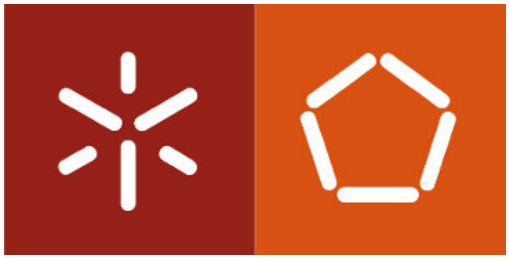


Universidade do Minho

Mestrado integrado em Engenharia Informática

Trabalho Prático

Diogo Pinto Ribeiro, A84442

João Nuno Cardoso Gonçalves de Abreu, A84802

José Diogo Xavier Monteiro, A83638

Vasco António Lopes Ramos, PG42852

Administração de Bases de Dados
4º Ano, 1º Semestre
Departmento de Informática

28 de abril de 2021

Índice

1 Introdução

O presente relatório pretende descrever qual a abordagem usada para realizar o trabalho prático proposto no âmbito da UC **Administração de Base de Dados**, incluída no Perfil de Engenharia de Aplicações, do Mestrado Integrado em Engenharia Informática (MIEI) da Universidade do Minho.

O trabalho consistiu em três grandes fases distintas:

- Instalação e Configuração de Referência, tanto do servidor de Base de Dados *PostgreSQL* como do servidor de *benchmarking*.
- Otimização do desempenho de quatro *queries* (interrogações analíticas).
- Otimização do desempenho do sistema da Base de Dados, relativamente à carga transacional de referência.

Deste modo, ao longo dos próximos capítulos será descrita e analisada a abordagem feita para cumprir com os requisitos acima descritos.

2 Setup e Configuração de Referência

O primeiro passo necessário para a realização deste trabalho prático passa pela instalação e configuração do *benchmark* TPC-C, tentando obter uma configuração de referência.

2.1 Setup

Para o setup deste *benchmark* recorremos à **Google Cloud Platform** (GCP), usando instâncias de VM no Compute Engine. A instância que suportará a nossa base de dados será denominada de **server**. Foi utilizada uma instância de **Primeira Geração**, N1, do tipo **n1-standard-2** (2 vCPU e 7,5 GB memória). O sistema operativo instalado foi o **Ubuntu 20.04 LTS**. Para o disco escolheu-se um **disco SSD** com **250GB** de capacidade, isto porque na GCP (*Google Cloud Platform*), a capacidade de débito do disco está diretamente relacionada com a capacidade física (tamanho) do disco.

Para a máquina que irá suportar o nosso *benchmark* também foi utilizada uma instância de **Primeira Geração**, N1. Desta vez, foi usado o tipo **f1-micro** (1 vCPU e 614 MB memória) dado que nos fornece bastantes horas sem custos. O sistema operativo manteve-se igual à instância anterior.

Na instância **server** foi instalado o PostgreSQL 12 e foram feitas as devidas alterações aos ficheiros de configuração para permitir ligações de outras máquinas da mesma rede local. Na instância **benchmark** foi instalado o Java, o cliente para o PostgreSQL 12 e o *benchmark* TPC-C.

Usando a instância **benchmark**, criámos uma base de dados chamada **tpcc** e carregámos todos os scripts sql fornecidos.

Para evitar repetir a utilização do comando **./load.sh** mais vezes do que o desejável e assim evitar custos, corremos o seguinte comando: **pg_dump -h server -Fc tpcc > tpcc.dump**. Para repor a base de dados usando estes ficheiros, basta correr o comando **pg_restore -h server -c -d tpcc < tpcc.dump**, sendo a reposição bastante mais rápida.

2.2 Análise da Configuração de Referência

Após termos o nosso processo de setup terminado, avançámos para a criação de uma configuração de referência. A nossa configuração de referência possui dois parâmetros que iremos aperfeiçoar de acordo com as nossas exigências: o número de ***warehouses*** e o número de **clientes**.

2.2.1 Warehouses

O número de ***Warehouses*** é o parâmetro que irá influenciar o tamanho que a nossa base de dados irá ter. Para este parâmetro o nosso objetivo é obter uma base de dados da mesma ordem de grandeza da quantidade de memória disponível. Assim sendo, o nosso objetivo será obter uma base de dados que possua um tamanho aproximado aos **4 GB**. Este valor permite-nos alocar recursos suficientes à base de dados em si, não esgotando estes mesmos recursos para outros processos paralelos existentes na máquina. Neste processo, fomos sequencialmente populando a base de dados para número de *warehouses* crescentes, tendo sempre o cuidado de guardar **dump's** da base de dados. Estes dump's permitem-nos repor a base de dados de maneira mais eficiente, maximizando os recursos e diminuindo as despesas na GCP. Outra técnica que utilizámos para acelerar este processo foi a desativação do **fsync**, dado que estamos a popular a base de dados e não nos preocupamos com perca de dados. Desativar esta opção acelera consideravelmente o PostgreSQL, existindo o problema da perca de dados. Após a população da base de dados, esta opção será novamente ativada. Os valores obtidos encontram-se na seguinte tabela:

# **Warehouses**	**Tamanho da BD (MB)**
2	254
8	953
16	1879
34	4010

Tabela 1: Relação entre Nº Warehouses e Tamanho da BD

Desta forma, e como se pode ver na tabela 1, fixou-se como número de *warehouses* para a nossa configuração de referência os **34 *warehouses***.

2.2.2 Clientes

Estando definido o número de *warehouses*, falta saber qual o número de clientes a utilizar.

Para encontrar o valor deste parâmetro, teve-se em consideração o *Throughput*, o *Response Time*, o *Abort Rate* e a carga de CPU no servidor da BD. De modo a que não houvesse qualquer tipo de ruído/interferência nas métricas obtidas, a cada execução eliminou-se a base de dados e recreou-se a mesma através do ficheiro de *dump* associado aos 34 *warehouses*. Como *measurement time* utilizámos o valor de 10 minutos.

# Clientes	Throughput (tx/s)	Resp. Time (s)	Abort Rate (%)	CPU Usage (%)
10	29.74178	0.00400344	0.000372197	5
20	60.35399	0.00408177	0.001116383	10
30	89.41264	0.00428246	0.002006482	13
40	118.50972	0.00441000	0.002929960	16
50	156.05366	0.00452711	0.003711980	20
60	192.02696	0.00464861	0.004638262	23
70	234.57260	0.00478677	0.002929960	27
80	262.70408	0.00500425	0.006656693	30
90	309.16489	0.00527188	0.008596313	35
100	343.11096	0.00571290	0.010140637	37
110	371.21880	0.00588778	0.011565085	40
120	393.62435	0.00606921	0.012642674	43
130	406.49945	0.00669994	0.014718805	47
140	410.62435	0.00720160	0.017094865	50
150	412.46355	0.00806134	0.020732669	56

Tabela 2: Métricas para definir número de clientes

Figura 1: *Response Time* tendo em conta o número de Clientes

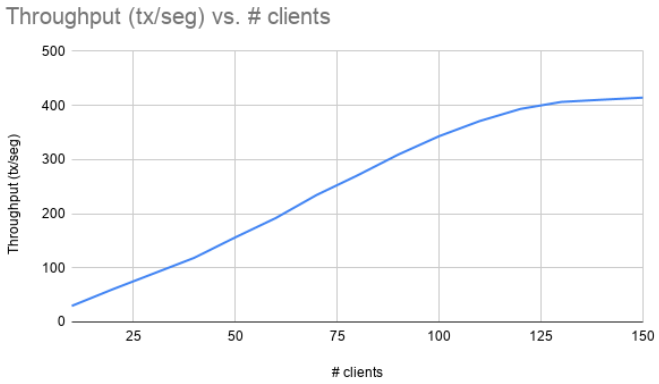


Figura 2: *Throughput* tendo em conta o número de Clientes

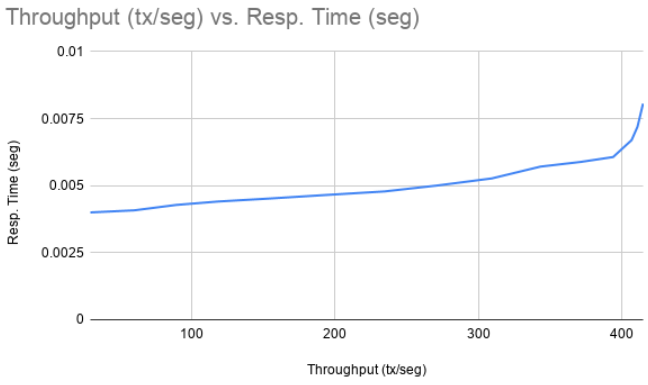


Figura 3: *Throughput* tendo em conta o *Response Time*

Desde o início que o objetivo era encontrar as configurações que permitissem obter uma taxa de utilização do CPU de cerca de 50%, uma taxa de ocupação da RAM de cerca de 50% a 60% e uma taxa de *abort rate* no máximo de 2%.

