.20 rows=10000 width=34) (actual time=26783.580..26927.155 loops=1)
Sort Key: (count(*)) DESC, q21_mv1.su_name
Sort Method: quicksort Memory: 55kB
-> Finalize GroupAggregate  (cost=1547774.55..1550952.81 rows=10000 width=34) (actual time=26702.032..26926.684 rows=396 loops=1)
  Group Key: q21_mv1.su_name
-> Gather Merge  (cost=1547774.55..1550752.81 rows=20000 width=34) (actual time=26701.578..26926.209 rows=1188 loops=1)
  Workers Planned: 2
  Workers Launched: 2
-> Partial GroupAggregate  (cost=1546774.53..1547444.29 rows=10000 width=34) (actual time=26413.035..26463.579 rows=396 loops=3)
  Group Key: q21_mv1.su_name
-> Sort  (cost=1546774.53..1546964.45 rows=75968 width=26) (actual time=26412.938..26445.893 rows=85365 loops=3)
  Sort Key: q21_mv1.su_name
  Sort Method: external merge Disk: 2952kB
  Worker 0: Sort Method: external merge Disk: 2848kB
  Worker 1: Sort Method: external merge Disk: 2850kB
-> Nested Loop Anti Join  (cost=93486.56..1070686.74 rows=113951 width=46) (actual time=13005.520..19453.782 rows=257514 loops=3)
  -> Nested Loop  (cost=93486.56..1070686.74 rows=113951 width=46) (actual time=13015.451..25651.259 rows=85365 loops=3)
    -> Parallel Hash Join  (cost=93486.56..1070686.74 rows=113951 width=46) (actual time=12996.237..16441.232 rows=257514 loops=3)
      Hash Cond: ((l1.ol_w_id = q21_mv2.s_w_id) AND (l1.ol_i_id = q21_mv2.s_l_id))
      -> Parallel Seq Scan on order_line l1  (cost=0.00..342900.28 rows=8500128 width=24) (actual time=0.096..3211.910 rows=68000102 loops=3)
      -> Parallel Hash  (cost=90444.09..90444.09 rows=133336 width=34) (actual time=5756.216..5756.221 rows=100800 loops=3)
        Buckets: 65536  Batches: 8  Memory Usage: 3232kB
        -> Hash Join  (cost=32.60..96444.09 rows=133336 width=34) (actual time=866.384..5702.409 rows=100800 loops=3)
          Hash Cond: (q21_mv2.supplier_no = q21_mv1.su_supplier)
          -> Parallel Seq Scan on mv2  (cost=0.00..76577.92 rows=3333392 width=12) (actual time=0.034..4474.192 rows=2666667 loops=3)
          Buckets: 1024  Batches: 1  Memory Usage: 33kB
          -> Hash  (cost=27.60..27.60 rows=400 width=30) (actual time=787.168..787.170 rows=396 loops=3)
            -> Seq Scan on nation  (cost=0.00..1.31 rows=1 width=4) (actual time=762.471..762.477 rows=1 loops=3)
              Filter: (n_name = 'GERMANY'::bpchar)
              Rows Removed by Filter: 24
            -> Index Scan using q21_mv1_i on q21_mv1  (cost=0.29..22.29 rows=400 width=34) (actual time=23.112..24.415 rows=396 loops=3)
              Index Cond: (su_nationkey = nation.n_nationkey)
            -> Index Scan using pk_orders on orders  (cost=0.43..1.36 rows=1 width=20) (actual time=0.010..0.010 rows=1 loops=772543)
              Index Cond: ((o_w_id = l1.ol_w_id) AND (o_d_id = l1.ol_d_id) AND (o_id = l1.ol_o_id))
              Filter: (l1.ol_delivery_d > o_entry_d)
            -> Index Scan using pk_order_line on order_line l2  (cost=0.56..5.35 rows=3 width=20) (actual time=0.023..0.023 rows=1 loops=772543)
              Index Cond: ((ol_w_id = l1.ol_w_id) AND (ol_d_id = l1.ol_d_id) AND (ol_o_id = l1.ol_o_id))
              Filter: (ol_delivery_d > l1.ol_delivery_d)
              Rows Removed by Filter: 7
Planning Time: 149.362 ms
JIT:
  Functions: 141
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 20.600 ms, Inlining 382.023 ms, Optimization 1107.365 ms, Emission 769.919 ms, Total 2279.907 ms
Execution Time: 26932.615 ms
(46 rows)
  
```

Figura 27: Plano de execução da query 21 com a segunda vista

Além disso, é possível verificar que aconteceu algo interessante. Como a nova vista foi também ordenada por *s\_w\_id* e *s\_i\_id*, deixou de ser necessário o programa realizar o *sort* por essas colunas que executava a seguir ao *Parallel Hash Join* onde fazia as combinações das chaves em questão. Graças a isto, o PostgreSQL alterou a sua estratégia para um *Nested Loop Anti Join*, em vez do anterior *Parallel Hash Anti Join*, passando a percorrer os índices de chaves primárias já existentes sobre as tabelas *orders* e *order\_line*. Assim, deixou de realizar os scans sequenciais que levava a cabo anteriormente nessas tabelas e as operações subsequentes para fazer as respetivas combinações de chaves. Isto explica a enorme diminuição do tempo

de execução e realça a eficácia desta vista.

De seguida, tendo em conta a condição  $supplier\_no = su\_suppkey$ , ordenou-se a vista por  $supplier\_no$  e implementou-se um índice na mesma coluna, de forma a agilizar esta igualdade.

