

**Universidade do Minho**  
Escola de Engenharia

Administração de Bases de Dados  
Engenharia de Aplicações

Grupo 10

Alexandre Miranda a84462

Alexandre Ferreira a84961

João Azevedo a85227

Paulo Araújo a85729

Braga,  
Janeiro 2021

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>6</b>
<b>2</b>	<b>Instalação e configuração do <i>benchmark</i> TPC-C</b>	<b>7</b>
<b>3</b>	<b>Proposta de configuração de referência</b>	<b>8</b>
3.1	Definição do número de <i>warehouses</i> . . . . .	8
3.2	Definição do número de clientes . . . . .	9
<b>4</b>	<b>Adaptação e otimização de interrogações analíticas</b>	<b>11</b>
4.1	Interrogação Analítica 1 - Query 1 . . . . .	11
4.1.1	Índice . . . . .	12
4.1.2	<i>Materialized views</i> . . . . .	13
4.2	Interrogação Analítica 2 - Query 21 . . . . .	15
4.2.1	Índice . . . . .	16
4.2.2	<i>Materialized views</i> . . . . .	17
4.3	Interrogação Analítica 3 - Query 17 . . . . .	20
4.3.1	Índices . . . . .	20
4.3.2	<i>Materialized views</i> . . . . .	22
4.4	Interrogação Analítica 4 - Query 18 . . . . .	24
4.4.1	Índices . . . . .	24
4.4.2	<i>Materialized Views</i> + Índices . . . . .	25
<b>5</b>	<b>Otimização do desempenho da carga transacional</b>	<b>31</b>
5.1	<i>Default</i> . . . . .	31
5.2	<i>Settings</i> . . . . .	32
5.2.1	<i>fsync</i> . . . . .	32
5.2.2	<i>synchronous_commit</i> . . . . .	33
5.2.3	<i>wal_sync_method</i> . . . . .	34
5.2.4	<i>full_page_writes</i> . . . . .	34
5.2.5	<i>wal_writer_delay</i> . . . . .	35
5.2.6	<i>wal_writer_flush_after</i> . . . . .	36
5.2.7	<i>commit_delay</i> & <i>commit_siblings</i> . . . . .	36
5.3	<i>Checkpoints</i> . . . . .	37
5.3.1	<i>checkpoint_timeout</i> . . . . .	37
5.3.2	<i>max_wal_size</i> . . . . .	38
5.3.3	<i>min_wal_size</i> . . . . .	39
5.3.4	<i>checkpoint_completion_target</i> . . . . .	40
5.3.5	Análise de resultados . . . . .	40
5.4	<i>Archiving</i> . . . . .	42
5.4.1	<i>archive_mode</i> . . . . .	42
5.5	Combinações . . . . .	42
5.5.1	<i>max</i> & <i>min wal_size</i> . . . . .	42
5.5.2	<i>max</i> & <i>min wal_size</i> e <i>sync_commit</i> . . . . .	43
5.5.3	<i>max</i> & <i>min wal_size</i> , <i>sync_commit</i> e <i>checkpoint_timeout</i> . . . . .	44
5.5.4	<i>max</i> & <i>min wal_size</i> , <i>sync_commit</i> , <i>checkpoint_timeout</i> e <i>commit delay</i> & <i>siblings</i> . . . . .	45
5.6	Análise de Resultados . . . . .	46

<b>6</b>	<b>Conclusão</b>	<b>48</b>
<b>7</b>	<b>Scripts</b>	<b>49</b>
7.1	Run 1 . . . . .	49
7.2	Run 2 . . . . .	50

## Lista de Figuras

1	Valor do CPU, medido pela aplicação <i>htop</i> , para a máquina server . . . . .	9
2	<i>Query plan</i> - Interrogação analítica 1 - sem otimizações . . . . .	12
3	<i>Query plan</i> - Interrogação analítica 1 - utilização do índice em <i>order_line</i>	12
4	<i>Query plan</i> - Interrogação analítica 1 - com utilização da <i>materialized view matq1</i> . . . . .	13
5	<i>Query plan</i> - Interrogação analítica 1 - com utilização da <i>materialized view matq1all</i> . . . . .	14
6	<i>Query plan</i> - Interrogação analítica 2 - sem otimizações . . . . .	15
7	<i>Query plan</i> - Interrogação analítica 2 - com utilização do índice <i>query2_supp</i>	16
8	Tabela de índices . . . . .	17
9	<i>Query plan</i> - Interrogação analítica 2 - com utilização da opção <i>seq_scan</i> a <i>off</i> . . . . .	17
10	<i>Query plan</i> - Interrogação analítica 2 - com utilização da <i>materialized view query2_mat</i> . . . . .	19
11	<i>Query plan</i> - Interrogação analítica 3 - sem otimizações . . . . .	20
12	<i>Query plan</i> - Interrogação analítica 3 - sem recorrer à utilização do índice <i>query3_item</i> criado . . . . .	21
13	<i>Query plan</i> - Interrogação analítica 3 - com a opção <i>seq_scan</i> a <i>off</i> . . . . .	21
14	<i>Query plan</i> - Interrogação analítica 3 - com utilização da <i>materialized view query3_mat</i> . . . . .	22
15	<i>Query plan</i> - Interrogação analítica 3 - com utilização da <i>materialized view query3_mat</i> e <i>seq_scan</i> a <i>off</i> . . . . .	23
16	<i>Query plan</i> - Interrogação analítica 4 - sem otimizações . . . . .	24
17	<i>Query plan</i> - Interrogação analítica 4 - com a opção <i>seq_scan</i> a <i>off</i> . . . . .	25
18	<i>Query plan</i> - Interrogação analítica 4 - com utilização da <i>materialized view query4_custord</i> . . . . .	26
19	<i>Query plan</i> - Interrogação analítica 4 - com utilização do índice <i>matindex</i>	27
20	<i>Query plan</i> - Interrogação analítica 4 - com utilização da <i>materialized view query4_ordol</i> . . . . .	28
21	<i>Query plan</i> - Interrogação analítica 4 - com utilização do índice <i>matindexq4</i>	28
22	<i>Query plan</i> - Interrogação analítica 4 - com utilização da <i>materialized view matq4alljoins</i> . . . . .	29
23	<i>Query plan</i> - Interrogação analítica 4 - com utilização do índice <i>indq4alljoins</i>	30
24	Gráficos obtidos da configuração <i>default</i> . . . . .	32
25	Gráficos obtidos da configuração <i>fsync</i> a <i>off</i> . . . . .	33
26	Gráficos obtidos da configuração <i>sync. commit</i> a <i>off</i> . . . . .	34
27	Gráficos obtidos da configuração <i>full_page_writes</i> a <i>off</i> . . . . .	35
28	Gráficos obtidos da configuração <i>wal_writer_delay</i> a 400ms . . . . .	36
29	Gráficos obtidos da configuração <i>commit_delay</i> a 500ms e <i>commit_siblings</i> a 6. . . . .	37
30	Gráficos obtidos da configuração <i>max_wal_size</i> a 8GB . . . . .	39
31	Gráficos obtidos da configuração <i>min_wal_size</i> a 160MB . . . . .	40
32	Valores de <i>wal files usage</i> para conf. <i>default</i> . . . . .	41
33	Valores de <i>wal files usage</i> para <i>max_wal_size</i> = 8GB. . . . .	41
34	Valores de <i>wal files usage</i> para <i>min_wal_size</i> = 160MB. . . . .	41
35	Gráficos obtidos da combinação <i>max &amp; min wal_size</i> . . . . .	43

36	Gráficos obtidos da combinação $\max \otimes \min wal\_size$ e $sync\_commit$ . . . . .	44
37	Gráficos obtidos da combinação $\max \otimes \min , sync\_commit$ e $checkp\_timeout$ . .	45
38	Gráficos obtidos da combinação $\max \otimes \min , sync\_commit, checkpoint\_timeout$ e $commit\_delay \otimes siblings$ .	46
39	Gráfico <i>Throughput</i> comparativo das diferentes combinações e <i>default</i> . . . . .	46
40	Gráfico <i>Response-time</i> comparativo das diferentes combinações e <i>default</i> . . . . .	47
41	Gráfico <i>Abort-rate</i> comparativo das diferentes combinações e <i>default</i> . . . . .	47
42	Gráfico <i>Response-time/Throughput</i> comparativo das diferentes combinações e <i>default</i> .	47

## Lista de Tabelas

1	Relação entre o n.º de <i>warehouses</i> e o tamanho da base de dados . . . . .	8
2	Valores da análise Escada TPC-C para a variação no n.º de clientes. . . . .	9
3	Tempos de execução com e sem <i>materialized view</i> e <i>seq_scan</i> a <i>on</i> e <i>off</i> . . . . .	23
4	Valores da análise Escada TPC-C para configuração <i>default</i> . . . . .	31
5	Valores da análise Escada TPC-C para a variação na configuração <i>fsync</i> . . . . .	32
6	Valores da análise Escada TPC-C para a variação na configuração <i>synchronous_commit</i> . . . . .	33
7	Valores da análise Escada TPC-C para a variação na conf. <i>full_page_writes</i> . . . . .	34
8	Valores da análise Escada TPC-C para a variação na conf. <i>wal_writer_delay</i> . . . . .	35
9	Valores da análise Escada TPC-C para a variação na configuração <i>wal_writer_flush_after</i> . . . . .	36
10	Valores da análise Escada TPC-C para a variação na combinação das configurações <i>commit_delay</i> & <i>commit_siblings</i> . . . . .	37
11	Valores da análise Escada TPC-C para a variação na configuração <i>checkpoint_timeout</i> . . . . .	38
12	Valores da análise Escada TPC-C para variação na configuração <i>max_wal_size</i> . . . . .	38
13	Valores da análise Escada TPC-C para variação na configuração <i>min_wal_size</i> . . . . .	39
14	Valores da análise Escada TPC-C para a variação na configuração <i>checkpoint_completion_target</i> . . . . .	40
15	Valores da análise Escada TPC-C para a variação na configuração <i>archive_mode</i> . . . . .	42
16	Valores da análise Escada TPC-C para a variação na configuração <i>max_wal_size</i> & <i>min_wal_size</i> . . . . .	43
17	Valores da análise Escada TPC-C para a variação na configuração <i>max</i> & <i>min wal_size</i> e <i>synchronous_commit</i> . . . . .	43
18	Valores da análise Escada TPC-C para a variação na configuração <i>max_wal_size</i> , <i>min_wal_size</i> , <i>synchronous_commit</i> e <i>checkpoint_timeout</i> . . . . .	44
19	Valores da análise Escada TPC-C para a variação na configuração <i>max_wal_size</i> , <i>min_wal_size</i> , <i>synchronous_commit</i> , <i>checkpoint_timeout</i> e <i>commit delay</i> & <i>siblings</i> . . . . .	45

# 1 Introdução

Este relatório descreve a nossa resolução do trabalho prático da unidade curricular Administração de Bases de Dados.

O trabalho foi feito com o intuito de melhor perceber como funcionam os conceitos teóricos de uma base de dados relacional num ambiente prático real. Para isso foram feitos vários testes a uma base de dados *postgres* utilizando o *benchmark* TPC-C. Este consiste no processamento de transações que simulam uma base de dados de uma cadeia de lojas.

A tarefa inicial passou por criarmos máquinas virtuais com recurso à *Google Cloud* com as configurações referenciadas pelo docente tendo ainda em conta a nossa avaliação quanto ao tamanho dos discos das máquinas.

Seguiu-se a instalação do *benchmark* e a configuração da base de dados que foi feita recorrendo a alguns testes iniciais do TPC-C. Este passo serviu para termos uma configuração de referência para o resto do trabalho.

Tendo uma configuração bem definida passamos à adaptação e otimização de algumas interrogações analíticas originárias do TPC-H. Para isso tivemos em conta os mecanismos de redundância aprendidos ao longo deste semestre, tentando aplicar estes conceitos neste contexto prático.

Por fim passamos à otimização da carga transacional do TPC-C editando as configurações do *postgres* que o docente indicou como mais relevantes.

## 2 Instalação e configuração do *benchmark* TPC-C

Para conseguirmos fazer a configuração inicial do *benchmark* TPC-C, começamos por seguir o tutorial apresentado pelo docente para a criação de máquinas na *Google Cloud* e instalação do escada TPC-C. Neste sentido, configuramos então uma máquina com o *benchmark*, onde está instalado o cliente *postgres* - f1-micro (1 CPU e 0,6 GB de memória) - e uma máquina server, onde se encontra instalado o servidor *postgres* - n1-standard-2 (2 CPUs e 7,5 GB de memória).

Como o docente referiu no final do tutorial, a performance do disco SSD é proporcional ao tamanho do mesmo, com esta informação, de forma a obter uma maior performance, tivemos em consideração a definição do tamanho do disco, e por isso definimos um tamanho de **100GB**.

### 3 Proposta de configuração de referência

Para encontrarmos uma configuração de referência, decidimos tirar o máximo partido da máquina onde corre o servidor *postgres*. Como sabemos que esta tem dois CPU's e 7,5GB de RAM procuramos encontrar uma configuração que tirasse o máximo partido da memória RAM existente e ainda permitisse uma utilização aceitável do CPU, isto é nunca abaixo dos 50%, para que não haja um sub aproveitamento do CPU, mas também nunca acima dos 90%, de modo a garantir que não existe uma grande saturação de recursos do processador. Assim sendo começamos por definir o número de *warehouses* e em seguida passamos à definição do número de clientes.

#### 3.1 Definição do número de *warehouses*

Para esta decisão baseámo-nos apenas no tamanho da base de dados. Após discussões com o docente, percebemos que o tamanho da mesma deve se encontrar no intervalo de 6 a 10 GB, de modo a tirar o máximo partido da memória RAM.

Não tendo qualquer noção do tamanho da base de dados em relação ao número de *warehouses*, decidimos povoar a mesma com 10 *warehouses* e ver o tamanho da mesma. Com isto percebemos que o valor pouco ultrapassa 1GB, e por isso decidimos povoar a mesma com 70 *warehouses*, ou seja, fizemos a seguinte aproximação:

$$10 \approx 1GB$$

$$70 \approx 7GB$$

Sabemos que esta não foi a estimativa mais correta, mas permitiu-nos encontrar um valor dentro do aceitável.

Como o *load* é um processo muito demorado decidimos desligar a opção '*fsync*' no servidor *postgres*, isto aumenta a performance do processo, apesar de aumentar o risco de perda de dados, no caso da ocorrência de uma falha no sistema. No entanto, como estamos apenas a povoar a base de dados, um erro não originará uma perda de dados irreversível. Deste modo, conseguimos diminuir o tempo de execução do povoamento, que é exponencial dependendo do número de *warehouses* definidas, conseguindo também poupar recursos na *GCP - Google Cloud Platform*.

Obtemos então os seguintes valores de tamanho de base de dados:

N. <sup>º</sup> <i>warehouses</i>	Tamanho da base de dados (MB)
10	1158
70	7948

Tabela 1: Relação entre o n.<sup>º</sup> de *warehouses* e o tamanho da base de dados

Sendo o valor de 7948 MB um valor dentro do intervalo já mencionado, podemos assumir 70 *warehouses* como a nossa escolha definitiva.

### 3.2 Definição do número de clientes

Para a definição do número de clientes tivemos que avaliar os valores de *Throughput* e *Response Time*, bem como os valores de *abort rate*, apesar de este último não ter tido tanta influência na tomada de decisão.

Para chegar ao valor ideal tivemos que correr o *benchmark*, utilizando um tempo de teste de 5 minutos. Achamos que este tempo seria suficiente para conseguirmos ter valores claros sobre qual a melhor opção para o número de clientes a utilizar.

Começamos por avaliar o valor de 20 clientes por *warehouse* (dando um total de 1400 clientes) e fomos incrementando esse valor em 20 clientes por cada execução diferente, até observarmos que os valores deixavam de ser indicativos de melhoria. Para não existir diferença no ambiente de teste, de cada vez que corríamos o *benchmark* limpávamos a base de dados para a execução seguinte, recorrendo ao ficheiro *dump* criado após o povoamento dos 70 *warehouses*.

N.º clientes	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
20	121,5541044	0,010583649	0,002831333
40	221,4360909	0,064804093	0,035922014
60	244,2958729	0,078553715	0,05120664
80	256,9159783	0,074574283	0,049388362
100	242,5893733	0,079181101	0,04897481

Tabela 2: Valores da análise Escada TPC-C para a variação no n.º de clientes.

Para conseguir tirar conclusões tivemos que analisar a performance do CPU aquando da realização de cada um dos testes para o diferente número de clientes. O valores do CPU observados para a máquina servidor *postgres* foram dentro dos 50% de utilização do mesmo, no entanto, para a máquina *benchmark* esses valores ultrapassavam bastante os 100% de utilização.



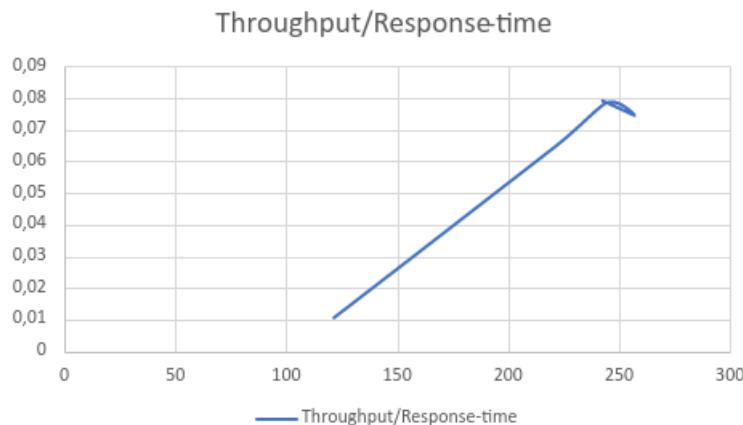
Figura 1: Valor do CPU, medido pela aplicação *htop*, para a máquina server

Sendo que os valores obtidos na máquina servidor *postgres* estão dentro dos valores aceitáveis definido inicialmente, (entre 50% e 90%), podemos assumir que a escolha do número de clientes apenas terá em consideração outras variáveis. Porém, depois de discutir com o docente o valor de CPU obtido para a máquina *benchmark*, e como sugerido pelo mesmo, fizemos um *upgrade*, aumentando o número de CPUs para dois.

De modo a confirmar que a melhoria da máquina permitiu baixar a utilização de CPU, voltamos a realizar o processo anterior para 80 clientes.

Como seria de esperar esse valor baixou, e os valores obtidos de *Throughput*, *Response Time* e *abort rate* mantiveram-se na mesma ordem de grandeza dos valores obtidos anteriormente.

Deste modo conseguimos agora selecionar o valor aceitável para o número de clientes. Queremos então maximizar o *throughput*, diminuindo o *response time*, sem que o *abort rate* suba para valores exorbitantes. Para fazer a tomada de decisão apoiamo-nos, principalmente, no seguinte gráfico:



Assim sendo o valor de clientes por *warehouse* escolhido foi o de 80 clientes, dando um total de 5600 clientes no sistema. Escolhemos este valor pois é o que nos permite maximizar o *throughput* e minimizar o *response time* para valores considerados aceitáveis.