Assim, procurou-se uma configuração que permitisse um *Throughtput* próximo do máximo (perto do "joelho", como se pode ver no gráfico na figura 2), que tenha um rácio aceitável entre o *Throughtput* e o *Response Time*, como se pode ver no "joelho"do gráfico na figura 3.

Tendo em conta estes requisitos, e como se pode ver na tabela 2, o número de clientes que melhor se aproximava de cumprir todos os requisitos é 140, pelo que a nossa escolha final foi a de utilizar **140 clientes** como referência para os próximos testes e otimizações.

2.3 Configuração de Referência

Considerando todos os testes apresentados e respetiva análise, conclui-se que a configuração de referência inclui um servidor de base de dados com **2 CPUs**, **7.5GB de memória RAM**, **250GB de disco SSD**, **34 *warehouses*** e **140 clientes**. A figura 4, mostra uma vista simplificada da capacidade de resposta do configuração de referência escolhida.

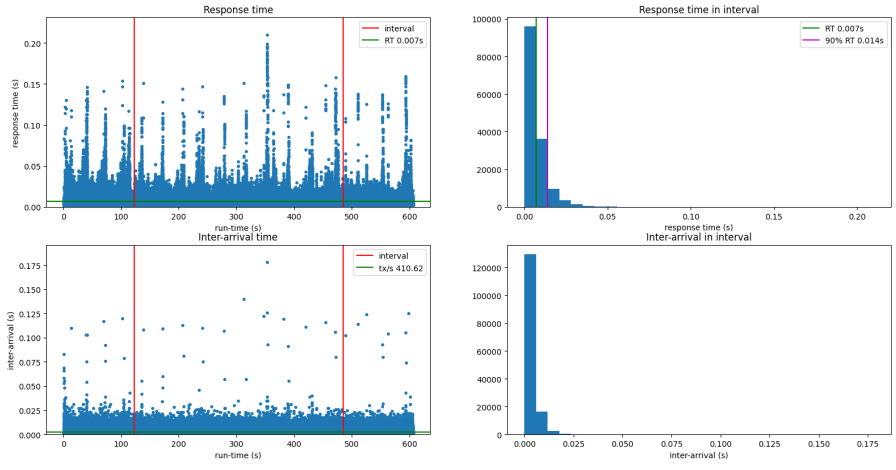


Figura 4: Métricas Configuração de Referência

3 Otimização de Queries Analíticas

Nesta secção iremos proceder à análise e otimização de desempenho de interrogações analíticas. Serão tidos em conta mecanismos de redundância usados, bem como os respetivos planos das interrogações.

Tal como visto na componente teórica desta UC, os índices são uma parte importante no bom desempenho de uma base de dados PostgreSQL. Dado isto, achámos pertinente descobrir se existem índices criados. Ao listar os índices presentes descobrimos que previamente já existem **21 índices**.

```
 tablename  |   indexname
------------+---------------
 customer   | keycustomer
 customer   | pk_customer
 customer   | ix_customer
 district   | keydisctrict
 district   | pk_district
 history    | keyhistory
 item       | keyitem
 item       | pk_item
 new_order  | keyneworder
 new_order  | ix_new_order
 order_line | keyorderline
 order_line | pk_order_line
 order_line | ix_order_line
 orders     | keyorders
 orders     | ix_orders
 orders     | pk_orders
 stock      | keystock
 stock      | pk_stock
 stock      | ix_stock
 warehouse  | keywarehouse
 warehouse  | pk_warehouse
(21 rows)
```

Figura 5: Índices encontrados

Outra componente que achámos pertinente investigar acerca da sua presença foram as ***Materialized Views***. Neste caso, a base de dados não possui qualquer registo significando a ausência das mesmas.

Para a análise das interrogação iremos utilizar o comando ***EXPLAIN ANALYZE*** tal como visto na componente teórica da UC. Com este comando iremos ser capazes de verificar qual o plano de cada interrogação, bem como tempos de execução e possibilidades de melhoria.

3.1 1ª Interrogação

A 1ª interrogação que escolhemos foi a ***Query* 13**. O objetivo desta *query* consiste em listar os clientes, agrupados e ordenados pelo tamanho das encomendas que realizaram. A relação entre os clientes e o tamanho das suas encomendas é ordenada pelo tamanho e conta o número de clientes que negociaram da mesma forma.

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

O primeiro passo realizado, foi executar o *EXPLAIN ANALYZE*, o que nos permitiu perceber qual a carga efetiva associada à *query* e como o *postgresql* operacionaliza a mesma. De seguida, estão apresentados os resultados obtidos:

```
                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=136167.06..136167.56 rows=200 width=16) (actual time=1248.287..1248.296 rows=46 loops=1)
   Sort Key: (count(*)) DESC, c_orders.c_count DESC
   Sort Method: quicksort  Memory: 27kB
   ->  GroupAggregate  (cost=136134.92..136159.42 rows=200 width=16) (actual time=1247.786..1248.241 rows=46 loops=1)
         Group Key: c_orders.c_count
         ->  Sort  (cost=136134.92..136142.42 rows=3000 width=8) (actual time=1247.765..1247.969 rows=3000 loops=1)
               Sort Key: c_orders.c_count DESC
               Sort Method: quicksort  Memory: 237kB
               ->  Subquery Scan on c_orders  (cost=135901.66..135961.66 rows=3000 width=8) (actual time=1246.191..1247.012 rows=3000 loops=1)
                     ->  HashAggregate  (cost=135901.66..135931.66 rows=3000 width=12) (actual time=1246.187..1246.697 rows=3000 loops=1)
                           Group Key: customer.c_id
                           ->  Hash Right Join  (cost=101031.00..130801.66 rows=1020000 width=8) (actual time=618.234..1079.868 rows=1020000 loops=1)
                                 Hash Cond: ((orders.o_w_id = customer.c_w_id) AND (orders.o_d_id = customer.c_d_id) AND (orders.o_c_id = customer.c_id))
                                 ->  Seq Scan on orders  (cost=0.00..22283.00 rows=142052 width=16) (actual time=0.048..130.201 rows=142346 loops=1)
                                       Filter: (o_carrier_id > 8)
                                       Rows Removed by Filter: 877654
                                 ->  Hash  (cost=78200.00..78200.00 rows=1020000 width=12) (actual time=617.225..617.226 rows=1020000 loops=1)
                                       Buckets: 131072  Batches: 16  Memory Usage: 3772kB
                                       ->  Seq Scan on customer  (cost=0.00..78200.00 rows=1020000 width=12) (actual time=105.395..399.556 rows=1020000 loops=1)
 Planning Time: 1.041 ms
 JIT:
   Functions: 26
   Options: Inlining false, Optimization false, Expressions true, Deforming true
   Timing: Generation 3.050 ms, Inlining 0.000 ms, Optimization 23.147 ms, Emission 79.969 ms, Total 106.166 ms
 Execution Time: 1458.857 ms
(25 rows)
```

Figura 6: Query 13 Explain Analyze Inicial

Através de uma breve análise dos resultados, conseguimos, de forma imediata, detetar que os valores do tempo de execução já se encontram dentro de parâmetros bastante aceitáveis. No entando, decidimos dar uso à ferramenta *explain.dalibo.com*, que nos habilita uma análise minuciosa dos diferentes dados obtidos, para verificar se ainda temos espaço para otimização.

