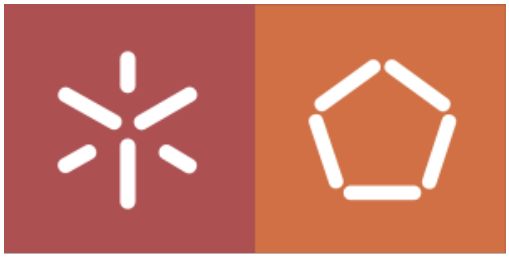

Universidade do Minho
Escola de Engenharia

Mestrado em Engenharia Informática

Perfil Engª de Aplicações

Administração de Base de Dados

Trabalho Prático

1º Ano, 1º Semestre

Ano letivo 2020/2021

Diogo Lopes

PG42823

Fábio Gonçalves

PG42827

Joel Carvalho

PG42837

Diogo Ferreira

PG42824

Conteúdo

Lista de Figuras

Lista de Tabelas

Siglas e Acrónimos

OLPT Online Transaction Processing

GCP Google Cloud Platform

SQL Structured Query Language

VM Virtual Machine

CPU Central Processing Unit

SO Sistema Operativo

WAL Write Ahead Log

1. Introdução

O desenvolvimento deste Projeto recai, na íntegra, no âmbito da Administração de Base de Dados Relacionais. Com a Administração de uma Base de Dados, é necessário ter uma especial atenção devido à solidez, disponibilidade e escalabilidade dos seus dados. Caso alguma das anteriores características fracassar, poderá provocar múltiplas mazelas, por vezes algumas delas irreparáveis.

Com o propósito de aprimorar estas características, o principal objetivo da implementação deste Trabalho Prático consiste na necessidade de otimizar e avaliar o *benchmark* TPC-C, que simula o registo de transações online (OLTP), com dados e interrogações adicionais. Assim sendo, efetuamos várias análises utilizando o sistema PostgreSQL, de modo a comparar os diferentes resultados, para posterior análise de performance.

1. Objetivos

Ao nível dos principais objetivos definidos, o plano de trabalho para este projeto está centrado essencialmente em quatro tópicos:

- Instalar e Configurar o *benchmark* TPC-C usando uma máquina virtual "n1-standard-2"para o servidor PostgreSQL;
- Propor uma configuração de referência em termos de número de *warehouses* e número de clientes;
- Escolher e Adaptar n interrogações analíticas (1 por cada elemento do grupo) para Otimizar o seu desempenho tendo em conta, os respetivos planos e mecanismos de redundância que estão a ser utilizados;
- Otimizar o desempenho da carga transacional tendo em consideração os parâmetros de configuração do PostgreSQL.

2. Instalação e configuração do benchmark TPC-C

Inicialmente, e tratando-se da base de trabalho, foi necessário instalar e configurar o *benchmark* TPC-C recorrendo aos serviços da *Google Cloud Platform.*

a. *Benchmark* TPC-C

Este *benchmark* consiste numa simulação de um sistema de bases de dados de uma cadeia de lojas, suportando a operação diária de gestão de vendas e stocks. É composto por 9 tabelas, que incluem entrada e entrega de pedidos, registo de pagamentos, verificação do estado dos pedidos e monitorização do nível de stock nos armazéns.

Na tabela seguinte é possível observar a composição da base de dados, e a respetiva descrição de cada tabela:

Tabela	Descrição
Warehouse	Armazém que disponibiliza o stock
Stock	Número de quantidade existente
Item	Detalhe do produto
District	Distritos de vendas (10)
Customers	Clientes do serviço
Order	Lista de encomendas
New_Order	Nova encomenda/entrada, em média com 10 itens diferentes
History	Relacionada com as encomendas, mantendo o histórico da mesma
Order_Line	Linha de cada encomenda
Tabelas Extras TPC-H para as *queries* analíticas	
Tabela	**Descrição**
Nation	Estado
Region	Região
Supplier	Fornecedor

Tabela 1 Composição da Base de Dados

De acordo com a Figura 1, onde é apresentado o esquema relacional, observamos a cardinalidade entre tabelas.

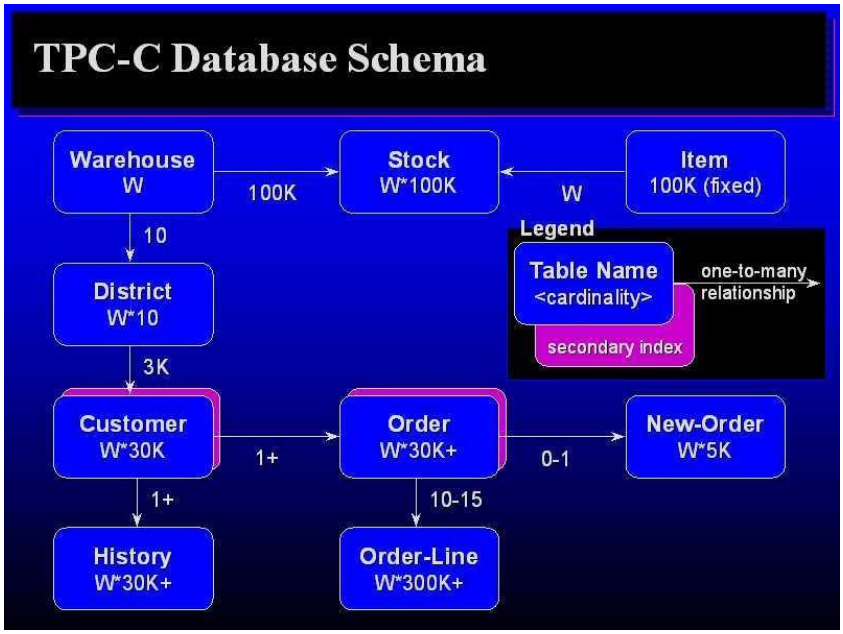

Figura 1 Cardinalidade entre tabelas

b. Google Cloud Platform

Disponibilizado pela *Google*, permite criar, implementar e dimensionar aplicações, sites e serviços na mesma infraestrutura que o *Google*, via *cloud.*

Algumas soluções oferecidas:

1. *Compute Engine* - Máquinas virtuais em execução nos *data center* da *Google*;
2. *Cloud SQL* - Serviços de Base de Dados relacional;
3. *Google Kubernetes Engine* - Ambiente para gerir aplicações em *containers*;
4. *Cloud Storage* - Armazenamento seguro, estável e escalável;
5. *Big Query* - Armazenamento de dados em várias nuvens sem servidor.

c. Arquitetura das Máquinas Virtuais

Preliminarmente, decidimos utilizar a máquina servidor com um disco de 7.5 GB HDD, baseado no tutorial fornecido pelo professor. Porém, vários problemas despoletaram, os inconstantes valores de *throughput*, tempo de resposta e taxa de aborto, isto é, quando executávamos duas vezes seguidas o comando *./run.sh*, para efetuar as transações, os valores iam sofrendo alterações muito elevadas, por vezes

superiores ao dobro, o que nos impedia de prosseguir o trabalho com a qualidade e performance desejada.

Posto isto, e após um esclarecimento por parte do professor, aumentamos o disco do servidor de 7.5 GB HDD para **250 GB SSD**, pelo facto de que a carga era demasiado elevada para a capacidade do *hardware*. Após esta pequena grande alteração, o problema da carga e dos valores inconstantes foram superados com sucesso.

Características	Servidor	*Benchmark*
Tipo	*n1-standard-2*	*f1-micro*
CPU	2	1 *shared core*
Memória	250 GB SSD	614 MB
Sistema Operativo	Ubuntu 20.04 LTS	Ubuntu 20.04 LTS
RAM	7457 MB	576 MB

Tabela 2 Configuração das Máquinas Virtuais

3. Configuração de número de *warehouses* e número de clientes

Após a instalação das VM's, o objetivo passou por propor uma configuração de referência em termos de número de *warehouses* e número de clientes. O valor *default* para o número de *warehouses* é igual a **2** e para o número de clientes é igual a **10**. Objetivo definir esses termos para alcançar:

1. Tamanho da Base de Dados - **3/4 GB**
2. CPU - Carga razoável entre **30%-50%**

Número de *warehouses*

Na tabela seguinte, demonstramos o número de tentativas efetuadas de modo a atingir o tamanho pretendido. Após análise dos resultados obtidos na segunda tentativa, conseguimos perceber que o crescimento era linearmente proporcional ao número de *warehouses*, isto é, triplicando o número de *warehouses*, o tamanho aproximou-se do triplo, o que nos levou a concluir que para atingir sensivelmente 3.5 GB, seria $\frac{2*3500}{256} =$ 27 *warehouses*.

Nº de *warehouses*	Tamanho da Base de Dados
2	256 MB
6	710 MB
27	**3172 MB**

Tabela 3 Número de Warehouses testados e definidos

Com o objetivo de não gastar recursos e desperdiçar tempo, guardamos a base de dados povoada com 27 *warehouses*, através do comando *pg_dump*. Assim, sempre que fosse necessário repor a base de dados não seria obrigatório executar o ficheiro *load.sh.*

Repor a Base de Dados:

pg_restore -h _nome_servirdor_ -c -d tpcc < path/nome_do_ficheiro

Número de clientes

Seguindo a ordem das tarefas delineadas no enunciado, passámos para a definição do número de clientes. Através de um processo iterativo e aumentando o número de clientes de forma gradual, avaliamos o número ideal de clientes tendo em conta os três parâmetros já estudados em ambiente de aula, transações por segundo, tempo de resposta e taxa de falha. Todos estes testes foram realizados utilizando como medida de tempo 10 minutos.

Número de Clientes (total)	Throughput (tx/s)	Response Time(s)	Abort Rate (%)	CPU (%) ≈
10 (270)	23.5309394417	0.003392156862	0.0005867167331	10%
30 (810)	71.4839721225	0.003275414171	0.0012429115202	11%
50 (1350)	118.445851657	0.004290642821	0.0029017387031	16%
70 (1890)	166.233766233	0.003881467245	0.0049718152945	22%
90 (2430)	212.280920169	0.004388192362	0.0060623332210	28%
110 (2970)	259.836957876	0.004214724318	0.0081103086569	28%
130 (3510)	305.428046979	0.004539222065	0.0099849688640	33%
150 (4050)	353.104672612	0.005005758232	0.0128037100514	35%

170 (4590)	346.523340520	0.007040882533	0.0255709790856	38%
190 (5130)	**441.020478521**	**0.006282252365**	**0.0205982874572**	**45%**
210 (5670)	486.444823773	0.006735644759	0.0233154310392	51%

Tabela 4 Número de clientes testados e definidos

Como podemos observar nos gráficos seguintes, onde evidenciamos o que foi estudado de forma semelhante às tarefas desenvolvidas em ambiente de sala de aula, à medida que se aumenta os clientes, o tempo de resposta e *throughput* também aumentam.