Concluindo temos então um número de *warehouses* estabelecido em 70, e um número de clientes por *warehouse* estabelecido em 80.

Para a realização desta tarefa criamos um *script*(7.1) de configuração do *workload* e execução do *benchmark* para nos ajudar a obter os resultados de maneira mais rápida.

## 4 Adaptação e otimização de interrogações analíticas

Como somos um grupo de 4 elementos analisamos e otimizamos 4 interrogações analíticas.

Para uma análise inicial corremos cada uma das queries alvo com os comandos ***Explain Analyze***, tal como foi indicado pelo docente desta unidade curricular. Com isto conseguimos ver, não só o tempo total da execução da *query*, mas também os passos e os seus respetivos tempos dados pela base de dados para o processamento das queries.

Recorrendo a diferentes mecanismos aprendidos na aula, como **índices** e **materialized views**, tentamos otimizar as diferentes queries escolhidas.

### 4.1 Interrogação Analítica 1 - Query 1

Para primeira *query* a ser analisada decidimos escolher a *query 1* apresentada no site fornecido. Esta escolha deveu-se pela simplicidade da mesma para uma melhor ambientação à base de dados.

A primeira tarefa a ser feita foi a de proceder à edição da *query* para que esta fosse compatível com a base de dados do TPC-C. Isto apenas consistiu em alterar o nome da tabela *orderline* para *order\_linet*. De seguida analisamos o corpo da *query* e reparamos que esta utilizava uma data como um filtro. Decidimos alterar essa data, nomeadamente a data presente no campo *ol\_delivery\_dt*, para uma data que realmente distinguisse algumas linhas da tabela e não estivesse apenas a aceitar todas as linhas. Ora como o povoamento da base de dados que está a ser utilizado foi feito no dia 27 de dezembro de 2020 e a tabela *order\_line* foi povoada entre as 19h30 e as 20h30, colocamos o filtro no meio desse horário para podermos ter um filtro que realmente diferenciasse algumas linhas da tabela.

Então a *query* estudada foi a seguinte:

```
select ol_number,
       sum(ol_quantity) as sum_qty,
       sum(ol_amount) as sum_amount,
       avg(ol_quantity) as avg_qty,
       avg(ol_amount) as avg_amount,
       count(*) as count_order
  from orderline
 where ol_delivery_d > '2020-12-27 20:00:00.000000'
 group by ol_number
 order by ol_number
```

Como já foi referido anteriormente corremos esta *query* com a opção *explain analyze* para podermos ver os passos dados pelo *postgres* e o tempo de execução. O resultado obtido foi o seguinte:

```

-----  

QUERY PLAN  

-----  

Finalize GroupAggregate (cost=342466.72..342470.50 rows=13 width=116) (actual time=3241.191..3244.526 rows=13 loops=1)
  Group Key: ol_number
    -> Gather Merge (cost=342466.72..342469.75 rows=26 width=116) (actual time=3241.141..3244.396 rows=39 loops=1)
      Workers Planned: 2
      Workers Launched: 2
        -> Sort (cost=341466.69..341466.73 rows=13 width=116) (actual time=3204.635..3204.639 rows=13 loops=3)
          Sort Key: ol_number
          Sort Method: quicksort Memory: 28kB
          Worker 0: Sort Method: quicksort Memory: 28kB
          Worker 1: Sort Method: quicksort Memory: 28kB
            -> Partial HashAggregate (cost=341466.26..341466.45 rows=13 width=116) (actual time=3204.570..3204.587 rows=13 loops=3)
              Group Key: ol_number
              -> Parallel Seq Scan on order_line (cost=0.00..318743.07 rows=1514879 width=11) (actual time=1662.650..2271.116 rows=1234557 loops=3)
                Filter: (ol_delivery_d > '2020-12-27 20:00:00'::timestamp without time zone)
                Rows Removed by Filter: 4717640
Planning Time: 0.120 ms
JIT:
  Functions: 30
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 6.456 ms, Inlining 0.000 ms, Optimization 7.994 ms, Emission 87.021 ms, Total 101.471 ms
Execution Time: 3246.476 ms
(21 rows)

```

Figura 2: *Query plan* - Interrogação analítica 1 - sem otimizações

Como podemos ver o tempo de execução da *query* obtido foi de 3246 ms. Não sendo um valor muito elevado ainda assim acreditamos ser possível implementar melhorias na *query*.

#### 4.1.1 Índice

Ao analizarmos os valores do *query plan* conseguimos perceber que metade do tempo de execução da *query* era usado para a filtragem das linhas cujo o valor da data *ol\_delivery\_d* era superior a ”2020-12-27 20:00:00.000000”. Como pudemos verificar estava a ser executado um *sequential scan* para esta operação.

Assim sendo percebemos que aplicando um índice na tabela *order\_line* por valores de *ol\_delivery\_d*, a pesquisa poderia ser otimizada. Criamos então esse índice:

*Create index oldd on order\_line(ol\_delivery\_d);*

Sem termos que desligar o *Seq scan*, ou qualquer outro tipo de configuração do *postgres*, este conseguiu perceber que, através dos seus mecanismos de estatística, usando o índice criado a execução da *query* seria otimizada.

```

-----  

QUERY PLAN  

-----  

Finalize GroupAggregate (cost=152030.81..152034.59 rows=13 width=116) (actual time=2001.809..2006.535 rows=13 loops=1)
  Group Key: ol_number
    -> Gather Merge (cost=152030.81..152033.85 rows=26 width=116) (actual time=2001.780..2006.424 rows=39 loops=1)
      Workers Planned: 2
      Workers Launched: 2
        -> Sort (cost=151030.79..151030.82 rows=13 width=116) (actual time=1967.668..1967.670 rows=13 loops=3)
          Sort Key: ol_number
          Sort Method: quicksort Memory: 28kB
          Worker 0: Sort Method: quicksort Memory: 28kB
          Worker 1: Sort Method: quicksort Memory: 28kB
            -> Partial HashAggregate (cost=151030.35..151030.55 rows=13 width=116) (actual time=1967.602..1967.619 rows=13 loops=3)
              Group Key: ol_number
              -> Parallel Index Scan using oldd on order_line (cost=0.56..128307.17 rows=1514879 width=11) (actual time=0.081..887.758 rows=1234557 loops=3)
                Index Cond: (ol_delivery_d > '2020-12-27 20:00:00'::timestamp without time zone)
Planning Time: 0.399 ms
JIT:
  Functions: 30
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 6.767 ms, Inlining 0.000 ms, Optimization 3.524 ms, Emission 89.392 ms, Total 99.683 ms
Execution Time: 2008.761 ms
(20 rows)

```

Figura 3: *Query plan* - Interrogação analítica 1 - utilização do índice em *order\_line*

Como pudemos verificar o tempo de execução da *query* melhorou em mais de um terço, o que achamos satisfatório, tendo em conta o tempo global inicial obtido.

Com alguns testes realizados conseguimos perceber que quanto maiores fossem os valores aceites na filtragem menor seria o impacto da otimização da indexação, ou seja, se a *query* pedisse uma hora mais perto das 19h, mais valores seriam selecionados e o índice deixaria de ser automaticamente utilizado pelo *postgres*, sendo necessário desligar o *sequencial scan* para forçar a utilização dos índices. Isto acontecia pois as melhorias não são tão acentuadas, chegando às vezes a piorar em casos onde todos os valores existentes eram aceites na filtragem.

#### 4.1.2 Materialized views

Após a criação da indexação pensamos em criar uma *materialized view* de modo a melhorar ainda mais o tempo de execução. Começamos então por criar uma *materialized view* em que já estão filtrados todos os valores cuja data corresponde aos valores pedidos. Desta forma esperamos poupar ainda mais tempo do que com a indexação, uma vez que já ficam os valores gravados estaticamente em memória.

```
create materialized view matq1 as (
    select *
        from order_line
       where ol_delivery_d > '2020-12-27 20:00:00.000000');

select ol_number,
       sum(ol_quantity) as sum_qty,
       sum(ol_amount) as sum_amount,
       avg(ol_quantity) as avg_qty,
       avg(ol_amount) as avg_amount,
       count(*) as count_order
    from matq1
   group by ol_number
  order by ol_number;
```

QUERY PLAN	
Finalize GroupAggregate	(cost=86406.68..86410.46 rows=13 width=116) (actual time=1497.579..1497.701 rows=13 loops=1)
Group Key:	ol_number
-> Gather Merge	(cost=86406.68..86409.72 rows=26 width=116) (actual time=1497.550..1497.621 rows=39 loops=1)
Workers Planned:	2
Workers Launched:	2
-> Sort	(cost=85406.66..85406.69 rows=13 width=116) (actual time=1469.555..1469.558 rows=13 loops=3)
Sort Key:	ol_number
Sort Method:	quicksort Memory: 28kB
Worker 0:	Sort Method: quicksort Memory: 28kB
Worker 1:	Sort Method: quicksort Memory: 28kB
-> Partial HashAggregate	(cost=85406.22..85406.42 rows=13 width=116) (actual time=1469.500..1469.516 rows=13 loops=3)
Group Key:	ol_number
-> Parallel Seq Scan on matq1	(cost=0.00..62257.49 rows=1543249 width=11) (actual time=0.046..352.262 rows=1234557 loops=3)
Planning Time:	0.095 ms
Execution Time:	1497.762 ms
(15 rows)	

Figura 4: *Query plan* - Interrogação analítica 1 - com utilização da *materialized view* *matq1*

Como podemos ver houve uma melhoria de meio segundo em relação ao uso da indexação, no entanto, não achamos muito significativo uma vez que ter que atualizar ao longo do tempo uma vista materializada para ganhar apenas meio segundo à pesquisa dinâmica não compensa. Com este estudo conseguimos entender que o agrupamento das

linhas também é custoso (tanto ou mais que a filtragem). Assim sendo, uma maneira de contornar isto seria a de colocar toda a *query* numa *materialized view*.

```
create materialized view matq1all as (
    select ol_number,
        sum(ol_quantity) as sum_qty,
        sum(ol_amount) as sum_amount,
        avg(ol_quantity) as avg_qty,
        avg(ol_amount) as avg_amount,
        count(*) as count_order
    from order_line
    where ol_delivery_d > '2020-12-27 20:00:00.000000'
    group by ol_number
    order by ol_number);
```

```
tpcc=# explain analyze select * from matq1all;
                                         QUERY PLAN
-----
Seq Scan on matq1all  (cost=0.00..15.60 rows=560 width=116) (actual time=0.008..0.010 rows=13 loops=1)
Planning Time: 0.038 ms
Execution Time: 0.020 ms
(3 rows)
```

Figura 5: *Query plan* - Interrogação analítica 1 - com utilização da *materialized view* *matq1all*

Como podemos constatar agora a *query* é praticamente instantânea, tal como era de esperar.

Sendo então esta *query* muito simples, com um tempo total otimizado de 2 segundos usando indexação, não achamos portanto muito relevante a criação de uma *materialized view*, uma vez que isto implicaria uma atualização periódica da informação da mesma e achamos que o tempo de espera de 2 segundos não justifica tanto trabalho. Talvez com o aumento considerável da base de dados esta *query* se torne mais lenta e aí sim possa-se pensar na utilização desta estratégia.

## 4.2 Interrogação Analítica 2 - Query 21

Para a segunda interrogação analítica decidimos escolher a *query* 21.

Tal como na interrogação anterior tivemos que fazer pequenas alterações sintáticas para que a *query* fosse compatível com a base de dados. A *query* usada foi a seguinte:

```
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;
```

Executando a *query* com **Explain Analyze** obtivemos a seguinte análise.

```
QUERY PLAN
Sort  (cost=1042338.38..1042363.38 rows=10000 width=34) (actual time=24444.718..24593.144 rows=396 loops=1)
  Sort Key: (count(*)) DESC, supplier.su_name
  Sort Method: quicksort Memory: 55kB
-> Finalize GroupAggregate  (cost=1039066.97..1041073.99 rows=10000 width=34) (actual time=24395.529..24592.007 rows=396 loops=1)
    Group Key: supplier.su_name
    -> Gather  (cost=1039066.97..1041476.95 rows=19408 width=34) (actual time=24395.317..24592.146 rows=1188 loops=1)
        Workers Planned: 2
        Workers Launched: 2
          -> Partial GroupAggregate  (cost=1038066.94..1038236.76 rows=9704 width=34) (actual time=24313.624..24352.886 rows=396 loops=3)
              Group Key: supplier.su_name
              -> Sort  (cost=1038066.94..1038091.20 rows=9704 width=26) (actual time=24313.558..24337.971 rows=76649 loops=3)
                  Sort Key: supplier.su_name
                  Sort Method: external merge Disk: 2686kB
                  Worker 0: Sort Method: external merge Disk: 2686kB
                  Worker 1: Sort Method: external merge Disk: 2738kB
                  -> Nested Loop Anti Join  (cost=797673.96..1037424.33 rows=9704 width=26) (actual time=14320.287..23679.291 rows=76649 loops=3)
                      -> Parallel Hash Join  (cost=797673.40..934559.09 rows=14555 width=46) (actual time=14318.581..15053.471 rows=226677 loops=1)
                          Hash Cond: ((orders.o_id = l1.ol_o_id) AND (orders.o_id = l1.ol_w_id) AND (orders.o_d_id = l1.ol_d_id))
                          Join Filter: (l1.ol_delivery_d > orders.o_entry_d)
                          -> Parallel Seq Scan on orders  (cost=0..1033363.00 rows=14555 width=50) (actual time=0..050..212.869 rows=700000 loops=1)
                              -> Parallel Hash  (cost=795999.00..795999.00 rows=41686 width=50) (actual time=13702.513..13702.792 rows=226677 loops=3)
                                  Buckets: 65536 (originally 111072) Batches: 16 (originally 1) Memory Usage: 4224kB
                                  -> Parallel Hash Join  (cost=366380.24..796999.00 rows=43680 width=50) (actual time=9462.098..13266.878 rows=226677 loops=3)
                                      Hash Cond: (((l1.ol_w_id = stock.s_w_id) AND (l1.ol_l_id = stock.s_l_id)))
                                      -> Parallel Seq Scan on order_line_l1  (cost=0..00..300142.46 rows=7448246 width=24) (actual time=0..050..2033.041 rows=5952197 loops=3)
                                      -> Parallel Hash  (cost=366380.24..796999.00 rows=41686 width=50) (actual time=14195..3944.202 rows=88840 loops=3)
                                          Buckets: 65536 (originally 45536) (estimated cardinality: 45536) (actual time=2911.000..3824.807 rows=88840 loops=3)
                                          -> Hash Join  (cost=372.96..366122.88 rows=17157 width=34) (actual time=1873.778..3824.807 rows=88840 loops=3)
                                              Hash Cond: (mod((stock.s_w_id * stock.s_l_id), 10000) = supplier.su_suppkey)
                                              -> Parallel Seq Scan on stock  (cost=0..00..347348.67 rows=2916667 width=8) (actual time=0.055..1320.874 rows=2333333 loops=3)
                                              -> Hash  (cost=372.23..372.23 rows=59 width=30) (actual time=1873.661..1873.666 rows=396 loops=3)
                                              Buckets: 1024 Batches: 1 Memory Usage: 3kB
                                              -> Hash  (cost=372.23..372.23 rows=59 width=30) (actual time=1869.698..1873.520 rows=396 loops=3)
                                              Hash Cond: (supplier.su_nationkey = nation.nationkey)
                                              -> Seq Scan on supplier  (cost=0..00..322.00 rows=10000 width=34) (actual time=0.029..2.208 rows=10000 loops=3)
                                              -> Hash  (cost=12.12..12.12 rows=1 width=4) (actual time=1869.625..1869.627 rows=1 loops=3)
                                                  Buckets: 1024 Batches: 1 Memory Usage: 9kB
                                                  -> Seq Scan on nation  (cost=0..00..12.12 rows=1 width=4) (actual time=1869.609..1869.617 rows=1 loops=3)
                                                      Filter: (n_name = 'GERMANY'::bpchar)
                                                      Rows Removed by Filter: 24
-> Index Scan using pk_order_line on order_line_l2  (cost=0..50..53.37 rows=1 width=20) (actual time=0.037..0.037 rows=1 loops=800031)
  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: 13.124 ms
 JIT:
  Executions: 189
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 43.185 ms, Inlining 491.614 ms, Optimization 3124.746 ms, Emission 1976.993 ms, Total 5636.538 ms
Execution Time: 24988.590 ms
(50 rows)
```

Figura 6: *Query plan* - Interrogação analítica 2 - sem otimizações

Como podemos observar, para obter o resultado, esta *query* utiliza, automaticamente,

índices já existentes, como por exemplo o '*pk\_order\_line*', para otimizar a procura dos elementos da *order\_line*, isto é:  $l2.ol_o_id = l1.ol_o_id$ ,  $l2.ol_w_id = l1.ol_w_id$  e  $l2.ol_d_id = l1.ol_d_id$ , deste modo é possível otimizar os respetivos joins.

#### 4.2.1 Índice

Sendo a execução da *query* um pouco demorada, acreditamos que o facto de acrescentarmos ainda mais alguns índices, este tempo possa ser ainda melhorado.

