Universidade de Coimbra

Faculdade de Ciências e Tecnologia

Departamento de Engenharia Informática

Mestrado em Engenharia Informática

Sistemas de Gestão de Dados

Projeto 1, 2020/2021, 2º Semestre

Docentes

Professor Pedro Furtado
(PNF@DEI.UC.PT)

David Silva de Paiva	davidpaiva@student.dei.uc.pt	2020178529
Ricardo David Da Silva Briceño	ricardodavid120@gmail.com	2020173503

Coimbra, 24 de março de 2021

Registo de Trabalhos

Lista de funcionalidades/objetivos implementados

Funcionalidades/Objetivos	Responsável	Esforço (horas)
Carregamento dos dados no Postgres	Ambos	3
Cálculo de MB/s no Carregamento dos dados no Postgres	Ambos	1/4
Cálculo do número médio de linhas carregas por segundo no Carregamento dos dados no Postgres	Ambos	1/4
Registo do tempo médio de execução de queries s/chave	Ambos	3
Registo do tempo médio de criação de chaves primárias e estrangeiras	Ambos	2
Registo do tempo médio de execução de queries c/chave	Ambos	4
Construção de gráficos	David	1
Query Plan Q1.2	David	2
Query Plan Q2.2	Ricardo	2
Query Plan Q3.2	David	2
Query Plan Q2.4	Ricardo	2
Relatório	Ambos	10

Tabela 1 – Objetivos Alcançados

Autoavaliação individual e global do projeto

O aluno, David Paiva, autoavalia o desempenho do grupo com uma nota de dezanove valores - numa escala de zero a vinte. Relativamente a uma autoavaliação individual, avalia o seu desempenho e o do colega, Ricardo Briceño, com dezanove valores para ambos.

O aluno, Ricardo Briceño, autoavalia o desempenho do grupo com uma nota de dezanove valores - numa escala de zero a vinte. Relativamente a uma autoavaliação

individual, avalia o seu desempenho e o do colega, David Paiva, com dezanove valores para ambos.

Índice de Figuras

Índice de Tabelas

Índice

Esta página foi deixada propositadamente em branco.

1. Enquadramento

Nesta secção são apresentados o objetivo geral do primeiro projeto da unidade curricular de Sistemas e Gestão de Dados e o processo experimental utilizado para realizar o estudo de benchmarking solicitado.

1.1 Objetivo Geral do Projeto

Este primeiro projeto tem como principal objetivo a aquisição de competências no domínio de estudos de benchmarking em base de dados. Com isso em mente, foi analisado a performance do motor de base de dados Postgres.

1.2 Processo Experimental

Todos os passos desde projeto, excepto a geração dos dados no SSB, foram realizados em duas máquinas distintas e com configurações de hardware ligeiramente diferentes. Com isso em mente, o trabalho apresentado neste documento visa não só uma análise do comportamento do motor de base de dados Postgres, mas tem também como objetivo comparar o seu comportamento em máquinas diferentes.

O primeiro passou consistiu em gerar aproximadamente 10GB de dados (para um scale-factor de 15 o SSB não gerou 15GB). Tal como referido anteriormente, este foi o único passo que foi realizado apenas numa máquina, uma vez que o que pretendíamos avaliar era a performance do Postgres e não do SSB.

Todos os passos que se seguiram, foram realizados em duas máquinas distintas. De modo a ser possível assumir uma distribuição normal da amostra dos dados recolhidos, foram realizadas 35 repetições de cada operação de inserção (load), pesquisa (search) e criação de chaves primárias (PK) e estrangeiras (FK).

Depois de realizadas todas as repetições das operações, os dados recolhidos foram processados, analisados e apresentados no presente documento.

Com base no objetivo geral do projeto e dos requisitos descritos no enunciado a performance será medida e avaliada com base no tempo de execução das pesquisas. Neste caso, os resultados com um tempo de execução mais baixos significam que são mais rápidos e por sua vez mais eficientes. Em adição, no processo de carregamento dos dados, serão também avaliados o número de linhas e os megabytes carregados por segundos.