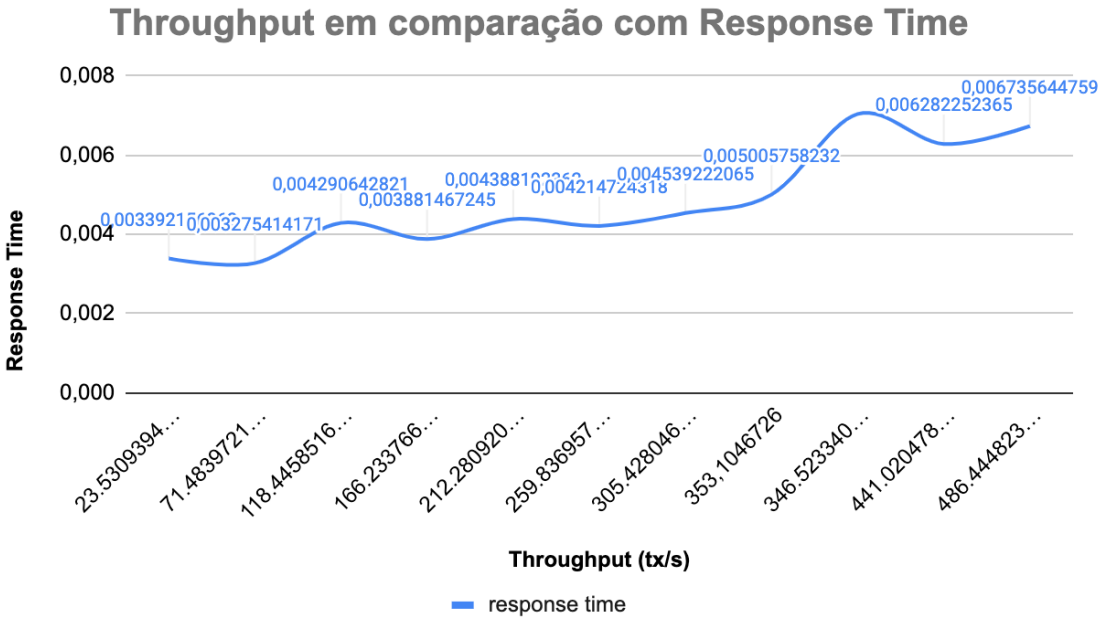

Figura 2 Throuhgput em comparação com Response Time

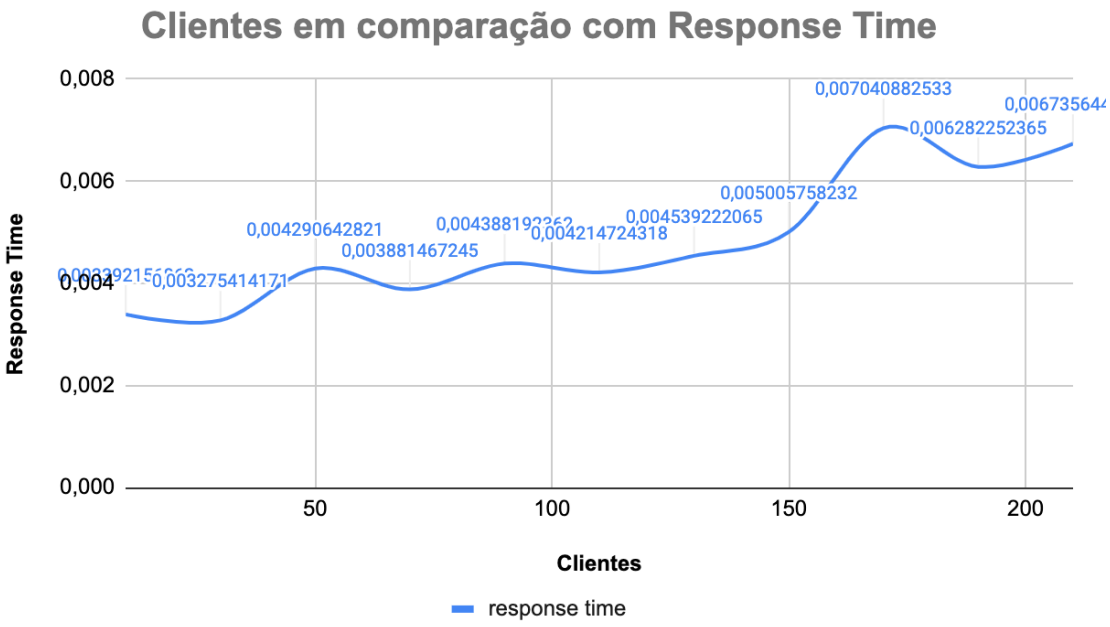

Figura 3 Clientes em comparação com Response Time

Analisando a tabela e os gráficos de comparação, concluímos que o número indicado de clientes seria **190**, isto porque, obtivemos uma média de aproximadamente **440 transações por segundo**, **0.006 segundos de tempo de resposta** e **taxa de falha de 0.021%.**

Para concluir é visível na imagem posterior, que a carga de **CPU** rondava os **50%**, como o delineado nos objetivos acima definidos. Como já foi referido, se continuássemos a aumentar o número de clientes e não ter em conta a carga do CPU, obteríamos melhores resultados.

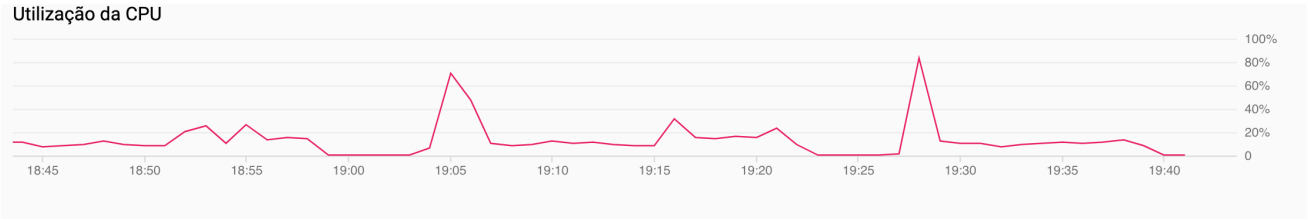

Figura 4 Historial da utilização do CPU durante as tentativas

Configuração final

Tendo por base as configurações definidas pelo Professor, a máquina terá as seguintes configurações de referência:

1. Número de *warehouses*: 27
2. Número de clientes: 190

A imagem seguinte demonstra os gráficos onde visualizamos detalhes como por exemplo a qualidade das transações e o tempo de resposta. No gráfico superior esquerdo visualizamos que as transações, representadas por um ponto azul, estão bastante concentradas e o tempo de resposta médio é de 0.006s, um número que consideramos bastante satisfatório. É relevante salientar que o gráfico inferior esquerdo, é importante para o tempo de resposta, uma vez que mostra que obtemos uma média de aproximadamente **440 transações por segundo.**

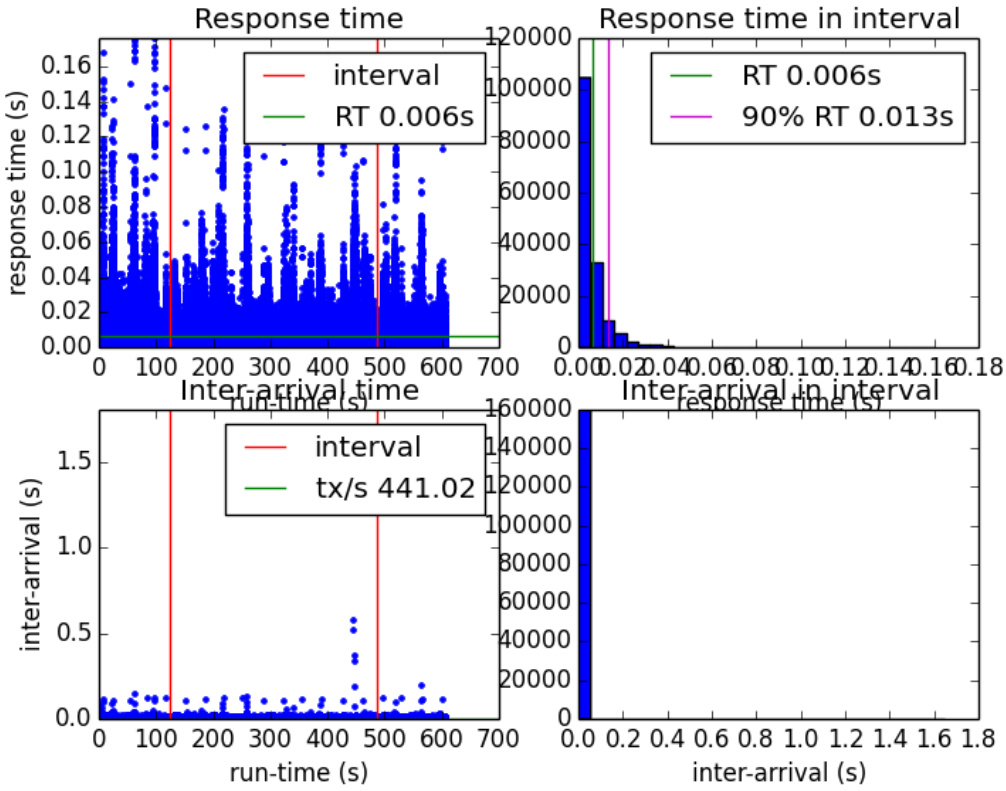

Figura 5 Output showtpc após número de clientes e warehouses definidos

4. Otimização do desempenho da carga transacional, configuração do PostgreSQL

O objetivo da criação deste tópico é definir a configuração adequada para o PostgreSQL de forma a alcançar os melhores resultados possíveis. Após o estudo e leitura de documentação conveniente, testamos cada parâmetro de forma individual mantendo as restantes como *default,* de maneira a que um parâmetro não influenciasse os demais. De seguida vamos discriminar, individualmente e de forma resumida, cada parâmetro com diferentes opções e o seu impacto nas configurações gerais. No final, apresentamos as nossas conclusões, que estiveram na génese da escolha das melhores configurações.

Configurações

a. *shared_buffers*

Com os *shared_buffers*, determinamos quanta memória é dedicada ao PostgreSQL para armazenar dados em cache.

Lista de valores:

1. 128 MB - Valor default
2. 1 GB (12.5%) - Valor inicial razoável para *shared_buffers* é 1/4 da memória do sistema.
3. 2 GB (25%) - Como a máquina tem aproximadamente 8 GB, ou seja, é superior a 1 GB, um valor inicial razoável para *shared_buffers* é ¼ dessa memória.

shared_buffers	Throughput (tx/s)	Response Time(s)	Abort Rate (%)
128 MB (*default*)	441.020478521	0.006282252365	0.0205982874572
1 GB	442.975849569	0.005421824885	0.0174518574724
2 GB	**443.950359660**	**0.005157252373**	**0.0165821863783**

Tabela 5 Testes com shared_buffers

De acordo com esperado e lido na documentação *online,* utilizando ¼ da memória obtemos uma pequena melhoria nos 3 parâmetros acima referidos.