Como pudemos reparar, não existem índices definidos para a tabela *supplier*, o que desencadeia *seq scans* nessa tabela. Estes tinham alguns custos, sendo até que um deles tinha uma duração acima dos dois segundos, o que nos levou a pensar qual seria a forma de melhorar esse tempo. Assim sendo, criamos indexação em relação aos valores de *su\_nationkey*, que era o valor desta tabela mais importante para esta *query*, nomeadamente na junção da tabela *supplier* com a tabela *nation*.

```
Create index query2_supp on supplier(su_nationkey);
```

Após a criação deste índice executamos de novo a *query* e obtivemos os seguintes resultados:

```
QUERY PLAN
Sort: (cost=1042225.54..1042250.54 rows=10000 width=34) (actual time=22306.542..22529.430 rows=396 loops=1)
  Sort Key: (count(*)) DESC, supplier.su_name
  Sort Method: quicksort Memory: 55KB
-> Finalize GroupAggregate (cost=1038954.13..1041561.15 rows=10000 width=34) (actual time=22267.900..22529.014 rows=396 loops=1)
  Group Key: supplier.su_name
    -> Gather Merge (cost=1038954.13..1041364.11 rows=19408 width=34) (actual time=22267.758..22528.577 rows=1188 loops=1)
      Workers Planned: 2
      Workers Launched: 2
        -> Partial GroupAggregate (cost=1037954.11..1038123.93 rows=9784 width=34) (actual time=22153.299..22189.853 rows=396 loops=3)
          Group Key: supplier.su_name
          -> Sort (cost=1037954.11..1037978.37 rows=9784 width=26) (actual time=22153.216..22174.872 rows=76649 loops=3)
            Sort Key: supplier.su_name
            Sort Method: external temp Disk: 2880kB
            Worker 0: Sort Method: external Disk: 2728kB
            Worker 1: Sort Method: external merge Disk: 2721kB
              -> Nested Loop Anti Join (cost=797561.13..1037311.49 rows=9704 width=26) (actual time=14766.611..21388.729 rows=76649 loops=3)
                -> Parallel Hash Join (cost=797560.56..934464.25 rows=14555 width=46) (actual time=14766.473..15617.810 rows=226677 loops=3)
                  Hash Cond: ((orders.o_id = l1.ol_w_id) AND (orders.o_id = l1.ol_d_id))
                  Join Filter: (l1.ol_delivery > orders.o_entry_d)
                  -> Parallel Seq Scan on orders (cost=0.00..2887.00 rows=875000 width=20) (actual time=0.042..208.485 rows=700000 loops=3)
                  -> Parallel Hash (cost=9567.16..103679.16 rows=13600 width=50) (actual time=14093.051..14093.662 rows=226677 loops=3)
                    Buckets: 65536 (originally 131072) Batches: 16 (originally 1) Memory Usage: 4224kB
                    -> Parallel Hash Join (cost=360267.40..790796.16 rows=43689 width=50) (actual time=10166.318..13653.811 rows=226677 loops=3)
                      Hash Cond: ((l1.ol_w_id = stock.s_w_id) AND (l1.ol_d_id = stock.s_t_id))
                      Hash Cond: ((l1.ol_w_id = stock.s_w_id) AND (l1.ol_d_id = stock.s_t_id))
                      -> Parallel Seq Scan on order_line l1 (cost=0.00..300142.46 rows=7440246 width=24) (actual time=2.752..2231.856 rows=5952197 loops=3)
                      -> Parallel Hash (cost=360610.04..366610.04 rows=17157 width=34) (actual time=4229.842..4229.848 rows=88840 loops=3)
                        Buckets: 65536 (originally 131072) Batches: 8 (originally 16) Memory Usage: 12kB
                        -> Hash Join (cost=260.13..601000 rows=17157 width=34) (actual time=1637.730..1637.734 rows=17157 width=34) (actual time=1637.730..1637.734 rows=17157 width=34)
                          Hash Cond: (mod(stock.s_w_id * stock.s_t_id, 10000) = supplier.su_supplierkey)
                          -> Parallel Seq Scan on stock (cost=0.00..347348.07 rows=2916667 width=8) (actual time=0.027..1846.623 rows=2333333 loops=3)
                          Hash (cost=259.39..259.39 rows=59 width=30) (actual time=1637.648..1637.651 rows=396 loops=3)
                          Buckets: 1024 Batches: 1 Memory Usage: 33kB
                          -> Nested Loop (cost=11.38..259.39 rows=59 width=30) (actual time=1635.452..1637.488 rows=396 loops=3)
                            -> Seq Scan on nation (cost=0.00..12.12 rows=59 width=4) (actual time=1635.298..1635.304 rows=1 loops=3)
                              Rows Removed by Filter: 24
                            -> Bitmap Heap Scan on supplier (cost=11.38..243.27 rows=400 width=34) (actual time=0.135..2.096 rows=396 loops=3)
                              Recheck Cond: (su_nationkey = nation.n_nationkey)
                              Heap Blocks: exact=191
                            -> Bitmap Index Scan on query2_supp (cost=0.00..11.29 rows=400 width=0) (actual time=0.083..0.083 rows=396 loops=3)
                              Index Cond: (su_nationkey = nation.n_nationkey)
                              Index Cond: (l1.ol_w_id = l1.ol_d_id) AND (l1.ol_d_id = l1.ol_o_id) AND (l1.ol_o_id = l1.ol_e_id)
                              Filter: (l1.ol_delivery > l1.ol_delivery_d)
                              Rows Removed by Filter: 7
Planning Time: 14.899 ms
JIT:
  Functions: 174
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 48.063 ms, Inlining 419.460 ms, Optimization 2558.493 ms, Emission 1924.627 ms, Total 4949.643 ms
Execution Time: 22536.731 ms
(51 rows)
```

Figura 7: *Query plan* - Interrogação analítica 2 - com utilização do índice *query2\_supp*

É possível verificar que as melhorias não foram muito significativas, no entanto, o *postgres* reconheceu o índice criado como uma possível melhoria e utilizou o mesmo na execução da *query*. Isto mostra que a adição deste índice foi uma boa estratégia, porém, a melhoria de 2 segundos não é satisfatória pelo que tentamos melhorar esta *query* através de outras otimizações.

No entanto, antes de proceder a mais otimizações, vimos que mais igualdades estavam a ser usadas, tal como as que ativaram a indexação *pk\_order\_line*, referida anteriormente,

mas que, neste caso, não utilizavam essa indexação. Posto isto decidimos desligar a opção `enable_seqscan`, para forçar o uso do índice.

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

Figura 8: Tabela de índices

Porém, e tal como seria de esperar, já que o *postgres* faz cálculos para determinar se o uso de indexação melhora o tempo de execução, e como era utilizado o *seq scan*, o tempo de execução disparou para além do 1 minuto e 30 segundos, como podemos ver na imagem abaixo. Podemos também observar que realmente foi forçado o uso do índice *pk\_order\_line* e *pk\_orders*.

```
QUERY PLAN
Sort : (cost=10020321190.53..10020321215.53 rows=10000 width=34) (actual time=101612.461..101612.523 rows=396 loops=1)
  Sort Key: (count(*)) DESC, supplier_name
  Sort Method: quicksort Memory: 55KB
-> GroupAggregate (cost=10020320251.48..10020320526.15 rows=10000 width=34) (actual time=101551.712..101612.119 rows=396 loops=1)
   Group Key: supplier.su_name
-> Sort (cost=10020320251.48..10020320309.70 rows=23289 width=26) (actual time=101551.581..101590.093 rows=229946 loops=1)
   Sort Key: supplier.su_name
   Sort Method: external merge Disk: 8128KB
-> Nested Loop Anti Join (cost=10018176211.19..10020071670.01 rows=234933 width=46) (actual time=74924.915..100881.236 rows=229946 loops=1)
   > Nested Loop
     > Join Filter: (supplier.su_nationkey = nation.n_nationkey)
     Rows Removed by Join Filter: 17166748
-> Merge Join (cost=10018176211.19..10020071670.01 rows=234933 width=46) (actual time=74921.745..91938.987 rows=17846779 loops=1)
   Merge Cond: ((orders.o_id = l1.ol_i_id) AND (orders.o_d_id = l1.ol_d_id) AND (orders.o_l_id = l1.ol_o_id))
   Join Filter: (keycustomer = keyneworder AND order_line = order_line)
-> Index Scan using pk_orders on orders (cost=0.43..1345152.80 rows=2100000 width=20) (actual time=74253.381..86611.878 rows=17846779 loops=1)
-> Materialize (cost=18176210.76..18205310.75 rows=17821197 width=54) (actual time=74253.372..84305.733 rows=17846779 loops=1)
-> Sort (cost=18176210.76..182020763.75 rows=17821197 width=54) (actual time=74253.372..84305.733 rows=17846779 loops=1)
   Sort Key: stock.s_w_id, l1.ol_d_id, l1.ol_o_id
   Sort Method: external merge Disk: 15260KB
-> Hash Join (cost=18176210.76..182020763.80 rows=17821197 width=54) (actual time=8778.588..36614.376 rows=17846779 loops=1)
   Hash Cond: (mod(stock.s_w_id * stock.s_l_id), 10000) = supplier.su_suppkey
   Hash Join (cost=18176210.76..182020763.80 rows=17821197 width=54) (actual time=8778.588..36614.376 rows=17846779 loops=1)
   Hash Cond: ((l1.ol_d_id = stock.s_w_id) AND (l1.ol_l_id = stock.s_l_id))
   > Index Scan using pk_order_line on order_line l1 (cost=0.56..12936168.85 rows=17856590 width=24) (actual time=8.122..10303.066 rows=17856590 loops=1)
   > Hash (cost=500073.43..500073.43 rows=7000000 width=8) (actual time=8717.330..8717.332 rows=7000000 loops=1)
   Buckets: 131072 Batches: 128 Memory Usage: 3143KB
-> Index Scan using ix_stock on stock (cost=0.43..500073.43 rows=7000000 width=8) (actual time=0.037..6828.920 rows=7000000 loops=1)
   > Hash (cost=13073.94..13073.94 rows=10000 width=34) (actual time=39.844..39.844 rows=10000 loops=1)
   Buckets: 16384 Batches: 1 Memory Usage: 793KB
-> Index Scan using query2_supp on supplier (cost=0.29..13873.94 rows=14000 width=24) (actual time=11.153..37.424 rows=10000 loops=1)
-> Set Operation (cost=1000000000.00..1000000000.00 rows=1 width=4) (actual time=0.000..0.000 rows=1 loops=1)
  Filter: (n.name = 'GERMANY'::bpchar)
  Rows Removed by Filter: 24
-> Index Scan using pk_order_line on order_line l2 (cost=0.56..9.37 rows=3 width=20) (actual time=0.000..0.000 rows=1 loops=680031)
  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: 52.196 ms
JIT:
  Functions: 56
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 7.881 ms, Inlining 81.993 ms, Optimization 338.941 ms, Emission 242.893 ms, Total 671.707 ms
Execution Time: 101867.704 ms
(45 rows)
```

Figura 9: *Query plan* - Interrogação analítica 2 - com utilização da opção *seq\_scan* a off

Como a maior parte das otimizações possível, em termos de índices, ou já se encontram definidas, ou não são viáveis no que diz respeito a melhorias relevantes no tempo de execução, decidimos passar para outro tipo de otimizações.

#### 4.2.2 Materialized views

Após as otimizações com indexação não terem sido muito bem sucedidas procedemos à utilização de *materialized views*.

Reparamos que a operação mais custosa era o *anti-join*, que resulta da cláusula *not exists* referente ao *select* aninhado. Com isto, e para manter a lógica da *query*, decidimos, ao invés de fazer *select* a toda a tabela de *order\_line*, criar uma *materialized view* em que a exclusão dos valores feita pela cláusula *not exists* já estivesse previamente calculada. Assim sendo, procedemos à criação da seguinte *materialized view*:

```
Create materialized view query2_mat as
  select l1.ol_o_id, l1.ol_w_id, l1.ol_d_id, l1.ol_i_id, l1.ol_delivery_d
  from order_line l1
  where not exists (
    select *
    from order_line l2
    where l2.ol_o_id = l1.ol_o_id
    and l2.ol_w_id = l1.ol_w_id
    and l2.ol_d_id = l1.ol_d_id
    and l2.ol_delivery_d > l1.ol_delivery_d));
```

Tendo agora a tabela '*query2\_mat*' com todo o trabalho do *not exists* já efectuado, procedemos à recriação da *query*, tentando manter o máximo de lógica possível da mesma. A *query* ficou da seguinte forma:

```
select su_name, count(*) as numwait
from supplier, query2_mat, orders, stock, nation
where ol_o_id = o_id
  and ol_w_id = o_w_id
  and ol_d_id = o_d_id
  and ol_w_id = s_w_id
  and ol_i_id = s_i_id
  and mod((s_w_id * s_i_id), 10000) = su_suppkey
  and ol_delivery_d > o_entry_d
  and su_nationkey = n_nationkey
  and n_name = 'GERMANY'
group by su_name
order by numwait desc, su_name;
```

Depois analisamos os tempos de execução, deparamo-nos com o seguinte cenário:

```

-----  

QUERY PLAN  

-----  

Sort  (cost=484719.38..484744.38 rows=10000 width=34) (actual time=6596.675..6652.984 rows=396 loops=1)  

  Sort Key: (cont(r)_DESC, supplier.su_name  

  Sort Method: quicksort Memory: 55KB  

-> Finalize GroupAggregate (cost=482648.52..484055.00 rows=10000 width=34) (actual time=6557.087..6652.454 rows=396 loops=1)  

  Group Key: supplier.su_name  

    -> Gather Merge (cost=482648.52..483904.43 rows=10114 width=34) (actual time=6556.947..6652.010 rows=1188 loops=1)  

      Workers Planned: 2  

      Workers Launched: 2  

      -> Partial GroupAggregate (cost=481648.50..481737.00 rows=5057 width=34) (actual time=6495.819..6533.164 rows=396 loops=3)  

        Group Key: supplier.su_name  

        -> Sort  (cost=481648.50..481661.14 rows=5057 width=26) (actual time=6495.757..6517.005 rows=76649 loops=3)  

          Sort Key: supplier.su_name  

          Sort Method: external merge Disk: 2776kB  

          Worker 0: Sort Method: external merge Disk: 2768kB  

          Worker 1: Sort Method: external merge Disk: 2768kB  

-> Nested Loop (cost=482648.50..482657.39 rows=5057 width=26) (actual time=3824.613..5829.899 rows=76649 loops=3)  

  Hash Cond: ((query2_mat.ol_w_id = stock.s_w_id) AND (query2_mat.ol_i_id = stock.r_i_id))  

-> Parallel Seq Scan on query2_mat  (cost=366010.04..366010.04 rows=17157 width=34) (actual time=2215.219..2215.225 rows=88840 loops=3)  

-> Parallel Hash  (cost=366010.04..366010.04 rows=17157 width=34) (actual time=2215.219..2215.225 rows=88840 loops=3)  

  Buckets: 65536 (originally 65536) Batches: 8 (originally 1) Memory Usage: 2912kB  

-> Hash Join (cost=260.13..366010.04 rows=17157 width=34) (actual time=56.616..2078.880 rows=88840 loops=3)  

  Hash Cond: (mod((stock.s_w_id * stock.s_l_id), 10000) = supplier.su_suppkey)  

-> Parallel Seq Scan on stock  (cost=0.00..347348.67 rows=2916667 width=8) (actual time=0.023..1190.806 rows=2333333 loops=3)  

-> Hash  (cost=259.39..259.39 rows=59 width=30) (actual time=56.509..56.513 rows=396 loops=3)  

  Buckets: 1024 Batches: 1 Memory Usage: 33kB  

-> Nested Loop (cost=11.38..259.39 rows=59 width=30) (actual time=54.994..56.368 rows=396 loops=3)  

  -> Seq Scan on nation  (cost=0.00..12.12 rows=1 width=4) (actual time=54.836..54.843 rows=1 loops=3)  

  Filter: (n_name = 'GERMANY')  

  Rows Removed by Filter: 24  

-> Bitmap Heap Scan on supplier  (cost=11.38..243.27 rows=400 width=34) (actual time=0.142..1.418 rows=396 loops=3)  

  Recheck Cond: (su_nationkey = nation.n_nationkey)  

  Heap Blocks: exact=191  

-> Bitmap Index Scan on query2_supp  (cost=0.00..11.29 rows=400 width=0) (actual time=0.089..0.089 rows=396 loops=3)  

  Index Cond: (su_nationkey = nation.n_nationkey)  

Index Cond: ((o_w_id = query2_mat.ol_w_id) AND (o_d_id = query2_mat.ol_d_id) AND (_id = query2_mat.ol_o_id))  

Filter: (query2_mat.ol_delivery_d > o_entry_d)  

Planning Time: 1.276 ms
JIT:
  Functions: 123
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 26.656 ms, Inlining 0.000 ms, Optimization 6.597 ms, Emission 155.634 ms, Total 188.887 ms
Execution Time: 6658.264 ns
(44 rows)

```

Figura 10: *Query plan* - Interrogação analítica 2 - com utilização da *materialized view* *query2\_mat*

Observa-se então uma melhoria muito significativa, tal como foi prevista. Realmente a grande demora na execução desta *query* era devida ao *anti-join*. No entanto, ainda apresenta um tempo de execução de 6 segundos. Isto deve-se aos muitos *joins* que ainda estão presentes na *query*.

Uma maneira simples de otimizar ainda mais a *query* seria colocar toda a *query* numa *materialized view*, o que seria muito simples e óbvio, e não requeria qualquer estudo da mesma, no entanto, destrói por completo a estrutura lógica da *query*.

De referir mais uma vez que poderíamos recorrer a *triggers* para atualizar a *materialized view* de maneira a que esta estivesse sempre devidamente atualizada. No caso de esta atualização ser demasiado custosa, podemos ainda fazer *refresh* desta num intervalo de tempo fixo, sabendo que a informação da *query* poderia estar desatualizada.

### 4.3 Interrogação Analítica 3 - Query 17

Para a terceira interrogação analítica seguimos com a escolha da *query* 17.

Uma vez mais a *query* precisou de algumas adaptações não só ao nível da sintaxe mas também ao nível da lógica para que os resultados da nossa *query* não fossem nulos ou vazios. A *query* ficou então definida da seguinte forma:

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

Começamos então por analisar o tempo de execução e o planeamento da *query* utilizando ***Explain Analyze***. Os resultados obtidos foram os seguintes:

```
QUERY PLAN
Aggregate (cost=819385.19..819385.20 rows=1 width=32) (actual time=8391.611..8391.973 rows=1 loops=1)
  > Hash Join (cost=342462.86..819068.39 rows=126720 width=3) (actual time=4586.075..8379.579 rows=106653 loops=1)
    Hash Cond: (order_line.ol_i_id = item.i_id)
    Join Filter: (item.i_data::text::numeric < (avg(order_line.i_ol_quantity)))
    Rows Removed by Join Filter: 160274
    > Seq Scan on order_line (cost=0.00..427889.52 rows=18862852 width=11) (actual time=0.039..1986.336 rows=18637557 loops=1)
    > Hash (cost=342437.61..342437.61 rows=2020 width=36) (actual time=4585.955..4586.314 rows=1474 loops=1)
      Buckets: 2048 Batches: 1 Memory Usage: 82k
      > Finalize GroupAggregate (cost=341900.60..342417.41 rows=2020 width=36) (actual time=4578.589..4585.881 rows=1474 loops=1)
        Group Key: item.i_id
        > Gather Merge (cost=341900.60..342371.96 rows=4040 width=36) (actual time=4578.572..4583.935 rows=4422 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          > Sort (cost=340900.57..340905.62 rows=2020 width=36) (actual time=4549.954..4550.157 rows=1474 loops=3)
            Sort Key: item.i_id
            Sort Method: quicksort Memory: 164kB
            Worker 0: Sort Method: quicksort Memory: 164kB
            Worker 1: Sort Method: quicksort Memory: 164kB
            > Partial HashAggregate (cost=340769.47..340789.67 rows=2020 width=36) (actual time=4548.771..4549.330 rows=1474 loops=3)
              Group Key: item.i_id
              > Parallel Hash Join (cost=2280.14..339977.47 rows=158400 width=8) (actual time=905.983..4481.707 rows=96976 loops=3)
                Hash Cond: (order_line.i_ol_i_id = item.i_id)
                > Parallel Seq Scan on order_line order_line_1 (cost=0.00..317056.22 rows=7859522 width=8) (actual time=0.041..1777.349 rows=6212519 loops=3)
                > Parallel Hash (cost=2274.29..2274.29 rows=1188 width=4) (actual time=905.623..905.624 rows=491 loops=3)
                  Buckets: 2048 Batches: 1 Memory Usage: 144kB
                  > Parallel Seq Scan on item (cost=0.00..2274.29 rows=1188 width=4) (actual time=733.963..735.573 rows=491 loops=3)
                    Filter: (i_data ~~ '%b%':text)
                    Rows Removed by Filter: 32842
Planning Time: 1.305 ns
JIT:
  Functions: 70
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 10.247 ms, Inlining 554.135 ms, Optimization 1137.813 ms, Emission 507.880 ms, Total 2210.075 ms
Execution Time: 8412.764 ms
(34 rows)
```

Figura 11: *Query plan* - Interrogação analítica 3 - sem otimizações

O tempo de execução é de apenas 8,6 segundo, no entanto apresenta vários setores que podem ser explorados e otimizados.

