UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

Benchmark TPC-C

Ana Marta Santos Ribeiro A82474
Jéssica Andreia Fernandes Lemos A82061
Miguel José Dias Pereira A78912

MIEI - 4º Ano - Administração de Bases de Dados

Braga, 24 de Outubro de 2020

Conteúdo

1 Introdução

Este relatório descreve todo o processo, bem como as tomadas de decisão e análise das mesmas, no âmbito do trabalho prático de Administração de Base de Dados do perfil de Engenharia de Aplicações no Mestrado Integrado em Engenharia Informática da Universidade do Minho.

O trabalho prático tem como objetivo obter uma configuração de referência, bem como otimizar o desempenho da carga transacional e das interrogações analíticas, de uma determinada base de dados, o *Benchmark TPC-C*.

Desta forma, realizou-se a análise da execução e de *logs* gerados pelo *PostgreSQL*, através do analisador pgBadger, bem como da comparação dos obtidos da execução de diversos testes.

2 Contextualização

De forma a atingir o objetivo pretendido começamos por analisar o *benchmark TPC-C*. Este corresponde a um sistema de base de dados de uma rede de armazéns. Deste modo, de seguida iremos apresentar a estrutura da mesma.

Tabela	Cardinalidade
warehouse	w
district	w*10
customer	w*30k
history	w*30k+
orders	w*30k+
new_order	w*30k+
item	100k
stock	w*100k
order_line	w*300k+

Tabela 1. Entidades

Como podemos verificar pela **Tabela 1**, o número de registos das tabelas da base de dados é definido em função do número de *warehouses* existente. A única que possui um número fixo é a *item* que neste caso corresponde a cem mil. De seguida, iremos apresentar o modelo conceptual onde é possível verificar os relacionamentos existentes, bem como a cardinalidade correspondente, que também está associado ao número de armazéns definido. Assim, podemos verificar que o tamanho da base de dados é estabelecido pelo número de *warehouses*.

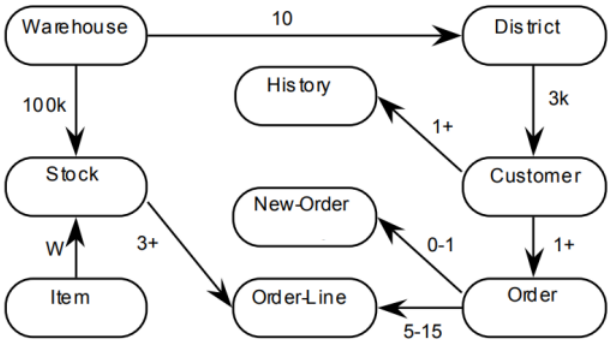


Figura 1. Modelo Conceptual

3 Scripts

Com o intuito de agilizar a realização dos diferentes testes optamos por desenvolver 3 scripts, nomeadamente:

- ***installation.sh*** - responsável por instalar os requisitos necessários para executar o *benchmark* como o *java* e *postgresql*.
- ***tests.sh*** - permite alterar as configurações do *workload-config.properties* de forma a testar com diferentes parâmetros. Para além disso, inicializa o *benchmark* e ainda copia para o *bucket* os resultados dos testes. É importante referir que neste ficheiro podem ser executados vários testes consecutivos.
- ***database.sh*** - responsável por criar, inicializar e restaurar a base de dados a partir de um *dump*.

4 Configuração de Referência

Com a intenção de otimizar e avaliar o *benchmark* definimos um conjunto de características base que se revelassem adequadas face ao objetivo pretendido. Assim sendo, de seguida iremos apresentar tanto a nível de hardware como de base de dados as especificações estabelecidas.

4.1 Especificação da Máquina

Tendo em conta que era pretendido a utilização do *Google Cloud Platform* como suporte à instalação e análise, começamos por definir as características das máquinas que pretendiamos utilizar.

Característica	**Especificação**
Sistema Operativo	Ubuntu 18.04 LTS
Disco	*HDD*
vCPUs	4
Memória	8 GB
Plataforma de CPU	Intel Haswell
Disco Adicional	*SSD*, 30 GB

Tabela 2. Características da Máquina

4.2 Base de dados

Como já foi referido na contextualização, o tamanho da base de dados é definido em função do número de *warehouse*. Como com 100 *warehouses* o tamanho da base de dados seria aproximadamente 11GB, optamos por este valor uma vez que é superior aos 8GB de memória existentes.

Para decidir o tempo de execução dos testes, começamos por ter em conta que para a obtenção da média dos dados é necessário permitir o aquecimento da cache de modo a ser possível avaliar na parte estável da mesma. Assim sendo, realizamos testes para 10, 15, 20 e 30 minutos cujos resultados do tempo de resposta são apresentados de seguida. De forma a analisar os gráficos tivemos em consideração o percentil 90, RT, que nos permite analisar o comportamento dos dados dado que possibilita a identificação

da gama dos valores onde se encontram 90% dos dados.

Na execução do teste de 10 minutos, apresentado na **Figura 2**, podemos verificar que existem alguns tempos de resposta que se afastam de forma considerável do *RT*.

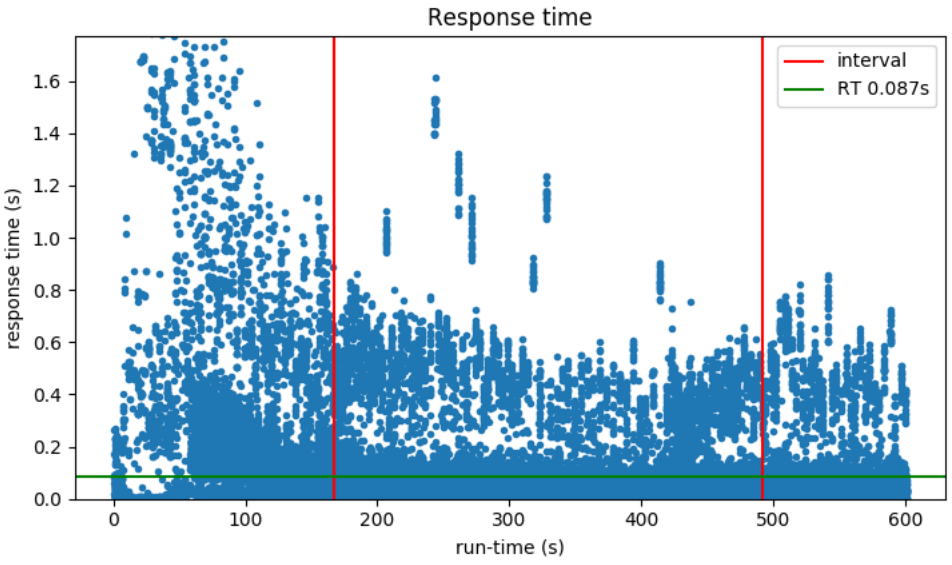


Figura 2. Execução para 10 min

Na **Figura 3**, para o teste de 15 min, é possível observar que continuam a existir valores dispersos, contudo o *RT* e o tempo de resposta máximo atingido diminuiram.