```
create index q21_mv2_i on q21_mv2(supplier_no);
```

Mais uma vez, o tempo de execução desceu, para 21102.229 ms, como seria de esperar, passando a realizar-se um scan sobre o índice introduzido em vez da vista, como comprova a imagem seguinte:

```
Sort (cost=1529514.30..1529539.30 rows=10000 width=34) (actual time=21096.701..21096.836 rows=396 loops=1)
  Sort Key: (count(*)) DESC, q21_mv1.su_name
  Sort Method: quicksort Memory: 55KB
-> Finalize GroupAggregate (cost=1525671.68..1528849.92 rows=10000 width=34) (actual time=21065.003..21096.560 rows=396 loops=1)
    Group Key: q21_mv1.su_name
    -> Gather Merge (cost=1525671.68..1528649.92 rows=20000 width=34) (actual time=21064.840..21096.198 rows=1188 loops=1)
      Workers Planned: 2
      Workers Launched: 2
    -> Partial GroupAggregate (cost=1524671.65..1525341.40 rows=10000 width=34) (actual time=20985.051..21037.102 rows=396 loops=3)
      Group Key: q21_mv1.su_name
      -> Sort (cost=1524671.65..1524861.57 rows=75966 width=26) (actual time=20984.979..21015.065 rows=85365 loops=3)
        Sort Key: q21_mv1.su_name
        Sort Method: external merge Disk: 3856kB
        Worker 0: Sort Method: external merge Disk: 2976kB
        Worker 1: Sort Method: external merge Disk: 3824kB
        -> Nested Loop Anti Join (cost=25444.51..1516693.44 rows=75966 width=26) (actual time=1485.545..20251.998 rows=85365 loops=3)
          -> Nested Loop (cost=25444.95..1048598.98 rows=113949 width=46) (actual time=1485.456..13854.992 rows=257514 loops=3)
            -> Hash Join (cost=25443.52..580127.56 rows=342017 width=50) (actual time=1485.345..10688.638 rows=257514 loops=3)
              Hash Cond: ((l1.ol_w_id = q21_mv2.s.w_id) AND (l1.ol_i_id = q21_mv2.s.i_id))
              -> Parallel Seq Scan on order_line l1 (cost=0.00..342990.28 rows=8500128 width=24) (actual time=1049.469..3223.678 rows=6800102 loops=3)
              -> Hash (cost=18143.52..18143.52 rows=320000 width=34) (actual time=435.155..435.159 rows=302400 loops=3)
                Buckets: 65536 Batches: 8 Memory Usage: 2996kB
                -> Nested Loop (cost=0.72..18143.52 rows=320000 width=34) (actual time=0.773..264.398 rows=302400 loops=3)
                  -> Nested Loop (cost=0.29..27.68 rows=400 width=30) (actual time=0.681..1.033 rows=396 loops=3)
                    -> Seq Scan on nation (cost=0.00..1.31 rows=1 width=4) (actual time=0.636..0.642 rows=1 loops=3)
                      Filter: (n_name = 'GERMANY' ::bpchar)
                      Rows Removed by Filter: 1
                    -> Index Scan on q21_mv1 on q21_mv1 (cost=0.29..22.29 rows=400 width=34) (actual time=0.039..0.266 rows=396 loops=3)
                      Index Cond: (cu.nationkey = nation.n.nationkey)
                      Index Scan using q21_mv1_i on q21_mv1 (cost=0.43..36.18 rows=911 width=12) (actual time=0.052..0.463 rows=764 loops=1188)
                      Index Cond: (supplier_no = q21_mv1.su_suppkey)
                      Index Cond: (o.w_id = l1.ol_w_id) AND (o.d_id = l1.ol_d_id) AND (o.id = l1.ol_o_id)
                      Filter: (l1.ol_delivery_d > o_entry_d)
                      Rows Removed by Filter: 7
Planning Time: 2.202 ms
JIT:
  Functions: 129
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timings: Generation 25.589 ms, Inlining 328.663 ms, Optimization 1695.367 ms, Emission 1121.316 ms, Total 3170.936 ms
Execution Time: 21102.229 ms
(44 rows)
```

Figura 28: Plano de execução da query 21 com o índice na segunda vista

Finalmente, decidiu-se colocar mais alguma da lógica do interrogação numa vista, isto é, as condições de união das tabelas, pré-processando mais informação para aliviar a carga computacional da *query*. De fora ficaram as condições com a query embebida e o nome do país, de maneira a preservar alguma da estrutura e flexibilidade da interrogação.

```
create materialized view q21_mv3 as
select ol_o_id, ol_w_id, ol_d_id, ol_delivery_d, su_name, n_name
from q21_mv1, order_line l1, orders, q21_mv2, nation
where ol_o_id = o_id
      and ol_w_id = o_w_id
      and ol_d_id = o_d_id
      and ol_w_id = s_w_id
      and ol_i_id = s_i_id
      and supplier_no = su_suppkey
      and l1.ol_delivery_d > o_entry_d
```

```

    and su_nationkey = n_nationkey
order by n_name, ol_o_id, ol_w_id, ol_d_id;

select su_name, count(*) as numwait
from q21_mv3 v
where not exists (select *
                   from order_line l2
                  where l2.ol_o_id = v.ol_o_id
                    and l2.ol_w_id = v.ol_w_id
                    and l2.ol_d_id = v.ol_d_id
                    and l2.ol_delivery_d > v.ol_delivery_d)
      and n_name = 'GERMANY'
group by su_name
order by numwait desc, su_name;
  
```

Esta terceira vista agilizar ainda mais a interrogação, diminuindo o tempo de execução para 14577.364 ms, uma vez que permitia realizar a união de todas as tabelas muito mais rapidamente, dado que a maior parte das condições de união já tinham sido feitas em avanço.