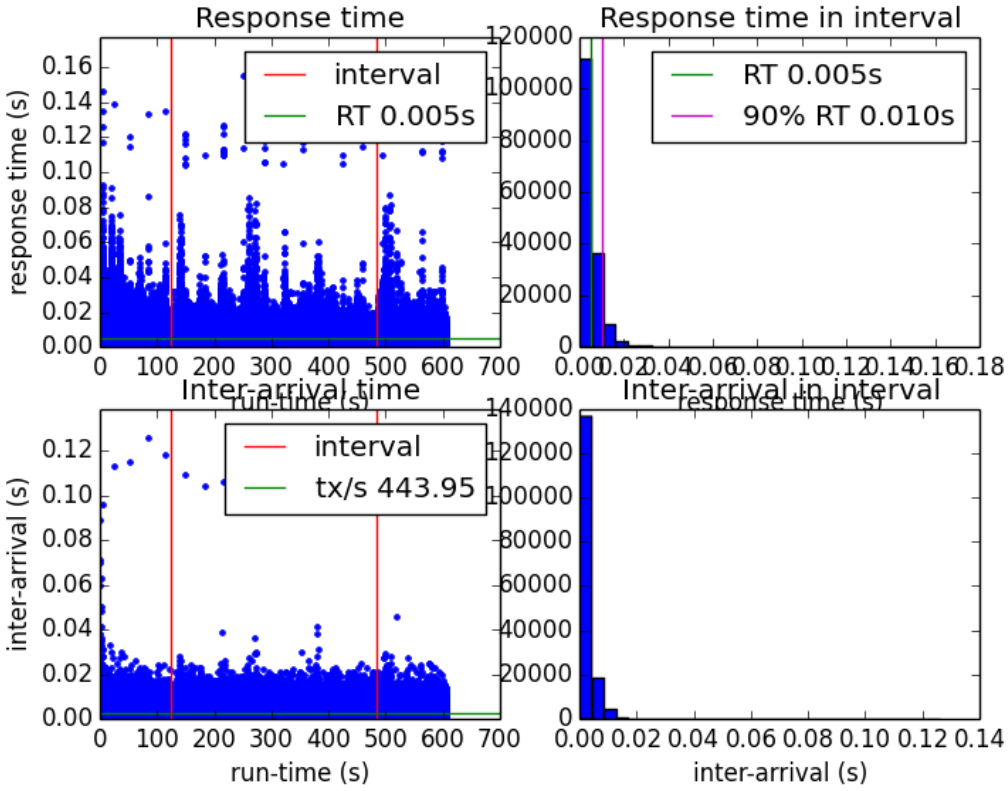

Figura 6 Output showtpc com melhores resultados do shared_buffers

b. *work_mem*

A *work_mem* é a quantidade máxima de memória base a ser usada por uma operação de consulta (como *hash*) antes de gravar os arquivos. É ótima para consultas grandes.

work_mem	Throughput (tx/s)	Response Time(s)	Abort Rate (%)
4 MB (*default*)	441.020478521	0.006282252365	0.0205982874572
10 MB	**441.27427268359**	**0.0079316533243**	**0.0251108369871**

Tabela 6 Testes com work_mem

Como é bom para consultas grandes, os valores são muito idênticos ao *default*. Então, definindo com 10 MB e com 190 clientes, será utilizado cerca de 2 GB de memória.

$$10 * 190 = 1.9 \text{ GB}$$

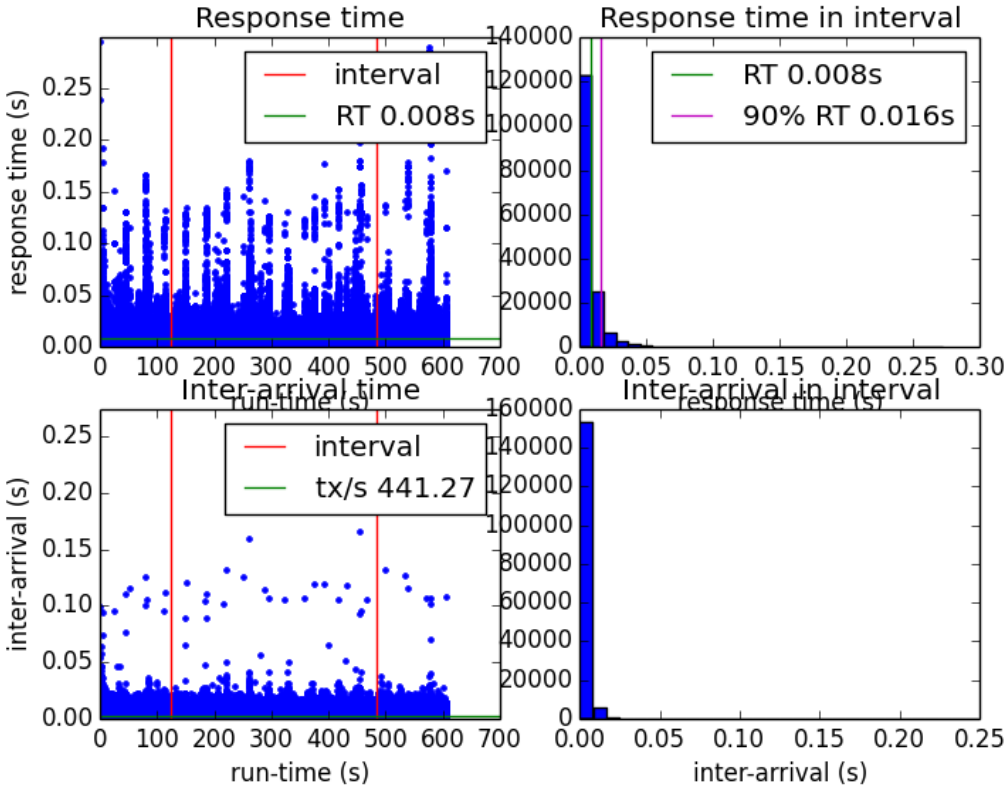

Figura 7 Output showtpc com melhores resultados do work_mem

c. *fsync*

Este parâmetro está definido por padrão, com o valor *on*. O Postgresql garante que a base de dados é recuperável em caso de falha, com o apoio de funções de sincronização. Alternando para *off*, o sistema em caso de falha, não consegue recuperar as transações que não foram escritas, com isto, é evidente um aumento no desempenho da base de dados.

fsync	Throughput (tx/s)	Response Time(s)	Abort Rate (%)
on (default)	**441.020478521**	**0.006282252365**	**0.0205982874572**
off	445.98759022862	0.0039663512875	0.0109784851164

Tabela 7 Testes com fsync

Obviamente que este parâmetro deve estar sempre ativo, apenas apresentamos este aqui para evidenciar uma experiência e um detalhe observado. Finalizando, a melhoria foi notada como era esperado.

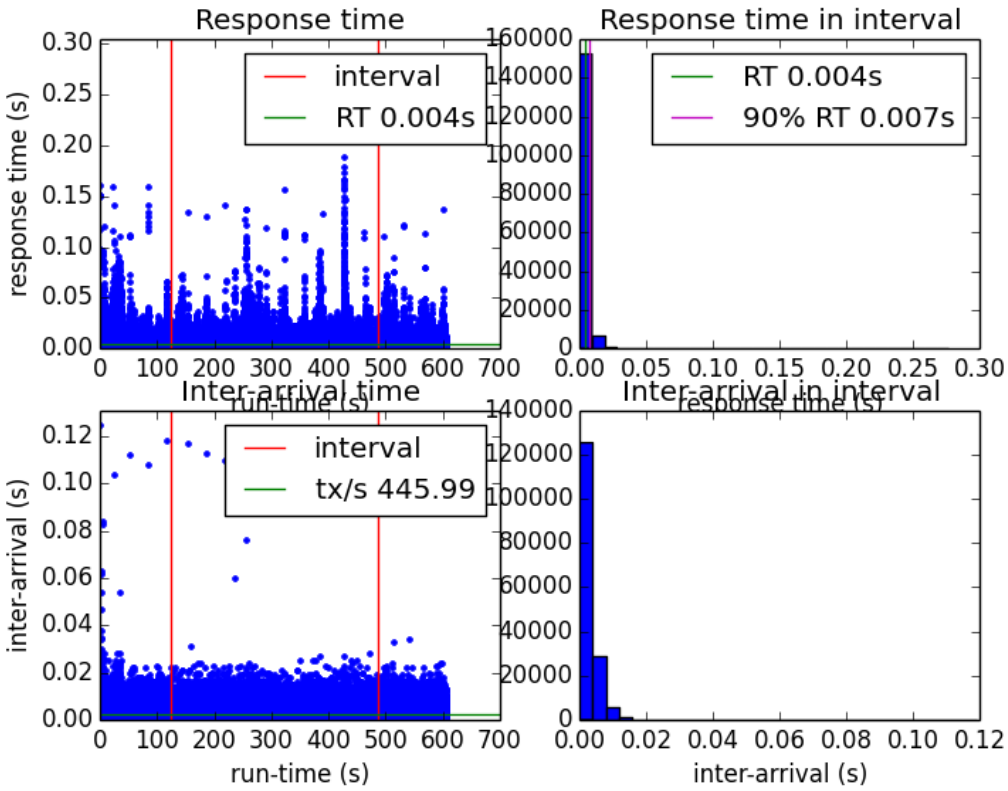

Figura 8 Output showtpc com melhores resultados do fsync

d. *synchronous_commit*

Este especifica se a confirmação da transação aguardará que os registos do *WAL* sejam gravados no disco antes da indicação de "sucesso" para o cliente. Esta configuração contém 5 valores possíveis.

synchronous_commit	**Throughput (tx/s)**	**Response Time(s)**	**Abort Rate (%)**
on (default)	441.020478521	0.006282252365	0.0205982874572
off	**447.742053171**	**0.0039833701650**	**0.0106146479699**
remote_apply	440.108835093	0.0081738719851	0.0253778140473
remote_write	440.139940144	0.0075005807638	0.0238232409904
local	440.139940144	0.007500580763	0.0238232409904

Tabela 8 Testes com synchronous_commit

Observando a tabela acima, podemos concluir que com este parâmetro desligado obtemos uma melhoria de performance acentuada. Uma vez que não cria nenhum risco de inconsistência na base de dados, pode resultar na perda de

algumas transações supostamente confirmadas recentemente, mas o estado da BD será o mesmo como se essas transações tivessem sido abortadas de forma limpa. Assim sendo este parâmetro pode ser desligado quando o desempenho é mais importante do que a certeza exata sobre a durabilidade de uma transação.