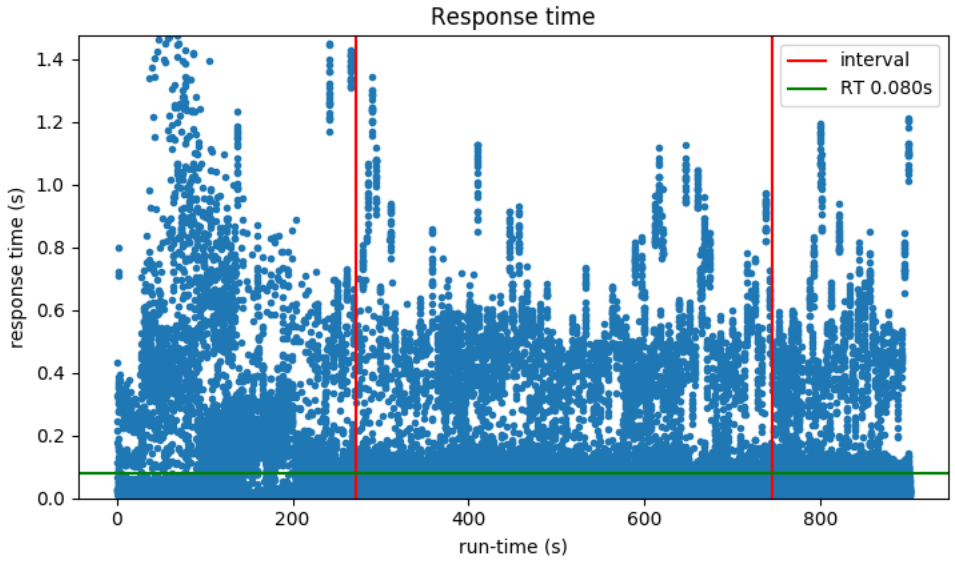


Figura 3. Execução para 15 min

Podemos concluir que o teste para 20 minutos, apresentado na **Figura 4**, é mais estável que os anteriores uma vez que apresenta menos picos, sendo que o valor máximo do tempo de resposta é consideravelmente mais baixo. Para além disso, constatamos que o *RT* reduziu.

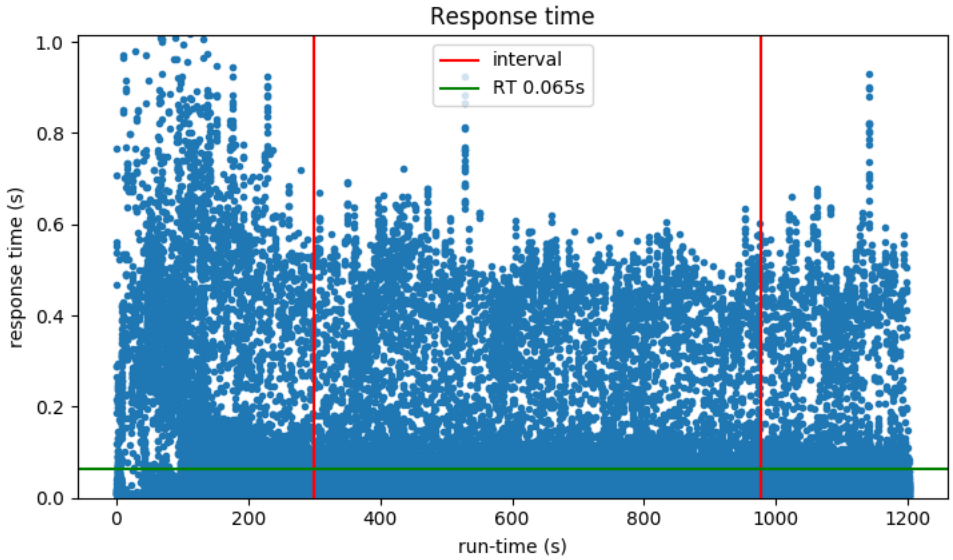


Figura 4. Execução para 20 min

Por fim, para o teste de 30 minutos, na **Figura 5**, verificamos que comparativamente ao anterior não melhorou a nível de estabilidade nem quanto ao tempo de resposta.

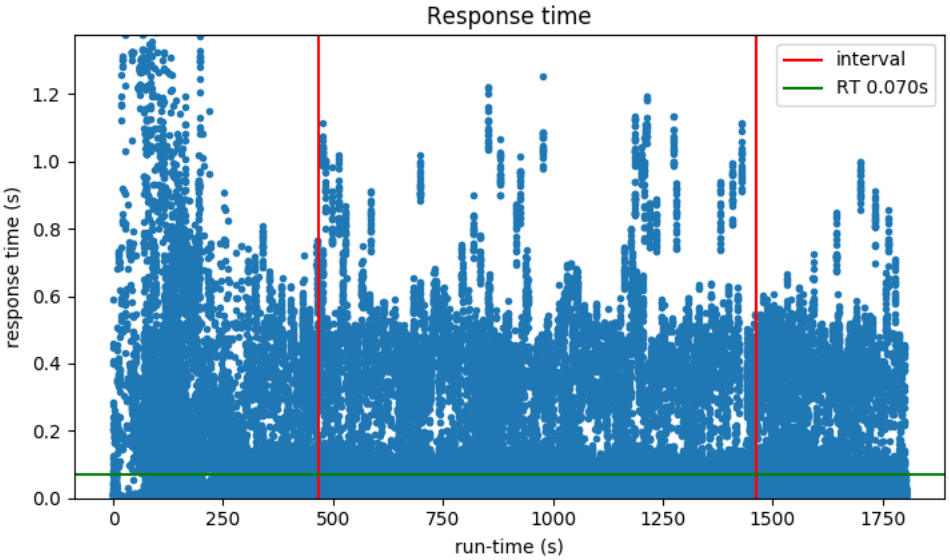


Figura 5. Execução para 30 min

Assim sendo, optamos por selecionar o teste de 20 minutos uma vez que é o mais estável e apresenta melhor tempo de resposta.

Para além disso, foi necessário definir o número de clientes por *warehouse* pelo que na **Tabela 3** apresentaremos para cada teste os respestivos resultados.

nº clientes/warehouse	**throughput (tx/s)**	**response_time (s)**	**abort_rate (s)**
10	64.09422	0.06478	0.26764
30	117.80331	0.16467	0.11743
50	122.01739	0.15909	0.11165
100	117.54494	0.16424	0.11215
150	134.79698	0.14447	0.09232
155	135.52039	0.14371	0.080251
200	130.85455	0.14802	0.10679

Tabela 3. Médias obtidas para os diferentes números de clientes por warehouse

Por fim, com o intuito de avaliar a estabilidade bem como possíveis problemas da configuração optamos por a cada execução guardar os *logs* gerados pelo PostgreSQL. Assim sendo é possível utilizando um analisador de *logs*, o *pgBadger*, analisá-los.

De forma a avaliarmos os resultados, optamos por apresentar o gráfico de *queries* por segundo, na **Figura 6**. Este representa as várias fases, contudo a mais interessante de analisar corresponde à parte intermédia do gráfico, que se refere à execução de *queries* por um período de 15 minutos. No gráfico apresentado é possível verificar que esta é relativamente estável não apresentando picos de performance.