```

-----
Sort (cost=1496247.27..1496270.21 rows=9174 width=34) (actual time=13375.829..14574.329 rows=396 loops=1)
  Sort Key: (count(*)) DESC, v_su_name
  Sort Method: quicksort Memory: 55kB
-> Finalize GroupAggregate (cost=1493319.24..1495643.47 rows=9174 width=34) (actual time=13374.148..14573.889 rows=396 loops=1)
      Group Key: v_su_name
      -> Gather Merge (cost=1493319.24..1495459.99 rows=18348 width=34) (actual time=13374.118..14573.477 rows=1188 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          -> Sort (cost=1492319.22..1492342.15 rows=9174 width=34) (actual time=13300.311..13300.623 rows=396 loops=3)
              Sort Key: v_su_name
              Sort Method: quicksort Memory: 55kB
              Worker 0: Sort Method: quicksort Memory: 55kB
              Worker 1: Sort Method: quicksort Memory: 55kB
              -> Partial HashAggregate (cost=1491623.68..1491715.42 rows=9174 width=34) (actual time=13297.363..13297.803 rows=396 loops=3)
                  Group Key: v_su_name
                  -> Parallel Hash Anti Join (cost=541458.52..1490585.43 rows=207650 width=26) (actual time=11411.985..13244.660 rows=85365 loops=3)
                      Hash Cond: ((v.ol_o_id = l2.ol_o_id) AND (v.ol_w_id = l2.ol_w_id) AND (v.ol_d_id = l2.ol_d_id))
                      Join Filter: (l2.ol_delivery_d > v.ol_delivery_d)
                      Rows Removed by Join Filter: 750880
                      -> Parallel Seq Scan on q21_mv3 v  (cost=0.00..378016.38 rows=311474 width=46) (actual time=1337.231..3107.299 rows=257514 loops=3)
                          Filter: (n_name = 'GERMANY':;b:char)
                          Rows Removed by Filter: 6538282
                      -> Parallel Hash (cost=342980.28..342980.28 rows=8500128 width=20) (actual time=8000.021..8000.022 rows=6800102 loops=3)
                          Buckets: 65536 Batches: 512 Memory Usage: 2816kB
                          -> Parallel Seq Scan on order_line l2 (cost=0.00..342980.28 rows=8500128 width=20) (actual time=695.640..3276.514 rows=6800102 loops=3)
Planning Time: 0.349 ms
JIT:
  Functions: 69
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 11.064 ms, Inlining 530.667 ms, Optimization 878.412 ms, Emission 672.870 ms, Total 2093.013 ms
Execution Time: 14577.364 ms
(31 rows)
  
```

Figura 29: Plano de execução da query 21 com a terceira vista

Contudo, esta solução já não é tão recomendável visto que restringe muito mais a query, devido à grande quantidade de informação pré-processada, relativa inclusivé a tabelas dinâmicas. Isto levanta uma necessidade mais urgente de atualizar a vista materializada com maior frequência, numa tentativa de manter os seus dados mais precisos e viáveis.

Dito isto, era possível otimizar ainda mais esta estratégia, caso se optasse pela mesma, colocando um índice na coluna *n\_name* da nova vista, uma vez que vai haver uma filtragem inicial das linhas correspondentes ao país que o utilizador escolher (p.e. *n\_name = 'GERMANY'*). Foi também por este motivo que o grupo ordenou a vista por essa coluna (*order by n\_name*).

```
create index q21_mv3_i on q21_mv3(n_name);
```

```
Sort (cost=1155103.74..1155126.67 rows=9174 width=34) (actual time=10534.051..10860.946 rows=396 loops=1)
  Sort Key: (count(*)) DESC, v_su.name
  Sort Method: quicksort Memory: 55kB
-> Finalize GroupAggregate (cost=1152175.71..1154499.94 rows=9174 width=34) (actual time=10532.348..10860.507 rows=396 loops=1)
    Group Key: v_su.name
-> Gather Merge (cost=1152175.71..1154316.46 rows=18348 width=34) (actual time=10532.318..10860.088 rows=1188 loops=1)
  Workers Planned: 2
  Workers Launched: 2
-> Sort (cost=1151175.68..1151198.62 rows=9174 width=34) (actual time=10485.089..10485.260 rows=396 loops=3)
  Sort Key: v_su.name
  Sort Method: quicksort Memory: 55kB
  Worker 0: Sort Method: quicksort Memory: 55kB
  Worker 1: Sort Method: quicksort Memory: 55kB
-> Partial HashAggregate (cost=1150480.14..1150571.88 rows=9174 width=34) (actual time=10482.131..10482.437 rows=396 loops=3)
  Group Key: v_su.name
-> Parallel Hash Anti Join (cost=541459.08..1149441.89 rows=207650 width=26) (actual time=8635.193..10434.709 rows=85365 loops=3)
  Hash Cond: ((v.ol_o_id = l2.ol_o_id) AND (v.ol_w_id = l2.ol_w_id) AND (v.ol_d_id = l2.ol_d_id))
  Join Filter: (l2.ol_delivery_d > v.ol_delivery_d)
  Rows Removed by Join Filter: 751010
-> Parallel Index Scan using q21_mv3_i on q21_mv3 v  (cost=0.56..36872.84 rows=311474 width=46) (actual time=0.121..205.784 rows=257514 loops=3)
  Index Cond: (n_name = 'GERMANY'::bpchar)
-> Parallel Hash (cost=342900.28..342900.28 rows=8500128 width=20) (actual time=8141.286..8141.288 rows=6800102 loops=3)
  Buckets: 65536 Batches: 512 Memory Usage: 2816kB
-> Parallel Seq Scan on order_line l2 (cost=0.00..342900.28 rows=8500128 width=20) (actual time=702.600..3442.864 rows=6800102 loops=3)

Planning Time: 0.324 ms
JIT:
  Functions: 69
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 18.965 ms, Inlining 340.802 ms, Optimization 1024.838 ms, Emission 738.932 ms, Total 2123.538 ms
Execution Time: 10863.969 ms
(30 rows)
```

Figura 30: Plano de execução da query 21 com a terceira vista e respetivo índice

Com este índice, o tempo de execução é reduzido para 10863.969 ms, o que é um valor bastante melhor que os 70582.875 ms iniciais, tendo sido removido cerca de 1 minuto de *run time*.

Uma vez que o grupo experimentou esta estratégia que, como já foi dito, é pouco viável, decidiu-se levá-la um passo mais longe ainda e colocar praticamente a *query* inteira numa vista materializada, deixando-se de fora apenas a condição do nome do país, o agrupamento dos valores e as ordenações. Esta tática foi testado sobretudo para fins de experimentação, para ver o efeito que tinha no tempo de execução, visto que era extremamente inviável e dificilmente seria útil num contexto realista.