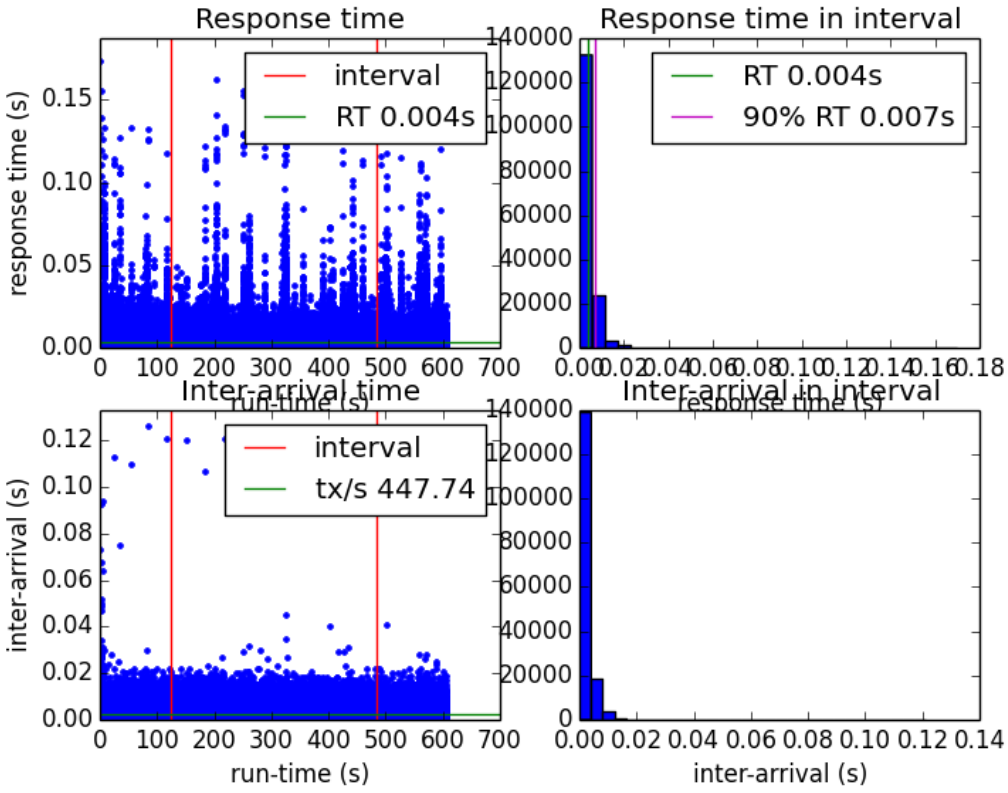

Figura 9 Output showtpc com melhores resultados do synchronous_commit

e. ***wal_sync_method***

Este é utilizado para obrigar as atualizações do *WAL* para o disco. Se o *fsync* estiver desativado, esta configuração é irrelevante, uma vez que as atualizações do arquivo *WAL* não serão forçadas a sair. Esta configuração contém 4 valores possíveis.

wal_sync_method	**Throughput (tx/s)**	**Response Time(s)**	**Abort Rate (%)**
fdatasync (default)	441.020478521	0.006282252365	0.0205982874572
fsync	436.218407908	0.010024461070	0.0329028855941
open_datasync	**442.344102927**	**0.006678960970**	**0.0216003085758**
open_sync	434.990579948	0.010655759287	0.0336254793981

Tabela 9 Testes com wal_sync_method

O valor *default* e o *open_datasync* são bastantes similares, e aumentam ligeiramente no *throughput,* mas nos restantes parâmetros evoluem negativamente. Apesar disso, optamos pelo *open_datasync* porque com a combinação dos restantes parâmetros otimizados acreditamos que será a melhor opção.

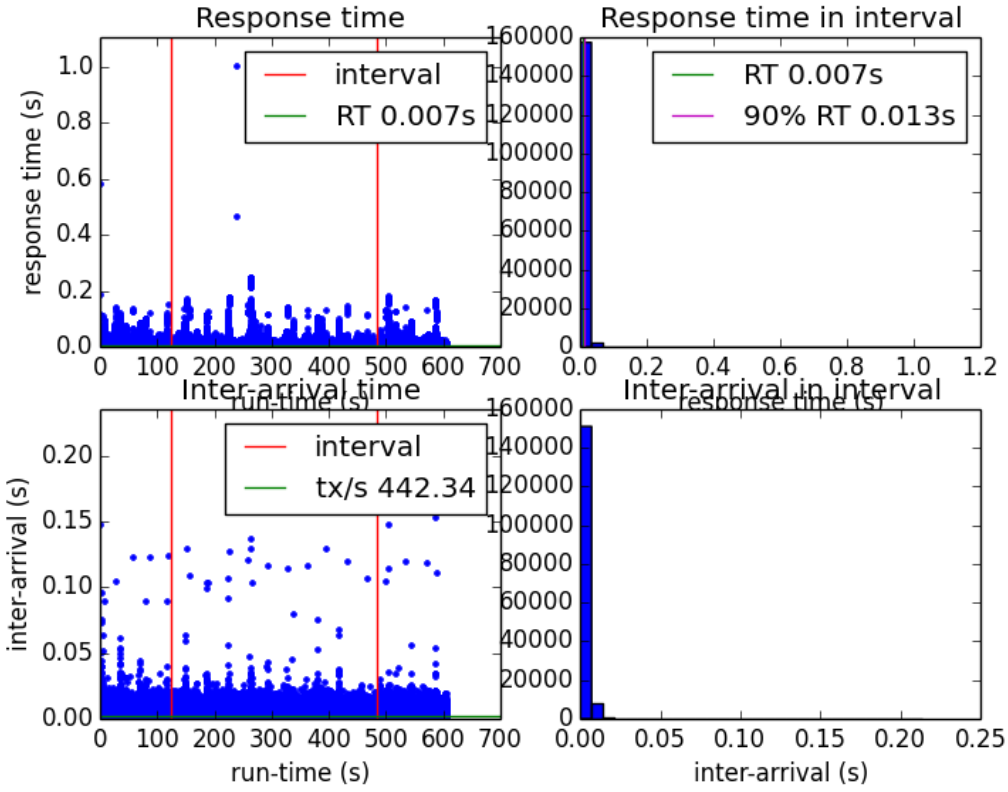

Figura 10 Output showtpc com melhores resultados do wal_sync_method

f. *wal_buffers*

É a quantidade de memória partilhada para os dados *WAL* que ainda não foram gravados no disco.

wal_buffers	Throughput (tx/s)	Response Time(s)	Abort Rate (%)
4 MB *(default)*	**441.020478521**	**0.006282252365**	**0.0205982874572**
16 MB	440.002655668	0.0065854190636	0.0216813669409
32 MB	437.532361451	0.0101990988311	0.0288666002925

Tabela 10 Testes com wal_buffers

De acordo com a visualização de dados presente na tabela acima, o valor *default* continua com os melhores resultados, e posto isto, não alteramos o parâmetro *wal_buffers.*

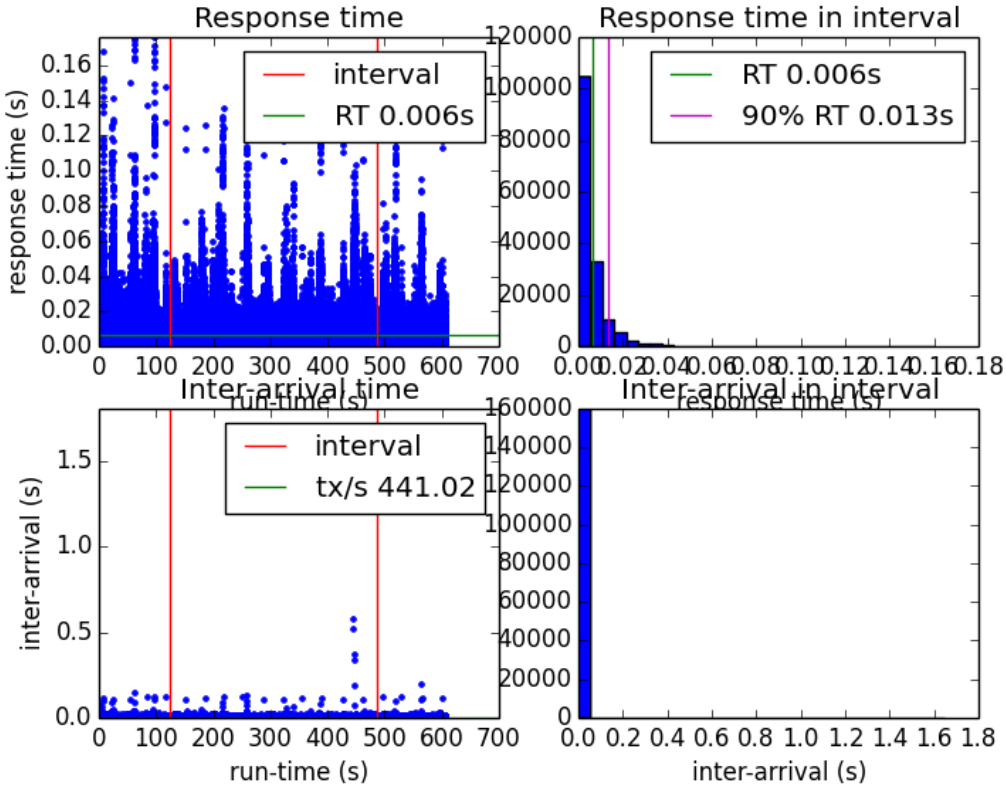

Figura 11 Output showtpc com melhores resultados do wal_buffers

g. *effective_cache_size*

Define uma suposição sobre o tamanho efetivo da cache de disco que está disponível para uma única consulta. Isto é, calcula o valor estimado para utilizar um índice, ou seja, um valor mais alto torna mais plausível o uso de *scans* nos índices.

effective_cache_size	Throughput (tx/s)	Response Time(s)	Abort Rate (%)
4 GB *(default)*	441.020478521	0.006282252365	0.0205982874572
6 GB	**443.152780976**	**0.006777957251**	**0.0212867647058**
8 GB	440.740331186	0.006808115211	0.0220339710834