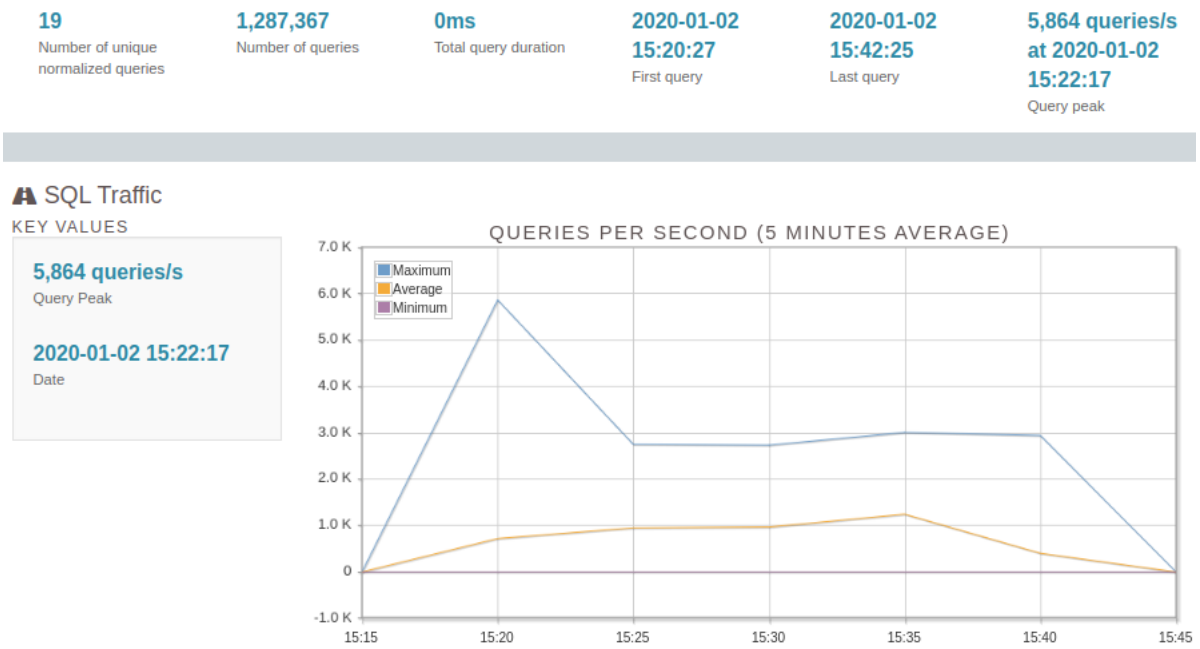


Figura 6. Gráfico queries por segundo

Para além disso, optamos por apresentar também um gráfico que indica as percentagens existentes de cada tipo de *log*, o que possibilita uma análise mais completa do comportamento da base de dados.

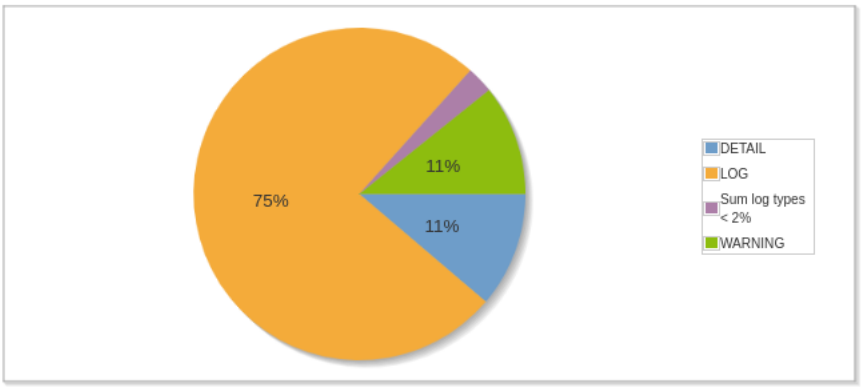


Figura 7. Logs por tipo

Nesta fase inicial testamos as interrogações analíticas de forma a registar quais os tempos de respostas para a configuração de referência. Assim sendo, na **Tabela 4** são indicados os tempos para cada das quatro interrogações, que se encontram em **Anexo**.

Interrogação	response_time (ms)
A1	74664.840
A2	76612.249
A3	171880.997
A4	103427.356

Tabela 4. Valores de referência das interrogações analíticas

5 Configuração do PostgreSQL

5.1 Memória

5.1.1 shared_buffers

Com o intuito de otimizar os valores obtidos através da configuração referência foi necessário realizar testes para verificar qual o valor mais adequado para o parâmetro *shared_buffers*, ou seja, a quantidade de memória para caching de dados. Através do estudo realizado previamente optamos por testar:

- 128 MG - É o valor *default*
- 1GB - Teste para um valor intermédio atendendo aos que consideramos importantes analisar
- 25% a 40% RAM - Tendo em conta que para sistemas com mais de um 1GB de RAM deve-se iniciar com um valor correspondente a 25% de RAM e para além disso não são esperadas melhorias para valores superiores a 40%

shared_buffers	**throughput (tx/s)**	**response_time (s)**	**abort_rate (s)**
128MB - Default	135.52039	0.14371	0.080251
	135.01145	0.14442	0.06078
1GB	148.44266	0.13159	0.05512
	188.69010	0.10348	0.04240
25% RAM - 2GB	174.84022	0.11131	0.07153
	167.11722	0.11679	0.05023
35% RAM - 2867MB	233.03181	0.08348	0.05067
	205.59085	0.09489	0.04486
40% RAM - 3267MB	240.5099	0.08113	0.04179
	217.17114	0.08972	0.03726

Tabela 5. Otimização dos *shared_buffers*

Como é expectável pretendemos selecionar o valores de *shared_buffer* que apresentam melhores resultados, contudo é importante ter em consideração que estes valores

devem ser consistentes. Por exemplo, na **Tabela 5**, podemos observar que para os valores de 1GB, 35% de RAM e 40% de RAM os valores obtidos de *throughput* em ambos os testes realizados variam consideravelmente. Posto isto, optamos por não selecioná-los. Relativamente ao de 128MB e ao de 25% de RAM constatámos que ao nível da percentagem de rollbacks, indicado pelo *abort_rate*, é semelhante assim como o tempo de resposta. Contudo, o de 2GB apresenta um *throughput* superior pelo que optamos por escolhê-lo para este parâmetro.

5.1.2 work_mem

De forma a obter o valor mais indicado para o *work_mem*, que corresponde à quantidade de memória despendida para operações internas de ordenação e *hashing* executamos os testes realizados para o parâmetro anterior.

work_mem	throughput (tx/s)	response_time (s)	abort_rate (s)
4MB - Default	179.51984	0.10882	0.04153
	178.40659	0.10920	0.04149
1GB	206.35191	0.09433	0.04705
	199.18553	0.09779	0.03942
25% RAM - 2GB	148.44266	0.13159	0.05512
	177.41314	0.10989	0.04368
35% RAM - 2867MB	195.27431	0.09923	0.08704
	163.2031	0.11927	0.04034
40% RAM - 3267MB	182.28254	0.10719	0.04045
	175.82787	0.11097	0.04242

Tabela 6. Otimização dos *work_mem*

Seguindo o raciocínio adotado na secção anterior podemos excluir as opções de 25%, 35% e 40% de RAM uma vez que os valores de *throughput* não se revelam consistentes. Tanto com 4MB como 1GB obtemos tempos consistentes de *throughput*, bem como valores semelhantes de tempo de resposta e percentagem de rollback. Tendo em conta que os valores de débito obtidos com 1GB são superiores optamos por selecionar este valor.

5.2 Write Ahead Log

5.2.1 wal_level

A quantidade de informação que é escrita nos logs é determinada pelo parâmetro *wal_level*. Assim, optamos 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*
- logical - adiciona as informações necessárias para suportar a descodificação lógica

Este parâmetro admite ainda outro valor, nomeadamente o *minimal*, que remove todos os *logs*, exceto as informações necessárias para recuperar de uma falha ou paragem. No entanto, não conseguimos testar com o mesmo.

wal_level	throughput (tx/s)	response_time (s)	abort_rate (s)
replica - Default	197.62352	0.09875	0.03987
	192.06281	0.10150	0.04082
logical	185.12791	0.10478	0.05388
	190.87882	0.10195	0.04688

Tabela 7. Otimização dos *wal_level*

Como podemos verificar na **Tabela 7** os valores para ambos os cenários de teste são estáveis. No entanto, decidimos manter o valor *default*, *replica*, dado que apresenta um débito superior. A nível de tempo de resposta e de percentagem de rollbacks estes têm um comportamento semelhante.