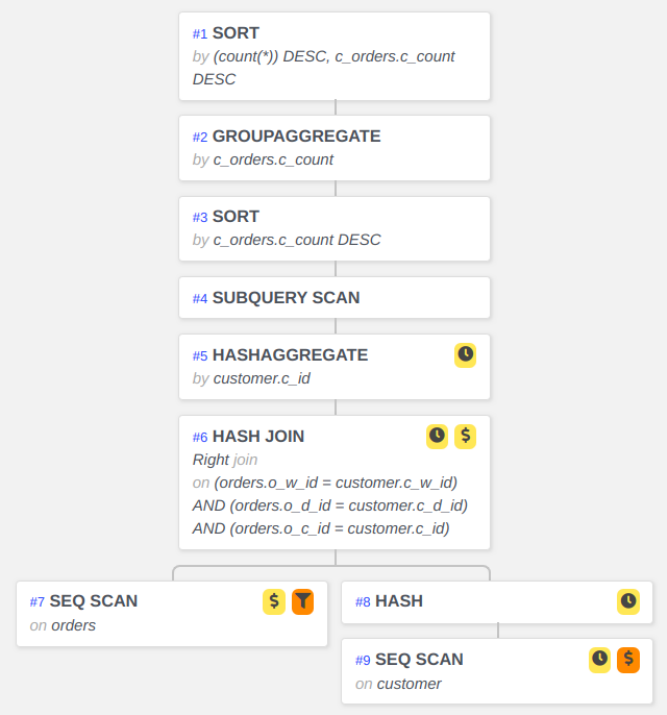
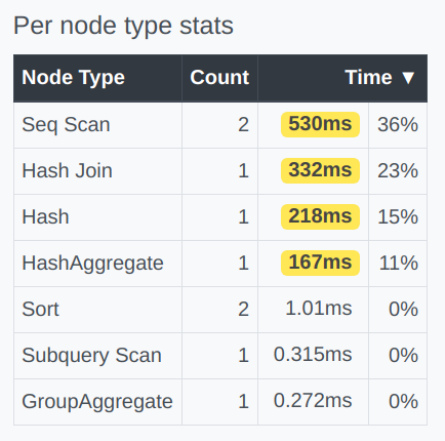


Figura 7: Query 13 Dalibo Plano Geral Inicial

Destes valores, conseguimos destacar vários fatores que poderão estar a afetar a "eficiência"da execução, tal como o hash join que poderia ser evitado, ou então, os sequencial scans nas diferentes tabelas. Mais uma vez, a ferramenta dalibo, demonstrou ser bastante útil, visto que nos permite visualizar as estatísticas do programa de forma a verificar quais as "ações"que estarão a causar maior impacto.

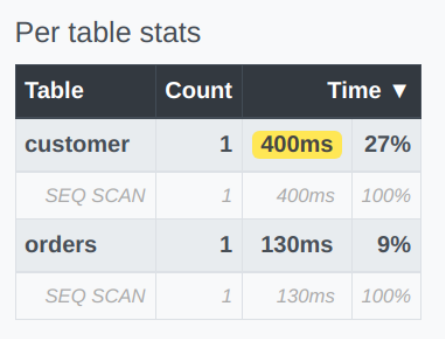


Per node type stats

Node Type	Count	Time ▼	
Seq Scan	2	530ms	36%
Hash Join	1	332ms	23%
Hash	1	218ms	15%
HashAggregate	1	167ms	11%
Sort	2	1.01ms	0%
Subquery Scan	1	0.315ms	0%
GroupAggregate	1	0.272ms	0%

Figura 8: Query 13 Estatísticas Gerais

De facto, os dois sequencial scans estão a ter o maior impacto no tempo de execução, mas ainda podemos recolher mais informação. A seguinte tabela de estatísticas, por exemplo, destaca a tabela dos **customers** como tendo o maior custo de ambas.



Per table stats

Table	Count	Time ▼	
customer	**1**	**400ms**	**27%**
SEQ SCAN	*1*	*400ms*	*100%*
orders	**1**	**130ms**	**9%**
SEQ SCAN	*1*	*130ms*	*100%*

Figura 9: Query 13 Estatísticas das Tabelas

Assim, tendo analisado toda esta informação, decidimos que o melhor método de otimização a ser aplicado nesta query, seria criar um índice para a tabela dos customers. No entando, sendo **c_id** uma chave primária, o indíce já existe. Deparados com este percalço, passamos à criação do índice para a tabela das orders:

```
CREATE INDEX indice_q13o ON orders (o_carrier_id);
```

Os resultados obtidos foram os seguintes:

```
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=127858.04..127858.54 rows=200 width=16) (actual time=1089.065..1089.073 rows=46 loops=1)
   Sort Key: (count(*)) DESC, c_orders.c_count DESC
   Sort Method: quicksort  Memory: 27kB
   ->  GroupAggregate  (cost=127825.90..127850.40 rows=200 width=16) (actual time=1088.649..1089.054 rows=46 loops=1)
         Group Key: c_orders.c_count
         ->  Sort  (cost=127825.90..127833.40 rows=3000 width=8) (actual time=1088.638..1088.805 rows=3000 loops=1)
               Sort Key: c_orders.c_count DESC
               Sort Method: quicksort  Memory: 237kB
               ->  Subquery Scan on c_orders  (cost=127592.64..127652.64 rows=3000 width=8) (actual time=1087.517..1088.214 rows=3000 loops=1)
                     ->  HashAggregate  (cost=127592.64..127622.64 rows=3000 width=12) (actual time=1087.514..1087.929 rows=3000 loops=1)
                           Group Key: customer.c_id
                           ->  Hash Right Join  (cost=103696.33..122492.64 rows=1020000 width=8) (actual time=526.049..919.600 rows=1020000 loops=1)
                                 Hash Cond: ((orders.o_w_id = customer.c_w_id) AND (orders.o_d_id = customer.c_d_id) AND (orders.o_c_id = customer.c_id))
                                 ->  Bitmap Heap Scan on orders  (cost=2665.33..13973.98 rows=142052 width=16) (actual time=10.736..77.060 rows=142346 loops=1)
                                       Recheck Cond: (o_carrier_id > 8)
                                       Heap Blocks: exact=6988
                                       ->  Bitmap Index Scan on ind_q13  (cost=0.00..2629.82 rows=142052 width=0) (actual time=9.731..9.731 rows=142346 loops=1)
                                             Index Cond: (o_carrier_id > 8)
                                 ->  Hash  (cost=78200.00..78200.00 rows=1020000 width=12) (actual time=514.876..514.877 rows=1020000 loops=1)
                                       Buckets: 131072  Batches: 16  Memory Usage: 3772kB
                                       ->  Seq Scan on customer  (cost=0.00..78200.00 rows=1020000 width=12) (actual time=12.698..302.717 rows=1020000 loops=1)
 Planning Time: 0.597 ms
 JIT:
   Functions: 26
   Options: Inlining false, Optimization false, Expressions true, Deforming true
   Timing: Generation 2.235 ms, Inlining 0.000 ms, Optimization 0.647 ms, Emission 11.665 ms, Total 14.547 ms
 Execution Time: 1091.483 ms
(27 rows)
```

Figura 10: Query 13 Explain Analyze com Index

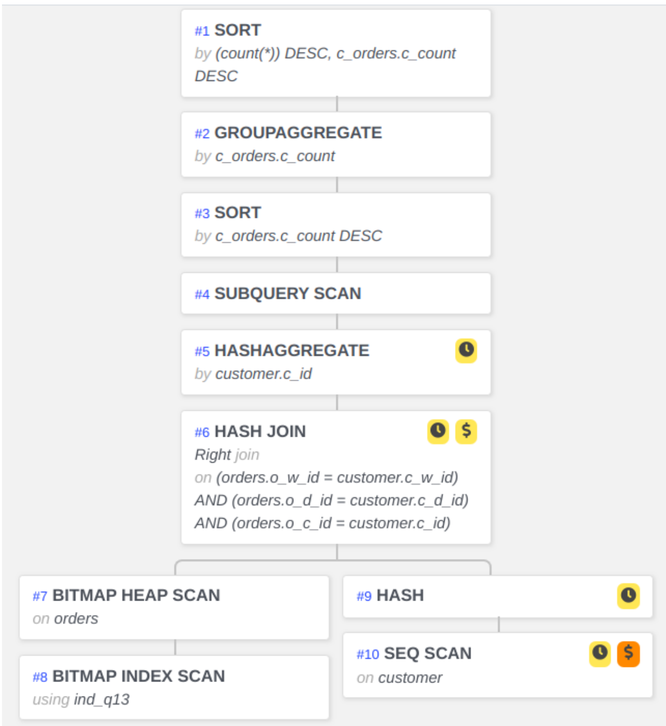


Figura 11: Query 13 Dalibo Index

Podemos verificar que a criação do índice resultou numa melhoria significativa em relação aos resultados anteriores. Porém, tal como tínhamos verificado, continuam a existir certas ações que podem ser melhoradas.