Tabela 11 Testes com effective_cache_size

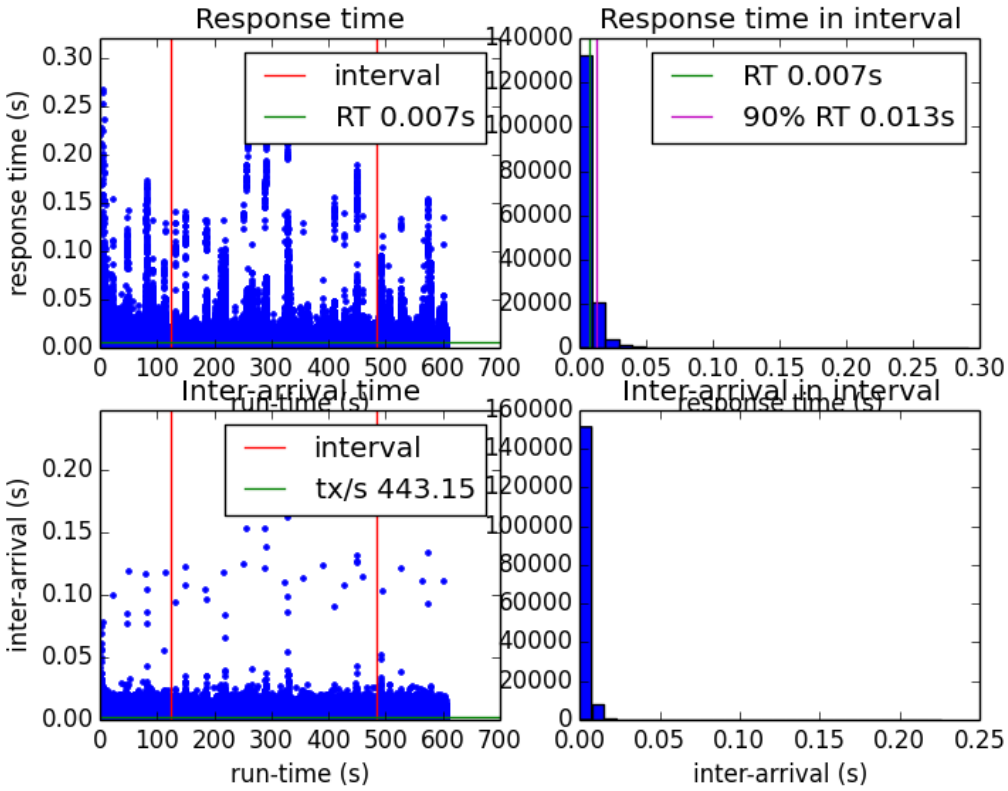

Figura 12 Output showtpc com melhores resultados do effective_cache_size

h. *max_wal_size*

Permite definir o tamanho máximo do crescimento do *WAL* durante os *checkpoints* automáticos. Este tamanho pode ser ultrapassado em circunstâncias especiais, como carga pesada, um *archive_command* com falha ou uma configuração alta de *wal_keep_size*.

max_wal_size	Throughput (tx/s)	Response Time(s)	Abort Rate (%)
1 GB *(default)*	441.020478521	0.006282252365	0.0205982874572
2 GB	441.911224326	0.006158745342	0.0197823209295
4 GB	**443.549145807**	**0.006427509108**	**0.0202974018592**

Tabela 12 Testes com max_wal_size

Este parâmetro de configuração foi alterado devido a um *warning* por parte do servidor, informando que deveria ser aumentado. Mediante isso, concluímos que 4 GB seria o valor adequado. Graças a essa alteração o aviso dissipou-se e melhoramos ligeiramente a performance no *running* das transações.

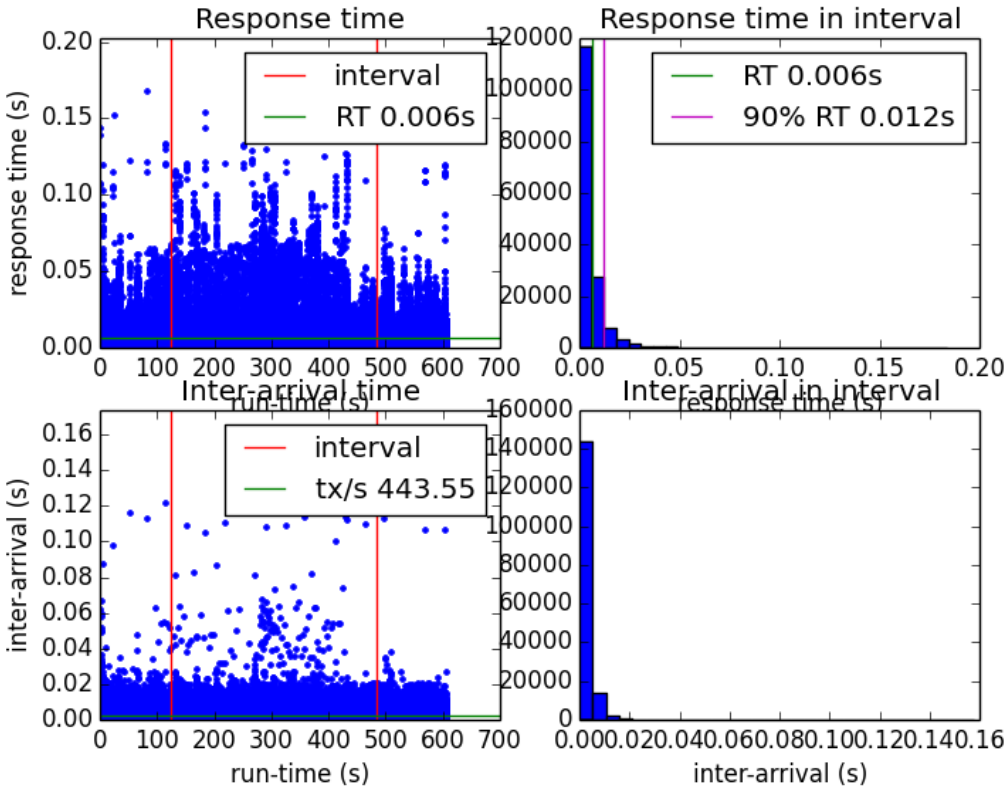

Figura 13 Output showtpc com os melhores resultados do max_wal_size

Combinações finais

Mediante a análise descrita acima, resta agregar os melhores parâmetros de modo a obter a melhor otimização da carga transacional. Deste modo, os parâmetros seguintes foram os selecionados.

Parâmetros

1. *shared_buffers*: 2GB
2. *work_mem*: 10 MB
3. *synchronous_commit*: *off*
4. *wal_sync_method*: *open_datasync*
5. *effective_cache_size*: 6 GB
6. *max_wal_size*: 4 GB

Throughput (tx/s)	Response Time(s)	Abort Rate (%)	CPU (%) ≈
Antes da otimização			
441.020478521	0.006282252365	0.0205982874572	45 %

Depois da otimização			
448.278243001	0.0029408165905	0.0085759835810	42 %

Tabela 13 Antes e depois da otimização

Como previsto, depois da combinação de todos os parâmetros de configuração conseguimos melhorar consideravelmente os três pontos básicos que temos vindo a analisar, bem como a performance do CPU.

Relativamente ao *throughput,* conseguimos superar os 441 iniciais e alcançamos **448 transações por segundo**, reduzindo também o tempo de resposta para **3 vezes menos e a taxa de aborto para menos de metade.** Desta forma e analisando a figura abaixo podemos concluir que as otimizações das configurações do PostgreSQL foram concluídas com êxito.

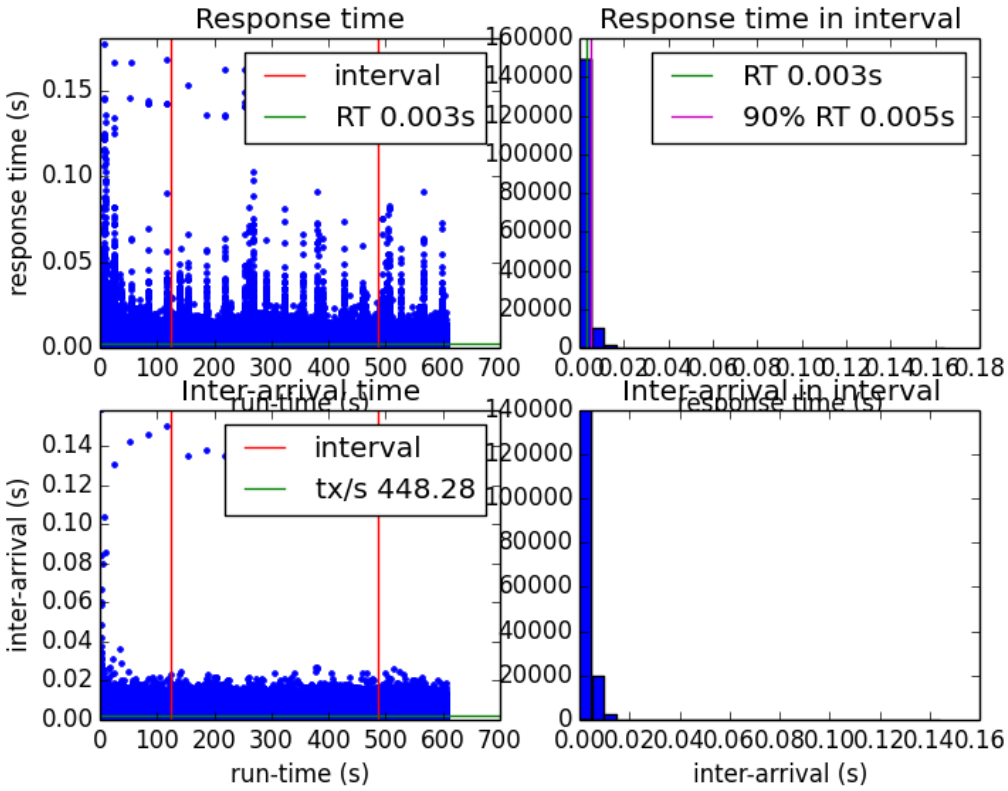

Figura 14 Output showtpc com os melhores parâmetros definidos

5. Interrogações analíticas e otimização do seu desempenho

Para a otimização das interrogações analíticas do TPC-H manipulamos cada uma com a ferramenta *online,* ***explain.dalibo.com,*** que disponibiliza de forma gráfica e intuitiva o comando *explain* do PostgreSQL.

Baseado no estudo já realizado, aqui serão aplicadas as seguintes técnicas:

1. Utilização de índices e vistas materializadas;
2. Ajuste de pesos relativos;
3. Detalhes estatísticos e atuais.

De seguida iremos demonstrar passo a passo a otimização de 4 queries, em que o objetivo será escolher o plano mais pequeno.