5.2.2 wal_buffers

De modo a criar os vários cenários de testes começamos por verificar que é aconselhável testarmos o parâmetro *wal_buffers*, ou seja, a quantidade de memória partilhada para informações WAL que não se encontram no disco, com megabytes.

wal_buffers	throughput (tx/s)	response_time (s)	abort_rate (s)
4MB	102.87357	0.18876	0.03862
	118.04256	0.16438	0.04932
8GB	103.21614	0.18872	0.03748
	117.69349	0.16515	0.05047
12MB	191.61230	0.10093	0.08727
	197.37651	0.9806	0.09892
16MB	196.46878	0.09942	0.04017
	207.56155	0.09383	0.03793
20MB	189.02744	0.10306	0.04306
	221.50692	0.08807	0.03760

Tabela 8. Otimização dos *wal_buffers*

Como já foi referido anteriormente começamos por excluir os valores cujo *throughput* não era estável. Desta forma, eliminamos o 4MB, 8 MB e 20MB. Tendo em conta que o de 12MB e 16MB apresentam tempos de resposta e percentagens de rollback idênticos optamos pelo de 16MB uma vez que apresenta um *throughput* superior.

6 Interrogações analíticas

Primeiramente analisamos os mecanismos de redundância já existentes no TPC-C. Constatámos que este já se encontra munido de vários índices, como se pode observar na **Figura 8**. Estes já aumentam o desempenho do TPC-C consideravelmente.

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

Figura 8. Índices TPC-C

Para a otimização das interrogações analíticas começamos por obter os respetivos planos, que se encontram em anexo, e fazendo uso do site *explain.dalibo.com* analisamos cada um dos planos.

Assim sendo vamos de seguida explicar o processo efetuado na tentativa de otimizar o desempenho para cada uma das interrogações analíticas.

6.1 A1

Começamos por analisar o respetivo plano na configuração de referência através da ferramenta *explain dalibo*.

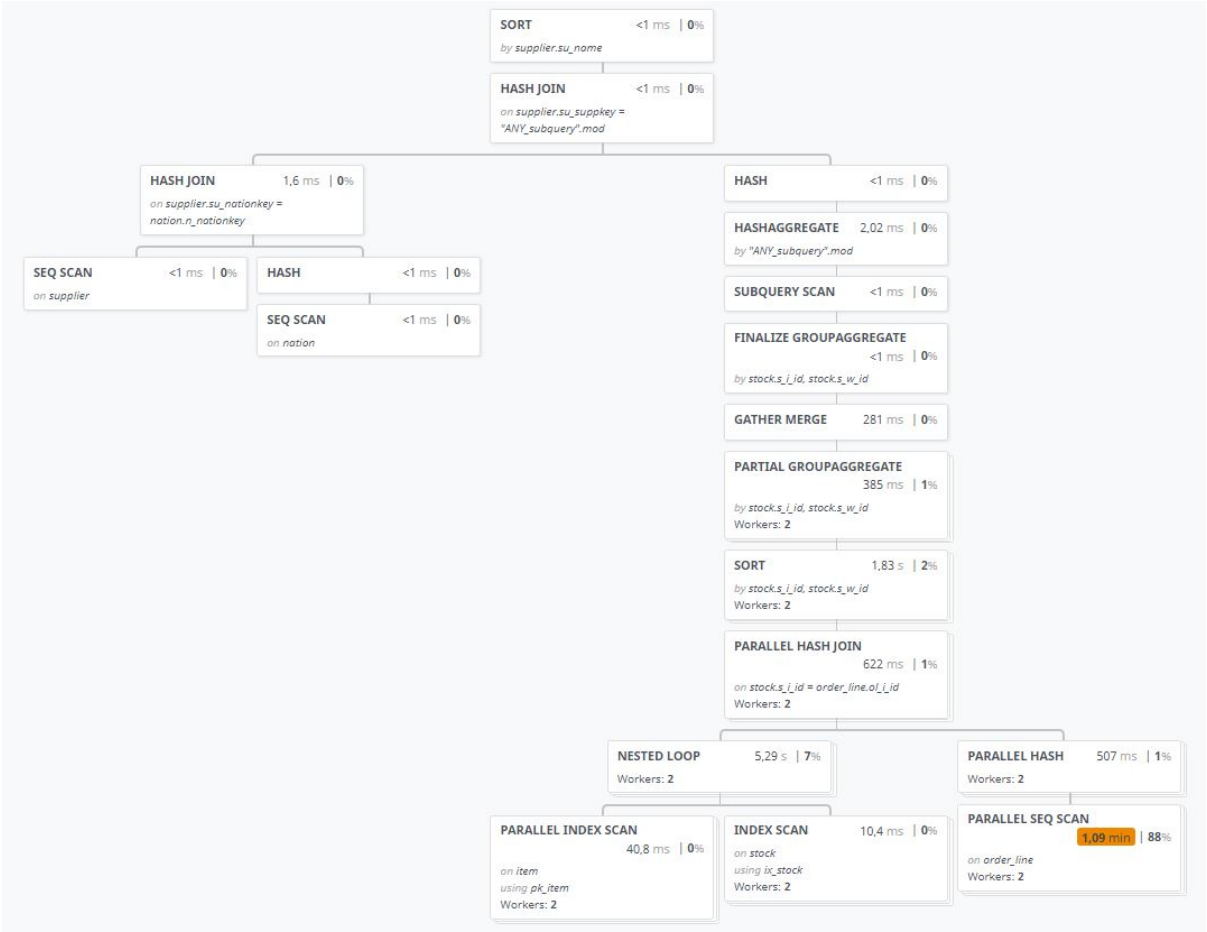


Figura 9. Análise inicial A1

Começamos por criar uma MATERIALIZED VIEW para a query que seleciona apenas os *i_id* que tenham o campo *i_data* iniciado pelo carater 'c', visto que a tabela *item* não sofrerá alterações regulares, e no *benchamark TPC-C* esta é estática:

```
CREATE MATERIALIZED VIEW A1_2
AS SELECT i_id
FROM item
WHERE i_data LIKE 'c%';
```

Sendo assim, alteramos então a query A1:

```
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 * from A1_2)
            and ol_i_id=s_i_id
            and extract(second from ol_delivery_d) > 50
            group by s_i_id, s_w_id, s_quantity
            having 2*s_quantity > sum(ol_quantity)
        )
    and su_nationkey = n_nationkey
    and n_name = 'GERMANY'
    order by su_name;
```

Os resultados obtidos indicam que houve uma redução de 40.8ms para valores inferiores a 1ms, como se pode observar na **Figura 10**.

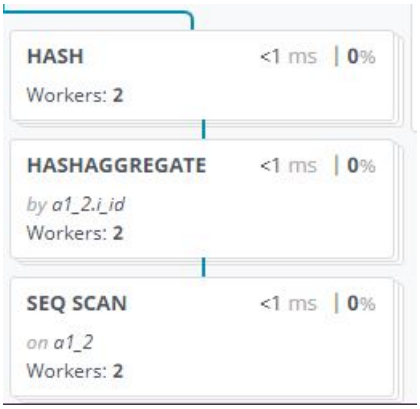


Figura 10. Resultado da otimização

Também se pode observar que grande parte do tempo perdido encontra-se na secção:

```
and extract(second from ol_delivery_d) > 50
```

Mas como a tabela *order_line* será uma tabela atualizada com regularidade, decidimos não efetuar o mesmo processo realizado para a tabela *item*.