Esta página foi deixada propositadamente em branco.

2. Setup Experimental

Nesta secção são apresentados os setups experimentais utilizados para a realização do trabalho.

2.1 Setup Experimental David

A Tabela 2 apresenta a configuração do setup experimental da máquina do David.

Processador	Intel® Core™ i7 7700HQ
Nº de Núcleos	4 Núcleos 4 Threads
RAM	16.0 GB
Disco	Samsung SSD 970 EVO Plus 1TB
Sistema Operativo	Windows 10 Home 64bits
Postgres	Versão 10

Tabela 2 - Configuração do Setup Experimental do David

2.2 Setup Experimental Ricardo

A Tabela 3 apresenta a configuração do setup experimental da máquina do Ricardo.

Processador	Intel® Core™ i7-9750H
Nº de Núcleos	6 Núcleos 6 Threads
RAM	16.0 GB
Disco	Western Digital SSD SN550 500GB
Sistema Operativo	Windows 10 Home 64bits
Postgres	Versão 10

Tabela 3 - Configuração do Setup Experimental do Ricardo

Ambas as configurações são muitos semelhantes, no entanto a máquina do Ricardo possuí um processador com mais núcleos- 6 núcleos. Deste modo, é possível comparar se o número de núcleos tem alguma influência na performance do motor de base de dados Postgres.

Esta página foi deixada propositadamente em branco.

3. Análise de Resultados

Nesta secção são apresentados os resultados obtidos após as várias repetições de execuções efetuadas, assim como a sua análise.

3.1 Carregamento dos Dados para o Postgres

A primeira fase consiste no carregamento dos dados gerados pelo SSB para o Postgres. O gráfico apresentado na Figura 1 mostra que ambos os setups, PC Ricardo e PC David, apresentam uma performance de carregamento muito semelhante, no entanto o PC Ricardo é ligeiramente mais rápido a carregar os dados para o Postgres.

Os dados relativos às tabelas com um número relativamente pequeno de linhas – tabelas customer, supplier e date - são carregados para o Postgres muitíssimo rápido uma vez que o tempo de carregamento é inferior a um segundo. A tabela part embora demore pouco mais de dois segundos, apresenta um tempo de carregamento bastante acessível. Em contraste, o cenário altera-se um pouco no carregamento dos dados da tabela lineorder, uma vez que este tempo ultrapassa os quatro minutos em ambas as máquinas. Este fenómeno é completamente normal, uma vez que se trata de 8.6GB de dados.