#### 4.3.1 Índices

Verificamos a falta de uso de índices na execução da *query*, uma vez que já se encontram criados muitos índices por defeito na base de dados. Por exemplo na tabela *order\_line* não há uso de índices, no entanto, sabemos pelas análises anteriores, da existência de índices nessa tabela, nomeadamente na coluna *ol\_i\_id* (referente ao índice *ix\_order\_line*) que é usada na nossa *query*.

Verificamos ainda que havia a possibilidade de otimizar a filtragem do campo *i\_data* criando um índice desta coluna na tabela *item*. Foi então esta a primeira abordagem a ser tomada. O índice criado foi o seguinte:

*Create index query3\_item on item(i\_data);*

Os resultados obtidos foram:

```
QUERY PLAN
Aggregate (cost=819385.19..819385.20 rows=1 width=32) (actual time=8370.490..8370.775 rows=1 loops=1)
  > Hash Join (cost=342462.86..819668.39 rows=126720 width=3) (actual time=4551.575..8358.289 rows=106653 loops=1)
    Hash Cond: (order_line.ol_i_id = item.i_id)
    Join Filter: ((order_line.ol_quantity)::numeric < (avg(order_line.iol_quantity)))
  Rows Removed by Join Filter: 166274
  -> Seq Scan on order_line (cost=0.00..427089.52 rows=18862852 width=11) (actual time=0.039..1999.343 rows=18637557 loops=1)
  -> Hash (cost=342437.61..342437.61 rows=2020 width=36) (actual time=4551.484..4551.766 rows=1474 loops=1)
    Buckets: 2048 Batches: 1 Memory Usage: 82KB
    -> Finalize GroupAggregate (cost=341900.60..342417.41 rows=2020 width=36) (actual time=4542.941..4551.361 rows=1474 loops=1)
      Group Key: item.i_id
      -> Gather Merge (cost=341900.60..342371.96 rows=4040 width=36) (actual time=4542.926..4549.570 rows=4422 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Sort (cost=340900.57..340905.62 rows=2020 width=36) (actual time=4509.124..4509.326 rows=1474 loops=3)
          Sort Key: item.i_id
          Sort Method: quicksort Memory: 164KB
          Worker 0: Sort Method: quicksort Memory: 164KB
          Worker 1: Sort Method: quicksort Memory: 164KB
          -> Partial HashAggregate (cost=340769.47..340789.67 rows=2020 width=36) (actual time=4507.959..4508.538 rows=1474 loops=3)
            Group Key: item.i_id
            -> Parallel Hash Join (cost=2389.14..330977.47 rows=158400 width=8) (actual time=855.739..4425.737 rows=90976 loops=3)
              Hash Cond: (order_line.ol_i_id = item.i_id)
              -> Parallel Seq Scan on order_line order_line_1 (cost=0.00..317056.22 rows=7859522 width=8) (actual time=0..340..1760.841 rows=6212519 loops=3)
              -> Parallel Hash (cost=2274.29..2274.29 rows=188 width=4) (actual time=854.996..854.997 rows=491 loops=3)
                Buckets: 2048 Batches: 1 Memory Usage: 144KB
                -> Parallel Seq Scan on item (cost=0.00..2274.29 rows=1188 width=4) (actual time=722.726..737.861 rows=491 loops=3)
                  Filter: (i_data ~~ '%::text')
                  Rows Removed by Filter: 32842
Planning Time: 0.802 ms
JIT:
  Functions: 70
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 10.987 ms, Inlining 399.191 ms, Optimization 1283.309 ms, Emission 483.671 ms, Total 2177.158 ms
Execution Time: 8375.139 ms
(34 rows)
```

Figura 12: *Query plan* - Interrogação analítica 3 - sem recorrer à utilização do índice *query3\_item* criado

Como podemos ver, não houve qualquer alteração na execução o que nos deixou um pouco insatisfeitos. Reparando que o uso de índices persistia inexistente decidimos alterar o ficheiro *postgresql.conf* colocando o campo *enable\_seqscan* a *off*. Com isto pretendemos impedir o uso de *sequential scan* para forçar o uso de índices, não só o da nossa autoria, como os já predefinidos, com o intuito de verificar algumas melhorias. Os resultados obtidos foram os seguintes:

```
QUERY PLAN
Aggregate (cost=1364730.90..1364730.92 rows=1 width=32) (actual time=2738.555..2738.655 rows=1 loops=1)
  > Nested Loop (cost=1012.16..1364414.10 rows=126720 width=3) (actual time=521.881..2713.254 rows=106653 loops=1)
    -> Finalize GroupAggregate (cost=1006.25..509479.29 rows=2020 width=36) (actual time=21.441..650.419 rows=1474 loops=1)
      Group Key: item.i_id
      -> Gather Merge (cost=1006.25..509443.94 rows=2020 width=36) (actual time=520.094..644.493 rows=1474 loops=1)
        Workers Planned: 1
        Workers Launched: 1
        -> Partial GroupAggregate (cost=6.24..508216.08 rows=2020 width=36) (actual time=448.263..1611.673 rows=737 loops=2)
          Group Key: item.i_id
          -> Nested Loop (cost=6.24..507078.36 rows=223623 width=8) (actual time=446.418..1586.900 rows=136464 loops=2)
            -> Parallel Index Scan using pk_item on item (cost=0.29..3881.59 rows=1188 width=4) (actual time=446.272..469.721 rows=737 loops=2)
              Filter: (i_data ~~ '%::text')
              Rows Removed by Filter: 49263
            -> Bitmap Heap Scan on order_line order_line_1 (cost=5.95..421.69 rows=188 width=8) (actual time=0.068..1.465 rows=185 loops=1474)
              Recheck Cond: (ol_i_id = item.i_id)
              Heap Blocks: exact=14838
              -> Bitmap Index Scan on ix_order_line (cost=0.00..5.90 rows=188 width=0) (actual time=0.037..0.037 rows=185 loops=1474)
                Index Cond: (ol_i_id = item.i_id)
  -> Bitmap Heap Scan on order_line (cost=5.91..442.60 rows=63 width=11) (actual time=0.833..1.380 rows=72 loops=1474)
    Recheck Cond: (ol_i_id = item.i_id)
    Filter: ((ol_quantity)::numeric < (avg(order_line.iol_quantity)))
    Rows Removed by Filter: 113
    Heap Blocks: exact=272794
    -> Bitmap Index Scan on ix_order_line (cost=0.00..5.90 rows=188 width=0) (actual time=0.035..0.035 rows=185 loops=1474)
      Index Cond: (ol_i_id = item.i_id)
Planning Time: 0.548 ms
JIT:
  Functions: 34
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 6.023 ms, Inlining 150.498 ms, Optimization 456.194 ms, Emission 284.345 ms, Total 897.061 ms
Execution Time: 2742.418 ms
(31 rows)
```

Figura 13: *Query plan* - Interrogação analítica 3 - com a opção *seq-scan* a *off*

Como podemos ver, os resultados são bastante animadores, apesar do índice criado pelo grupo não ser usado, o uso dos índices predefinidos pelo tpcc, nomeadamente o

índice *ix\_order\_line*, melhoraram significativamente o tempo de execução, que se encontra agora em 2,7 segundos. Isto mostra que realmente a indexação é uma grande melhoria na execução desta *query*.

#### 4.3.2 Materialized views

Apesar de já termos alcançado resultados muito animadores com apenas a alteração de uma configuração do *postgresql* acreditamos que conseguimos ainda atingir resultados mais satisfatórios.

Com a análise do código da *query* conseguimos verificar que esta apresenta um *select* aninhado tratando-se assim de uma *query* composta por uma *subquery* independente. O grupo decidiu criar uma *materialized view* com o resultado do *select* aninhado com o intuito de aliviar trabalho à *query* principal, não tirando lógica à composição da mesma. De seguida apresenta-se a criação dessa *materialized view*:

```
create materialized view query3_mat as (
    select i.id, avg(ol.quantity) as a
    from item, order_line
    where i_data like 'b%'
        and ol_i_id = i_id
    group by i_id);
```

A *query* ficou transformada ficando da seguinte forma:

```
select sum(ol_amount) / 2.0 as avg_yearly
from order_line, query3_mat t
where ol_i_id = t.i_id
    and ol_quantity < t.a;
```

Testamos então com o ambiente original, isto é com o *seq scan* a *on* e os resultados obtidos foram:

```
QUERY PLAN
Finalize Aggregate (cost=349401.91..349401.92 rows=1 width=32) (actual time=4327.162..4331.612 rows=1 loops=1)
  -> Gather (cost=349401.69..349401.90 rows=2 width=32) (actual time=4326.822..4331.575 rows=3 loops=1)
      Workers Planned: 2
      Workers Launched: 2
      -> Partial Aggregate (cost=348401.69..348401.70 rows=1 width=32) (actual time=4296.419..4296.422 rows=1 loops=3)
          -> Hash Join (cost=42.17..348305.36 rows=38528 width=3) (actual time=33.425..4281.625 rows=35551 loops=3)
              Hash Cond: (order_line.ol_i_id = t.i_id)
              Join Filter: ((order_line.ol_quantity)::numeric < t.a)
              Rows Removed by Join Filter: 55425
              -> Parallel Seq Scan on order_line (cost=0.00..317056.22 rows=7859522 width=11) (actual time=0.042..1873.137 rows=6212519 loops=3)
                  -> Hash (cost=23.74..23.74 rows=1474 width=13) (actual time=32.984..32.985 rows=1474 loops=3)
                      Buckets: 2048  Batches: 1  Memory Usage: 82kB
                      -> Seq Scan on query3_mat t (cost=0.00..23.74 rows=1474 width=13) (actual time=0.020..0.292 rows=1474 loops=3)
Planning Time: 1.044 ms
JIT:
  Functions: 44
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 19.611 ms, Inlining 0.000 ms, Optimization 5.213 ms, Emission 82.408 ms, Total 107.232 ms
Execution Time: 4351.454 ms
(19 rows)
```

Figura 14: *Query plan* - Interrogação analítica 3 - com utilização da *materialized view* *query3\_mat*

Como podemos ver temos uma melhoria de 50% em relação ao tempo original, mas houve uma perda de performance em relação ao uso dos índices provocados pela desativação do *seq scan*.

	Mat. View Seq. Scan	Without	With
On		8,6s	4,3s
Off		2,7s	1,6s

Tabela 3: Tempos de execução com e sem *materialized view* e *seq\_scan* a *on* e *off*

Deste modo o grupo decidiu combinar as duas otimizações (**uso de *materialized view* e *enable\_seqscan* a *off***) com o objetivo de obter um melhor resultado.

```
QUERY PLAN
-----
Aggregate (cost=10000704652.46..10000704652.47 rows=1 width=32) (actual time=1635.762..1635.764 rows=1 loops=1)
  -> Nested Loop (cost=10000000005.94..10000704421.29 rows=92468 width=3) (actual time=124.244..1623.270 rows=106653 loops=1)
    -> Seq Scan on query3_mat t (cost=10000000000.00..10000000023.74 rows=1474 width=13) (actual time=0.019..0.802 rows=1474 loops=1)
    -> Bitmap Heap Scan on order_line (cost=5.94..477.25 rows=63 width=11) (actual time=0.691..1.090 rows=72 loops=1474)
      Recheck Cond: (ol_i_id = t.i_id)
      Filter: ((ol_quantity)::numeric < t.a)
      Rows Removed by Filter: 113
      Heap Blocks: exact=272794
      -> Bitmap Index Scan on ix_order_line (cost=0.00..5.92 rows=188 width=0) (actual time=0.024..0.024 rows=185 loops=1474)
        Index Cond: (ol_i_id = t.i_id)
Planning Time: 0.333 ms
JIT:
  Functions: 9
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 1.323 ms, Inlining 14.446 ms, Optimization 67.298 ms, Emission 41.085 ms, Total 124.151 ms
Execution Time: 1637.185 ms
(16 rows)
```

Figura 15: *Query plan* - Interrogação analítica 3 - com utilização da *materialized view* *query3\_mat* e *seq\_scan* a *off*

Como era esperado este foi o melhor resultado obtido até agora, já que combina dois tipos de otimizações, forçando os índices e uso de *materialized views*.

Apresentamos de seguida uma análise comparativa dos tempo de execução estudados.

Para esta análise, o grupo encontra-se satisfeito com as melhorias efetuadas, no entanto, é importante referir que o uso de *materialized view* pode gerar resultados desatualizados, logo pensamos que melhor opção seria apenas o uso forçado da indexação, ou seja desligar o *seq scan*. Apesar de representar um valor temporal um pouco mais elevado, temos a certeza que os resultados obtidos estariam garantidamente atualizados, o que, a nosso ver, compensa o segundo extra de espera.

## 4.4 Interrogação Analítica 4 - Query 18

Para esta última interrogação, escolhemos a *query* 18. Esta escolha deveu-se ao seu elevado tempo de execução (71s) que nos motivou para o tentar diminuir.

```
select c_last, c_id o_id, o_entry_d, o.ol_cnt, sum(ol_amount)
from customer, orders, order_line
where c_id = o_c_id
    and c_w_id = o_w_id
    and c_d_id = o_d_id
    and ol_w_id = o_w_id
    and ol_d_id = o_d_id
    and ol_o_id = o_id
group by o_id, o_w_id, o_d_id, c_id, c_last, o_entry_d, o.ol_cnt
having sum(ol_amount) > 200
order by sum(ol_amount) desc, o_entry_d;
```

Executando a *query* com o ***Explain Analyze*** conseguimos ver a *query plan*, e analisar onde podemos fazer otimizações.

```
QUERY PLAN
*sort  (cost=7362405.79..7378861.07 rows=6582114 width=77) (actual time=7189.551..71806.241 rows=630000 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC, orders.o_entry_d
  Sort Method: external merge Disk: 40760KB
-> Finalize GroupAggregate (cost=3046646.62..6032843.44 rows=582114 width=77) (actual time=65421.444..70543.277 rows=630000 loops=1)
  Group Key: orders.o_id, orders.o_w_id, orders.o_d_id, customer.c_id, customer.c_last, orders.o_entry_d, orders.o.ol_cnt
  Filter: (sum(order_line.ol_amount) > '200'::numeric)
  Rows Removed by Filter: 1548068
-> Gather Merge (cost=3046646.62..5233962.16 rows=16455284 width=77) (actual time=54845.322..66919.011 rows=2178068 loops=1)
  Workers Planned: 2
  Workers Launched: 2
-> Partial GroupAggregate (cost=3045646.68..3333614.07 rows=8227642 width=77) (actual time=54553.897..64128.582 rows=726023 loops=3)
  Group Key: orders.o_id, orders.o_w_id, orders.o_d_id, customer.c_id, customer.c_last, orders.o_entry_d, orders.o.ol_cnt
  -> Sort (cost=3045646.68..3866215.78 rows=8227642 width=48) (actual time=54553..828..60213.513 rows=6212519 loops=3)
    Sort Key: orders.o_id, orders.o_w_id, orders.o_d_id, customer.c_id, customer.c_last, orders.o_entry_d, orders.o.ol_cnt
    Sort Method: external merge Disk: 367200KB
    Worker 0: Sort Method: external merge Disk: 366720KB
    Worker 1: Sort Method: external merge Disk: 385336KB
    -> Nested Loop (cost=173355.68..1594419.68 rows=8227642 width=48) (actual time=3428.079..25981.932 rows=6212519 loops=3)
      Join Filter: ((customer.c_w_id = order_line.ol_w_id) AND (customer.c_d_id = order_line.ol_d_id))
      -> Parallel Hash Join (cost=173355.04..228517.90 rows=926595 width=53) (actual time=3427.941..4442.491 rows=726023 loops=3)
        Hash Cond: ((orders.o_c_id = customer.c_id) AND (orders.o_w_id = customer.c_w_id) AND (orders.o_d_id = customer.c_d_id))
        -> Parallel Seq Scan on orders (cost=0.00..29478.71 rows=908971 width=28) (actual time=0.129..263.522 rows=726023 loops=3)
        -> Parallel Hash (cost=151646.38..151646.38 rows=892038 width=29) (actual time=2643.314..2643.315 rows=700000 loops=3)
          Buckets: 65536 Batches: 64 Memory Usage: 2624KB
          -> Parallel Seq Scan on customer (cost=0.00..151646.38 rows=892038 width=29) (actual time=1102.375..2102.732 rows=700000 loops=3)
-> Index Scan using pk_order_line on order_line (cost=0.56..1.34 rows=9 width=15) (actual time=0.021..0.026 rows=9 loops=2178068)
  Index Cond: ((ol_w_id = orders.o_w_id) AND (ol_d_id = orders.o_d_id) AND (ol_o_id = orders.o_id))

Planning Time: 1.280 ms
JIT:
  Functions: 88
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 31.858 ms, Inlining 325.775 ms, Optimization 1758.109 ms, Emission 1220.086 ms, Total 3335.828 ms
Execution Time: 71922.690 ms
(33 rows)
```

Figura 16: *Query plan* - Interrogação analítica 4 - sem otimizações

### 4.4.1 Índices

Reparamos que apesar de alguns dos índices *default* serem usados, como por exemplo *pk\_order\_line*, nem todas as vezes estes substituem os *Seq scans*. Claro que, como já vimos nas interrogações analíticas anteriores, por vezes o uso dos índices em detrimento de *Seq Scan* apenas piora o tempo de execução, mas também já vimos, como por exemplo na interrogação analítica 3, que forçar o uso de índices pode trazer melhorias. Logo, começamos por desligar o *Seq Scan* e obtivemos o seguinte resultado.

```

-----+-----+-----+-----+
-----| QUERY PLAN |-----+
-----+-----+-----+-----+
Sort  (cost=7660950.20..7677405.49 rows=6582114 width=77) (actual time=62366.111..62711.710 rows=630000 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC, orders.o_entry_d
  Sort Method: external merge Disk: 40760KB
    > Finalize GroupAggregate (cost=3345191.04..6330587.86 rows=6582114 width=77) (actual time=56357.178..61075.127 rows=630000 loops=1)
      Group Key: orders.o_id, orders.o_w_id, orders.o_d_id, customer.c_id, customer.c_last, orders.o_entry_d, orders.o.ol_cnt
      Filter: (sum(order_line.ol_amount) > '200'::numeric)
      Rows Removed by Filter: 1548868
        > Gather Merge (cost=3345191.04..5532506.58 rows=16455284 width=77) (actual time=46415.143..57496.639 rows=2178068 loops=1)
          Workers Planned: 2
          Workers Launched: 2
            > Partial GroupAggregate (cost=3344191.82..3632158.49 rows=8227642 width=77) (actual time=46219.770..55054.996 rows=726023 loops=3)
              Group Key: orders.o_id, orders.o_w_id, orders.o_d_id, customer.c_id, customer.c_last, orders.o_entry_d, orders.o.ol_cnt
              > Sort  (cost=3344191.02..3364760.12 rows=8227642 width=48) (actual time=46219.634..51105.444 rows=6212519 loops=3)
                Sort Key: orders.o_id, orders.o_w_id, orders.o_d_id, customer.c_id, customer.c_last, orders.o_entry_d, orders.o.ol_cnt
                Sort Method: external merge Disk: 370752KB
                Worker 0: Sort Method: external merge Disk: 374728KB
                Worker 1: Sort Method: external merge Disk: 373768KB
                > Nested Loop (cost=1.42..1892964.10 rows=8227642 width=48) (actual time=1074.212..16059.151 rows=6212519 loops=3)
                  Join Filter: ((customer.c_w_id = order_line.ol_w_id) AND (customer.c_d_id = order_line.ol_d_id))
                  > Merge Join (cost=0.86..527062.32 rows=926595 width=53) (actual time=1074.122..4567.201 rows=726023 loops=3)
                    Merge 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))
                    > Parallel Index Scan using ix_orders on orders (cost=0.43..100397.05 rows=908971 width=28) (actual time=0.055..557.111 rows=726023 loops=3)
                    > Index Scan using pk_customer on customer (cost=0.43..394525.37 rows=2140889 width=29) (actual time=0.076..1837.000 rows=2099743 loops=3)
                  > Index Scan using pk_order_line on order_line (cost=0.56..1.34 rows=9 width=15) (actual time=0.068..0.012 rows=9 loops=2178068)
                  Index Cond: ((ol_w_id = orders.o_w_id) AND (ol_d_id = orders.o_d_id) AND (ol_c_id = orders.o_c_id))
Planning Time: 1.008 ms
 JIT:
  Functions: 91
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 23.027 ms, Inlining 357.567 ms, Optimization 1766.186 ms, Emission 1094.682 ms, Total 3241.461 ms
Execution Time: 62827.709 ms
(31 rows)

```

Figura 17: *Query plan* - Interrogação analítica 4 - com a opção *seq-scan* a off

De facto como podemos ver, houve uma melhoria em volta dos 9 segundos. Achamos que uma das razões que levou a esta melhoria foi a utilização do índice *ix\_orders* na tabela *orders* e do *pk\_customer* da tabela *customer*. No entanto, também podemos ver que a alteração, automática, no uso de *Parallel Hash Join* para *Merge Join* trouxe um ligeiro agravamento do tempo de execução de 1s para 3s. Este último diz respeito ao *Join* que ocorre entre a tabela *customer* e a tabela *orders* → (*c\_id = o\_c\_id and c\_w\_id = o\_w\_id and c\_d\_id = o\_d\_id*).