6.2 A2

Mais uma vez analisamos o plano inicial obtido:

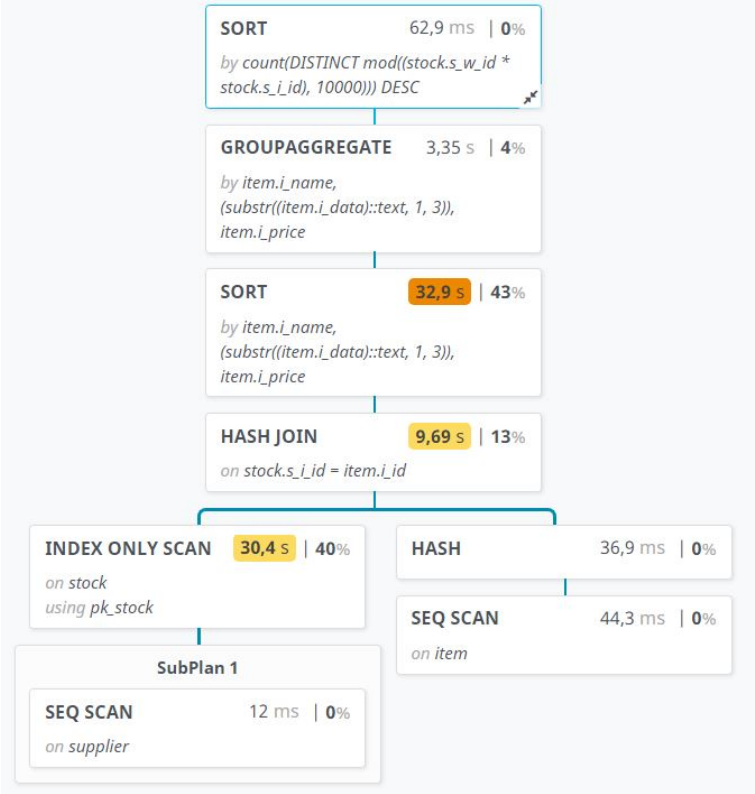


Figura 11. Análise inicial A2

Seguindo o mesmo processo que em A1, começamos por criar uma MATERIALIZED VIEW para a tabela *supplier*:

```
CREATE MATERIALIZED VIEW A2_1
AS SELECT su_suppkey
FROM supplier
WHERE su_comment LIKE '%bean%';
```

E uma MATERIALIZED VIEW para a tabela *item*:

```
CREATE MATERIALIZED VIEW A2_2
AS SELECT i_id, i_name, i_price, substr(i_data, 1, 3) AS i_data_substr
FROM item
WHERE i_data NOT LIKE 'z%'
GROUP BY i_name, i_data_substr, i_price;
```

E adaptando a interrogação A2 para utilizar as MATERIALIZED VIEWS:

```
select i_name, i_data_substr as brand, i_price,
    count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
from A2_2, stock
where i_id = s_i_id
and (mod((s_w_id * s_i_id),10000) not in
    (select * from A2_1))
group by i_name, i_data_substr, i_price
order by supplier_cnt desc;
```

Diminuindo assim o tempo total da interrogação em aproximadamente 4 segundos:

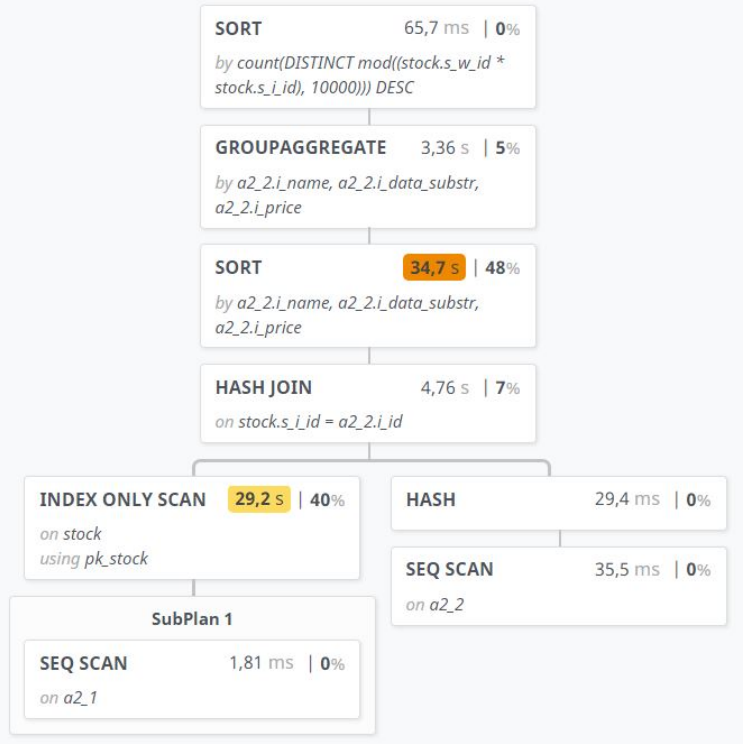


Figura 12. Análise a A2

6.3 A3 e A4

Estas duas interrogações são as que demoram mais tempo a executar uma vez que ambas fazem *join* de várias tabelas com várias condições. Torna-se então difícil à partida melhorar os tempos para estas duas interrogações.

Numa primeira análise aos planos de ambas as interrogações constatámos que alguns dos índices que o TPC-C implementa estão a ser utilizados. Assim sendo, decidimos criar novos índices na tentativa de melhorar o desempenho de ambas as interrogações:

```
CREATE INDEX idx_7 ON customer(c_w_id,c_d_id);
CREATE INDEX idx_8 ON customer(c_w_id);
CREATE INDEX idx_9 ON customer(c_d_id);
CREATE INDEX idx_10 ON order_line(ol_w_id);
CREATE INDEX idx_11 ON order_line(ol_d_id);
CREATE INDEX idx_12 ON stock(s_w_id);
CREATE INDEX idx_13 ON stock(s_i_id);
```

No entanto, não conseguimos que os índices fossem úteis para as interrogações e como tal não conseguimos melhorar os desempenhos de ambas.

7 Replicação

De modo a utilizar replicação na configuração, recorremos à plataforma *Dremio*. Esta caracteriza-se pela sua flexibilidade, pela capacidade de realizar consultas extremamente rápidas e pela facilidade de realizar JOINs. O *Dremio* recorre às *reflections*, que consistem em representações físicas e otimizadas dos dados fonte. Assim sendo, podem ser utilizadas para acelarar o processamento das *queries*. Os tipos de *reflections* existentes são:

- Raw Reflection - Permite particionar e classificar os dados, com eficiência para consultas.
- Aggregate Reflection - Os dados estão pré-agregados, com base numa seleção de dimensões e medidas, sendo permitido também particinar e classificar os mesmos.
- External - Permite a utilização de conjuntos de dados e tabelas existentes em sistemas externos.

Optamos por criar uma *Raw Reflection*, através das sub-queries da A1, sendo apresentados os resultados da execução na **Figura 13**.