Query 1

Esta consulta relata a quantidade e quantidade total de todas as linhas de pedidos enviadas fornecidas por um período de tempo específico. Além disso, existe também a informação sobre o valor médio e a quantidade, além da contagem total de todas essas linhas de pedidos solicitadas por número de linha de pedido individual.

```
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 > '2020-12-23 23:53:00.000000'
group by ol_number order by ol_number
```

Resultados Iniciais

```
Finalize GroupAggregate  (cost=188460.66..188465.02 rows=15 width=116) (actual time=3172.594..3172.749 rows=13 loops=1)
  Group Key: ol_number
  ->  Gather Merge  (cost=188460.66..188464.16 rows=30 width=116) (actual time=3172.538..3172.629 rows=39 loops=1)
      Workers Planned: 2
      Workers Launched: 2
      ->  Sort  (cost=187460.63..187460.67 rows=15 width=116) (actual time=3138.327..3138.331 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=187460.12..187460.34 rows=15 width=116) (actual time=3138.261..3138.276 rows=13
loops=3)
              Group Key: ol_number
              ->  Parallel Seq Scan on order_line  (cost=0.00..144943.64 rows=2834432 width=11) (actual time=32.563..1266.031
rows=2295918 loops=3)
                  Filter: (ol_delivery_d > '2020-12-23 23:53:00'::timestamp without time zone)
```

Rows Removed by Filter: 395451 Planning Time: 0.208 ms JIT: Functions: 30 Options: Inlining false, Optimization false, Expressions true, Deforming true Timing: Generation 6.964 ms, Inlining 0.000 ms, Optimization 3.005 ms, Emission 93.123 ms, Total 103.092 ms Execution Time: 3175.022 ms
Operações
1. CREATE MATERIALIZED VIEW query_1 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 > '2020-12-23 23:53:00.000000' group by ol_number order by ol_number; 2. explain analyze select * from query_1; 3. REFRESH MATERIALIZED VIEW query_1;
Resultados Finais
Seq Scan on query_1 (cost=0.00..15.60 rows=560 width=116) (actual time=0.006..0.007 rows=13 loops=1) Planning Time: 0.108 ms Execution Time: 0.018 ms

Considerações Finais	Custo
Antes	2318.236
Depois	0.050

Tabela 14 Trabalho realizado na Query 1

Observamos inicialmente, os *indexes* presentes na tabela *order_line*, para termos a noção se poderíamos adicionar mais indexes para uma determinada *query.* Posto isto, e de acordo com a *query* em questão, concluímos que apenas seria benéfico adicionar indexes para as colunas que representam as datas, *ol_delivery_d*, e para a coluna que representa o número do pedido, *ol_number*. Porém, estes já eram existentes e ao realizarmos o *explain analyze,* deparámo-nos com um tempo de execução bastante elevado. Facto que pudemos confirmar com a plataforma *explain.dalibo*.

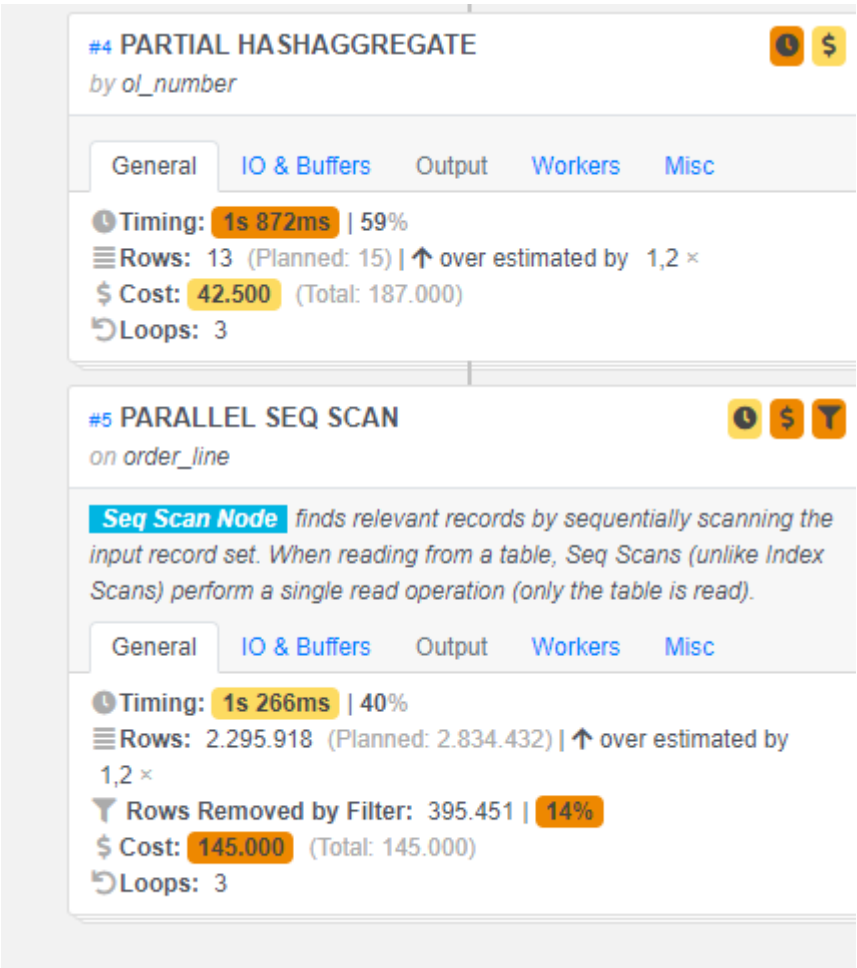

Figura 15 Query 1

Posto este problema e, visto que não era possível melhorar o desempenho desta consulta com a adição de mais indexes, decidimos então criar uma vista materializada para armazenar o conteúdo devolvido por esta consulta. Analisando então o *explain analyse*, vimos que esta consulta é sem dúvida alguma muito mais rápida, contudo teremos que proceder ao '*REFRESH MATERIALIZED VIEW query_1;*', para atualizarmos os dados contidos nesta tabela.

Query 4

Lista todos os pedidos com linhas que foram enviadas após a data de reserva dos pedidos num determinado intervalo de tempo.

```
select o_ol_cnt, count(*) as order_count from orders where o_entry_d >= '2020-12-23 23:51:00.000000' and o_entry_d < '2020-12-23 23:52:00.000000' and exists (select * from order_line where o_id = ol_o_id and o_w_id = ol_w_id and o_d_id = ol_d_id and ol_delivery_d >= o_entry_d) group by o_ol_cnt order by o_ol_cnt;
```

Resultados Iniciais
Finalize GroupAggregate (cost=259496.12..259498.90 rows=11 width=12) (actual time=1047.542..1054.291 rows=10 loops=1) Group Key: orders.o_ol_cnt -> Gather Merge (cost=259496.12..259498.68 rows=22 width=12) (actual time=1047.530..1054.272 rows=30 loops=1) Workers Planned: 2 Workers Launched: 2 -> Sort (cost=258496.09..258496.12 rows=11 width=12) (actual time=1015.664..1015.667 rows=10 loops=3) Sort Key: orders.o_ol_cnt Sort Method: quicksort Memory: 25kB Worker 0: Sort Method: quicksort Memory: 25kB Worker 1: Sort Method: quicksort Memory: 25kB -> Partial HashAggregate (cost=258495.79..258495.90 rows=11 width=12) (actual time=1015.626..1015.630 rows=10 loops=3) Group Key: orders.o_ol_cnt -> Nested Loop Semi Join (cost=0.86..258276.40 rows=43878 width=4) (actual time=28.703..964.067 rows=105337 loops=3) -> Parallel Index Scan using orders_idx_o_d on orders (cost=0.42..12260.92 rows=131634 width=24) (actual time=7.634..105.901 rows=105337 loops=3) Index Cond: ((o_entry_d >= '2020-12-23 23:51:00'::timestamp without time zone) AND (o_entry_d < '2020-12-23 23:52:00'::timestamp without time zone)) -> Index Scan using pk_order_line on order_line (cost=0.43..2.35 rows=3 width=20) (actual time=0.007..0.007 rows=1 loops=316011) Index Cond: ((ol_w_id = orders.o_w_id) AND (ol_d_id = orders.o_d_id) AND (ol_o_id = orders.o_id)) Filter: (ol_delivery_d >= orders.o_entry_d) Planning Time: 0.782 ms JIT: Functions: 51 Options: Inlining false, Optimization false, Expressions true, Deforming true Timing: Generation 8.114 ms, Inlining 0.000 ms, Optimization 3.610 ms, Emission 58.211 ms, Total 69.935 ms Execution Time: 1056.887 ms
Resultados Finais
1056.887 ms

Tabela 15 Trabalho realizado na Query 4

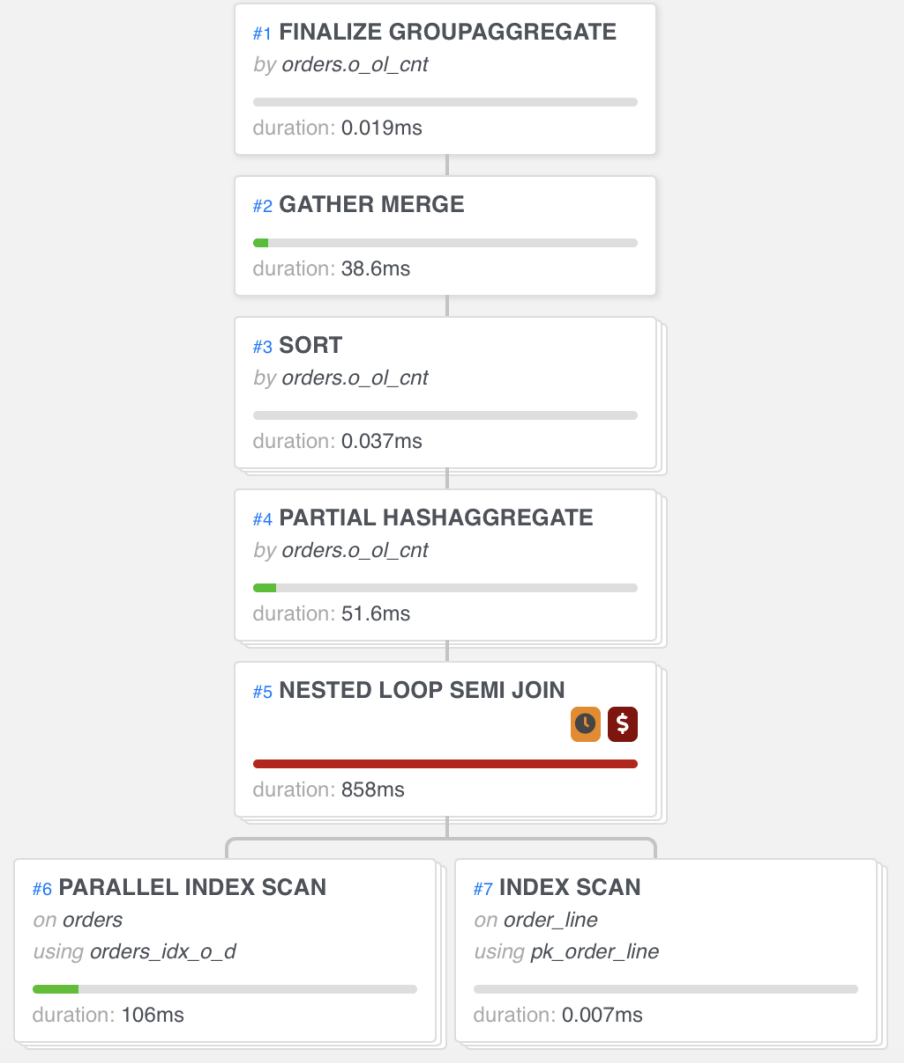

Figura 16 Query 4

Analisando o plano de execução da query em questão podemos observar que já está a fazer uso de índices implementados nas tabelas *orders* e *order_line*. Podemos também observar que o maior custo desta operação é o *nested loop semi joint* devido ao elevado número de linhas resultante da seleção de linhas da tabela orders. Devido ao facto de estarmos a filtrar os *orders* com base num período limitado de tempo, não faz muito sentido estarmos a criar uma vista materializada pois não queremos estar sempre a consultar o mesmo intervalo de tempo dado que os resultados não serão alterados pois a menos que criemos um intervalo cuja data final ainda não tenha acontecido. Como tal, vamos tentar melhorar a performance desta query experimentando novos algoritmos para executar o *join*.