Previamente, tínhamos mencionado o **hash join** como um desses problemas e acreditamos que a criação de uma materialized view será uma boa solução. Esta solução, permite-nos consolidar numa só tabela informação relativa a encomendas e os clientes que as realizaram. A sua criação vai reduzir o número de travessias nas tabelas customers e orders, permitindo evitar tanto o scan da tabela customers como o hash join. A criação do índice não se torna redundante visto que será utilizado durante a criação desta nova tabela.

```
CREATE MATERIALIZED VIEW mat_q13 AS
        select c_id,
        count(o_id) from customer left outer join orders on( c_w_id = o_w_id
            and c_d_id = o_d_id
                    and c_id = o_c_id
                and o_carrier_id > 8)
            group by c_id;
```

Assim sendo, em vez de se executar a interrogação analítica conforme apresentado anteriormente, basta a alterarmos para tirar proveito desta materialized view, passando a tomar a seguinte forma:

```
EXPLAIN ANALYZE select c_count, count(*) as custdist
                from (select * from mat_q13) as c_orders(c_id, c_count) group by c_count
                order by custdist desc, c_count desc;
```

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Sort  (cost=63.73..63.85 rows=46 width=16) (actual time=0.823..0.826 rows=46 loops=1)
   Sort Key: (count(*)) DESC, mat_q13.count DESC
   Sort Method: quicksort  Memory: 27kB
   ->  HashAggregate  (cost=62.00..62.46 rows=46 width=16) (actual time=0.788..0.795 rows=46 loops=1)
         Group Key: mat_q13.count
         ->  Seq Scan on mat_q13  (cost=0.00..47.00 rows=3000 width=8) (actual time=0.011..0.270 rows=3000 loops=1)
 Planning Time: 0.151 ms
 Execution Time: 0.883 ms
(8 rows)
```

Figura 12: Query 13 Explain Analyze com Index e Materialized Views

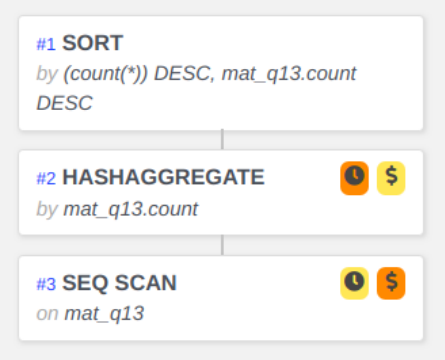


Figura 13: Query 13 Dalibo Index e Materialized View

A criação da materialized view mostrou ser extremamente eficaz, sendo que, como realiza pré-processamento da tabela com as informações necessárias à interrogação, ficam excluídas as ações com maior custo do tempo de execução e acabamos por obter uma melhoria de 99%.

Resultados Finais:

	Tempo médio de resposta (ms)	Melhoria face ao inicial
Inicial	1458	-
Index	1091	25%
Index e MV	0.883	99%

Tabela 3: Tempos de resposta obtidos

3.2 2ª Interrogação

A 2ª interrogação escolhida foi a ***Query* 16**. Esta consiste em descobrir quantos fornecedores são capazes de fornecer itens com determinados atributos, por ordem decrescente. O resultado é agrupado pelo identificador do item.

```
select      i_name,
            substr(i_data, 1, 3) as brand,
            i_price,
            count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
from        stock, item
where       i_id = s_i_id
            and i_data not like 'zz%'
            and (mod((s_w_id * s_i_id),10000) not in
```

```
                        (select su_suppkey
                         from supplier
                         where su_comment like '%bad%'))
group by i_name, substr(i_data, 1, 3), i_price
order by supplier_cnt desc;
```

Após executarmos a interrogação usando o *EXPLAIN ANALYZE* obtivemos o seguinte resultado:

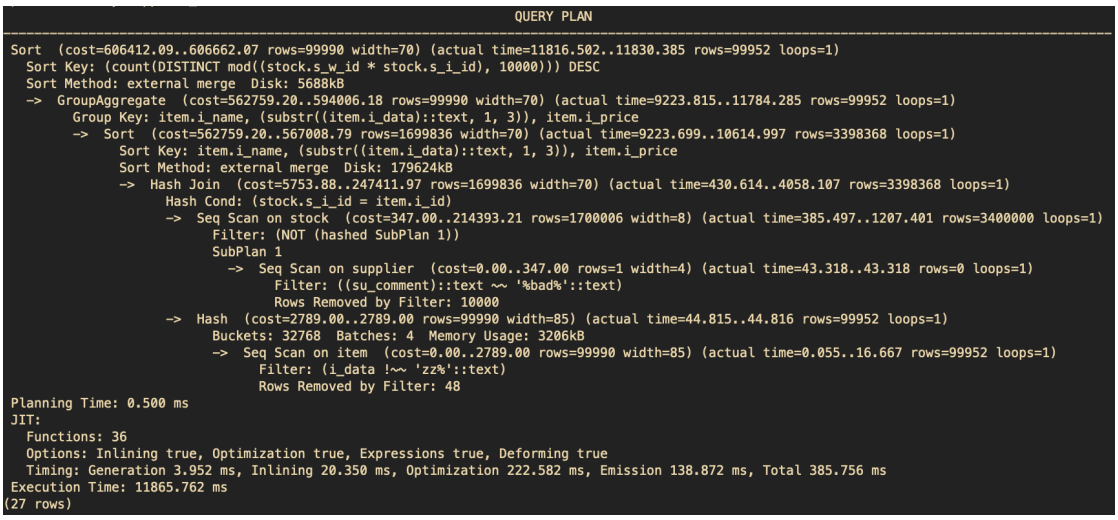


Figura 14: Explain Analyze Inicial

O uso da ferramenta *explain.dalibo.com* facilitou-nos a análise das queries mostrando-nos o custo de cada operação. De seguida podemos ver o resultado para esta versão da query:

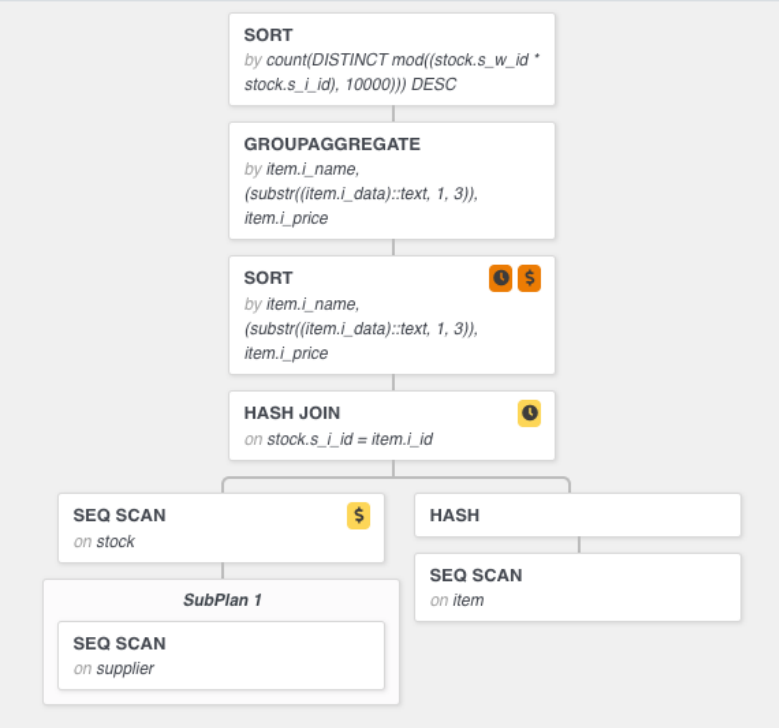


Figura 15: Plano inicial de execução

Tendo em conta que as transações realizadas durante o *benchmark* não alteram qualquer informação utilizada por esta interrogação, concluímos que poderia ser aplicada uma **materialized view** de forma a melhorar o desempenho.

Assim, criou-se uma materialized view *stock_item_mv*, da seguinte forma:

```
create materialized view stock_item_mv as
    select i_price, i_data, i_id, i_name, mod((s_w_id * s_i_id),10000) as calc
        from item inner join stock on i_id = s_i_id;
```

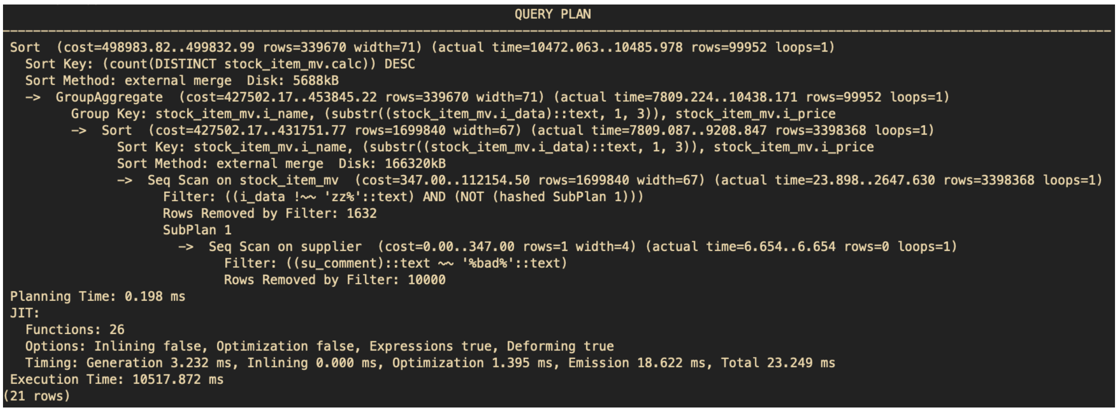
Esta vista materializada permite consolidar numa só tabela informação relativa a preços e nomes de itens. Pretende-se com esta *materialized view* reduzir o número de travessias nas tabelas stock e item, por consolidarmos a sua união com as informações relevantes para a interrogação analítica, o que permitirá evitar o scan da tabela item e o custoso hash join.

Assim sendo, em vez de se executar a interrogação analítica conforme apresentado anteriormente, esta é alterada para tirar proveito desta materialized view,

passando a tomar a seguinte forma:

```
select i_name, substr(i_data,1,3) as brand, i_price,
    count(distinct(calc)) as supplier_cnt
    from stock_item_mv where i_data not like 'zz%'
        and (calc not in (select su_suppkey from supplier
            where su_comment like '%bad%'))
    group by i_name, substr(i_data,1,3), i_price
    order by supplier_cnt desc;
```

Os resultados obtidos foram:



```
                                                     QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
Sort  (cost=498983.82..499832.99 rows=339670 width=71) (actual time=10472.063..10485.978 rows=99952 loops=1)
  Sort Key: (count(DISTINCT stock_item_mv.calc)) DESC
  Sort Method: external merge  Disk: 5688kB
  ->  GroupAggregate  (cost=427502.17..453845.22 rows=339670 width=71) (actual time=7809.224..10438.171 rows=99952 loops=1)
        Group Key: stock_item_mv.i_name, (substr((stock_item_mv.i_data)::text, 1, 3)), stock_item_mv.i_price
        ->  Sort  (cost=427502.17..431751.77 rows=1699840 width=67) (actual time=7809.087..9208.847 rows=3398368 loops=1)
              Sort Key: stock_item_mv.i_name, (substr((stock_item_mv.i_data)::text, 1, 3)), stock_item_mv.i_price
              Sort Method: external merge  Disk: 166320kB
              ->  Seq Scan on stock_item_mv  (cost=347.00..112154.50 rows=1699840 width=67) (actual time=23.898..2647.630 rows=3398368 loops=1)
                    Filter: ((i_data !~~ 'zz%'::text) AND (NOT (hashed SubPlan 1)))
                    Rows Removed by Filter: 1632
                    SubPlan 1
                      ->  Seq Scan on supplier  (cost=0.00..347.00 rows=1 width=4) (actual time=6.654..6.654 rows=0 loops=1)
                            Filter: ((su_comment)::text ~~ '%bad%'::text)
                            Rows Removed by Filter: 10000
Planning Time: 0.198 ms
JIT:
  Functions: 26
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 3.232 ms, Inlining 0.000 ms, Optimization 1.395 ms, Emission 18.622 ms, Total 23.249 ms
Execution Time: 10517.872 ms
(21 rows)
```

Figura 16: Explain analyze com vista materializada

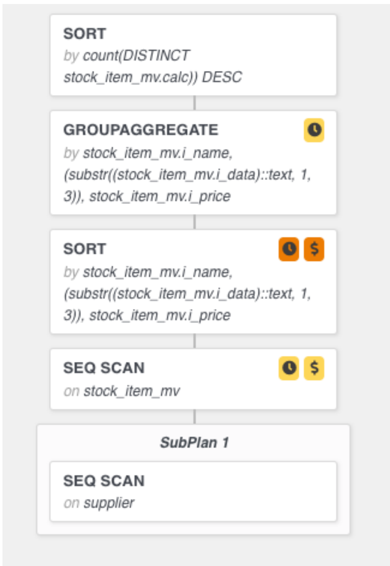
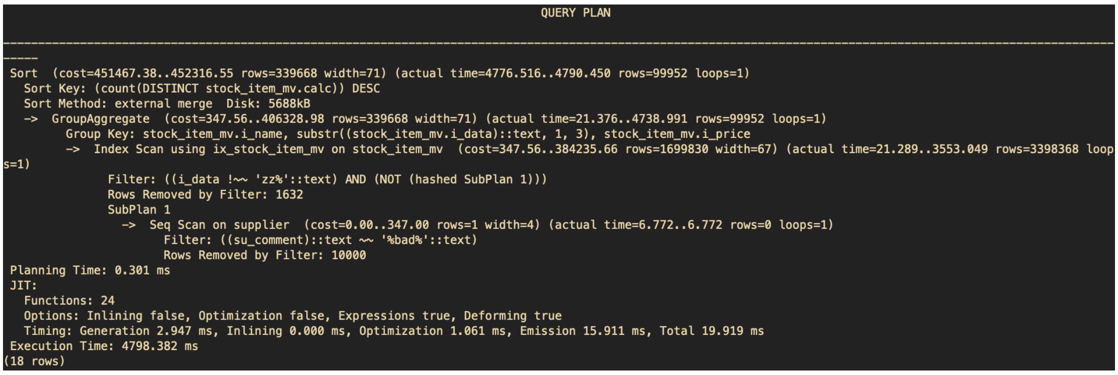


Figura 17: Plano de execução com vista materializada

Verificamos ainda que é feito um scan sequencial, seguido de uma ordenação, de acordo com algumas colunas. Por isso, decidimos criar um índice composto por essas mesmas colunas, de forma a facilitar a ordenação. Foi criado da seguinte forma:

```
create index ix_stock_item_mv
on stock_item_mv(i_name,(substr((i_data)::text,1,3)),i_price);
```

Os resultados obtidos foram:



```
                                                                    QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------------------
-----
 Sort  (cost=451467.38..452316.55 rows=339668 width=71) (actual time=4776.516..4790.450 rows=99952 loops=1)
   Sort Key: (count(DISTINCT stock_item_mv.calc)) DESC
   Sort Method: external merge  Disk: 5688kB
   ->  GroupAggregate  (cost=347.56..406328.98 rows=339668 width=71) (actual time=21.376..4738.991 rows=99952 loops=1)
         Group Key: stock_item_mv.i_name, substr((stock_item_mv.i_data)::text, 1, 3), stock_item_mv.i_price
         ->  Index Scan using ix_stock_item_mv on stock_item_mv  (cost=347.56..384235.66 rows=1699830 width=67) (actual time=21.289..3553.049 rows=3398368 loop
s=1)
               Filter: ((i_data !~~ 'zz%'::text) AND (NOT (hashed SubPlan 1)))
               Rows Removed by Filter: 1632
               SubPlan 1
                 ->  Seq Scan on supplier  (cost=0.00..347.00 rows=1 width=4) (actual time=6.772..6.772 rows=0 loops=1)
                       Filter: ((su_comment)::text ~~ '%bad%'::text)
                       Rows Removed by Filter: 10000
 Planning Time: 0.301 ms
 JIT:
   Functions: 24
   Options: Inlining false, Optimization false, Expressions true, Deforming true
   Timing: Generation 2.947 ms, Inlining 0.000 ms, Optimization 1.061 ms, Emission 15.911 ms, Total 19.919 ms
 Execution Time: 4798.382 ms
(18 rows)
```

Figura 18: Explain analyze com vista materializada e índice

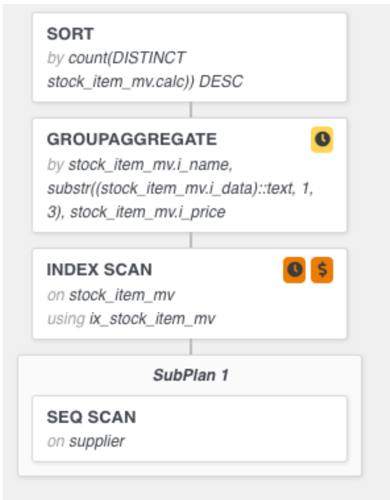


Figura 19: Plano de execução com vista materializada e índice

Na tabela 4 podemos encontrar todos os tempos médios de resposta para cada versão da query e a sua respetiva melhoria em relação à versão inicial.

	Tempo médio de resposta (ms)	Melhoria face ao inicial
Inicial	11865	-
Materialized View	10517	11%
MV e Index	4798	60%

Tabela 4: Tempos de resposta obtidos

3.3 3ª Interrogação

A 3ª interrogação escolhida foi a ***Query* 18**. Esta tem o objetivo de apresentar o *ranking* de todos os clientes com um total de encomendas com valor superior a uma dada quantia de dinheiro.

```
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
```

O primeiro passo foi executar o *EXPLAIN ANALYZE* para se perceber qual a carga efetiva associada a *query* e para se perceber como o sistema de base de dados iria operacionalizar a mesma.