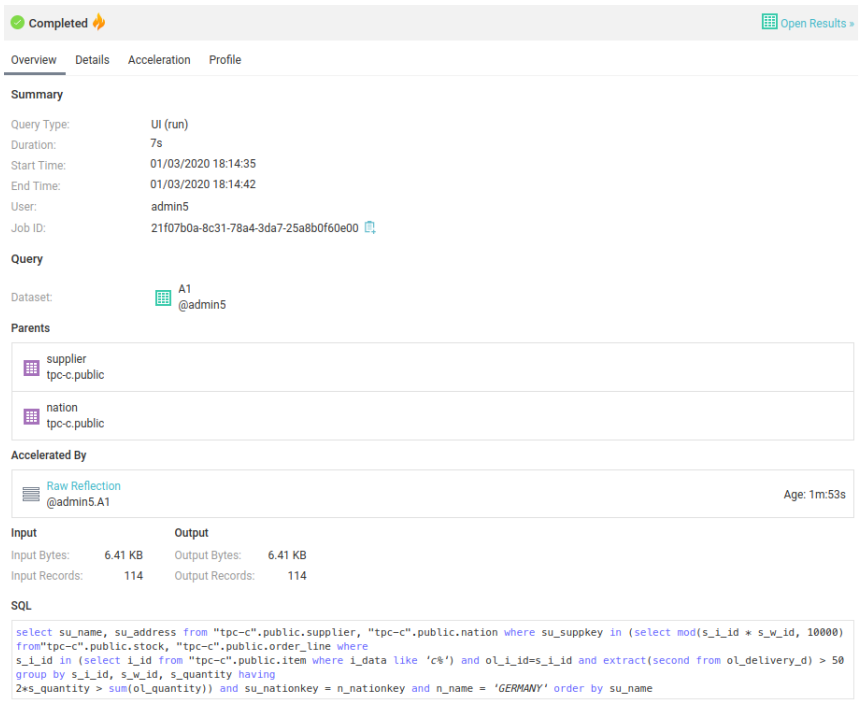


Figura 13. Execução da query A1

8 Conclusão

O desenvolvimento deste trabalho prático, que consistia em avaliar e otimizar o *benchmark TPC-C*, permitiu-nos aplicar os conteúdos lecionados na unidade curricular de Administração de Base de Dados.

Neste trabalho foram aplicados vários conceitos lecionados nas aulas, de modo a otimizar o desempenho da carga transacional e as interrogações analíticas. Para tal, recorremos a ferramentas como o *pgBager*, para analisar *logs* e *dremio*, para a replicação.

Tendo em conta que o grupo nunca tinha trabalhado com o *Google Cloud* tornou-se necessário procurar estratégias para poupar recursos na mesma, tal como armazenar os dados em *Cloud Storage*, bem como automatizar a instalação e execução do *benchmark*. Assim, para além de pouparmos recursos, foi possível obter resultados em maior quantidade, e como tal, uma análise mais profunda. Contudo, este processo tornou-se um pouco demorado e como tal atrasou as restantes etapas do projeto. Deste modo, apesar de concluirmos o projecto com sucesso, consideramos que pode, no futuro, ser feita uma maior exploração da base de dados, testando outros parâmetros para a configuração da mesma que não consideramos tão relevantes devido à escassez de tempo. Para além disso, na replicação pretendiamos implementar também uma *Raw Reflection* para a *query* A2 e uma *Aggregate Reflection* para a A3 e A4, contudo tivemos problemas durante os testes e não houve tempo para verificar o problema que estava a acontecer com *Dremio*.

Anexos

A. Interrogações analíticas

A1

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

Plano A1

```
Sort  (cost=2228680.02..2228680.52 rows=200 width=51) (actual time=74587.954..74587.960 rows=111 loops=1)
  Sort Key: supplier.su_name
  Sort Method: quicksort  Memory: 37kB
  ->  Hash Join  (cost=2228305.60..2228672.37 rows=200 width=51) (actual time=74585.398..74587.817 rows=111 loops=1)
        Hash Cond: (supplier.su_suppkey = "ANY_subquery".mod)
        ->  Hash Join  (cost=1.32..364.82 rows=400 width=55) (actual time=0.814..3.224 rows=396 loops=1)
              Hash Cond: (supplier.su_nationkey = nation.n_nationkey)
              ->  Seq Scan on supplier  (cost=0.00..322.00 rows=10000 width=59) (actual time=0.014..0.849 rows=10000 loops=1)
              ->  Hash  (cost=1.31..1.31 rows=1 width=4) (actual time=0.772..0.773 rows=1 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                    ->  Seq Scan on nation  (cost=0.00..1.31 rows=1 width=4) (actual time=0.764..0.767 rows=1 loops=1)
                          Filter: (n_name = 'GERMANY'::bpchar)
                          Rows Removed by Filter: 24
        ->  Hash  (cost=2228301.77..2228301.77 rows=200 width=4) (actual time=74584.507..74584.508 rows=2628 loops=1)
              Buckets: 4096 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 125kB
              ->  HashAggregate  (cost=2228299.77..2228301.77 rows=200 width=4) (actual time=74583.634..74584.161 rows=2628 loops=1)
                    Group Key: "ANY_subquery".mod
                    ->  Subquery Scan on "ANY_subquery"  (cost=1876026.71..2227466.02 rows=333502 width=4) (actual time=73663.354..74582.137 rows=3398 loops=1)
                          ->  Finalize GroupAggregate  (cost=1876026.71..2224131.00 rows=333502 width=16) (actual time=73663.353..74581.683 rows=3398 loops=1)
                                Group Key: stock.s_i_id, stock.s_w_id
                                Filter: ((2 * stock.s_quantity) > sum(order_line.ol_quantity))
                                Rows Removed by Filter: 148902
                                ->  Gather Merge  (cost=1876026.71..2192448.34 rows=2001010 width=20) (actual time=73661.769..74602.147 rows=152300 loops=1)
                                      Workers Planned: 2
                                      Workers Launched: 2
                                      ->  Partial GroupAggregate  (cost=1875026.69..1960482.11 rows=1000505 width=20) (actual time=73560.408..74321.301 rows=50767 loops=3)
                                            Group Key: stock.s_i_id, stock.s_w_id
                                            ->  Sort  (cost=1875026.69..1893889.28 rows=7545037 width=16) (actual time=73560.375..73936.489 rows=2280967 loops=3)
                                                  Sort Key: stock.s_i_id, stock.s_w_id
                                                  Sort Method: external merge  Disk: 85584kB
                                                  Worker 0:  Sort Method: external merge  Disk: 44240kB
                                                  Worker 1:  Sort Method: external merge  Disk: 44512kB
                                                  ->  Parallel Hash Join  (cost=557840.75..755221.75 rows=7545037 width=16) (actual time=71543.267..72108.800 rows=2280967 loops=3)
                                                        Hash Cond: (stock.s_i_id = order_line.ol_i_id)
                                                        ->  Nested Loop  (cost=0.73..138912.36 rows=84209 width=16) (actual time=4.227..5340.771 rows=50767 loops=3)
                                                              ->  Parallel Index Scan using pk_item on item  (cost=0.29..3667.13 rows=842 width=4) (actual time=2.074..40.781 rows=508 loops=3)
                                                                    Filter: (i_data ~~ 'c%'::text)
                                                                    Rows Removed by Filter: 32826
                                                              ->  Index Scan using ix_stock on stock  (cost=0.43..157.15 rows=347 width=12) (actual time=2.662..10.413 rows=100 loops=1523)
                                                                    Index Cond: (s_i_id = item.i_id)
                                                        ->  Parallel Hash  (cost=498167.67..498167.67 rows=3637148 width=8) (actual time=66145.916..66145.917 rows=1492383 loops=3)
                                                              Buckets: 131072  Batches: 128  Memory Usage: 2528kB
                                                              ->  Parallel Seq Scan on order_line  (cost=0.00..498167.67 rows=3637148 width=8) (actual time=7.433..65639.167 rows=1492383 loops=3)
                                                                    Filter: (date_part('second'::text, ol_delivery_d) > '50'::double precision)
                                                                    Rows Removed by Filter: 7236482
```

A2

```
EXPLAIN ANALYSE
select i_name,
substr(i_data, 1, 3) as brand,
i_price,
count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
from stock, item
where i_id = s_i_id
and i_data not like 'z%'
and (mod((s_w_id * s_i_id),10000) not in
(select su_suppkey
from supplier
where su_comment like '%bean%'))
group by i_name, substr(i_data, 1, 3), i_price
order by supplier_cnt desc;
```