A melhoria mais acentuada diz respeito à execução total do *nested loop* que faz a junção de todas as tabelas. Como agora é usada indexação para percorrer as 3 tabelas (o que antigamente apenas estava a ocorrer numa delas), observou-se uma melhoria significativa, passando de um tempo total de 25 segundos para apenas 16 neste passo da execução, daí os 9 segundo de melhoria.

Porém, estes resultados não são, de todo, satisfatórios no que diz respeito à generalidade da *query*, por isso continuamos à procura de possíveis otimizações.

#### 4.4.2 Materialized Views + Índices

De modo a conseguirmos obter resultados mais satisfatórios, fomos à procura de tempos mais baixos usando *Materialized views*. Durante a análise anterior reparámos que parte do tempo de execução era dedicado ao processamento dos *Joins* das 3 tabelas que são utilizadas nesta *query*. Para tentar manter a lógica da mesma, mas ao mesmo tempo otimizar os *joins* começámos por criar uma *materialized view* que tratasse da junção entre a tabela *customers* e a tabela *orders*.

```

create materialized view query4_custord as (
  select c_last,c_id,o_id,o_entry_d,o.ol_cnt,o.w_id,o.d_id
  from customer, orders
  where c_id = o.c_id
    and c.w_id = o.w_id
    and c.d_id = o.d_id);

```

À *materialized view*, *query4\_custord*, manteve-se o *join* à tabela *order\_line* e, portanto, a *query* passou a tomar o seguinte aspetto:

```

select c_last, c_id, o_id, o_entry_d, o.ol_cnt, sum(ol_amount)
from query4_custord, order_line where
    ol_w_id = o_w_id
    and ol_d_id = o_d_id
    and ol_o_id = o_id
group by o_id, o_w_id, o_d_id, c_id, c_last, o_entry_d, o.ol_cnt
having sum(ol_amount) > 200
order by sum(ol_amount) desc, o_entry_d;

```

Ao executar a *query* anterior com o ***explain analyze*** conseguimos ver que ouve uma ligeiro melhoria em relação ao tempo de execução da *query* original, que era cerca de 71,9 s, passando agora para 65 s. No entanto, comparando com a versão onde tínhamos forçado o uso de índices, este tempo aumentou. Isto deve-se ao uso do *seq scan* quando é necessário percorrer as colunas da tabela da *materialized view*, provocando esse aumento. Anteriormente eram usados índices das tabelas *customer* e *orders*, que já não estão a ser usados, por se encontrarem agora referentes à *materialized view*.

```

QUERY PLAN
Sort  (cost=10006324228.04..10006324541.58 rows=125417 width=77) (actual time=65095.102..65459.728 rows=630000 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC, query4_custord.o_entry_d
  Sort Method: external merge Disk: 43168KB
-> GroupAggregate (cost=10005837475.44..10006308031.99 rows=125417 width=77) (actual time=60370.791..64464.853 rows=630000 loops=1)
  Group Key: query4_custord.o_id, query4_custord.o_w_id, query4_custord.o_d_id, query4_custord.c_id, query4_custord.c_last, query4_custord.o_entry_d, query4_custord.o.ol_cnt
  Filter: (sum(order_line.ol_amount) > '200'::numeric)
  Rows Removed by Filter: 1548068
-> Sort (cost=10005837475.44..100058883872.60 rows=18558887 width=48) (actual time=52270.217..59506.579 rows=18637557 loops=1)
  Sort Key: query4_custord.o_id, query4_custord.o_w_id, query4_custord.o_d_id, query4_custord.c_id, query4_custord.c_last, query4_custord.o_entry_d, query4_custord.o.ol_cnt
  Sort Method: external merge Disk: 1189024KB
-> Merge Join (cost=10000407430.18..10000407479.50 rows=18558887 width=48) (actual time=5699.894..14115.350 rows=18637557 loops=1)
  Merge Cond: ((query4_custord.o_w_id = order_line.ol_w_id) AND (query4_custord.o_d_id = order_line.ol_d_id) AND (query4_custord.o_id = order_line.ol_o_id))
-> Sort (cost=10000407429.62..10000412874.79 rows=2178668 width=45) (actual time=5390.736..5837.459 rows=2178668 loops=1)
  Sort Key: query4_custord.o_w_id, query4_custord.o_d_id, query4_custord.o_id
  Sort Method: external merge Disk: 131272KB
-> Seq Scan on query4_custord (cost=100000000000.00..10000044129.68 rows=2178668 width=45) (actual time=0.044..247.451 rows=2178668 loops=1)
Planning Time: 0.342 ms
JIT:
  Functions: 21
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 2.889 ms, Inlining 11.130 ms, Optimization 181.409 ms, Emission 116.161 ms, Total 311.590 ms
Execution Time: 65896.940 ms
(23 rows)

```

Figura 18: *Query plan* - Interrogação analítica 4 - com utilização da *materialized view* *query4\_custord*

Para conseguirmos manter as otimizações ganhas, usando índices, temos então de os voltar a criar para as colunas da tabela da *materialized view*. Então procedemos à criação do seguinte índice.

```
Create index matindex on query4_custord(o_id,c_id,o_w_id,o_d_id);
```

Voltando a executar a *query* obtivemos, como seria de esperar, uma melhoria significativa, que se deve, e como podemos observar no *query plan*, ao uso do índice criado anteriormente. Com esta otimização conseguimos uma melhoria a rondar os 30% do valor original da *query*.

```

-----+-----+-----+-----+
-----| QUERY PLAN |-----+
-----+-----+-----+-----+
Sort  (cost=3111220.63 ..3111534.18 rows=125417 width=77) (actual time=47178.432..47547.331 rows=630000 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC, query4_custord.o_entry_d
  Sort Method: external merge  Disk: 3108K
    >  Finalize GroupAggregate (cost=2802196.59 ..3095024.59 rows=125417 width=77) (actual time=41656.869..46164.590 rows=630000 loops=1)
      Group Key: query4_custord.o_id, query4_custord.o_w_id, query4_custord.o_d_id, query4_custord.c_id, query4_custord.c_last, query4_custord.o_entry_d, query4_custord.o.ol_cnt
      Filter: (sum(order_line.ol_amount) > '200'::numeric)
      Rows Removed by Filter: 1548068
    ->  Gather Merge (cost=2802196.59 ..3067746.46 rows=752500 width=77) (actual time=32187.710..42651.870 rows=2178068 loops=1)
        Workers Planned: 2
        Workers Launched: 2
      ->  Partial GroupAggregate (cost=2801190.57 ..2979889.27 rows=376250 width=77) (actual time=20061.644..38602.073 rows=726023 loops=3)
          Group Key: query4_custord.o_id, query4_custord.o_w_id, query4_custord.o_d_id, query4_custord.c_id, query4_custord.c_last, query4_custord.o_entry_d, query4_custord.o.ol_cnt
          Sort Method: external merge  Disk: 399708KB
            >  Sort (cost=2801190.57 ..2828528.74 rows=7732870 width=48) (actual time=32061.562..34306.320 rows=6212519 loops=3)
              Sort Key: query4_custord.o_id, query4_custord.o_w_id, query4_custord.o_d_id, query4_custord.c_id, query4_custord.c_last, query4_custord.o_entry_d, query4_custord.o.ol_cnt
              Sort Method: external merge  Disk: 399708KB
              Worker 0: Sort Method: external merge  Disk: 399928KB
              Worker 1: Sort Method: external merge  Disk: 398408KB
            ->  Nested Loop (cost=0.99 ..1440701.77 rows=7732870 width=48) (actual time=658.272..15563.313 rows=6212519 loops=3)
              ->  Parallel Index Scan using MatIndex on query4_custord (cost=0.43 ..1440717.19 rows=907528 width=45) (actual time=0.041..1336.398 rows=726023 loops=3)
                Index Cond: ((ol_w_id = query4_custord.o_w_id) AND (ol_d_id = query4_custord.o_d_id) AND (ol_o_id = query4_custord.o_id))
Planning Time: 0.564 ms
JIT:
  Functions: 43
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 9.655 ms, Inlining 441.222 ms, Optimization 905.901 ms, Emission 625.785 ms, Total 1982.563 ms
Execution Time: 47666.124 ms
(27 rows)

```

Figura 19: *Query plan* - Interrogação analítica 4 - com utilização do índice *matindex*

Com os resultados obtidos anteriormente, decidimos utilizar o segundo *join* da *query* para criar uma *materialized view*, junção das tabelas *order\_line* com *orders*. Pensamos que seria uma melhoria, uma vez que a tabela *order\_line* é mais extensa, o que pouparia algum tempo. A *materialized view* criada foi a seguinte:

```

create materialized view query4_ordol as (
  select ol_amount, o_id, o_w_id, o_d_id, o_entry_d, o.ol_cnt, o_c_id
  from order_line, orders
  where ol_w_id = o_w_id
    and ol_d_id = o_d_id
    and ol_o_id = o_id);

```

Estando então as tabelas *order\_line* e *orders* já juntas na *materialized view*, a *query* fica definida da seguinte forma:

```

select c_last, c_id o_id, o_entry_d, o.ol_cnt, sum(ol_amount)
from customer, query4_ordol
where c_id = o_c_id
  and c_w_id = o_w_id
  and c_d_id = o_d_id
group by o_id, o_w_id, o_d_id, c_id, c_last, o_entry_d, o.ol_cnt
having sum(ol_amount) > 200
order by sum(ol_amount) desc, o_entry_d;

```

Os resultados obtidos foram os presentes na seguinte imagem:

```

QUERY PLAN
Sort (cost=10007332338.19..10007348182.85 rows=6337864 width=77) (actual time=67396.219..67755.268 rows=630000 loops=1)
  Sort Key: (sum(query4_ordol.ol_amount)) DESC, query4_ordol.o_entry_d
  Sort Method: external merge Disk: 40760kB
-> GroupAggregate (cost=10005244988.39..10006053066.09 rows=6337864 width=77) (actual time=62247.761..66730.056 rows=630000 loops=1)
  Group Key: query4_ordol.o_id, query4_ordol.o_w_id, query4_ordol.o_d_id, customer.c_id, customer.c_last, query4_ordol.o_entry_d, query4_ordol.o.ol_cnt
  Filter: (sum(query4_ordol.ol_amount) > '200':numeric)
  Rows Removed by Filter: 1548068
-> Sort (cost=10005244988.39..10005292522.37 rows=19013593 width=48) (actual time=53484.847..61704.353 rows=18637557 loops=1)
  Sort Key: query4_ordol.o_id, query4_ordol.o_w_id, query4_ordol.o_d_id, customer.c_id, customer.c.last, query4_ordol.o_entry_d, query4_ordol.o.ol_cnt
  Sort Method: external merge Disk: 1119136kB
-> Hash Join (cost=10000446625.95..10001191518.81 rows=19013593 width=48) (actual time=2132.708..17145.289 rows=18637557 loops=1)
  Hash Cond: ((query4_ordol.o_c_id = customer.c_id) AND (query4_ordol.o_w_id = customer.c_w_id) AND (query4_ordol.o_d_id = customer.c_d_id))
    -> Seq Scan on query4_ordol (cost=10000000000.00..10000328674.36 rows=18637636 width=31) (actual time=27.362..4073.568 rows=18637557 loops=1)
    -> Hash (cost=394525.37..394525.37 rows=2104890 width=29) (actual time=2104.499..2104.502 rows=2100000 loops=1)
      Buckets: 65536  Batches: 64  Memory Usage: 2564kB
        -> Index Scan using pk_customer on customer (cost=0.43..394525.37 rows=2104890 width=29) (actual time=322.609..1518.910 rows=2100000 loops=1)
Planning Time: 1.828 ms
JIT:
  Functions: 20
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 3.389 ms, Inlining 12.884 ms, Optimization 182.973 ms, Emission 125.880 ms, Total 325.126 ms
Execution Time: 68969.015 ms

```

Figura 20: *Query plan* - Interrogação analítica 4 - com utilização da *materialized view* *query4\_ordol*

Mais uma vez a inexistência de índices na *materialized view* gera um resultado insatisfatório forçando o uso de *sequential scan*. Assim sendo repetimos o processo, anteriormente seguido, criando o seguinte índice:

*Create index matindexq4 on query4\_ordol(o\_c\_id,o\_w\_id,o\_d\_id);*

Os resultados agora foram os seguintes:

```

QUERY PLAN
Sort (cost=7851719.14..7067563.73 rows=6337837 width=77) (actual time=60687.387..61038.770 rows=630000 loops=1)
  Sort Key: (sum(query4_ordol.ol_amount)) DESC, query4_ordol.o_entry_d
  Sort Method: external merge Disk: 40760kB
-> Finalize GroupAggregate (cost=2897855.14..5772457.29 rows=6337837 width=77) (actual time=54985.873..59768.222 rows=630000 loops=1)
  Group Key: query4_ordol.o_id, query4_ordol.o_w_id, query4_ordol.o_d_id, customer.c_id, customer.c.last, query4_ordol.o_entry_d, query4_ordol.o.ol_cnt
  Filter: (sum(query4_ordol.ol_amount) > '200':numeric)
  Rows Removed by Filter: 1548068
-> Gather Merge (cost=2897855.14..5003994.57 rows=15844592 width=77) (actual time=44998.910..56230.233 rows=2178068 loops=1)
  Workers Planned: 2
  Workers Launched: 2
    -> Partial GroupAggregate (cost=2896855.12..3174135.48 rows=7922296 width=77) (actual time=44853.920..53869.859 rows=726023 loops=3)
      Group Key: query4_ordol.o_id, query4_ordol.o_w_id, query4_ordol.o_d_id, customer.c_id, customer.c.last, query4_ordol.o_entry_d, query4_ordol.o.ol_cnt
      Filter: (sum(query4_ordol.ol_amount) > '200':numeric)
      Rows Removed by Filter: 1548068
      -> Sort (cost=2896855.12..2916660.86 rows=7922296 width=48) (actual time=44853.751..49793.026 rows=6212519 loops=3)
        Sort Key: query4_ordol.o_id, query4_ordol.o_w_id, query4_ordol.o_d_id, customer.c_id, customer.c.last, query4_ordol.o_entry_d, query4_ordol.o.ol_cnt
        Sort Method: external merge Disk: 381560kB
        Worker 0: Sort Method: external merge Disk: 377632kB
        Worker 1: Sort Method: external merge Disk: 366048kB
        -> Nested Loop (cost=0.99..1501649.54 rows=7922296 width=48) (actual time=660.246..14479.821 rows=6212519 loops=3)
          -> Parallel Index Scan using pk_customer on customer (cost=0.43..382036.85 rows=892638 width=29) (actual time=0.049..879.014 rows=700000 loops=3)
            -> Index Scan using matindexq4 on query4_ordol (cost=0.56..11.17 rows=9 width=31) (actual time=0.010..0.015 rows=9 loops=2100000)
              Index Cond: ((o_c_id = customer.c_id) AND (o_w_id = customer.c_w_id) AND (o_d_id = customer.c_d_id))
Planning Time: 0.566 ms
JIT:
  Functions: 43
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 23.134 ms, Inlining 364.122 ms, Optimization 1045.846 ms, Emission 568.361 ms, Total 2001.463 ms
Execution Time: 61162.955 ms
(27 rows)

```

Figura 21: *Query plan* - Interrogação analítica 4 - com utilização do índice *matindexq4*

Deparamo-nos com um cenário em que esta *materialized view* em nada melhora a *query* em relação à abordagem adotada anteriormente. Isto deve-se ao facto de a mesma ser muito extensa, provocando um elevado trabalho ao servidor para fazer a junção da *materialized view*, muito extensa, com a tabela *customers*.

Com este cenário, o grupo decidiu proceder à criação de uma *materialized view* em que as três tabelas já estivessem previamente juntas, com o objetivo de deixar para a *query* apenas o trabalho do somatório, do agrupamento e da ordenação do resultado. Desta maneira a lógica da *query* continua lá, apenas alterando a consulta das três tabelas para apenas uma.