```
                                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------
Sort  (cost=3173551.96..3180787.81 rows=2894338 width=77) (actual time=33206.559..33487.765 rows=306000 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC, orders.o_entry_d
  Sort Method: external merge  Disk: 20968kB
  ->  Finalize GroupAggregate  (cost=1292943.34..2605705.00 rows=2894338 width=77) (actual time=30529.951..32864.207 rows=306000 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: 714000
        ->  Gather Merge  (cost=1292943.34..2254766.55 rows=7235844 width=77) (actual time=25115.257..31125.608 rows=1020000 loops=1)
              Workers Planned: 2
              Workers Launched: 2
              ->  Partial GroupAggregate  (cost=1291943.32..1418570.59 rows=3617922 width=77) (actual time=24828.881..29547.498 rows=340000 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=1291943.32..1300988.12 rows=3617922 width=48) (actual time=24828.774..27530.844 rows=2890261 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: 185656kB
                          Worker 0:  Sort Method: external merge  Disk: 186288kB
                          Worker 1:  Sort Method: external merge  Disk: 184856kB
                          ->  Nested Loop  (cost=82593.93..675236.87 rows=3617922 width=48) (actual time=2224.744..12910.602 rows=2890261 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=82593.50..108441.38 rows=425111 width=53) (actual time=2224.072..2806.402 rows=340000 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..13783.00 rows=425000 width=28) (actual time=0.102..133.258 rows=340000 loops=3)
                                      ->  Parallel Hash  (cost=72250.00..72250.00 rows=425000 width=29) (actual time=1830.391..1830.392 rows=340000 loops=3)
                                            Buckets: 65536  Batches: 32  Memory Usage: 2560kB
                                            ->  Parallel Seq Scan on customer  (cost=0.00..72250.00 rows=425000 width=29) (actual time=1248.133..1563.034 rows=340000 loops=3)
                                ->  Index Scan using pk_order_line on order_line  (cost=0.43..1.20 rows=9 width=15) (actual time=0.021..0.026 rows=9 loops=1020000)
                                      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: 1.714 ms
JIT:
  Functions: 88
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 15.336 ms, Inlining 621.614 ms, Optimization 1810.183 ms, Emission 1308.719 ms, Total 3755.852 ms
Execution Time: 33632.059 ms
```

Figura 20: Explain Analyze Inicial

Tal como se pode ver na figura 20, o tempo total de execução foi de **33632.059 ms**. Ora, tendo em conta o plano de execução e considerando que este tipo de interrogação seria para ser atualizado uma/duas vezes por dia e que a sua execução deveria ser extremamente rápida, o primeiro passo que se achou por bem foi criar uma *Materialized View* de forma a albergar os dados da *query*.

```sql
create materialized view q18_view as
select c_last, c_id, o_id, o_entry_d, o_ol_cnt, sum(ol_amount) as total_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;
```

Nesta *materialized view* decidiu-se incluir a agregação por *SUM(ol_ amount)* já de forma a tornar a *query* mais rápida na execução, mesmo que dificultando o processo de atualização da mesma, pois a execução da *query* é mais crítica que a atualização da mesma (sendo que este processo seria feito de forma periódica, como 1/2 vezes por dia). Este *refresh* pode ser feito através de *REFRESH MATERIALZED VIEW q18_view*. Tal como se pode ver na figura 21, a criação da *materialized view* fez diminuir drasticamente o tempo de execução da *query* que passou a ser de **839.985 ms**.

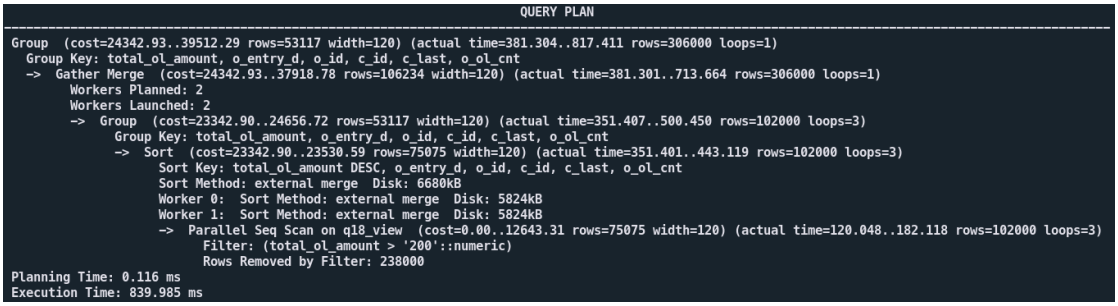


Figura 21: Explain Analyze depois da Materialized View

Para esta análise, foi necessário modificar ligeiramente a *query* de forma a utilizar a *materialized view*, pelo que a *query* utilizada passou a ser:

```
select c_last, c_id, o_id, o_entry_d, o_ol_cnt, total_ol_amount
from q18_view
group by o_id, c_id, c_last, o_entry_d, o_ol_cnt, total_ol_amount
having total_ol_amount > 200
order by total_ol_amount desc, o_entry_d;
```

Por fim, como se reparou que a operação de *sort* era extremamente custosa, decidiu-se criar um índice para colmatar esta dificuldade. O índice criado para colmatar a *key* de *sort* mostrada na figura 21 foi o seguinte:

```
create index q18_ind_order_by
on q18_view(total_ol_amount desc, o_entry_d, o_id, c_id, c_last, o_ol_cnt);
```

A criação deste índice teve um impacto positivo no plano de execução, pois como se pode ver na figura 22, reduziu o tempo de execução de **839.985 ms** para **349.368 ms**.

Figura 22: Explain Analyze depois do Índice

Em jeito de conclusão, com a aplicação de uma *materialized view* e um ín-

dice conseguiu-se reduzir o tempo de execução da *query* de **33632.059 ms** para **349.368 ms**, o que significa que se conseguiu otimizar a *query* para ser cerca de 100 vezes mais rápida.

3.4 4ª Interrogação

A 4ª interrogação que escolhemos foi a **Query 20**. O primeiro passo consistiu em adaptar a interrogação para que esta funcione corretamente. Para isso efetuámos as seguintes adaptações: corrigimos o nome da tabela *order_line* e o critério *n_name* teve de ser convertido para maiúsculas. O código sql referente à interrogação é o seguinte:

```
select      su_name, su_address
from        supplier, nation
where       su_suppkey in
           (select  mod(s_i_id * s_w_id, 10000)
           from     stock, order_line
           where    s_i_id in
                        (select i_id
                         from item
                         where i_data like 'c%')
                    and ol_i_id=s_i_id
                    and ol_delivery_d > '2010-05-23 12:00:00'
           group by s_i_id, s_w_id, s_quantity
           having   2*s_quantity > sum(ol_quantity))
    and su_nationkey = n_nationkey
    and n_name = 'GERMANY'
order by su_name;
```

Após executarmos a interrogação usando o *EXPLAIN ANALYZE* obtivemos o seguinte resultado:

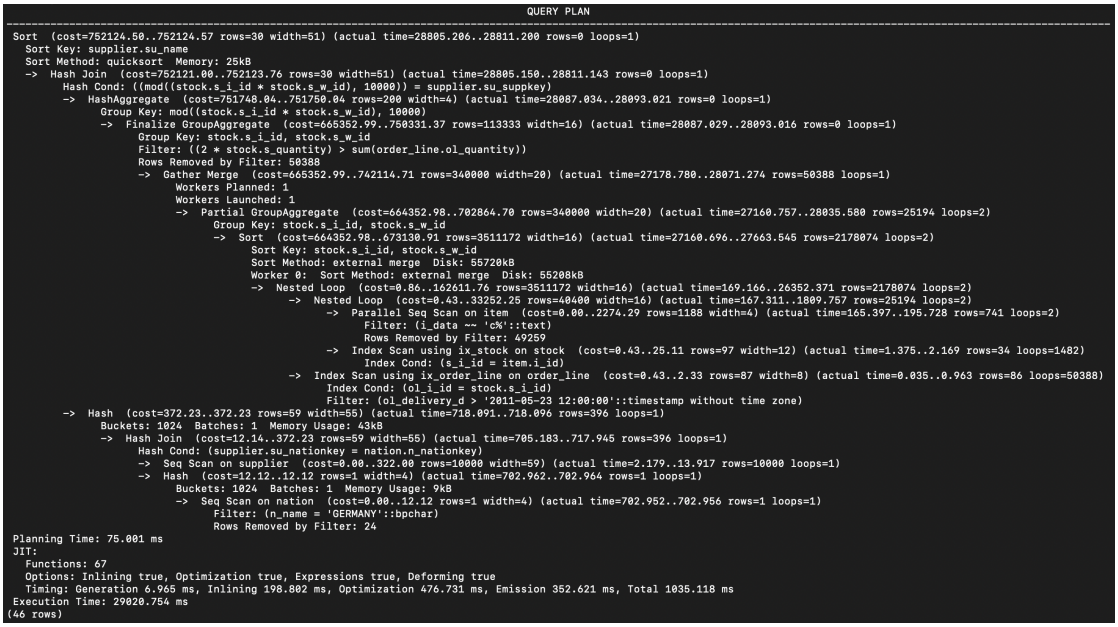


Figura 23: Query 4 - Explain Analyze Inicial

Ao analisar o resultado obtido conseguimos ver todas as decisões tomadas pelo *PostgreSQL* para a execução da interrogação, bem como os respetivos custos de cada operação e tempo de execução. Neste caso, o tempo de execução é de **29020.754 ms**, o que é bastante elevado, sendo o nosso objetivo baixar este valor.

A primeira grande melhoria que optámos por implementar foi a utilização de ***Materialized Views***, isto porque existe uma instrução que no resultado do *Explain Analyze* possui um custo bastante elevado. A instrução que tem esse custo elevado é a seguinte:

```
select  mod(s_i_id * s_w_id, 10000)
from      stock, order_line
where     s_i_id in
    (select i_id
    from item
    where i_data like 'co%')
and ol_i_id=s_i_id
and ol_delivery_d > '2010-05-23 12:00:00'
group by s_i_id, s_w_id, s_quantity
having    2*s_quantity > sum(ol_quantity);
```

Sendo que esta parte possui um custo tão elevado, considerámos que seria

uma boa ideia não a correr sempre que a interrogação é feita, pelo que optámos por substituí-la pela *Materialized View* seguinte:

```
CREATE MATERIALIZED VIEW mv_interrogacao4 AS (
    select  mod(s_i_id * s_w_id, 10000)
    from     stock, order_line
    where    s_i_id in
            (select i_id
        from item
        where i_data like 'co%')
    and ol_i_id=s_i_id
    and ol_delivery_d > '2010-05-23 12:00:00'
    group by s_i_id, s_w_id, s_quantity
    having   2*s_quantity > sum(ol_quantity)
);
```

Com a implementação desta *View*, passámos a ter a instrução previamente executada, sendo apenas necessário efetuar o *REFRESH* da *Materialized View*. Com isto, a nossa interrogação inicial passa a ter o seguinte formato:

```
select          su_name, su_address
from            supplier, nation
where           su_suppkey in
               (select * from mv_interrogacao4)
     and su_nationkey = n_nationkey
     and n_name = 'GERMANY'
order by su_name;
```

Com estas alterações feitas, procedemos à nova execução da interrgoação, obtendo o seguinte plano:

```
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=419.22..419.25 rows=15 width=51) (actual time=2.564..2.567 rows=0 loops=1)
   Sort Key: supplier.su_name
   Sort Method: quicksort  Memory: 25kB
   ->  Hash Join  (cost=58.51..418.92 rows=15 width=51) (actual time=2.535..2.537 rows=0 loops=1)
         Hash Cond: (supplier.su_suppkey = mv_interrogacao4.mod)
         ->  Hash Join  (cost=12.14..372.23 rows=59 width=55) (actual time=2.507..2.508 rows=1 loops=1)
               Hash Cond: (supplier.su_nationkey = nation.n_nationkey)
               ->  Seq Scan on supplier  (cost=0.00..322.00 rows=10000 width=59) (actual time=1.684..1.687 rows=33 loops=1)
               ->  Hash  (cost=12.12..12.12 rows=1 width=4) (actual time=0.798..0.798 rows=1 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 9kB
                     ->  Seq Scan on nation  (cost=0.00..12.12 rows=1 width=4) (actual time=0.783..0.786 rows=1 loops=1)
                           Filter: (n_name = 'GERMANY'::bpchar)
                           Rows Removed by Filter: 24
         ->  Hash  (cost=43.88..43.88 rows=200 width=4) (actual time=0.005..0.006 rows=0 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 8kB
               ->  HashAggregate  (cost=41.88..43.88 rows=200 width=4) (actual time=0.005..0.005 rows=0 loops=1)
                     Group Key: mv_interrogacao4.mod
                     ->  Seq Scan on mv_interrogacao4  (cost=0.00..35.50 rows=2550 width=4) (actual time=0.002..0.002 rows=0 loops=1)
 Planning Time: 11.645 ms
 Execution Time: 3.730 ms
(20 rows)
```

Figura 24: Query 4 - Explain Analyze com Materialized Views

Como podemos ver, o tempo de execução baixou para **3.730 ms**. O benefício em termos de custo das *Materialized Views* foi excelente neste caso. Esta melhoria por si só é notável, no entanto achámos que podemos tentar obter uma performance um pouco melhor, isto porque no plano de execução existe uma leitura sequencial na tabela *supplier* com um custo pouco elevado comparativamente com a restante interrogação. Assim sendo criámos o seguinte índice:

```sql
CREATE INDEX indice_sup2 ON supplier (su_nationkey);
```

Correndo a interrogação novamente, mas recorrendo a este índice, constatámos algo interessante: a utilização de índices prejudicou a nossa performance.