```
create materialized view q21_mv4 as
select ol_o_id, ol_w_id, ol_d_id, ol_delivery_d, su_name, n_name
from q21_mv1, order_line l1, orders, q21_mv2, nation
where ol_o_id = o_id
  and ol_w_id = o_w_id
  and ol_d_id = o_d_id
  and ol_w_id = s_w_id
  and ol_i_id = s_i_id
  and supplier_no = su_suppkey
  and l1.ol_delivery_d > o_entry_d
  and not exists (select *
    from order_line l2
    where l2.ol_o_id = l1.ol_o_id
      and l2.ol_w_id = l1.ol_w_id
      and l2.ol_d_id = l1.ol_d_id
      and l2.ol_delivery_d > l1.ol_delivery_d)
```

```

    and su_nationkey = n_nationkey
order by n_name, ol_o_id, ol_w_id, ol_d_id;

select su_name, count(*) as numwait
from q21_mv4
where n_name = 'GERMANY'
group by su_name
order by numwait desc, su_name;
  
```

Como seria de esperar, uma vez que a vasta maioria da interrogação já foi processada, o tempo de execução voltou a diminuir significativamente para 1125.987 ms, aproximadamente 1 segundo de *run time*. Nesta versão, a carga computacional resumia-se a uma filtragem das linhas da vista relativas ao país em questão e agrupamento/ordenação da informação segundo os critérios pretendidos.

```

Sort (cost=130538.23..130561.29 rows=9224 width=34) (actual time=1121.695..1124.407 rows=396 loops=1)
  Sort Key: (count(*)) DESC, su_name
  Sort Method: quicksort Memory: 55kB
-> Final GroupAggregate (cost=127593.88..129930.78 rows=9224 width=34) (actual time=1120.162..1124.014 rows=396 loops=1)
      Group Key: su_name
      -> Gather Merge (cost=127593.88..129746.30 rows=18448 width=34) (actual time=1120.139..1123.606 rows=1188 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          -> Sort (cost=126593.86..126616.92 rows=9224 width=34) (actual time=1091.247..1091.295 rows=396 loops=3)
              Sort Key: su_name
              Sort Method: quicksort Memory: 55kB
              Worker 0: Sort Method: quicksort Memory: 55kB
              Worker 1: Sort Method: quicksort Memory: 55kB
              -> Partial HashAggregate (cost=125894.16..125986.40 rows=9224 width=34) (actual time=1088.292..1088.479 rows=396 loops=3)
                  Group Key: su_name
                  -> Parallel Seq Scan on q21_mv4 (cost=0.00..125346.77 rows=109479 width=26) (actual time=344.506..1045.145 rows=85365 loops=3)
                      Filter: (n_name = 'GERMANY'::bpchar)
                      Rows Removed by Filter: 2168045

Planning Time: 0.076 ms
JIT:
  Functions: 27
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 4.647 ms, Inlining 0.000 ms, Optimization 2.065 ms, Emission 38.401 ms, Total 45.113 ms
Execution Time: 1125.987 ms
(24 rows)
  
```

Figura 31: Plano de execução da query 21 com a última vista implementada

Tal como foi o caso com as vistas anteriores, era ainda possível implementar um índice na coluna *n\_name* desta, agilizando ainda mais a *query*. Desta forma, o tempo de execução acabaria nos 111.892 ms, o que é praticamente instantâneo, embora o grupo não recomende esta solução, como já foi referido previamente.