Algoritmo de *Join*	**Tempo de execução**
Nested Loop	1056.887 ms

Hash Join	6198.426 ms
Merge Join	5000.824 ms

Tabela 16 Algoritmos Query 4

Podemos observar através da tabela anterior que a utilização do nested loop produz muito melhores resultados que as outras opções. é de observar que o PostgreSQL nem sempre faz as melhores opções pois num momento inicial quando desativamos o uso do *nested loop join* optou por executar o *hash join* apesar de se verificar que o *merge join* produzir melhores resultados. Para executar a *querie* com recurso ao *merge join* desativamos o uso do *nested* e *hash join.*

Query 6

Lista o valor total da receita guardada das linhas dos pedidos no ano de 2020, em que a quantidade não seja superior a 100000.

```
select sum(ol_amount) as revenue from order_line
where ol_delivery_d >= '2020-01-01 00:00:00.000000'
and ol_delivery_d < '2020-12-31 00:00:00.000000'
and ol_quantity between 1 and 100000;
```

Resultados Iniciais

```
Finalize Aggregate  (cost=178322.17..178322.18 rows=1 width=32) (actual time=2316.691..2316.757 rows=1 loops=1)
  -> Gather  (cost=178321.95..178322.16 rows=2 width=32) (actual time=2312.710..2316.709 rows=3 loops=1)
      Workers Planned: 2
      Workers Launched: 2
      -> Partial Aggregate  (cost=177321.95..177321.96 rows=1 width=32) (actual time=2286.246..2286.248 rows=1 loops=3)
          -> Parallel Seq Scan on order_line  (cost=0.00..170166.65 rows=2862119 width=3) (actual time=15.141..1449.842 rows=2295918 loops=3)
              Filter: ((ol_delivery_d >= '2020-01-01 00:00:00'::timestamp without time zone) AND (ol_delivery_d < '2020-12-31 00:00:00'::timestamp without time zone) AND (ol_quantity >= 1) AND (ol_quantity <= 100000))
              Rows Removed by Filter: 395451
 Planning Time: 0.196 ms
 Execution Time: 2318.236 ms
```

Operações

1. CREATE MATERIALIZED VIEW order_line_2020 AS select * from order_line where ol_delivery_d >= '2020-01-01 00:00:00.000000' and ol_delivery_d < '2020-12-31 00:00:00.000000';
2. Modificar a query para utilizar a view criada: select sum(ol_amount) as revenue from order_line_2020 where ol_quantity between 1 and 100000;

Resultados Finais

```
Finalize Aggregate  (cost=138297.93..138297.94 rows=1 width=32) (actual time=2160.697..2160.784 rows=1 loops=1)
  -> Gather  (cost=138297.70..138297.91 rows=2 width=32) (actual time=2156.228..2160.755 rows=3 loops=1)
      Workers Planned: 2
      Workers Launched: 2
      -> Partial Aggregate  (cost=137297.70..137297.71 rows=1 width=32) (actual time=2131.049..2131.050 rows=1 loops=3)
          -> Parallel Seq Scan on order_line_2020  (cost=0.00..130122.74 rows=2869983 width=3) (actual time=14.694..1281.674 rows=2295918 loops=3)
              Filter: ((ol_quantity >= 1) AND (ol_quantity <= 100000))
 Planning Time: 0.093 ms
 JIT:
   Functions: 17
   Options: Inlining false, Optimization false, Expressions true, Deforming true
   Timing: Generation 7.162 ms, Inlining 0.000 ms, Optimization 2.033 ms, Emission 41.052 ms, Total 50.248 ms
 Execution Time: 2161.881 ms
(13 rows)
```

Considerações Finais	Custo
Antes	2318.236
Depois	2161.881

Tabela 17 Trabalho realizado na Query 6

Para a *query* em questão decidimos que a melhor opção em termos de utilidade seria criar uma *materialized view* que contém todas as linhas da tabela *order_line* referentes ao ano de 2020 que podem depois ser utilizadas em mais interrogações. Esta não vai sofrer mais alterações visto que o intervalo temporal já terminou. Podem também ser criadas *views* com intervalos referentes a um trimestre ou a um semestre de acordo com a granularidade pretendida pelo cliente. *Views* como a que foram criadas permitem melhorar a performance de várias *queries* analíticas e na maioria dos casos tal como o da *query* aqui apresentada, serão referentes a intervalos temporais terminados e como tal não será necessário fazer *refresh* destas.

Após a criação da *materialized view* modificamos a *query* em questão para a utilizar. A *query* original está a fazer um *sequencial scan* porque todas as linhas da tabela *order_line* pertencem ao intervalo temporal fornecido, numa situação normal estaria a fazer um *index scan* utilizando o índice já existente para a coluna *ol_delivery_d,* daí não termos optado por criar *indexes*. A pouca diferença observada na performance entre as duas versões deve-se ao facto de a *view* ter o mesmo tamanho da tabela. Numa situação normal existiria uma maior diferença.

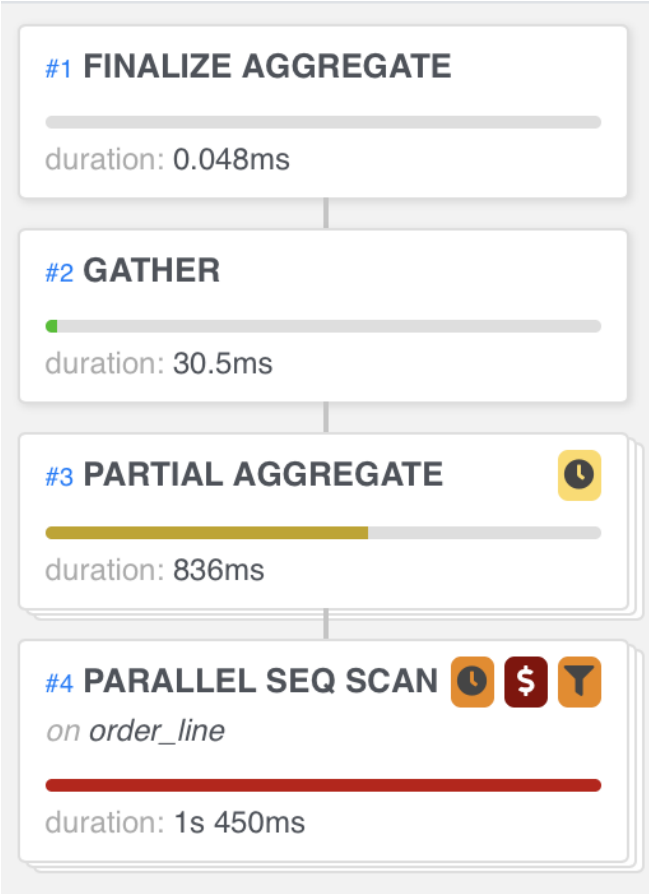

Figura 17 Query 6

Query 13

Número de clientes agrupados e classificados pelo tamanho dos pedidos que fizeram.

```
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;
```

Resultados Iniciais
Sort (cost=110864.15..110864.65 rows=200 width=16) (actual time=829.242..829.249 rows=50 loops=1) Sort Key: (count(*)) DESC, c_orders.c_count DESC Sort Method: quicksort Memory: 27kB -> GroupAggregate (cost=110832.00..110856.50 rows=200 width=16) (actual time=828.831..829.229 rows=50 loops=1) Group Key: c_orders.c_count -> Sort (cost=110832.00..110839.50 rows=3000 width=8) (actual time=828.822..828.987 rows=3000 loops=1)

```
Sort Key: c_orders.c_count DESC
Sort Method: quicksort Memory: 237kB
-> Subquery Scan on c_orders (cost=110598.74..110658.74 rows=3000 width=8) (actual time=827.656..828.352 rows=3000 loops=1)
-> HashAggregate (cost=110598.74..110628.74 rows=3000 width=12) (actual time=827.653..828.067 rows=3000 loops=1)
Group Key: customer.c_id
-> Hash Right Join (cost=80035.00..106548.74 rows=810000 width=8) (actual time=340.532..697.689 rows=810000 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..20166.51 rows=135521 width=16) (actual time=0.071..107.913 rows=135021 loops=1)
Filter: (o_carrier_id > 8)
Rows Removed by Filter: 793420
-> Hash (cost=61904.00..61904.00 rows=810000 width=12) (actual time=340.023..340.024 rows=810000 loops=1)
Buckets: 262144 Batches: 8 Memory Usage: 6390kB
-> Seq Scan on customer (cost=0.00..61904.00 rows=810000 width=12) (actual time=12.827..172.716 rows=810000 loops=1)
Planning Time: 0.291 ms
JIT:
Functions: 26
Options: Inlining false, Optimization false, Expressions true, Deforming true
Timing: Generation 2.490 ms, Inlining 0.000 ms, Optimization 0.578 ms, Emission 11.912 ms, Total 14.981 ms
Execution Time: 831.906 ms
```

Operações

1. CREATE MATERIALIZED VIEW **query_13** 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;
2. \d - **Confirmar criação**
3. explain analyze select c_count, count(*) as custdist from (select * from **query_13**) as c_orders (c_id, c_count) group by c_count order by custdist desc, c_count desc;
4. REFRESH MATERIALIZED VIEW query_13; - **Atualizar vista materializada**

Resultados Finais

```
Sort (cost=63.91..64.04 rows=50 width=16) (actual time=0.874..0.878 rows=50 loops=1)
Sort Key: (count(*)) DESC, query_13.count DESC
Sort Method: quicksort Memory: 27kB
-> HashAggregate (cost=62.00..62.50 rows=50 width=16) (actual time=0.845..0.853 rows=50 loops=1)
Group Key: query_13.count
-> Seq Scan on query_13 (cost=0.00..47.00 rows=3000 width=8) (actual time=0.009..0.269 rows=3000 loops=1)
Planning Time: 0.174 ms
Execution Time: 0.918 ms
```

Considerações Finais	Custo
Antes	831.906 ms

Depois	0.918 ms

Tabela 18 Trabalho realizado na Query 13

Observando os resultados obtidos concluímos com sucesso a otimização da *query* 13. Como a vista materializada guarda o conteúdo da consulta numa tabela, então a diferença do tempo de resposta foi brutal, seguidamente apresentamos alguns detalhes fornecidos pela ferramenta ***explain.dalibo.com.***

ANTES:

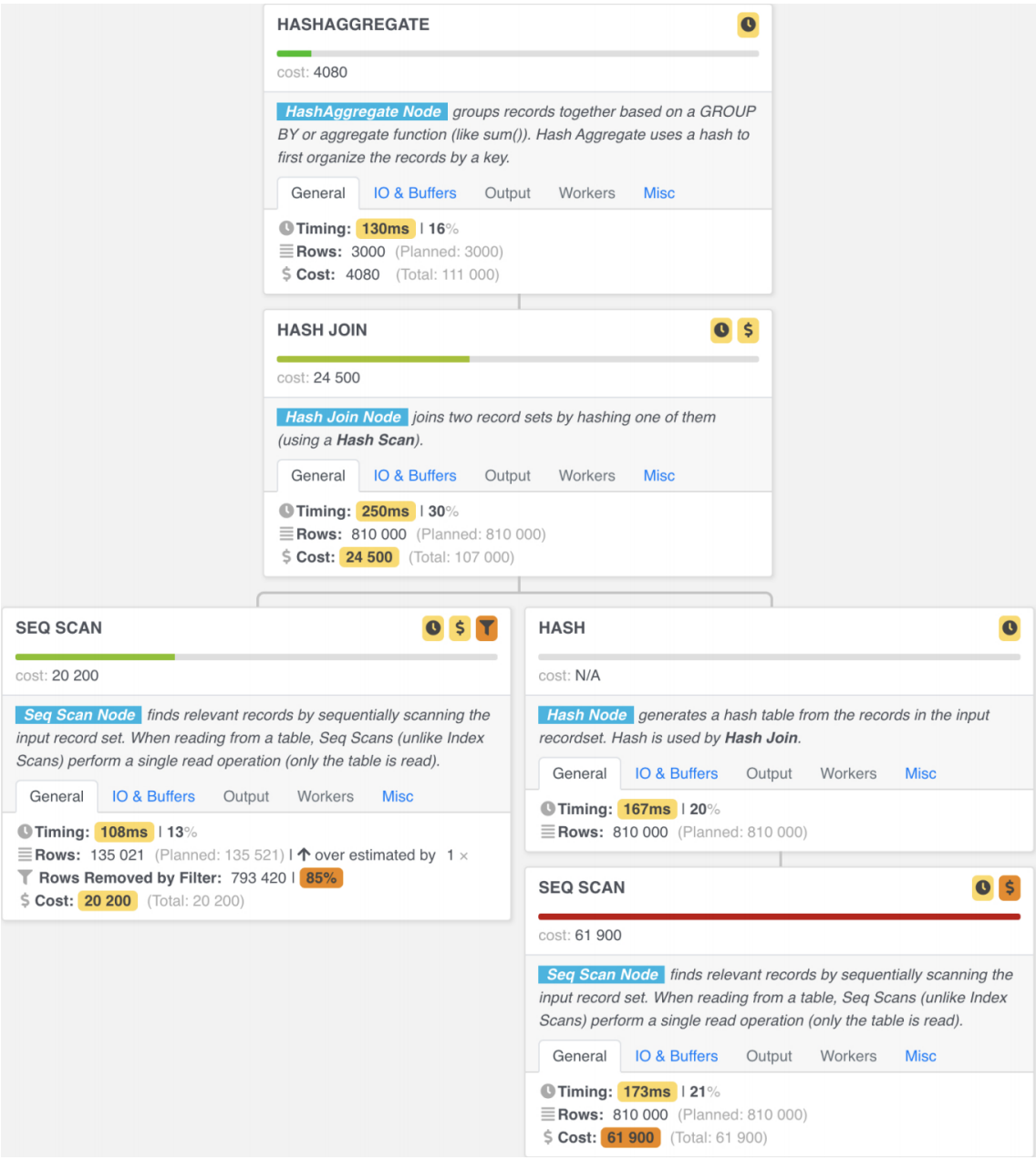

Figura 18 Query 13 antes da otimização

DEPOIS:

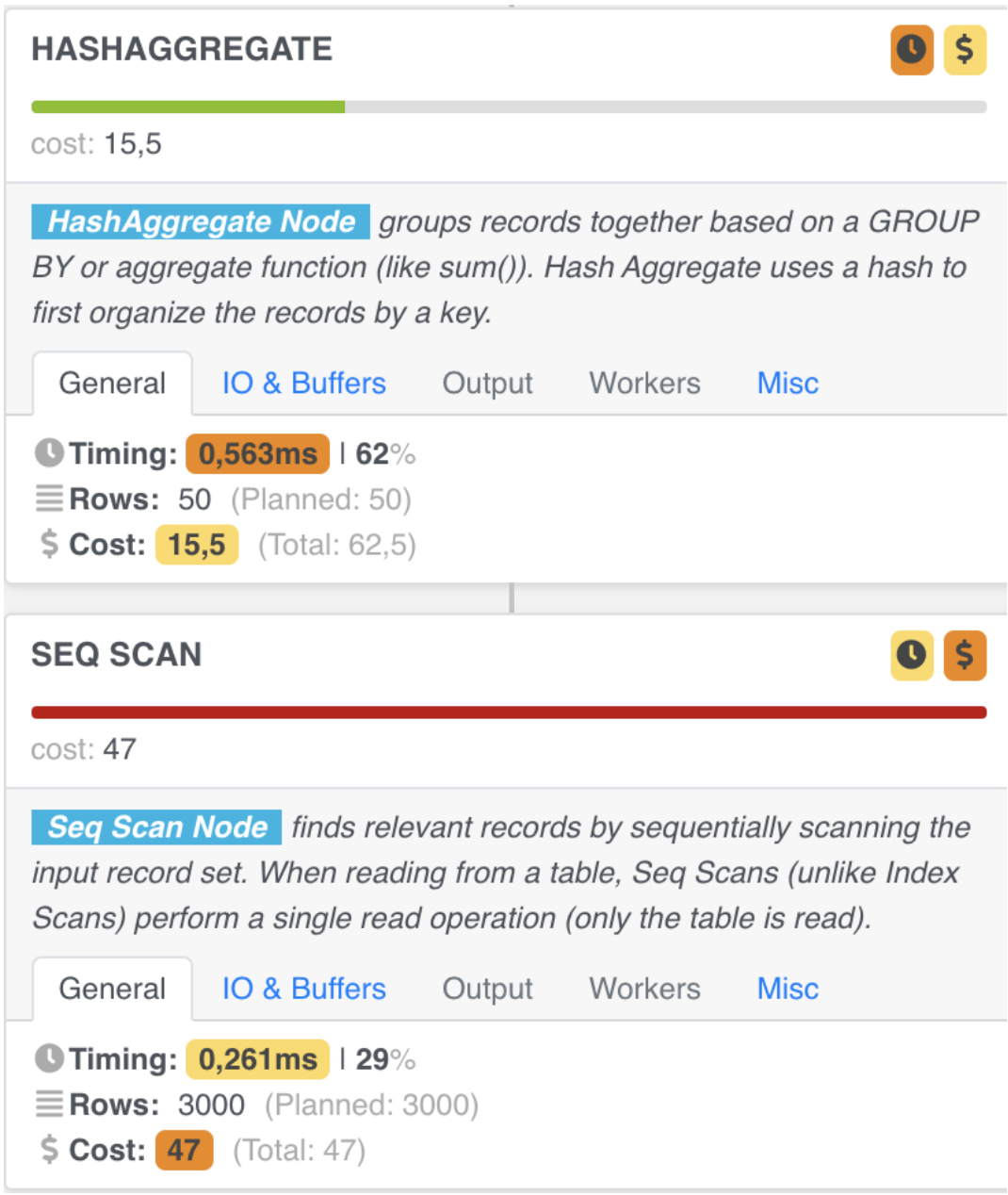

Figura 19 Query 13 depois da otimização

6. Análise demonstrativa

Operações
tpcc=# explain analyse select * from order_line; QUERY PLAN ---------- Seq Scan on order_line (cost=0.00..183632.06 rows=8074106 width=68) (actual time=0.130..855.492 rows=8074106 loops=1) Planning Time: 0.100 ms Execution Time: 1138.999 ms (3 rows)
tpcc=# explain analyse select * from order_line where ol_delivery_d >= '2020-01-01 00:00:00.000000'; QUERY PLAN ---------- Seq Scan on order_line (cost=0.00..203817.33 rows=6803256 width=68) (actual time=89.233..1330.896 rows=6887753 loops=1) Filter: (ol_delivery_d >= '2020-01-01 00:00:00'::timestamp without time zone) Rows Removed by Filter: 1186353 Planning Time: 6.677 ms JIT: Functions: 2 Options: Inlining false, Optimization false, Expressions true, Deforming true Timing: Generation 0.469 ms, Inlining 0.000 ms, Optimization 18.846 ms, Emission 68.524 ms, Total 87.838 ms Execution Time: 1770.464 ms (9 rows)

Tabela 19 Análise demonstrativa

Observando, a execução destas duas *queries* verificamos que ter menos linhas de resultado não implica ter menor custo. Isto é, a primeira *query* faz uma pesquisa geral, sem filtros, à tabela *order_line* obtendo um custo total de **1138.999 ms** e um resultado composto por 8074106 linhas, a segunda *query* é similar, contudo é constituída por um filtro para retornar as linhas das encomendas que foram entregues no ano de 2020, apenas esta condição o custo aumenta em **aproximadamente 638 ms** e o resultado final é composto por menos 1186353 linhas. Concluindo, o custo de execução de uma *query* depende principalmente da complexidade desta e não do tamanho do seu resultado final.

7. Conclusão

O desenvolvimento deste projeto prático, consistia em avaliar e otimizar o benchmark TPC-C, de modo a aplicar e ir mais além os conteúdos lecionados na Unidade Curricular de Administração de Base de Dados.

Mediante este desafio e tendo em conta todos os limites definidos pelo professor, de forma também a não gastar demasiados recursos monetários da Google Cloud Platform. Sendo assim, duas máquinas virtuais foram instaladas com o PostgreSQL e de modo a suportar as transações de forma gradual e equilibrada.

Seguidamente, idealizamos o número de *warehouses* e o número de clientes. Este processo foi bastante demorado e custoso pela maior adversidade encontrada na realização do projeto, os valores inconstantes das transações, várias tentativas eram efetuadas até que tudo se dissipou com a criação de um servidor com um disco mais veloz e maior.

Uma vez definidos o número de *warehouses* e o número de clientes, tudo ficou mais intuitivo. O processo de otimização do desempenho da carga transacional e configuração do PostgreSQL, baseou-se na leitura de documentação *online* e testes intensivos até conseguirmos um melhor resultado possível.

Já numa fase final, o objetivo debruçou-se na otimização de quatro interrogações analíticas, tendo em conta, principalmente, os respetivos planos e os mecanismos de redundância que estão a ser usados. Conceitos de sala de aula foram aplicados, como o *explain, explain analyze*, criação de índices e vistas materializadas, processo este que não trouxe transtornos inesperados.

De forma a concluir e reiterar, a elaboração deste trabalho permitiu a todos os constituintes do grupo alargar os seus conhecimentos relativamente a conceitos elaborados em contexto de aula e extra-aula.