```
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=300.54..300.80 rows=102 width=51) (actual time=3.733..3.736 rows=0 loops=1)
   Sort Key: supplier.su_name
   Sort Method: quicksort  Memory: 25kB
   ->  Hash Join  (cost=57.76..297.14 rows=102 width=51) (actual time=3.702..3.704 rows=0 loops=1)
         Hash Cond: (supplier.su_suppkey = mv_interrogacao4.mod)
         ->  Nested Loop  (cost=11.38..248.58 rows=400 width=55) (actual time=3.663..3.664 rows=1 loops=1)
               ->  Seq Scan on nation  (cost=0.00..1.31 rows=1 width=4) (actual time=0.759..0.759 rows=1 loops=1)
                     Filter: (n_name = 'GERMANY'::bpchar)
                     Rows Removed by Filter: 7
               ->  Bitmap Heap Scan on supplier  (cost=11.38..243.27 rows=400 width=59) (actual time=2.895..2.896 rows=1 loops=1)
                     Recheck Cond: (su_nationkey = nation.n_nationkey)
                     Heap Blocks: exact=1
                     ->  Bitmap Index Scan on indice_sup2  (cost=0.00..11.29 rows=400 width=0) (actual time=2.075..2.075 rows=396 loops=1)
                           Index Cond: (su_nationkey = nation.n_nationkey)
         ->  Hash  (cost=43.88..43.88 rows=200 width=4) (actual time=0.008..0.009 rows=0 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 8kB
               ->  HashAggregate  (cost=41.88..43.88 rows=200 width=4) (actual time=0.008..0.008 rows=0 loops=1)
                     Group Key: mv_interrogacao4.mod
                     ->  Seq Scan on mv_interrogacao4  (cost=0.00..35.50 rows=2550 width=4) (actual time=0.006..0.006 rows=0 loops=1)
 Planning Time: 15.122 ms
 Execution Time: 4.909 ms
(21 rows)
```

Figura 25: Query 4 - Explain Analyze com Materialized Views e Índices

Assim sendo, achámos que para esta interrogação os índices não serão usados, tendo apenas sido utilizadas *Materialized Views*. Considerámos que a melhoria que conseguimos foi significativamente elevada, sendo necessário encontrar um balanço entre o uso da *Materialized View* e a frequência com que esta é atualizada. Dado que o tempo de execução desceu de **29020.754 ms** para **3.730 ms**, ficámos com uma melhor noção do poder da afinação de interrogações num sistema.

4 Otimização do PostgreSQL

Nesta secção, veremos que impacto a mudança de algumas configurações do *PostgreSQL* irá ter no desempenho do nosso *Benchmark*.

Para a escolha das configurações a serem alteradas, seguindo o conselho do docente, optámos por escolher as que são relativas a transações, pois, foram as que tiveram mais foco durante as aulas teórico-práticas.

Deste modo, para as configurações escolhidas, testámos cada uma delas de forma isolada para a mesma configuração de referência e também para o mesmo estado da base de dados.

No final desta etapa de testes de configurações, iremos fazer uma análise com o objetivo de determinar os parâmetros a considerar na otimização efetiva do *PostgreSQL*.

4.1 Settings

Nesta secção encontram-se as principais configurações relativas a transações ao nível do *Write Ahead Log*, permitindo-nos alterar o comportamento das operações de escrita que são feitas neste.

4.1.1 fsync

O parâmetro *fsync*, quando ativado, tentará garantir que os *updates* estão escritos no disco, fazendo chamadas de funções de sincronização, garantindo assim que a base de dados é recuperável em caso de falha. No entanto, com o intuito de melhorar o desempenho, desativámos este parâmetro, de modo a não ocupar o CPU com processos aparentemente "desnecessários".

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
on (default)	410.624	0.0072	0.0170
off	507.391	0.0027	0.0079

Tabela 5: Estudo do parâmetro *fsync*

Apesar desta opção aumentar significativamente o desempenho, não deve ser utilizada, pelo facto de passarmos a comprometer a integridade dos dados. Sendo assim, esta opção foi usada apenas para fins de aprendizagem e não para otimização efetiva do servidor de base de dados *PostgreSQL*.

4.1.2 wal_level

A quantidade de informação que é escrita nos logs é determinada pelo parâmetro *wal_level*. Assim, optámos por realizar testes para os seguintes valores:

- replica - é o valor *default* e acrescenta a informação essencial à execução de *queries* de leitura e ao processo de arquivamento de *logs*.
- minimal - que remove todos os *logs*, exceto as informações necessárias para recuperar de uma falha ou interrupção do servidor.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
replica (default)	410.624	0.0072	0.0170
minimal	435.735	0.0057	0.0134

Tabela 6: Estudo do parâmetro *wal_level*

4.1.3 commit_delay

Com esta opção é possível adicionar um atraso antes do *flush* do WAL ser iniciado. Isto pode aumentar o *throughput*, permitindo que mais transações façam *commit* num só WAL *flush*, que vem com um custo em aumento da latência.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
0ms (default)	410.624	0.0072	0.0170
100ms	367.173	0.0069	0.0234
250ms	396.369	0.0063	0.0193
500ms	414.084	0.0074	0.0169
1000ms	415.127	0.0077	0.0171

Tabela 7: Estudo do parâmetro *commit_delay*

4.1.4 commit_siblings

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

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
5 (default)	410.624	0.0072	0.0170
10	419.241	0.0071	0.0168
15	417.104	0.0075	0.0169
20	413.428	0.0079	0.0173

Tabela 8: Estudo do parâmetro *commit_siblings*

4.2 Checkpoints

Os *checkpoints* são pontos no registo transacional em que todos os ficheiros de dados foram atualizados para refletir a informação presente nos registos, pelo que a alteração das configurações dos checkpoints podem trazer excelente otimização tanto para os **Tempos de Resposta** como para o **Throughput** de pedidos.

Um checkpoint começa sempre que acaba um **checkpoint_timeout** ou se o **max_wal_size** estiver quase a ser excedido.

4.2.1 checkpoint_timeout

O parâmetro **checkpoint_timeout** permite-nos modificar o tempo máximo, em segundos, entre o qual vão ocorrer checkpoints. Reduzir este valor provoca maior ocorrência de checkpoints. Neste parâmetro tentámos fazer um balanço entre a frequência com que os checkpoints são feitos e o *Throughput* resultante.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
2 min	375.223	0.0067	0.0161
5 min (default)	410.624	0.0072	0.0170
10 min	410.541	0.0064	0.0159
15 min	411.881	0.0065	0.0162

Tabela 9: Estudo do parâmetro *checkpoint_timeout*

4.2.2 checkpoint_completion_target

O parâmetro **checkpoint_completion_target** determina a fração do tempo total do **checkpoint_timeout** que já deve ter sido completada para que um checkpoint seja dado como concluído.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
0	412.813	0.0060	0.0144
0.2	404.587	0.0075	0.0206
0.5 (default)	410.624	0.0072	0.0170
0.7	424.172	0.0071	0.0165
1	410.521	0.0069	0.0166

Tabela 10: Estudo do parâmetro *checkpoint_completion_target*

4.2.3 max_wal_size

O parâmetro **max_wal_size** determina o tamanho máximo do crescimento que o WAL pode sofrer entre checkpoints automáticos. A redução do valor deste parâmetro tem como consequência um aumento da frequência dos checkpoints.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
1 GB (default)	410.624	0.0072	0.0170
2 GB	433.278	0.0058	0.0146
4 GB	459.586	0.0036	0.0104

Tabela 11: Estudo do parâmetro *max_wal_size*

4.2.4 min_wal_size

O parâmetro **min_wal_size** define um valor mínimo de espaço para o WAL. Enquanto o tamanho do WAL se manter abaixo do valor selecionado, os ficheiros WAL são reciclados para uso futuro num checkpoint, em vez de removidos. Pode ser utilizado para assegurar que existe espaço WAL suficiente para lidar com picos de trabalho, por exemplo.

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
80 MB (default)	410.624	0.0072	0.0170
160 MB	405.307	0.0076	0.0206
320 MB	412.942	0.0063	0.0154
640 MB	411.865	0.0062	0.0149

Tabela 12: Estudo do parâmetro *min_wal_size*

4.3 Combinação de Otimizações

Nesta secção, vamos apresentar o resultado da análise que fizemos dos diferentes parâmetros e da sua utilidade. Para isso, combinámos as opções com melhores resultados no desempenho da carga transacional, sendo o objetivo encontrar a melhor configuração possível do *PostgreSQL* para o nosso problema. Iremos começar por usar a combinação das melhores opções das *settings*, de seguida o mesmo para os *checkpoints* e, finalmente, a conjunção de ambas.

4.3.1 Settings

Nesta primeira combinação, selecionámos as opções da secção de *settings* que mais otimizaram o servidor de Base de Dados *PostgreSQL*, tendo em conta os valores dos testes realizados. Essas opções foram as seguintes:

- wal_level = minimal
- commit_delay = 1000
- commit_siblings = 10

Os resultados obtidos foram os seguintes:

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
configuração de referência	410.624	0.0072	0.0170
otimização	431.027	0.0051	0.0171

Tabela 13: Combinação das otimizações da secção *Settings*

Os resultados obtidos por esta combinação representam uma melhoria de performance (*throughput* e *response time*), o *abort rate* não sofre alterações relevantes, pelo que se pode considerar que a combinação tem sucesso na tarefa de otimizar o servidor de base de dados.

4.3.2 Checkpoints

À semelhança da combinação anterior, aqui o objetivo foi selecionar as melhores opções de configuração relativas à secção de *checkpoints*, tendo sempre em

consideração a configuração de referêcia do *PostgreSQL*. Analisando os valores obtidos, decidiu-se combinar as seguintes configurações:

- checkpoint_completion_target = 0.7
- max_wal_size = 4GB

Os resultados obtidos foram os seguintes:

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
configuração de referência	410.624	0.0072	0.0170
otimização	447.399	0.0043	0.0162

Tabela 14: Combinação das otimizações da secção *Checkpoints*

Mais uma vez, comparando com a configuração de referência, o processo de otimização teve sucesso, no sentido em que o *throughput* aumentou e tanto *response time* como o *abort rate* diminuíram.

4.3.3 Final

Por fim, como tentativa de otimização final, decidiu-se conciliar as duas combinações anteriores, ou seja, criar uma combinação final de configurações que englobassem as combinações de otimização dos *settings* e dos *checkpoints*, pelo que obtivemos a seguinte combinação de configurações:

- wal_level = minimal
- commit_delay = 1000
- commit_siblings = 10
- checkpoint_completion_target = 0.7
- max_wal_size = 4GB

Os resultados obtidos com esta combinação final foram os seguintes:

Opção	Throughput (tx/s)	Response Time (s)	Abort Rate (%)
configuração de referência	410.624	0.0072	0.0170
otimização final	481.001	0.0026	0.0123

Tabela 15: Resultados relativos à melhor combinação de otimizações

Assim, comparando com a configuração de referência, pode-se concluir que com esta combinação de configurações, de facto, conseguiu-se obter melhores resultados:

- Verificámos um **aumento** do *throughput* de 410.624 tx/s para 481.001 tx/s;
- Uma **diminuição** do *response time* de 0.0072 s para 0.0026 s;
- E uma **diminuição** o *abort rate* de 0.0170% para 0.0123%.

Em jeito de conclusão deste capítulo, isto significa que a tarefa de otimizar o servidor de Base de Dados *PostgreSQL*, tendo em conta a carga transacional do TPC-C, foi realizada com sucesso.

5 Conclusões

Com este trabalho foi feita uma investigação prática relativamente ao desempenho de um sistema de gestão de bases de dados. Tendo em conta os objetivos definidos no enunciado, foi possível alcançar melhorias no desempenho da base de dados, tanto ao nível da carga transacional como também ao nível do tempo de resposta das interrogações analíticas.

A fase inicial teve como objetivo encontrar uma configuração de referência, sendo que esta etapa foi bastante trabalhosa, pois foram necessários muitos testes até se chegar aos valores tanto das configurações da máquina, bem como do número de warehouses e clientes.

Na fase de otimização das queries analíticas, começou-se por analisar o plano das queries, e tentar perceber possíveis otimizações. Foi na inserção de views materializadas e índices que se conseguiu melhores resultados.

De seguida, a otimização do teste de carga baseou-se na análise dos parâmetros do *PostgreSQL*, para se conseguir melhor desempenho a partir da configuração de referência. Para tal, também foram realizados testes, para diferentes valores desses campos isoladamente, e de seguida, conjuntamente, para se chegar à configuração ideal. A principal dificuldade encontrada nesta fase foi a quantidade de testes necessários para se conseguir provar a otimização da configuração final, em que se repetiu algumas vezes os mesmos testes de forma a obtermos valores médios e garantir que as melhorias obtidas não eram apenas flutuações de performance sem relevância.

Em suma, a realização deste trabalho permitiu ao grupo alargar os seus conhecimentos relativamente a conceitos expostos nas aulas, pelo que, e tendo em conta o resultado final, consideramos o resultado do trabalho satisfatório.