A *materialized view* foi criada da seguinte forma:

```

create materialized view matq4alljoins as (
    select c_last, c_id, o_id, o_entry_d, o.ol_cnt, ol_amount, o_w_id, o_d_id
    from customer, orders, order_line
    where c_id = o_c_id
        and c_w_id = o_w_id
        and c_d_id = o_d_id
        and ol_w_id = o_w_id
        and ol_d_id = o_d_id
        and ol_o_id = o_id);

```

Tendo a *query* como versão final a seguinte forma:

```

select c_last, c_id, o_id, o_entry_d, o.ol_cnt, sum(ol_amount)
from matq4alljoins
group by o_id, o_w_id, o_d_id, c_id, c_last, o_entry_d, o.ol_cnt
having sum(ol_amount) > 200
order by sum(ol_amount) desc, o_entry_d;

```

Fizemos então o ***explain analyze*** e ficamos com os seguintes resultados:

```

QUERY PLAN
Sort  (cost=2520003.99..2522157.14 rows=621258 width=77) (actual time=53010.226..53374.962 rows=630000 loops=1)
  Sort Key: (sum(ol_amount)) DESC, o_entry_d
  Sort Method: external merge Disk: 43168kB
-> Finalize GroupAggregate (cost=1642207.21..2405607.96 rows=621258 width=77) (actual time=47068.200..52046.021 rows=630000 loops=1)
    Group Key: o_id, o_w_id, o_d_id, c_id, c.last, o_entry_d, o.ol_cnt
    Filter: (sum(ol_amount) > '200'::numeric)
    Rows Removed by Filter: 1548068
-> Gather Merge (cost=1642207.21..2270484.42 rows=3727546 width=77) (actual time=36667.297..48108.825 rows=2221802 loops=1)
  Workers Planned: 2
  Workers Launched: 2
    -> Partial GroupAggregate (cost=1641207.19..1839233.10 rows=1863773 width=77) (actual time=36365.242..45507.649 rows=740601 loops=3)
      Group Key: o_id, o_w_id, o_d_id, c_id, c.last, o_entry_d, o.ol_cnt
      -> Sort (cost=1641207.19..1660621.49 rows=7765722 width=48) (actual time=36365.149..41301.912 rows=6212519 loops=3)
        Sort Key: o_id, o_w_id, o_d_id, c_id, c.last, o_entry_d, o.ol_cnt
        Sort Method: external merge Disk: 380448kB
        Worker 0: Sort Method: external merge Disk: 404312kB
        Worker 1: Sort Method: external merge Disk: 404384kB
        -> Parallel Seq Scan on matq4alljoins (cost=0.00..274693.22 rows=7765722 width=48) (actual time=548.817..2935.791 rows=6212519 loops=3)
Planning Time: 0.198 ms
JIT:
  Functions: 25
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 6.226 ms, Inlining 387.540 ms, Optimization 738.054 ms, Emission 526.951 ms, Total 1650.770 ms
Execution Time: 53497.358 ms
(24 rows)

```

Figura 22: *Query plan* - Interrogação analítica 4 - com utilização da *materialized view* *matq4alljoins*

Uma vez mais não temos o melhor resultado até agora, já que com a criação da primeira *materialized view* obtivemos melhores resultados.

Com uma análise cuidada, percebemos que o *group by* consumia muito tempo. Para melhorar o desempenho do mesmo foi criado um índice na *materialized view* pelos campos das chaves que vão ser agrupadas, com o intuito de existir uma navegação mais rápida pela tabela que tem agora proporções bem maiores. Como podemos ver nas imagens do *query plan*, o *group key* é composto pelos campos: *o\_id*, *o\_w\_id*, *o\_d\_id*, *c\_id*, *c.last*, *o\_entry\_d*, *o.ol\_cnt*.

Assim sendo o índice criado foi o seguinte:

```

Create index indq4alljoins on matq4alljoins(o_id, o_w_id, o_d_id, c_id, c.last, o_entry_d,
o.ol_cnt);

```

Com todas estas otimizações o resultado foi:

```

-----  

QUERY PLAN  

-----  

Sort  (cost=2336425.00..2337978.13 rows=621252 width=77) (actual time=12626.252..12990.873 rows=630000 loops=1)
  Sort Key: (sum(o1_amount)) DESC, o_entry_d
  Sort Method: external merge Disk: 43168kB
-> GroupAggregate (cost=0.56..2221436.59 rows=621252 width=77) (actual time=7974.649..11934.873 rows=630000 loops=1)
  Group Key: o_id, o_w_id, o_d_id, c_id, c_last, o_entry_d, o.ol_cnt
  Filter: (sum(o1_amount) > '200'::numeric)
  Rows Removed by Filter: 1548068
-> Index Scan using indq4alljoins on matq4alljoins (cost=0.56..1769475.85 rows=18637556 width=48) (actual time=0.169..6899.622 rows=18637557 loops=1)
Planning Time: 0.551 ms
JIT:
  Functions: 6
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 1.477 ms, Inlining 12.091 ms, Optimization 72.499 ms, Emission 48.625 ms, Total 134.692 ms
Execution Time: 13060.725 ms
(14 rows)

```

Figura 23: *Query plan* - Interrogação analítica 4 - com utilização do índice *indq4alljoins*

Agora sim temos uma melhoria muito significativa do tempo de execução da *query*. Tendo começado com um tempo na ordem dos 71 segundos e estando agora com um tempo de execução de apenas 13, achamos ter atingido um bom resultado.

É óbvio que podíamos colocar mais trabalho na *materialized view*, mas isso tiraria toda a lógica à *query*. Se alterássemos mais o formato da *query* seria para a colocar toda em memória, passando a ser instantânea.

Desta forma conseguimos resolver mais do que um problema. Não só fica esta *query* otimizada, como todas as que precisem da junção destas mesmas 3 tabelas. Como é óbvio, diferentes queries poderão necessitar de índices em outros campos, mas o trabalho mais pesado fica já feito. A atualização periódica desta *materialized view* pode assim justificar-se para poder ser reutilizada por inúmeras queries diferentes.

## 5 Otimização do desempenho da carga transacional

Nesta secção tentamos alterar algumas configurações do *postgres* com o objetivo de aumentar os valores de *throughput* do *benchmark* TPC-C. Tendo muitas opções onde alterar, decidimos cingir-nos apenas aos campos destacados pelo docente numa das aulas teóricas da UC.

Isto implicou alterar os campos da secção de *Write Ahead Logs*, que foi a secção onde a maioria dos campos enfatizados pelo docente se situavam, estando esta dividida nas subsecções: *settings*, *checkpoints* e *achiving*.

A nossa abordagem passou por fazer uma avaliação individual de cada parâmetro para ver o impacto de cada um. De seguida, fizemos combinações com os campos que melhores resultados obtiveram tentando, com isto, maximizar o *throughput* ainda mais.

Cada avaliação foi feita com tempos de medição de 10 minutos e partiu sempre com o mesmo estado da base de dados (carregamento do ficheiro *dump* ao fim de cada teste), desta forma garantimos que todos os testes são feitos no mesmo ambiente de teste. Para automatizar estas avaliações, criamos um script (7.2) em que é executado o *benchmark*, o nome do ficheiro resultado é alterado para uma string dada (para depois sabermos a qual configuração aquele teste diz respeito) e ainda é corrido o *pg-restore* para voltar a por a base de dados pronta a ser testada novamente.

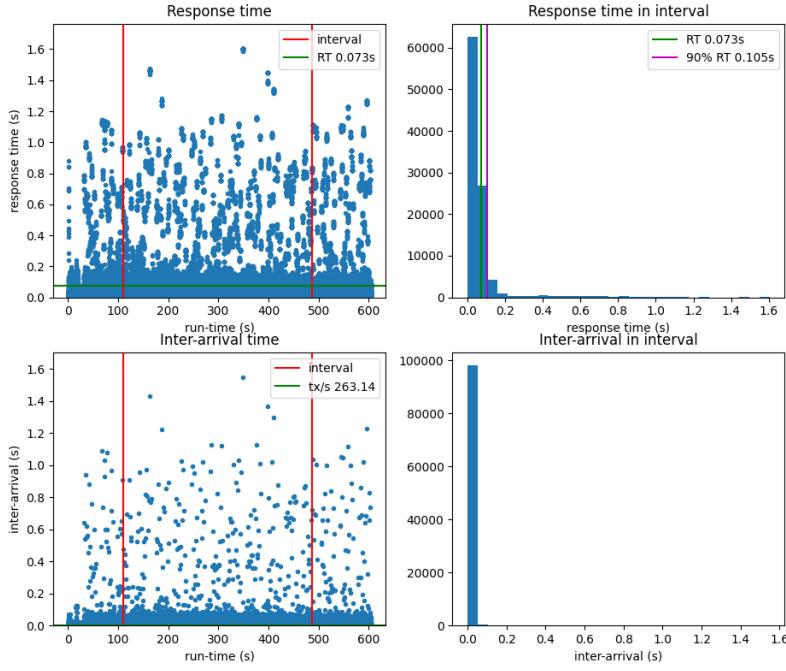
### 5.1 Default

Como as avaliações da máquina de referência foram feitos com tempos de 5 minutos, decidimos correr o *benchmark* de novo com todas as configurações por defeito para podemos fazer comparações mais realistas.

Throughput (tx/s)	Response-time (s)	Abort-rate (%)
263.1353736	0.0728543	0.0498829

Tabela 4: Valores da análise Escada TPC-C para configuração *default*.

Como era de esperar os valores não diferenciaram muito do que já tínhamos tirado antes, no entanto, temos os valores de referência para esta unidade de tempo.

Figura 24: Gráficos obtidos da configuração *default*.

## 5.2 Settings

### 5.2.1 *fsync*

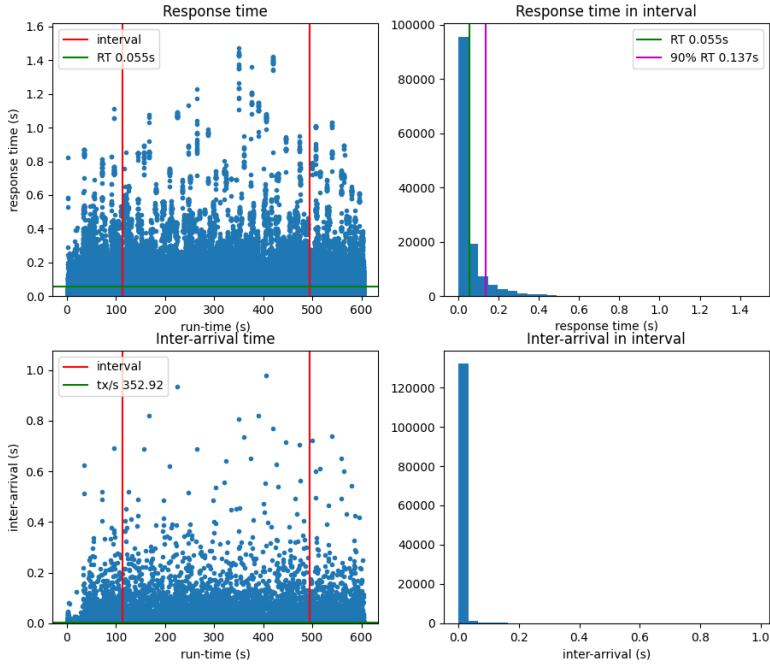
O valor por defeito deste parâmetro é a 'on', e com ele assim o *postgres* vai-se certificar que as atualizações são escritas em disco, ou seja, só marcando uma operação como terminada quando tudo foi escrito para disco. Isto ajuda na recuperação da base de dados em caso de falha, mas torna o processo bem mais lento.

Assim sendo, alteramos este campo para 'off' com o intuito de verificar melhorias nos valores de *throughput* aquando da execução do *benchmark*.

	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
off	352.9226265	0.0545099	0.0358665

Tabela 5: Valores da análise Escada TPC-C para a variação na configuração *fsync*.

Como podemos ver, as melhorias foram significativas, no entanto não é aconselhável usar esta configuração desta forma, já que é impossível recuperar transações que não foram escritas em disco. Assim sendo, sabemos que isto não deve ser usado pois existe a possibilidade de colocarmos a base de dados num estado incoerente.

Figura 25: Gráficos obtidos da configuração *fsync* a *off*

### 5.2.2 *synchronous\_commit*

O valor por defeito deste parâmetro é 'on'. Este parâmetro é parecido ao *fsync*, na medida em que estando a 'off', o cliente vê a sua operação realizada com sucesso ainda antes de ter sido escrita em disco, no entanto o *postgres* certifica-se que a transação é escrita usando sincronização. O que acontece em caso de falha é que a transação pode ser cancelada e o cliente não ver, porém a base de dados nunca fica num estado de incoerência ou com dados corrompidos (coisa que acontecia com *fsync* a 'off').

	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
off	304.2428756	0.0633112	0.0365405
local	266.1412483	0.0720887	0.0484978
remote_write	270.4872205	0.0709183	0.0491806
remote_apply	259.5015822	0.0739834	0.0487408

Tabela 6: Valores da análise Escada TPC-C para a variação na configuração *synchronous\_commit*.

Reparamos que os valores diferentes de 'off', que foram testados, apresentavam valores muito parecidos com o valor por defeito. Depois de alguma pesquisa percebemos que tinham o mesmo efeito a não ser que se acrescentasse valores à lista do campo *synchronous\_standby\_names*. Não tendo qualquer conhecimento deste parâmetro decidimos apenas considerar os valores de 'off'.

Realmente podemos ver que colocar o *synchronous commit* a 'off' traz-nos alguns benefícios a nível de *throughput* em relação à configuração *default*.

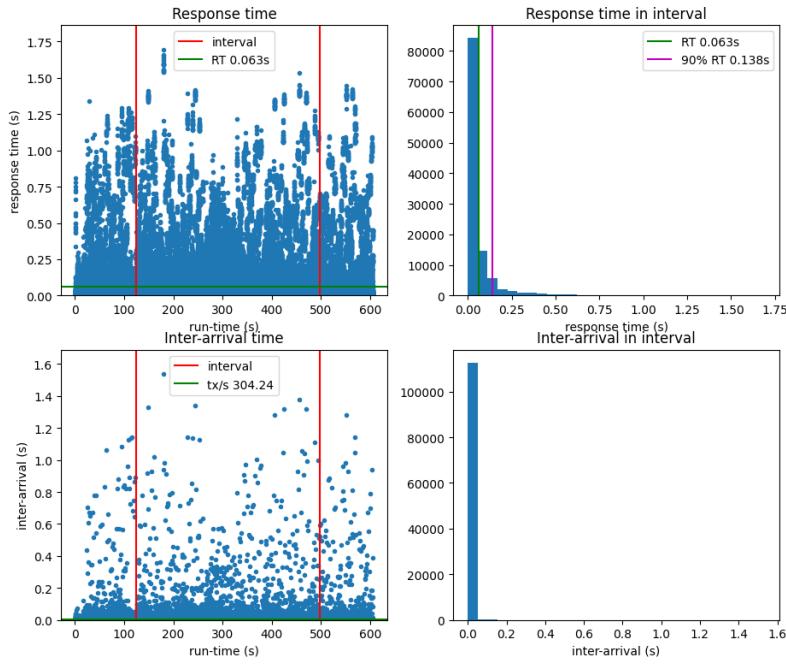


Figura 26: Gráficos obtidos da configuração *sync. commit* a *off*

### 5.2.3 wal\_sync\_method

Este parâmetro define qual é a operação usada pela base de dados para fazer a sincronização de escritas. O docente referiu que cada sistema operativo já traz a melhor opção (no nosso caso é o *fsync*), pelo que decidimos não efetuar alterações aqui.

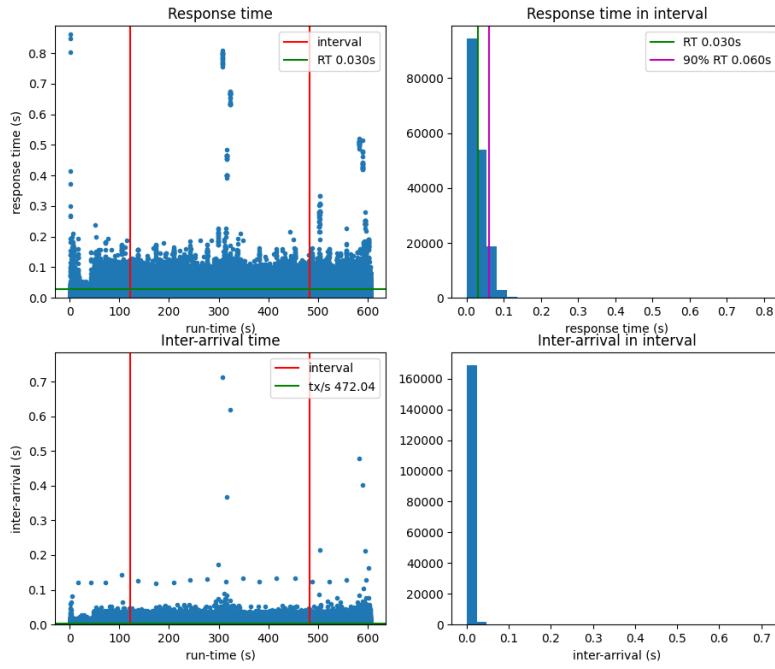
### 5.2.4 full\_page\_writes

O valor por defeito deste parâmetro é 'on'. Este parâmetro estando ativo, o servidor escreve para *WAL* uma página inteira na primeira modificação que a mesma sofre durante o *checkpoint* atual. Ao desligarmos isto podemos levar a que a base de dados tenha partes da página com valores atualizados e outras com valores antigos, estando assim num estado incoerente. No entanto, isto pode fazer com que o servidor seja bem mais rápido.

	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
off	472.0384730	0.0298681	0.0374766

Tabela 7: Valores da análise Escada TPC-C para a variação na conf. *full\_page\_writes*.

Como podemos constatar, as melhorias foram bastante significativas, mas, tal como no caso do *fsync*, isto não é sustentável num ambiente de produção, já que pode levar a uma base de dados com os dados corrompidos.

Figura 27: Gráficos obtidos da configuração *full\_page\_writes* a *off*

### 5.2.5 wal\_writer\_delay

Este parâmetro define o limite de tempo que o servidor espera antes de dar *flush* ao *WAL*. Ao modificar este parâmetro o servidor pode esperar mais ou menos tempo antes de escrever todo o *log* em memória. O valor *default* é 200ms.

	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
100ms	236.8811969	0.0811729	0.0484541
400ms	280.5160280	0.0684702	0.0472020
800ms	265.5742650	0.0716620	0.0461279

Tabela 8: Valores da análise Escada TPC-C para a variação na conf. *wal\_writer\_delay*.

Podemos reparar que baixar o valor faz com que haja mais escritas e assim tornar o processo um pouco mais lento. Fazer aumentos melhora um pouco, mas isto pode estar a ser restringido pelo campo que vamos analisar de seguida.

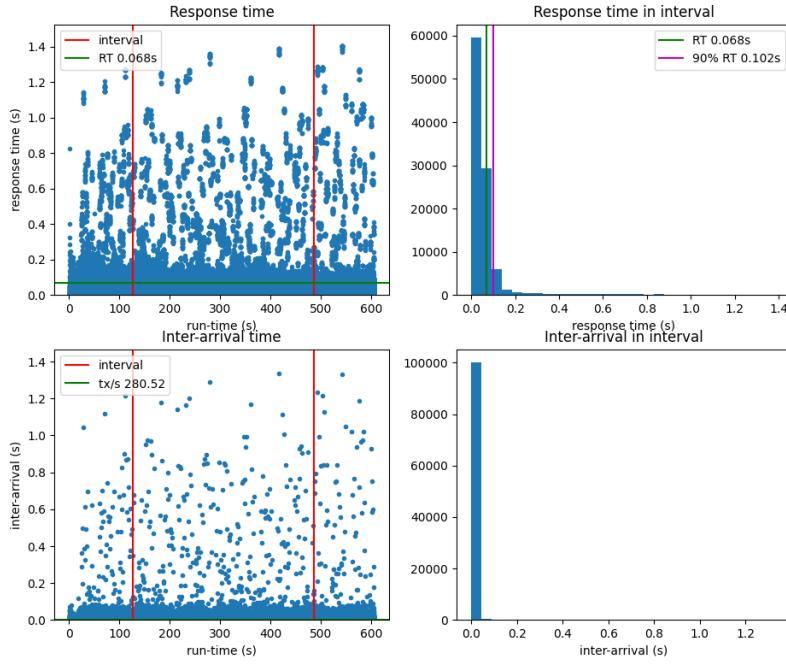


Figura 28: Gráficos obtidos da configuração *wal\_writer\_delay* a 400ms

### 5.2.6 *wal\_writer\_flush\_after*

Este parâmetro, tal como o anterior, define um limite que o servidor espera antes de dar *flush* ao *WAL*, mas agora em tamanho, ou seja, quando o *WAL* ultrapassa este tamanho, o servidor força a escrita do mesmo em memória. O valor *default* é *1MB*.