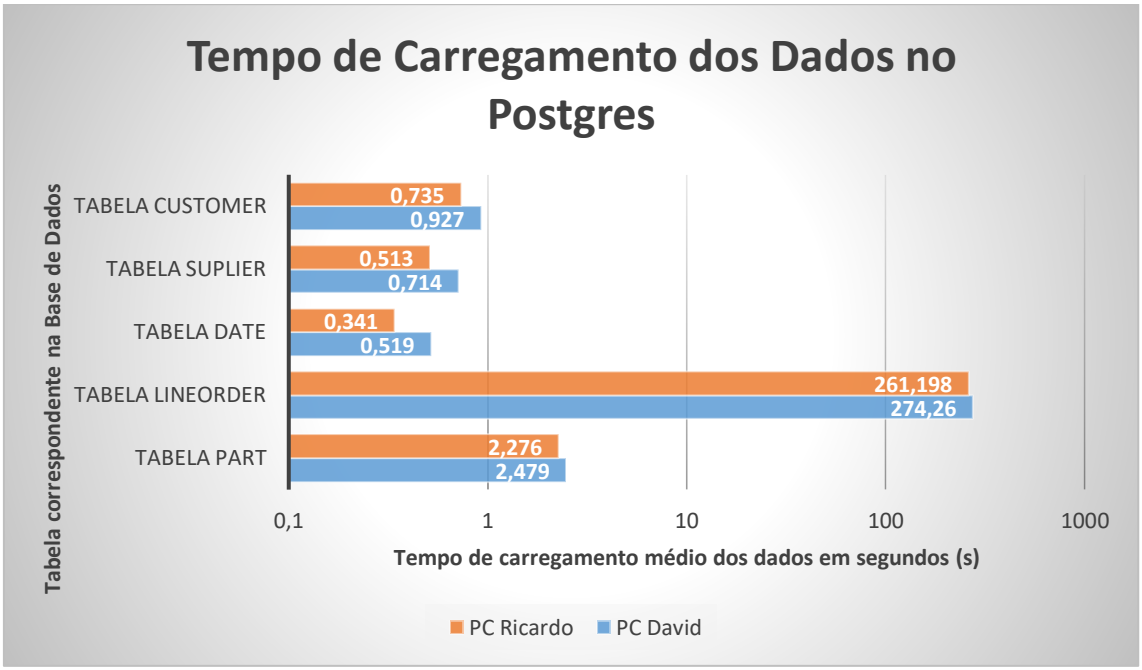


Figura 1 - Tempo de Carregamento dos Dados no Postgres

O tempo de carregamento dos dados para a base de dados não é a única variável dependente interessante de se analisar. Com isso em mente, o gráfico apresentado na

Figura 2 mostra a quantidade de informação carregada para a base de dados em megabytes num segundo. A tabela date e supplier apresentam um valor bastante inferior aos restantes uma vez que as tabelas são extremamente pequenas, tornando o cálculo pouco preciso e inconsistente. No entanto, observando os restantes valores é possível concluir que o valor tende a estabilizar, no intervalo dos 25 aos 31 megabytes por segundos, à medida que a quantidade de dados aumenta.