```

Sort (cost=16511.08..16534.14 rows=9224 width=34) (actual time=111.759..111.782 rows=396 loops=1)
  Sort Key: (count(*)) DESC, su_name
  Sort Method: quicksort Memory: 55kB
-> HashAggregate (cost=15811.39..15903.63 rows=9224 width=34) (actual time=111.436..111.541 rows=396 loops=1)
    Group Key: su_name
    -> Index Scan using q21_mv4_i on q21_mv4 (cost=0.56..14497.65 rows=262748 width=26) (actual time=0.033..52.620 rows=256096 loops=1)
        Index Cond: (n_name = 'GERMANY'::bpchar)
Planning Time: 0.083 ms
Execution Time: 111.892 ms
(9 rows)
  
```

Figura 32: Plano de execução da query 21 com a última vista e índice



## 4 Otimização do desempenho da carga transacional

A parte final do trabalho prático consistia em averiguar o impacto de diferentes parâmetros de configuração do *PostgreSQL* no desempenho da carga transacional, procurando encontrar uma configuração que produzisse resultados melhores do que a original.

O grupo focou-se na secção **Write Ahead Logs** da configuração, que possui os parâmetros mais relevantes para a execução de transações sobre a base de dados. O processo passou por testar os vários valores possíveis para as configurações desta secção, comparar os seus resultados e, no fim, discutir a viabilidade de cada parâmetro e a compatibilidade dos parâmetros mais eficazes entre si, procurando assim encontrar uma configuração de referência estável.

O grupo planeava testar todas as configurações de *Write Ahead Logs*, mas viu-se incapaz de fazer isso devido à falta de tempo e à grande quantidade de parâmetros existentes, pelo que testou apenas um conjunto com as que considerava mais relevantes.

Semelhante ao método usado em [2.2](#), todos os testes foram realizados com o mesmo ponto de partida, repondo o estado inicial da base de dados entre cada teste para garantir que os resultados obtidos não eram enviesados; cada parâmetro foi testado isoladamente, com valores predefinidos no resto da configuração. Nesta secção, todos os valores foram arredondados a poucas casas decimais para serem mais legíveis (e porque os dígitos removidos são praticamente irrelevantes).

A secção *Write Ahead Logs* divide-se em três subsecções diferentes: **settings**, **checkpoints** e **archiving**.

### 4.1 Settings

#### 4.1.1 fsync

Quando esta opção está ligada, como é a sua predefinição, o servidor *PostgreSQL* verifica que as atualizações estão a ser fisicamente escritas no disco, garantindo assim que a base de dados consegue recuperar para um estado consistente em caso de falha. Contudo, isto resulta numa grande penalização em termos de performance: sempre que se dá *commit* de uma transação, o *PostgreSQL* tem de esperar que o sistema operativo dê *flush* do *write ahead log* para o disco.

Desligando este parâmetro, o servidor deixa de fazer estas verificações, limitando-se a escrever para o disco logo que possível, o que pode resultar num aumento significativo de performance. Contudo, é bastante perigoso, uma vez que pode haver perda ou corrupção de dados em caso de falha.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
off	45.204	0.395	0.0301
on	37.921	0.512	0.0443

Como era de esperar, esta alteração levou a uma melhoria da performance da carga transacional. Contudo, o grupo decidiu não desligar este parâmetro, devido ao grande risco



associado.

#### 4.1.2 synchronous\_commit

Este parâmetro permite especificar o tipo de processamento de WAL que deve ser realizado antes de o servidor da base de dados retornar um indicador de sucesso ao cliente, regulando o nível de sincronização entre o cliente e as escritas no disco - por exemplo, desligar este parâmetro implica que há um intervalo de tempo entre o momento que o cliente recebe confirmação e o momento em que o resultado da transação é, de facto, escrito no disco.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
off	15.675	0.389	0.0359
local	35.161	0.447	0.0829
remote_write	32.265	0.486	0.0735
on	37.921	0.512	0.0443

Os valores alternativos ao *default* (on) desta configuração levaram todos a um degrada-  
mento de performance, pelo que se optou por manter o valor predefinido.

#### 4.1.3 WAL\_sync\_method

O método usado neste parâmetro é usado para escrever atualizações de WAL para o disco e só é relevante se o *fsync* estiver ligado (como se decidiu usar).

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
open_datasync	35.260	0.506	0.9927
fdatasync	36.378	0.484	0.0882
fsync	37.921	0.512	0.0443
open_sync	33.296	0.567	0.1033

Pelos valores obtidos nos testes realizados, é possível observar que o método usado por predefinição nesta configuração também é óptimo.

#### 4.1.4 full\_page\_writes

Quando este parâmetro está ligado, o servidor escreve todo o conteúdo de cada página do disco para a WAL durante a primeira alteração dessa página, a seguir a um *checkpoint*.

Logo, desligar este parâmetro pode levar a uma melhor performance, contudo surge o risco de perda ou corrupção de dados, em caso de falha do sistema.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
off	50.292	0.181	0.0416
on	37.921	0.512	0.0443

Tal como no caso do *fsync*, o grupo optou por não desligar este parâmetro, apesar da melhor performance, de maneira a garantir um maior nível de consistência à base de dados.



#### 4.1.5 wal\_buffers

Nesta configuração, é possível definir a quantidade de memória partilhada alocada para dados de WAL que ainda não tenham sido escritos para o disco. O valor predefinido é calculado a partir do espaço alocado para os *shared\_buffers* e corresponde, no nosso caso, a 4 MB.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
2 MB	36.997	0.513	0.0673
4 MB	37.921	0.512	0.0443
8 MB	38.065	0.503	0.0498
16 MB	38.207	0.490	0.0534
32 MB	36.170	0.484	0.0884

#### 4.1.6 commit\_delay

Este parâmetro define o *delay* antes de iniciar um *WAL flush*. Aumentar o *delay* pode levar a um maior *throughput*, permitindo dar *commit* a um número superior de transações em cada *flush* - é, portanto, um compromisso entre o número de transações *committed* em cada *flush* e a sua latência.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
100 ms	36.0422	0.451	0.0817
200 ms	37.426	0.483	0.0791
500 ms	38.921	0.468	0.0421
1000 ms	39.202	0.448	0.0440
2000 ms	37.711	0.475	0.0477

Como seria de esperar, aumentar o *commit\_delay* permitiu melhorar a performance do programa durante algum tempo, até eventualmente se tornar contraprodutivo. O valor de 1000 ms foi o que produziu melhores resultados.

#### 4.1.7 commit\_siblings

Este parâmetro define o número mínimo de transações concorrentes ativas necessário para incorrer num *commit\_delay*.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
2	34.175	0.289	0.0445
5	37.921	0.512	0.0443
10	38.706	0.389	0.0422
20	36.827	0.496	0.0794

Conclui-se que a subida ligeira do valor predefinido de transações concorrentes leva a uma melhor performance do programa.



## 4.2 Checkpoints

### 4.2.1 checkpoint\_timeout

Este parâmetro define o tempo entre *WAL checkpoints* automáticos. Dado que o valor predefinido é 5 minutos e um tempo maior significa um maior atraso na recuperação de falha, experimentou-se diminuir o *timeout*.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
30 s	38.501	0.503	0.0391
1 min	40.889	0.477	0.0208
2 min	40.991	0.463	0.0373
5 min	37.921	0.512	0.0443

A experiência teve resultados positivos, permitindo concluir que o *throughput* e os outros parâmetros do programa melhoram com um *checkpoint\_timeout* menor. Dado que os resultados com 1 e 2 minutos são muito próximos, o grupo optou por 1 minuto por ter um *abort rate* significativamente inferior.

### 4.2.2 max\_wal\_size

Este parâmetro representa o tamanho máximo aproximado que a WAL pode crescer durante *checkpoints* automáticos, sendo por *default* 1 GB. O grupo decidiu testar com valores superiores para ver o impacto de uma WAL menos limitada na performance.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
1 GB	37.921	0.512	0.0443
2 GB	46.138	0.349	0.0412
4 GB	50.206	0.301	0.0204

Como é possível observar, o aumento do tamanho máximo da WAL deu aso a melhorias bastante significativas da performance da carga transacional.

### 4.2.3 min\_wal\_size

O *min\_wal\_size* define um limite de tamanho para o qual sempre que a WAL foi inferior, os ficheiros WAL antigos são reciclados para usos futuros, em vez de removidos. Semelhante ao que se observou com o *max\_wal\_size*, a performance deve aumentar com um limite inferior maior.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
80 MB	37.921	0.512	0.0443
160 MB	37.769	0.517	0.0446
320 MB	38.476	0.501	0.0448
640 MB	38.203	0.507	0.0450



Tal como se pensava, o aumento do valor levou a uma melhor performance do programa, contudo traduz-se numa diferença mínima.

#### 4.2.4 checkpoint\_completion\_target

Este parâmetro define o objetivo de conclusão do *checkpoint* como uma fração do tempo total entre *checkpoints*, sendo por predefinição 0.5 segundos.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
0 s	36.854	0.564	0.0792
0.2 s	37.784	0.535	0.0753
0.5 s	37.921	0.512	0.0443
0.8 s	34.665	0.584	0.0834
1 s	32.687	0.618	0.0885

Nenhum dos valores testados produziu resultados melhores que o predefinido, pelo que o grupo optou por não o alterar.

#### 4.2.5 checkpoint\_flush\_after

Sempre que o mais bytes do que os definidos neste parâmetro forem escritos para disco durante um *checkpoint*, há um tentativa de forçar o sistema operativo a escrever no armazenamento subjacente. O *default* é 256 kb.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
0 kb	38.222	0.509	0.0426
512 kb	37.921	0.512	0.0443
1024 kb	36.870	0.526	0.0461

A alteração deste parâmetro tem pouco impacto nos resultados, todavia o uso de 0 kb aparenta ser ligeiramente melhor do que os 512 kb predefinidos.

#### 4.2.6 checkpoint\_warning

Este parâmetro define um valor temporal que, se for superior ao intervalo de tempo entre *checkpoints* causados pelo preenchimento de segmentos de ficheiros WAL, leva à escrita de uma mensagem para o *log* do servidor.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
15 s	40.121	0.463	0.0375
30 s	37.921	0.512	0.0443
45 s	39.666	0.489	0.0397
60 s	38.478	0.504	0.0433

Neste parâmetro, surpreendentemente, o valor predefinido foi o que produziu piores resultados, embora isto também possa estar relacionado com a incerteza dos valores obtidos.



## 4.3 Archiving

### 4.3.1 archive\_mode

Quando ativo, segmentos WAL terminados são enviados para o armazenamento de arquivos. Esta opção está desligada por predefinição.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
off	37.921	0.512	0.0443
on	30.827	0.630	0.0490
always	30.757	0.633	0.0473

Como seria de esperar, o uso deste parâmetro piora a performance, uma vez que aumenta o número de operações a ser realizadas em *run time*.

### 4.3.2 archive\_command

Este parâmetro é o comando usado para arquivar um ficheiros com segmentos WAL completos. Este parâmetro só é usado caso o *archive\_mode* esteja ativo e, uma vez que o uso desse parâmetro piora a performance, é de esperar que o mesmo se repita aqui.

Valor	Throughput (tx/s)	Response time (s)	Abort rate (%)
%p	36,586	0.533	0.0734
%f	36,387	0.564	0.0793

Os resultados estão de acordo com as expectativas, pelo que o grupo optou por continuar a não ligar o *archive\_mode* e ignorar este parâmetro também.

### 4.3.3 archive\_timeout

Este parâmetro precisa de algum contexto para ser entendido. O *archive\_command* apenas é invocado para segmentos WAL completos. Logo, se o servidor gerar pouco tráfego WAL, pode haver longos atrasos entre a conclusão de uma transação e o seu armazenamento seguro na *archive storage*. Para limitar este desfazamento surge o *archive\_timeout*, que serve para forçar o servidor a mudar para um segmento WAL novo.

Infelizmente, o grupo não teve tempo de fazer testes para este parâmetro. Contudo, tendo em conta que é necessário ativar tanto o *archive\_mode* como o *archive\_command* para poder usar este parâmetro e ambos pioram a performance, é de esperar que os testes ao *archive\_timeout* levassem também a resultados negativos.



## 4.4 Combinação dos parâmetros adotados

Depois de testar os parâmetros da configuração individualmente, o grupo escolheu a opção mais benificial de cada um e executou a carga transacional com essa combinação de valores, em busca de uma configuração que produzisse melhores resultados que a *predefinida*.

Começou-se por combinar apenas os parâmetros de cada secção do *Write-Ahead Log*, passando depois ao uso simultâneo de todos, de forma a observar a evolução da performance de forma mais faseada.

### 4.4.1 Settings

A configuração de *settings* a utilizar, com base nos resultados dos testes individuais de cada parâmetro, é a seguinte:

```
fsync = on
synchronous_commit = on
wal_sync_method = fsync
full_page_writes = on
wal_buffers = 16MB
commit_delay = 1000ms
commit_siblings = 10
```

Os resultados obtidos são apresentados de seguida:

Configuração	Throughput (tx/s)	Response time (s)	Abort rate (%)
Default	37.921286275895575	0.5117257814016696	0.04425271360979683
Nova	55.445151033386324	0.34774903631568277	0.0453224869262057

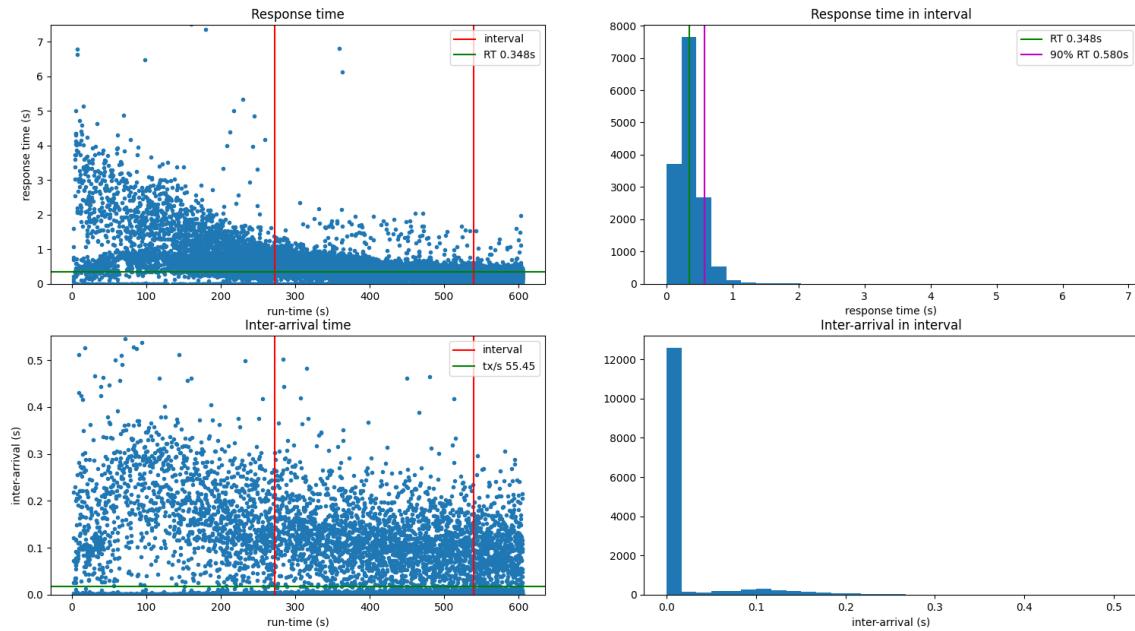


Figura 33: Métricas da configuração com a nova combinação de *settings*

A combinação dos melhores valores para cada parâmetro desta secção resultou num incremento bastante significativo do *throughput*, diminuição do *response time* e mantimento da taxa de aborto. A performance da carga transacional melhorou bastante em relação à configuração predefinida e pode considerar-se este teste um grande sucesso.

#### 4.4.2 Checkpoints

Relativamente à secção de *checkpoints*, a nova combinação de parâmetros a ter em conta é a seguinte:

```
checkpoint_timeout = 1min
max_wal_size = 4GB
min_wal_size = 320MB
checkpoint_completion_target = 0.5s
checkpoint_after_flush = 0
checkpoint_warning = 15s
```

Os resultados obtidos com esta configuração foram os seguintes:

Configuração	Throughput (tx/s)	Response time (s)	Abort rate (%)
<i>Default</i>	37.921286275895575	0.5117257814016696	0.04425271360979683
Nova	51.55431234261666	0.3748408150851582	0.0440193056928534

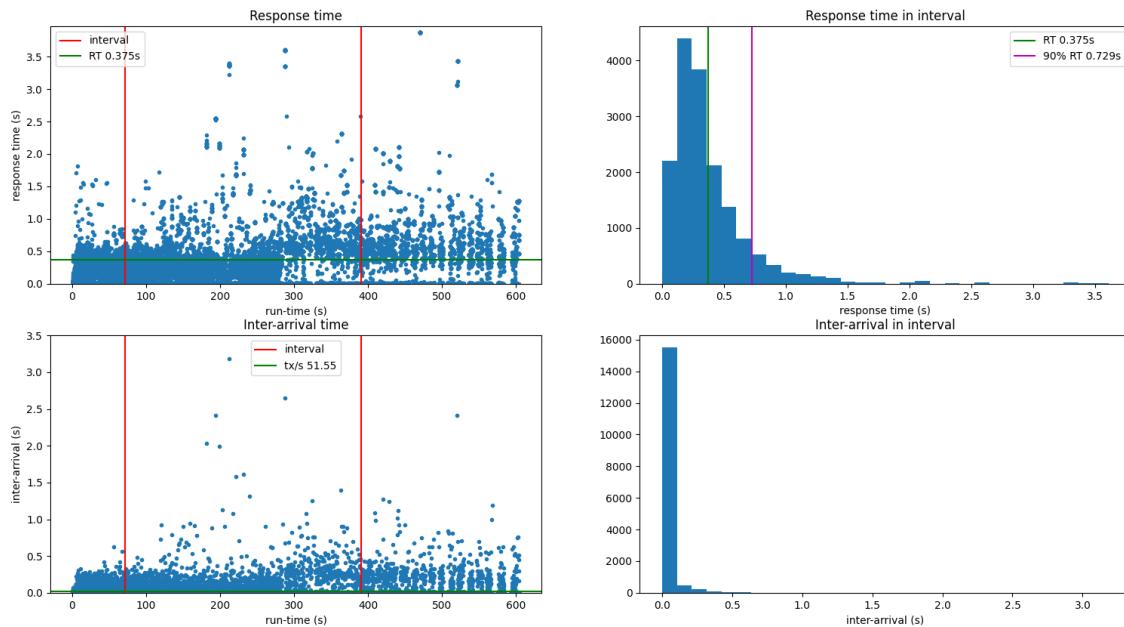


Figura 34: Métricas da configuração com a nova combinação de *checkpoints*

Tal como aconteceu com os *settings*, a nova configuração de *checkpoints* também levou a uma melhoria considerável do desempenho da carga transacional, pelo que este teste corrobora a viabilidade desta combinação de parâmetros.

#### 4.4.3 Archiving

Todos os valores alternativos dos parâmetros (testados) desta secção levaram ao degradaamento da performance, apresentando *throughputs* inferiores, bem como *response times* e *abort rates* superiores à configuração predefinida.

Por este motivo, o grupo não viu utilidade em testar a carga transacional com a combinação de parâmetros de *archiving*, uma vez que não havia motivo para a performance melhorar.

#### 4.4.4 Configuração de referência

Por fim, o grupo executou a carga transacional com os novos valores de todas as secções do *Write-Ahead Log*:

```
fsync = on
synchronous_commit = on
wal_sync_method = fsync
full_page_writes = on
wal_buffers = 16MB
commit_delay = 1000ms
commit_siblings = 10
```