Plano A2

```
Sort  (cost=1573598.14..1573845.62 rows=98990 width=71) (actual time=76529.761..76543.536 rows=98505 loops=1)
   Sort Key: (count(DISTINCT mod((stock.s_w_id * stock.s_i_id), 10000))) DESC
   Sort Method: external merge  Disk: 5608kB
   ->  GroupAggregate  (cost=1473295.03..1561320.95 rows=98990 width=71) (actual time=69171.682..76480.594 rows=98505 loops=1)
       Group Key: item.i_name, (substr((item.i_data)::text, 1, 3)), item.i_price
       ->  Sort  (cost=1473295.03..1485658.04 rows=4945204 width=71) (actual time=69171.442..73125.828 rows=8832678 loops=1)
             Sort Key: item.i_name, (substr((item.i_data)::text, 1, 3)), item.i_price
             Sort Method: external merge  Disk: 466744kB
             ->  Hash Join  (cost=5730.84..517783.72 rows=4945204 width=71) (actual time=97.319..40178.937 rows=8832678 loops=1)
                 Hash Cond: (stock.s_i_id = item.i_id)
                 ->  Index Only Scan using pk_stock on stock  (cost=350.46..434179.17 rows=4995660 width=8) (actual time=14.201..30411.235
                 rows=8966900 loops=1)
                       Filter: (NOT (hashed SubPlan 1))
                       Rows Removed by Filter: 1033100
                       Heap Fetches: 0
                       SubPlan 1
                       ->  Seq Scan on supplier  (cost=0.00..347.00 rows=1212 width=4) (actual time=0.834..12.032 rows=1071 loops=1)
                             Filter: ((su_comment)::text ~~ '%bean%'::text)
                             Rows Removed by Filter: 8929
                 ->  Hash  (cost=2789.00..2789.00 rows=98990 width=86) (actual time=81.278..81.279 rows=98505 loops=1)
                       Buckets: 32768  Batches: 4  Memory Usage: 3164kB
                       ->  Seq Scan on item  (cost=0.00..2789.00 rows=98990 width=86) (actual time=0.855..44.331 rows=98505 loops=1)
                           Filter: (i_data !~~ 'z%'::text)
                           Rows Removed by Filter: 1495
```

A3

```
EXPLAIN ANALYSE
select n_name,
sum(ol_amount) as revenue
from customer, orders, order_line, stock, supplier, nation, region
where c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and ol_o_id = o_id
and ol_w_id = o_w_id
and ol_d_id= o_d_id
and ol_w_id = s_w_id
and ol_i_id = s_i_id
and mod((s_w_id * s_i_id),10000) = su_suppkey
and ascii(substr(c_state,1,1))-ascii('a') = su_nationkey
and su_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'EUROPE'
group by n_name
order by revenue desc;
```

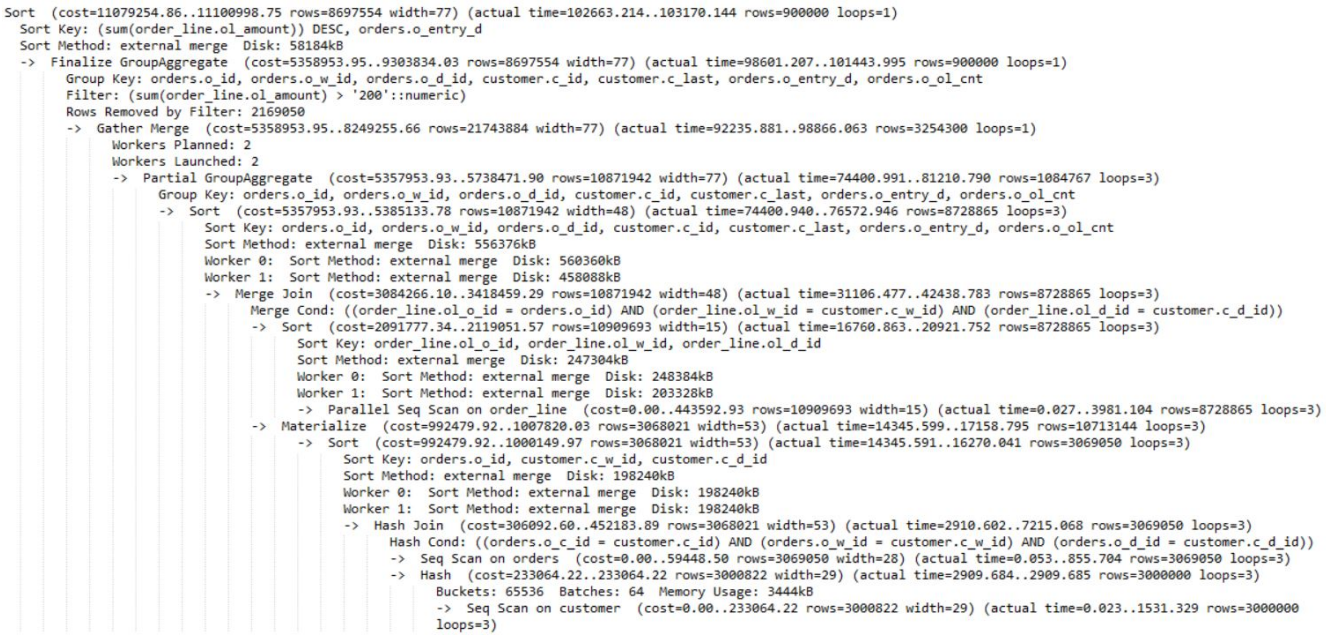
Plano A3

```
Sort  (cost=1183023.88..1183023.94 rows=25 width=58) (actual time=171702.662..171702.663 rows=5 loops=1)
  Sort Key: (sum(order_line.ol_amount)) DESC
  Sort Method: quicksort  Memory: 25kB
  ->  Finalize GroupAggregate  (cost=1182935.39..1183023.30 rows=25 width=58) (actual time=171694.219..171702.645 rows=5 loops=1)
        Group Key: nation.n_name
        ->  Gather Merge  (cost=1182935.39..1183022.61 rows=50 width=58) (actual time=171692.201..171880.064 rows=15 loops=1)
              Workers Planned: 2
              Workers Launched: 2
              ->  Partial GroupAggregate  (cost=1181935.37..1182016.81 rows=25 width=58) (actual time=171636.343..171644.627 rows=5 loops=3)
                    Group Key: nation.n_name
                    ->  Sort  (cost=1181935.37..1181962.41 rows=10818 width=29) (actual time=171634.205..171636.525 rows=25603 loops=3)
                          Sort Key: nation.n_name
                          Sort Method: quicksort  Memory: 3647kB
                          Worker 0:  Sort Method: quicksort  Memory: 2419kB
                          Worker 1:  Sort Method: quicksort  Memory: 2418kB
                          ->  Hash Join  (cost=623340.79..1181210.50 rows=10818 width=29) (actual time=170703.908..171621.371 rows=25603 loops=3)
                                Hash Cond: ((mod((stock.s_w_id * stock.s_i_id), 10000) = supplier.su_suppkey) AND (nation.n_nationkey = supplier.su_nationkey))
                                ->  Parallel Hash Join  (cost=622868.79..1177925.90 rows=270442 width=44) (actual time=170695.871..171455.125 rows=640349 loops=3)
                                      Hash Cond: ((customer.c_w_id = stock.s_w_id) AND (order_line.ol_i_id = stock.s_i_id))
                                      ->  Nested Loop  (cost=243549.50..775607.67 rows=271799 width=52) (actual time=32822.728..167703.604 rows=640349 loops=3)
                                            Join Filter: ((customer.c_w_id = order_line.ol_w_id) AND (customer.c_d_id = order_line.ol_d_id))
                                            ->  Hash Join  (cost=243548.94..325264.05 rows=31959 width=53) (actual time=32819.270..34117.407 rows=75044 loops=3)
                                                  Hash Cond: ((ascii(substr((customer.c_state)::text, 1, 1)) - 97) = nation.n_nationkey)
                                                  ->  Parallel Hash Join  (cost=243546.41..313756.44 rows=1278342 width=23) (actual time=32817.058..33638.950 rows=1023017 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..41545.71 rows=1278771 width=16) (actual time=3.222..8906.069 rows=1023017 loops=3)
                                                        ->  Parallel Hash  (cost=215559.42..215559.42 rows=1250342 width=15) (actual time=23445.009..23445.009 rows=1000000 loops=3)
                                                              Buckets: 131072  Batches: 64  Memory Usage: 3296kB
                                                              ->  Parallel Seq Scan on customer  (cost=0.00..215559.42 rows=1250342 width=15) (actual time=2.918..22882.661 rows=1000000 loops=3)
                                                  ->  Hash  (cost=2.47..2.47 rows=5 width=30) (actual time=2.176..2.176 rows=5 loops=3)
                                                        Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                                        ->  Hash Join  (cost=1.07..2.47 rows=5 width=30) (actual time=2.167..2.173 rows=5 loops=3)
                                                              Hash Cond: (nation.n_regionkey = region.r_regionkey)
                                                              ->  Seq Scan on nation  (cost=0.00..1.25 rows=25 width=34) (actual time=1.253..1.256 rows=25 loops=3)
                                                              ->  Hash  (cost=1.06..1.06 rows=1 width=4) (actual time=0.899..0.899 rows=1 loops=3)
                                                                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                                                    ->  Seq Scan on region  (cost=0.00..1.06 rows=1 width=4) (actual time=0.894..0.895 rows=1 loops=3)
                                                                          Filter: (r_name = 'EUROPE'::bpchar)
                                                                          Rows Removed by Filter: 4
                                            ->  Index Scan using pk_order_line on order_line  (cost=0.56..13.96 rows=9 width=19) (actual time=1.649..1.775 rows=9 loops=225131)
                                                  Index Cond: ((ol_w_id = orders.o_w_id) AND (ol_d_id = orders.o_d_id) AND (ol_o_id = orders.o_id))
                                      ->  Parallel Hash  (cost=300611.53..300611.53 rows=4163050 width=8) (actual time=2119.079..2119.079 rows=3333333 loops=3)
                                            Buckets: 131072  Batches: 128  Memory Usage: 4128kB
                                            ->  Parallel Index Only Scan using pk_stock on stock  (cost=0.43..300611.53 rows=4163050 width=8) (actual time=0.086..1170.678 rows=3333333 loops=3)
                                                  Heap Fetches: 0
                                ->  Hash  (cost=322.00..322.00 rows=10000 width=8) (actual time=7.813..7.813 rows=10000 loops=3)
                                      Buckets: 16384  Batches: 1  Memory Usage: 519kB
                                      ->  Seq Scan on supplier  (cost=0.00..322.00 rows=10000 width=8) (actual time=0.733..5.413 rows=10000 loops=3)
```