	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
0MB	280.4408711	0.0683869	0.0480550
2MB	259.8438456	0.0739538	0.0478186
4MB	269.4509102	0.0712922	0.0481206
8MB	270.7613978	0.0707179	0.04912240

Tabela 9: Valores da análise Escada TPC-C para a variação na configuração *wal\_writer\_flush\_after*.

O valor a 0MB força escritas imediatas. Seria de esperar que isto piorasse o desempenho devido ao maior número de escritas, no entanto, houve uma melhoria talvez devido ao facto de serem escritas mais pequenas. Nos restantes valores não se verificaram mudanças significativas, pois podem estar a ser influenciados pelo parâmetro anterior que se encontra default aquando destes testes. Talvez a combinação de mudanças traga alguns benefícios.

### 5.2.7 *commit\_delay & commit\_siblings*

O parâmetro de *commit\_delay* define um tempo extra que o servidor espera antes de dar *flush* do *WAL* para memória e o *commit\_siblings* define o número de transações que têm que estar ativas para que o *commit\_delay* seja inicializado. Decidimos fazer logo testes com estes parâmetros em conjunto, pois percebemos que estavam muito ligados.

Com isto esperamos que aumentando o *delay* e o número de *siblings* (aproximando-nos do número de transações paralelas que ocorrem no *benchmark*) seja obtido um resultado mais animador.

Delay	Sib.	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
100ms	3	268.6055211	0.0714656	0.0488384
500ms	3	250.7248674	0.0765476	0.0478795
500ms	5	269.0397008	0.0712585	0.0480067
500ms	6	287.6330594	0.0666952	0.0484777
500ms	10	249.1856290	0.0771075	0.0472574
1000ms	4	263.0502379	0.0730121	0.0490595

Tabela 10: Valores da análise Escada TPC-C para a variação na combinação das configurações *commit\_delay* & *commit\_siblings*.

O resultado não foi o esperado já que não se apresentaram melhorias muito significativas. Talvez fazer diferentes combinações pudesse trazer melhorias, mas é difícil perceber quais os valores ideais para este *benchmark*.

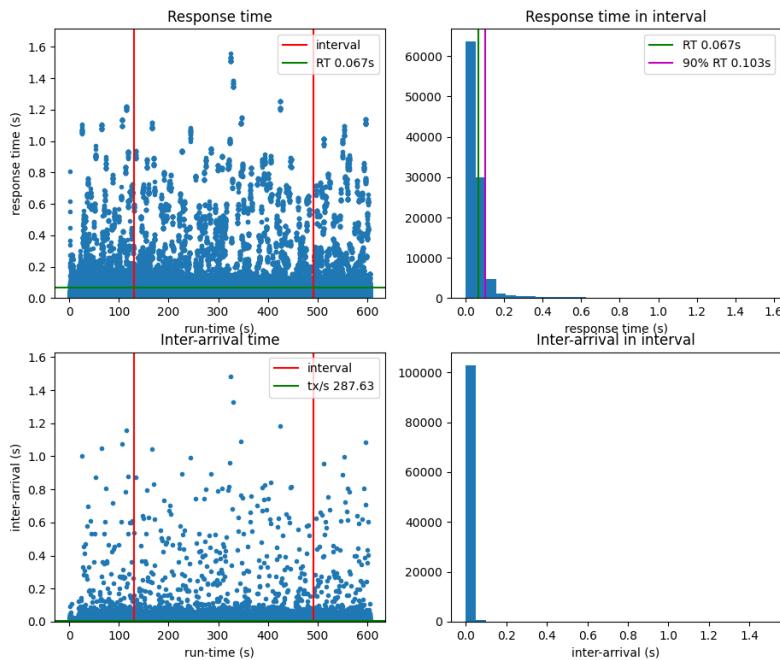


Figura 29: Gráficos obtidos da configuração *commit\_delay* a 500ms e *commit\_siblings* a 6.

### 5.3 Checkpoints

#### 5.3.1 *checkpoint\_timeout*

Este parâmetro consiste na definição de tempo máximo entre *checkpoints* automáticos no *log* (*WAL*). Este valor é por defeito de 5 minutos. O aumento deste valor pode resultar em menos escritas em memória e, por conseguinte melhores resultados de *throughput*.

	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
1min	253.8585501	0.0729832	0.0499654
2min	282.6342491	0.0677777	0.0489211
8min	269.2606854	0.0713016	0.0494638

Tabela 11: Valores da análise Escada TPC-C para a variação na configuração *checkpoint\_timeout*.

Como podemos ver, os resultados não diferem muito do valor original. Isto pode estar a acontecer devido ao valor de *max\_wall\_size* por defeito ter mais influência do que o *checkpoint-timeout*.

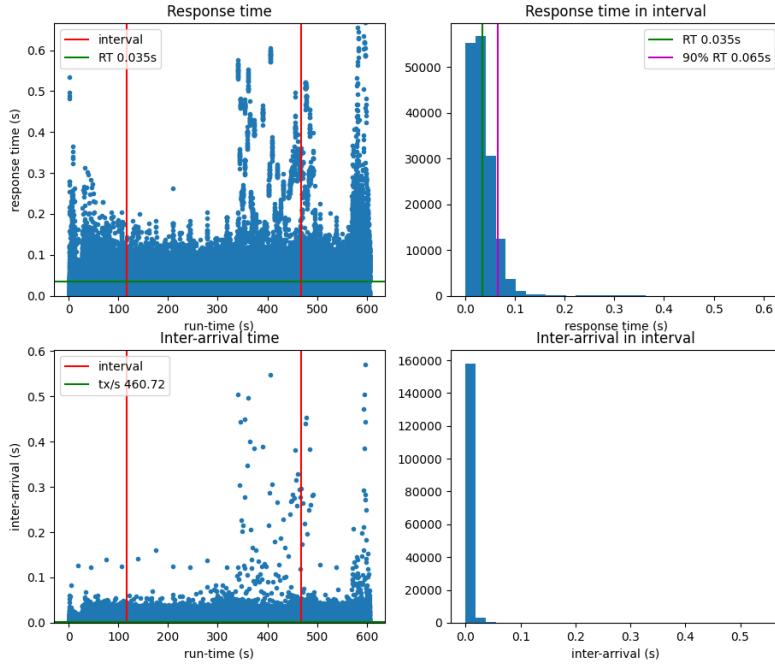
### 5.3.2 *max\_wal\_size*

Este parâmetro define o espaço máximo entre *checkpoints* automáticos no *WAL*. O valor por defeito é de 1GB. O aumento deste valor pode originar menos escritas em memória o que pode resultar em resultados de *throughput* melhores.

	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
2GB	375.9357271	0.0496221	0.0463187
4GB	449.4768867	0.0352027	0.0392723
8GB	460.7194777	0.0349773	0.0417906

Tabela 12: Valores da análise Escada TPC-C para variação na configuração *max\_wal\_size*.

Conseguimos ver agora que o desempenho melhorou muito, confirmando a nossa suspeita anterior de que o tamanho do log por defeito estava a ser mais restritivo do que o tempo entre checkpoints.

Figura 30: Gráficos obtidos da configuração *max\_wal\_size* a 8GB

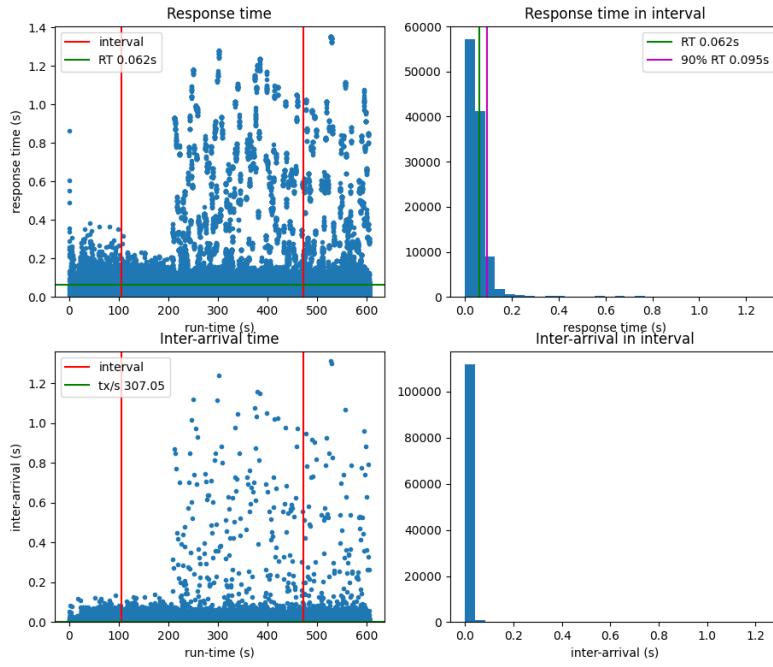
### 5.3.3 *min\_wal\_size*

Este campo define o valor mínimo do tamanho do log sendo que enquanto o log não tiver este tamanho, os valores antigos serão reutilizados e não eliminados.

	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
160MB	307.0546190	0.0624731	0.0473956
320MB	263.6202340	0.0728801	0.0483596
640MB	287.4286451	0.0667436	0.0491063

Tabela 13: Valores da análise Escada TPC-C para variação na configuração *min\_wal\_size*.

É de constatar que estes resultados apresentam variâncias um pouco anormais pelo que achamos que as diferenças sejam mais originadas pela própria *cloud* do que pelos valores do parâmetro, e que na nossa opinião não são muito conclusivos.

Figura 31: Gráficos obtidos da configuração *min\_wal\_size* a 160MB

### 5.3.4 *checkpoint\_completion\_target*

Este parâmetro especifica a fração do tempo definido no *checkpoint\_timeout* que esperamos que um *checkpoint* decorra. O servidor vai gerar *warnings* caso este valor seja excedido.

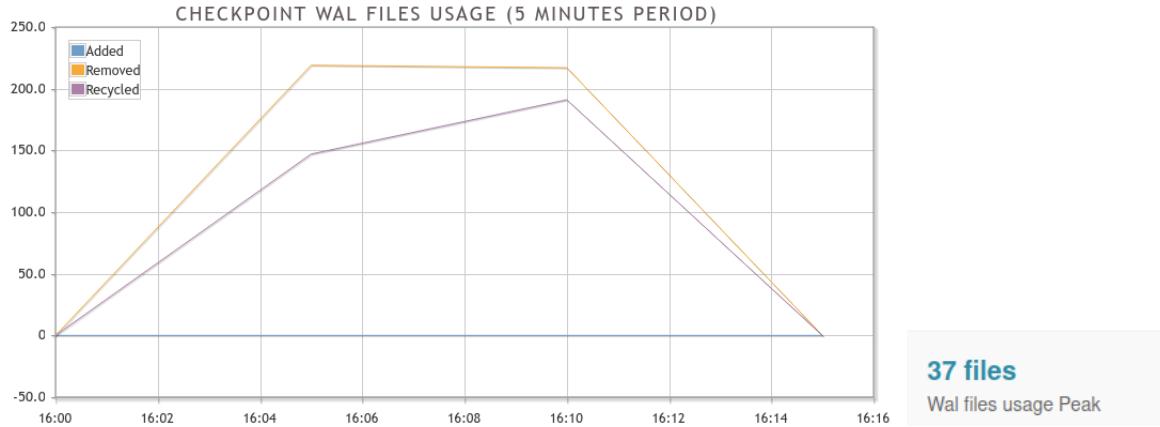
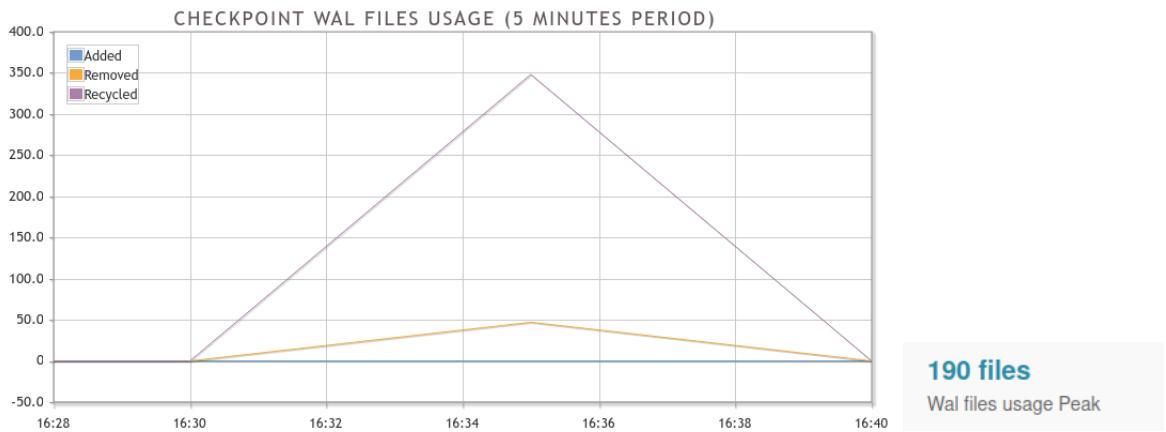
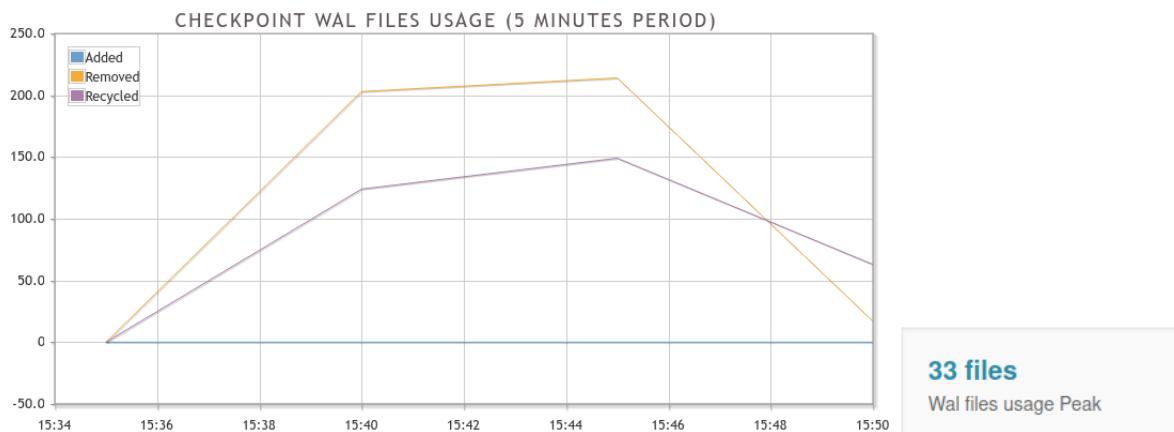
	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
0.0	324.8899078	0.0564812	0.0442211
0.3	263.6478342	0.0728561	0.0494275
0.7	293.6132490	0.0653144	0.0486436
1.0	304.9358570	0.0629880	0.0479853

Tabela 14: Valores da análise Escada TPC-C para a variação na configuração *checkpoint\_completion\_target*.

### 5.3.5 Análise de resultados

Para melhor entendermos como o *postgres* variava o uso de *checkpoints* nos *logs* decidimos recorrer à ferramenta *pgbadger*. Com ela conseguimos perceber as diferenças que o *postgres* apresenta quanto ao *management* do *WAL* e dos *checkpoints* nos diferentes tipos de configuração.

De seguida apresentam-se gráficos gerados pelo *pgbadger* em relação ao uso de ficheiros de *WAL* na configuração *default*, na configuração com *max\_wal\_size* igual a 8GB e ainda na configuração *min\_wal\_size* igual a 160MB.

Figura 32: Valores de *wal files usage* para conf. *default*.Figura 33: Valores de *wal files usage* para *max\_wal\_size* = 8GB.Figura 34: Valores de *wal files usage* para *min\_wal\_size* = 160MB.

Como podemos verificar, a configuração de *max\_wal\_size* apresenta uma quantidade de ficheiros removidos muito inferior, isto acontece pois os *checkpoints* são de tamanho consideravelmente maior, daí se confirmar o que já tínhamos referido: há menos escritas em disco o que leva a um processamento mais rápido. De notar que como o valor de

*checkpoint timeout* por defeito é 5 minutos, isso explica o porquê de nos 5 minutos haver um pico no uso de ficheiros *wal*. Quanto ao *min\_wal\_size*, reparámos que o impacto nos ficheiros reciclados não foi tão grande como desejávamos e daí os resultados de *throughput* não serem tão bons.

Olhando agora para o número máximo de ficheiros de *log* usados, conseguimos cimentar a nossa posição quanto ao porquê do *max\_wal\_size* igual a 8GB apresentar valores tão satisfatórios.

## 5.4 Archiving

### 5.4.1 archive\_mode

Este parâmetro define se os *logs* depois de utilizados são arquivados em algum lado ou não. O valor deste parâmetro por defeito é *off*.

	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
on	246.7155514	0.0780405	0.0477298
always	249.3263752	0.0768988	0.0480247

Tabela 15: Valores da análise Escada TPC-C para a variação na configuração *archive\_mode*.

Ao analisarmos estes valores percebemos que tanto a opção de 'always' e 'on' têm resultados similares e após uma pesquisa pela documentação do *postgres* percebemos que a opção 'always' só difere de 'on' quando a base de dados se encontra em *recovery mode*, pelo que no caso de estudo deste *benchmark* não haverá diferença.

Quando corremos este teste percebemos que o arquivamento não estava a ser efetuado porque aliada a esta opção vem a do comando usado para o efeito. Como por defeito este vem vazio, o arquivamento é tentado mas depois não é efetuado. Seria de bom tom testar com os diferentes tipos de comandos, mas como o nosso objetivo é melhorar o *throughput* deste *benchmark*, achamos que não seria necessário pois isso apenas acrescenta complexidade ao processamento das transações trazendo custos de performance.

## 5.5 Combinações

Todos os testes realizados até agora foram individualizados, ou seja todos os campos não especificados encontravam-se com valores *default*.

Nesta secção vamos explorar a combinação de configurações explorando aquelas que, individualmente, se apresentaram mais produtivas.

De referir que as configurações de *fsync* e *full\_page\_writes* não vão ser usadas aqui pelo facto de, tal como já foi referido, não serem indicadas para o uso em ambiente de produção.

### 5.5.1 max & min wal\_size

As configurações que melhor desempenho geraram foram as presentes na secção dos *checkpoints*. Juntando as duas mais positivas o resultado foi o seguinte.