```

checkpoint_timeout = 1min
max_wal_size = 4GB
min_wal_size = 320MB
checkpoint_completion_target = 0.5s
checkpoint_after_flush = 0
checkpoint_warning = 15s
  
```

Os resultados obtidos podem ser observados de seguida:

Configuração	Throughput (tx/s)	Response time (s)	Abort rate (%)
Default	37.921286275895575	0.5117257814016696	0.04425271360979683
Nova	55.445151033386324	0.34774903631568277	0.0453224869262057

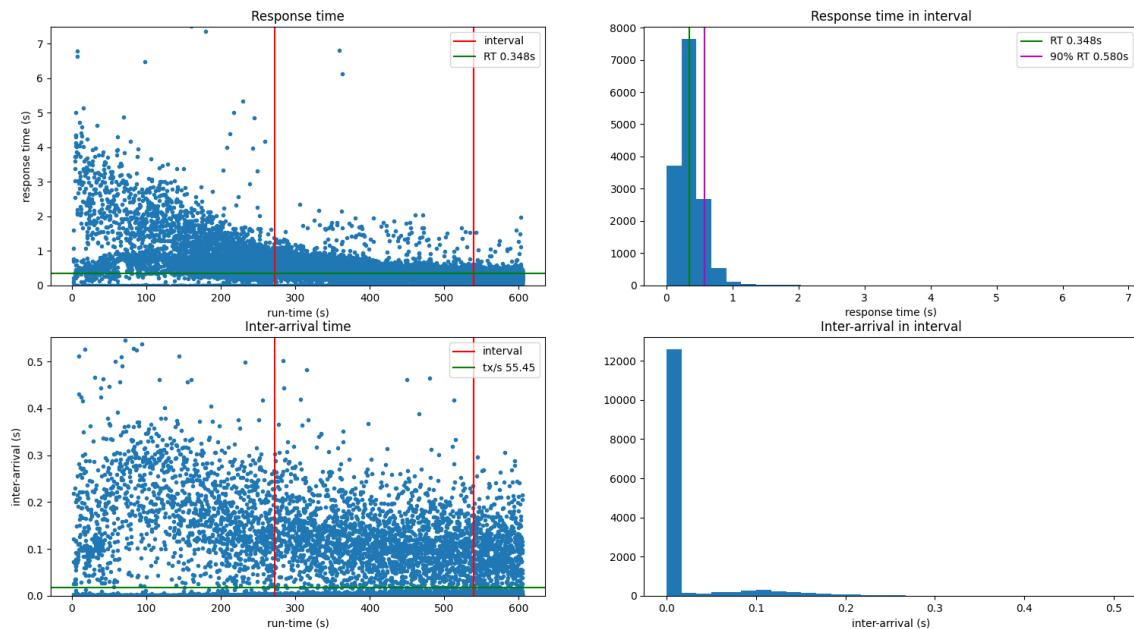


Figura 35: Métricas da configuração com a nova combinação de *checkpoints*

Esta nova configuração permite uma grande otimização do desempenho da carga transacional em relação à predefinida do *PostgreSQL*, pelo que o grupo considera que fez um trabalho bastante satisfatório e obteve uma boa configuração de referência.



## 5 Conclusão

O desenvolvimento deste projeto permitiu a consolidação da matéria teórico-prática lecionada nas aulas da unidade curricular e um melhor entendimento do funcionamento interno do *PostgreSQL*, nomeadamente o papel de cada parâmetro da configuração relativo a transações e o efeito dos seus valores possíveis no desempenho da carga transacional de uma base de dados. Além disso, permitiu também uma reforçada aprendizagem do funcionamento de ferramentas de otimização de consultas, como índices e vistas materializadas, bem como a identificação das circunstâncias em que devem ser utilizadas e a relevância da organização dos dados em disco para a sua eficácia.

O grupo está satisfeito com o resultado final e considera que cumpriu todos os requisitos propostos no enunciado. A realização do trabalho prático foi bastante demorada, devido à longa duração de operações como o populamento da base de dados e ao elevado número de vezes que se executou a carga transacional da base de dados (10 minutos por *run*) - além disso, o grupo restaurou o estado inicial da base de dados ao fim de cada *run*, de maneira a garantir a autenticidade dos resultados obtidos, o que também contribuiu bastante para tornar a realização do projeto mais demorosa.

O grupo termina assim este relatório agradecendo ao docente tanto pela clara exposição e explicação de conceitos nas aulas, que se provou fulcral para o desenvolvimento do projeto, como pela constante disponibilidade para esclarecer dúvidas.



## A Script de configuração do servidor

Este *script* foi criado para fazer a configuração inicial da máquina virtual do servidor de *PostgreSQL* e iniciar a sua execução. Deve ser corrido na *root* da máquina virtual.

```
#!/bin/bash

sudo apt-get update
sudo apt-get install postgresql-12
sudo systemctl stop postgresql
sudo systemctl disable postgresql

/usr/lib/postgresql/12/bin/initdb -D data

sudo sed -i "s/#listen_addresses = 'localhost'/listen_addresses =
'localhost,server'/g" data/postgresql.conf
sudo sed -i 's/# IPv6 local
connections:/host\tall\tall\t\t10.128.0.0\24\t\ttrust\n#
IPv6 local connections:/g' data/pg_hba.conf

/usr/lib/postgresql/12/bin/postgres -D data -k.
```



## B Script de configuração da benchmark

Este *script* foi criado para fazer a configuração inicial da máquina virtual da *benchmark*. Deve ser corrido na *root* da máquina virtual.

```
#!/bin/bash

USER = $hugo

sudo apt-get update
sudo apt-get install openjdk-8-jdk

tar xvf tpc-c-0.1-SNAPSHOT(tpc-c.tar.gz
tar xvf extra.tgz

cd tpc-c-0.1-SNAPSHOT/

sudo sed -i "s/db.connection.string=jdbc:postgresql:\/\/localhost\/tpcc/db.
connection.string=jdbc:postgresql:\/\/server\tpcc/g"
etc/database-config.properties
sudo sed -i "s/db.username=alfranio/db.username=$USER/g"
etc/database-config.properties
sudo sed -i "s/db.password=pass/db.password=/g"
etc/database-config.properties

sudo apt-get install postgresql-client-12
```



## C Script de populamento da base de dados para um dado número de *warehouses*

Este *script* foi criado para facilitar o procura pelo número ideal de *warehouses* para a configuração de referência. Elimina a base de dados existente e cria uma nova, executando o *load* para o número de *warehouses* indicado. Deve ser corrido na *root* da *benchmark*.

```
#!/bin/bash

declare -i WAREHOUSES = 80

cd tpc-c-0.1-SNAPSHOT/
sudo sed -i 's/tpcc.number.warehouses = [0-9]+/tpcc.number.warehouses =
$WAREHOUSES/g' etc/workload-config.properties

dropdb -h server tpcc
createdb -h server tpcc

psql -h server tpcc < etc/sql/postgresql/createtable.sql
psql -h server tpcc < etc/sql/postgresql/createindex.sql
for i in etc/sql/postgresql/*01; do psql -h server tpcc < $i; done

./load.sh

cd ../extra/
psql -h server tpcc < createExtraTables.sql
```



## D Script de execução da carga transacional para um dado número de clientes

Este *script* executa a carga transacional para o número de clientes especificado, depois de restaurar a base de dados ao seu estado inicial. Deve ser corrido na *root* da *benchmark*.

```
#!/bin/bash

declare -i CLIENTS = 80

pg_restore -h server -c -d tpcc < tpcc.dump

cd tpc-c-0.1-SNAPSHOT/
sudo sed -i 's/tpcc.numclients = [0-9]+/tpcc.numclients =
$CLIENTS/g' etc/workload-config.properties

./run.sh
```