A.4

```
EXPLAIN ANALYSE
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
```

Plano A4



B Automatização

Installation

```
dir=/home/$USER/EscadaTPC-C
tpcc_tar=tpc-c-0.1-SNAPSHOT-tpc-c.tar.gz
tpcc_folder=$dir/tpc-c-0.1-SNAPSHOT
db_name=tpcc
execution=$1

# Instalar java
sudo apt-get update
sudo apt-get install openjdk-8-jre
echo "export JAVA_HOME='/usr/lib/jvm/java-8-openjdk-amd64/'" >> /home/$USER/.bashrc
echo "export PATH=$PATH:$JAVA_HOME/bin" >> /home/$USER/.bashrc
source /home/$USER/.bashrc

# Instalar Postgresql 11
```

```
sudo apt-get install wget ca-certificates
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc
sudo apt-key add -
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt/ `lsb_release
-cs`-pgdg main" >> /etc/apt/sources.list.d/pgdg.list'
sudo apt-get update
sudo apt-get install postgresql-11 postgresql-contrib
echo "export PATH=/usr/lib/postgresql/11/bin/:$PATH" >> /home/$USER/.bashrc
source /home/$USER/.bashrc

# Instalar tmux
sudo apt-get install tmux

if [ ! -d $dir ]
then
mkdir -p $dir
fi

# Obter tpcc do bucket
gsutil cp gs://abd-bucket19/$tpcc_tar $dir
tar -xzf $dir/$tpcc_tar -C $dir
rm -rf $dir/$tpcc_tar

set_property(){
sed -i -e "s/\($1 \?= \?\).*$/\1$2/g" $tpcc_folder/etc/database-config.properties
}
set_property db.username $USER
set_property db.password

tmux new-session -d -s exe
sed -i 's/\r//' execute.sh
tmux send-keys -t exe "./execute.sh $execution" "C-m"
```

Tests

```
test=$1
dir=/home/$USER/EscadaTPC-C
tpcc_folder=$dir/tpc-c-0.1-SNAPSHOT
workload_config=$tpcc_folder/etc/workload-config.properties
postgres=$tpcc_folder/etc/sql/postgresql

run() {
     rm $tpcc_folder/*.dat
     date_time=$(date +"%T")
     echo "Init run: $date_time"
     cd $tpcc_folder
     time ./run.sh > /dev/null
     echo "Finish run"
}

restore() {
     date_time=$(date +"%T")
     echo "Init restore: $date_time"
     cd /home/$USER
     sed -i 's/\r//' restore.sh
     ./restore.sh
     echo "Finish restore"
}

to_bucket() {
     echo "Copy test"
     cd $tpcc_folder
     date_time=$(date +"%F_%T")
     new_test=$"$test"
     mv *.dat "$new_test$date_time".dat
     gsutil cp *.dat gs://abd-bucket19/logs/
     echo "Copy log"
```

```
    rm *.dat
    cd /mnt/disks/sdb/tpcc/log
    mv *.log "$new_test$date_time".log
    gsutil cp *.log gs://abd-bucket19/logs/psql_logs/
    rm *.log
}

set_property_workload() {
    sed -i -e "s/\($1 \?= \?\).*$/\1$2/g" $workload_config
}

# Tests
restore
set_property_workload measurement.time 20
set_property_workload tpcc.number.warehouses 100
set_property_workload tpcc.numclients 155
run
to_bucket
```

Database

```
db_name=tpcc
db_dump=dump-10gb.bak
dir=/mnt/disks/sdb

set_property_postgres() {
    sed -i -e "s/#\?\($1 \?= \?\).*$/\1$2/g" $dir/$db_name/postgresql.conf
}

if [ ! -d "$dir" ]
then
mkdir -p $dir
fi
```

```
rm -rf $dir/$db_name
pkill -f "postgres"
sudo service postgresql stop
sleep 5
export PATH=/usr/postgresql/11/bin/:$PATH
initdb -D $dir/$db_name
set_property_postgres log_statement \'all\'
nohup postgres -D $dir/$db_name -k . >/dev/null 2>&1 &
sleep 10
createdb -h localhost $db_name

if [ ! -f $dir/$db_dump ]
then
gsutil cp gs://abd-bucket19/$db_dump $dir
fi

time pg_restore -x -h localhost -d $db_name -Fc $dir/$db_dump > /dev/null
```