Opções		Resultados		
max	min	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
8GB	160MB	459.7238799	0.0357158	0.0410320

Tabela 16: Valores da análise Escada TPC-C para a variação na configuração  $\max_{wal\_size}$  &  $\min_{wal\_size}$ .

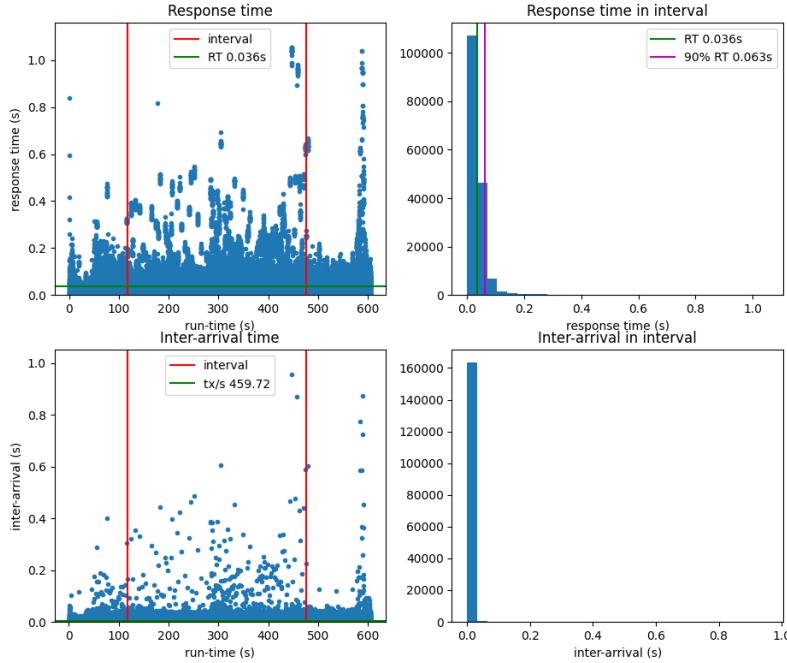


Figura 35: Gráficos obtidos da combinação  $\max$  &  $\min wal\_size$

### 5.5.2 $\max$ & $\min wal\_size$ e $sync\_commit$

Olhando agora também para as configurações noutras secções, a alteração do *synchronous\_commit* foi a mais apetecível, pelo que decidimos adicionar a que melhor resultado apresentou.

Opções			Resultados		
max	min	sync	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
8GB	160MB	off	479.1384566	0.0125960	0.0109078

Tabela 17: Valores da análise Escada TPC-C para a variação na configuração  $\max$  &  $\min wal\_size$  e  $synchronous\_commit$ .

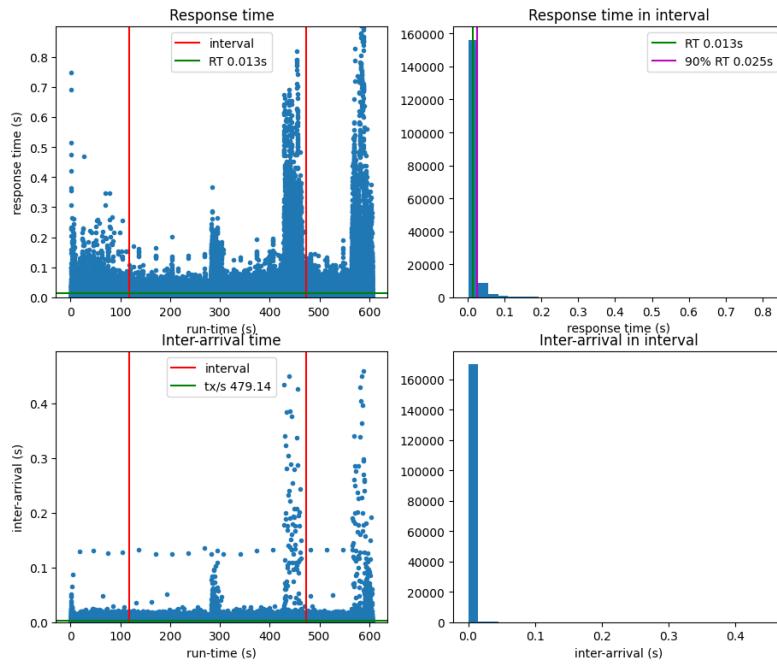


Figura 36: Gráficos obtidos da combinação *max & min wal\_size* e *sync\_commit*.

### 5.5.3 *max & min wal\_size, sync\_commit* e *checkpoint\_timeout*

Decidimos também acrescentar a configuração de *checkpoint\_timeout* que melhor resultados apresentou, com esperança de melhorar ainda mais a configuração, no entanto isto não aconteceu levando a uma ligeira decadência nos resultados.

Opções				Resultados		
max	min	sync	timeout	Throughput (tx/s)	Response-time (s)	Abort-rate (%)
8GB	160MB	off	2min	473.2711420	0.0188980	0.0167148
8GB	160MB	on	2min	421.6271253	0.0429000	0.0455702

Tabela 18: Valores da análise Escada TPC-C para a variação na configuração *max\_wal\_size*, *min\_wal\_size*, *synchronous\_commit* e *checkpoint\_timeout*.

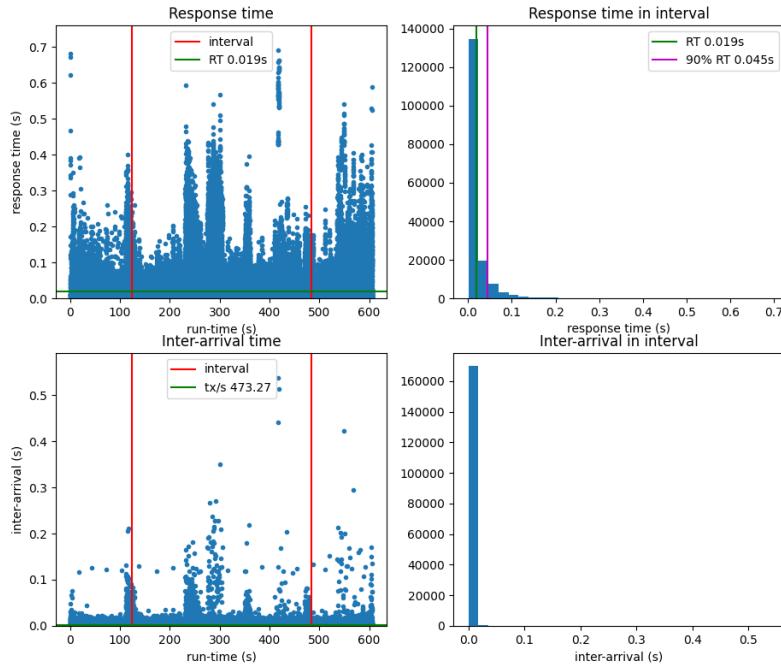


Figura 37: Gráficos obtidos da combinação *max & min*, *sync\_commit* e *checkp\_timeout*.

#### 5.5.4 *max & min wal\_size, sync\_commit, checkpoint\_timeout e commit delay & siblings*

Como houve melhorias quando alteramos os valores de *commit delay* e de *commit siblings* em simultâneo, apesar de mínimas, decidimos adicionar as configurações que tiveram melhor desempenho neste campo para ver se conseguíamos obter um resultado ainda melhor.

Opções					
max	min	sync	timeout	delay	sib
8GB	160MB	off	2min	500ms	6
8GB	160MB	on	2min	500ms	6

Resultados		
Throughput (tx/s)	Response-time (s)	Abort-rate (%)
476.1274240	0.0167652	0.0150564
429.3041845	0.0422880	0.0449248

Tabela 19: Valores da análise Escada TPC-C para a variação na configuração *max\_wal\_size*, *min\_wal\_size*, *synchronous\_commit*, *checkpoint\_timeout* e *commit delay & siblings*.

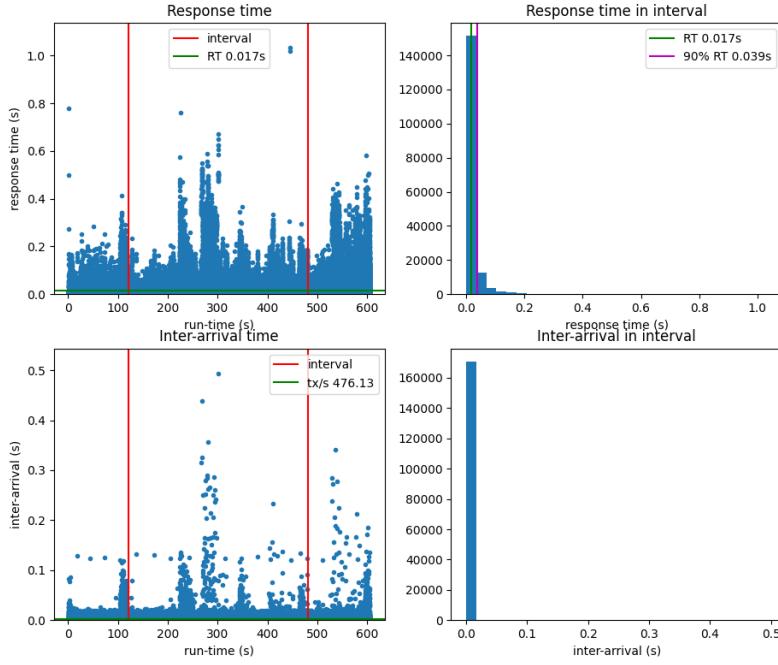


Figura 38: Gráficos obtidos da combinação *max & min*, *sync\_commit*, *checkpoint\_timeout* e *commit\_delay & siblings*.

Os resultados obtidos não superaram o melhor valor já obtido, pelo que este teste não nos satisfez totalmente.

## 5.6 Análise de Resultados

Com todos os testes e alterações feitas chegamos à conclusão que a melhor configuração é a presente na secção 5.5.2. De referir que esta não garante que a informação dada ao cliente da base de dados seja totalmente verdadeira, já que uma transação pode ser dada como completa, mas na realidade ter sido abortada. Caso a base de dados se encontre numa situação em que tal coisa não possa acontecer temos a configuração da secção 5.5.1 que tem resultados ligeiramente piores mas ainda assim bastante satisfatórios.

A nossa configuração já referida como ideal, não só aumentou o *throughput* para perto do dobro do original, como também fez baixar os valores de *abort-rate* e de *response-time* consideravelmente.

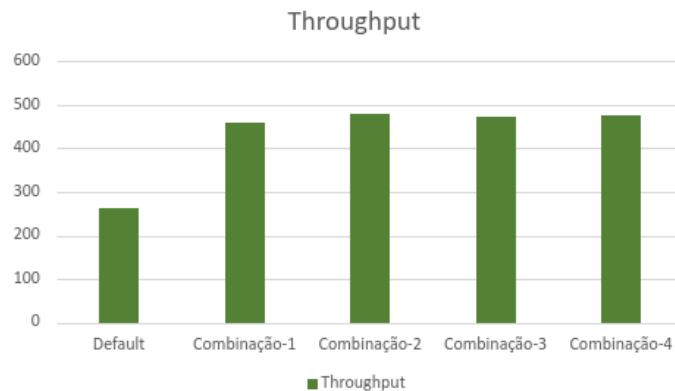


Figura 39: Gráfico *Throughput* comparativo das diferentes combinações e *default*.

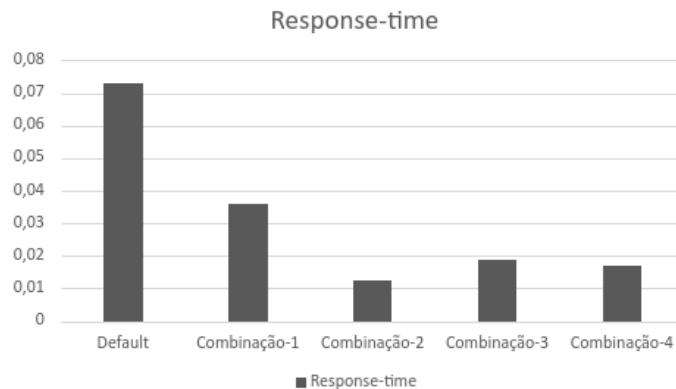


Figura 40: Gráfico *Response-time* comparativo das diferentes combinações e *default*.

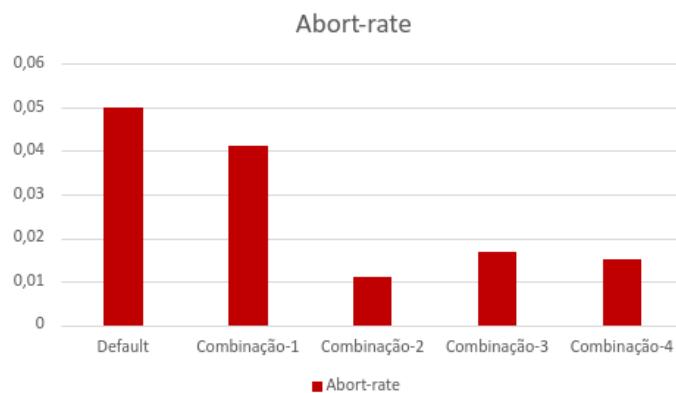


Figura 41: Gráfico *Abort-rate* comparativo das diferentes combinações e *default*.

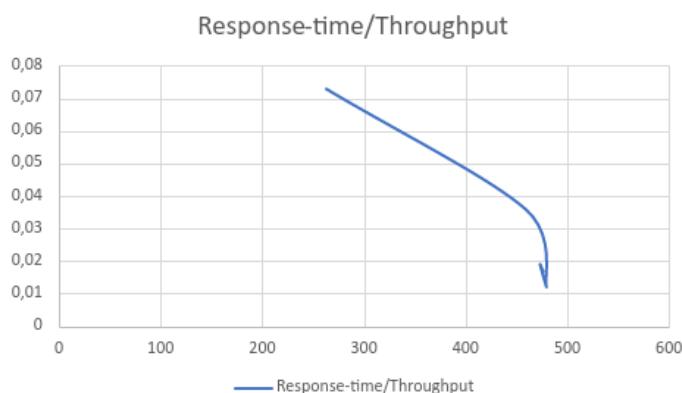


Figura 42: Gráfico *Response-time/Throughput* comparativo das diferentes combinações e *default*.

## 6 Conclusão

Este projeto teve como principal objetivo analisar e perceber os mecanismos de uma base de dados abordados na unidade curricular de Administração de Bases de Dados, que nós assumimos como cumprido.

A primeira tarefa foi um pouco fácil para o grupo, uma vez que o professor nos disponibilizou um tutorial de como fazer todas as configurações na *Google Cloud* e também de como configurar as próprias máquinas. Tendo os tipos de máquinas sido predefinidas pelo docente, não foi necessário fazer testes para fazer escolha das mesmas, no entanto houve uma pequena alteração na máquina *benchmark* que em certos momentos se mostrou insuficiente.

Na escolha de uma configuração de referência do *benchmark*, o grupo dedicou algum tempo uma vez que era um ponto chave para a resolução de todo o trabalho prático. Era então uma decisão importante para o grupo o que exigiu testes às máquinas para perceber de que maneira poderíamos tirar melhor partido da capacidade das mesmas (CPU e RAM). Com a escolha de *warehouses* tivemos em conta o tamanho da base de dados que tentamos manter entre os 6 e os 10 GB tal como tinha sido discutido com o docente. Após essa configuração definida, passamos à decisão de um número base de clientes por *warehouse* que nos levou a testar os níveis de CPU da máquina que corria o servidor *postgres*. Isto fez-nos usar ferramentas do sistema operativo como o *htop* para melhor monitorizar as máquinas e cumprindo assim com uma valorização do trabalho.

O passo seguinte foi então a adaptação e otimização de interrogações analíticas. Adaptar as interrogações foi um pouco difícil para nós, pois não só tínhamos que as tornar compatíveis com a nossa base de dados, como tivemos que alterar alguns filtros das mesmas para que estes fizessem sentido no contexto da nossa realidade (por exemplo: adaptar datas, etc...). Tendo então as otimizações escolhidas devidamente adaptadas, procedemos ao uso de mecanismos de redundância, como a indexação e *materialized views*, com o objetivo final de baixar o máximo que conseguíssemos o tempo de execução das *queries*. Ficamos bastante contentes com os resultados obtidos e por termos sido capazes de perceber como os diferentes mecanismos podem ser aplicados num contexto mais prático e real.

Como última tarefa tivemos então a otimização da carga transacional. Era nos pedido que nos focássemos maioritariamente na subida do *throughput* da mesma e pensamos ter alcançado essa meta. A manipulação das configurações do *postgres* que foram indicadas pelo docente realmente ajudaram-nos muito nesta tarefa. O uso de ferramentas de monitorização do *postgres*, nomeadamente o *pgbadger*, ajudou-nos a perceber como a alteração das configurações modificavam o modo de execução das transações por parte do *postgres*. Ficamos ainda mais satisfeitos pelo facto de termos conseguido também baixar os valores de *response-time* e de *abort-rate* que, apesar de não serem o objetivo principal, achamos valores importantes de otimizar também.

Em suma, achamos ter atingido todos os objetivos traçados para este projeto prático, tirando então um balanço bastante positivo em termos, não só de resultados como também de aprendizagem.

## 7 Scripts

### 7.1 Run 1

Automatização da execução do *benchmark*, tendo como parâmetros variáveis o n.º de clientes e o tempo de execução (nas configurações do *benchmark*).

---

```
#!/bin/bash

_NCLI=0
_TIME=0

_TPC_C="tpc-c-0.1-SNAPSHOT"
_CNF_FILE="$_TPC_C/etc/workload-config.properties"

read -p "Number of clients: " _NCLI
read -p "Runtime (minutes): " _TIME

if [ "$_NCLI" -eq "$_NCLI" ] 2>/dev/null && [ "$_TIME" -eq "$_TIME" ]
  2>/dev/null; then
  echo "ok."
else
  echo "invalid input inserted"
  exit 1
fi

echo "[1] Updating configuration $_CNF_FILE..."
echo -e "\tParameters: clients = $_NCLI, runtime=$_TIME"

# change configuration file
sed -i "s/^\\(tpcc.numclients\\s*=\\s*\\)\\.*/$\\1$_NCLI/" $_CNF_FILE
sed -i "s/^\\(measurement.time\\s*=\\s*\\)\\.*/$\\1$_TIME/" $_CNF_FILE

echo "[2] Running run.sh"

cd $_TPC_C
./run.sh

cd ..
pg_restore -h server -c -d tpcc < tpcc70.dump

echo "ok."
```

---

## 7.2 Run 2

Automatização da execução das transações (*run.sh* e *pg\_restore*.)

---

```
#!/bin/bash

TCP_C="input_name"

read -p "Output file name: " TPC_C
read -p "Are you sure? (y=continue/n=exit): " confirm && [[ $confirm == [yY] ]] || $confirm == [yY] [eE] [sS] ]] || exit 1

echo "File $TPC_C is being generated..."

cd tpc-c-0.1-SNAPSHOT
./run.sh
mv TPCC-20* $TPC_C
cd ..
pg_restore -h server -c -d tpcc < tpcc70.dump

echo "done"
```

---