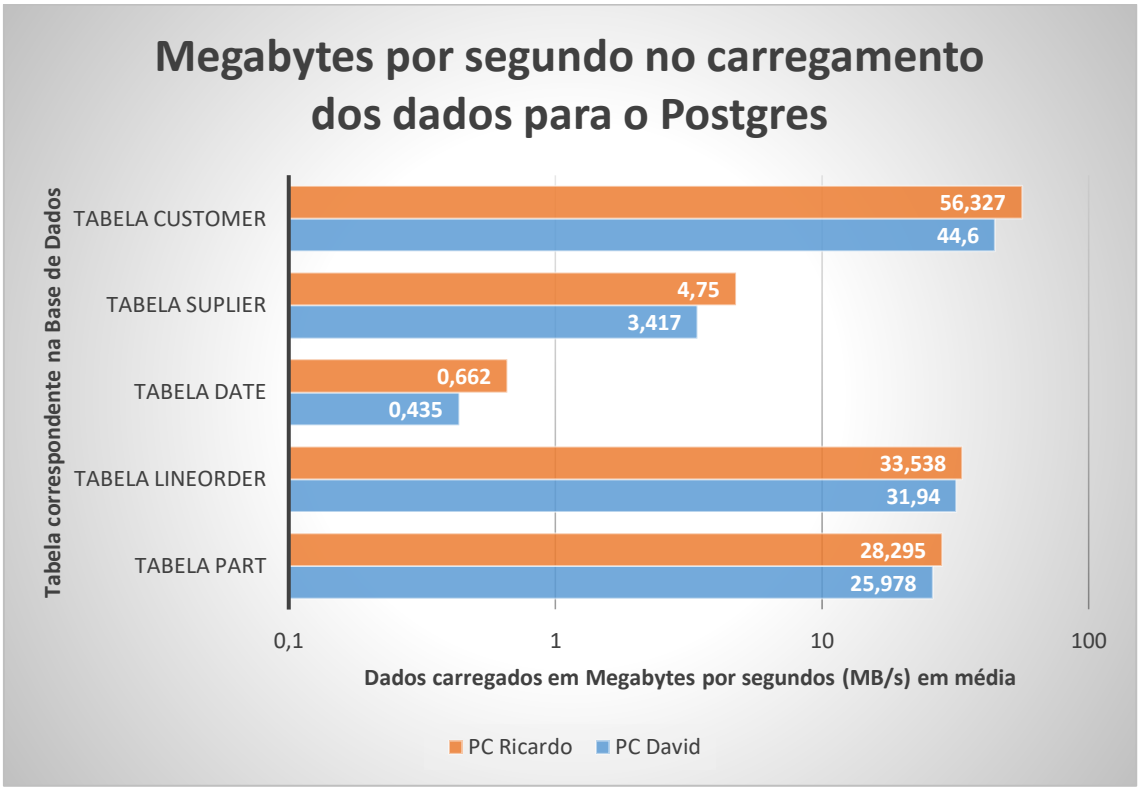


Figura 2 - Megabytes por segundo no carregamento dos dados para o Postgres

Por fim e não menos importante, o gráfico apresentado na Figura 3 mostra outra métrica interessante no momento do carregamento de dados para a base de dados, o número de linhas carregadas por segundos. À semelhança do que se passava na métrica anterior, quando observamos os valores para um carregamento muito pequeno de dados, o valor pode não ser muito preciso e apresentar alguma inconsistência. No entanto, esse valor estabiliza e é bastante coerente para quantidades de ficheiros um pouco mais elevas ou até mesmo na ordem dos 8GB – como é o caso da tabela lineorder. Nestes casos é possível concluir que ambas as máquinas carregam por segundo um número de linhas que se encaixa no intervalo de 320000 a 350000 linhas por segundo.

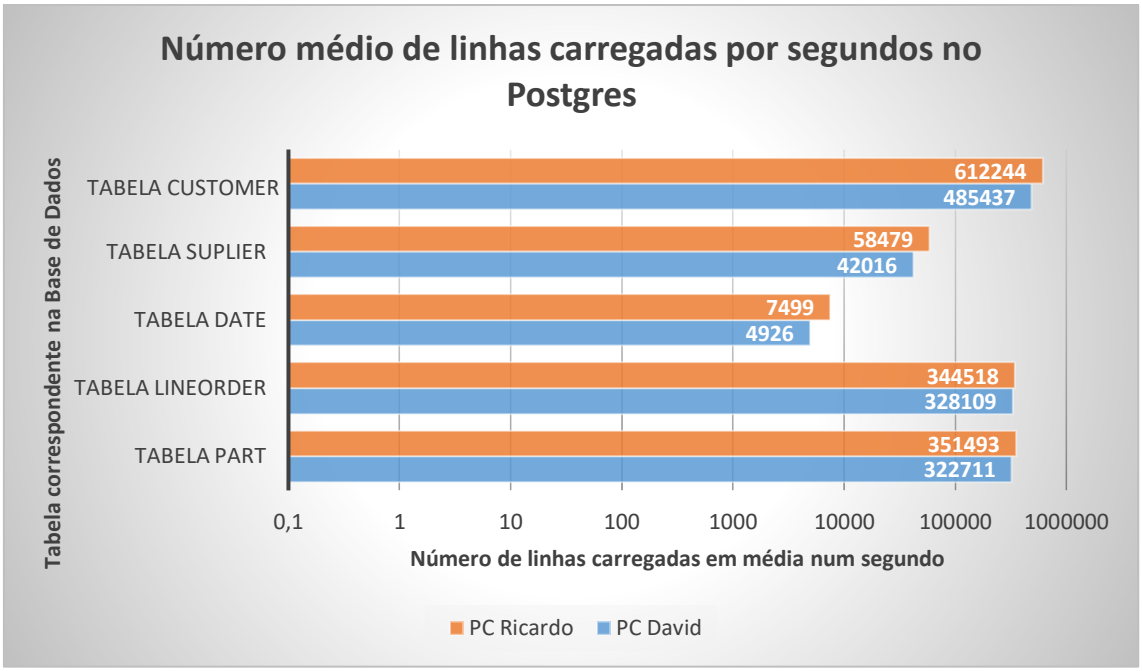


Figura 3 - Número médio de linhas carregadas por segundos no Postgres

Após a análise feita anteriormente, relativamente ao carregamento dos dados par ao Postgres, é possível concluir que as máquinas tiveram um comportamento muito semelhante entre si. Segue-se a análise da performance do Postgres no que toca a pesquisas.

3.2 Registo de performance das pesquisas

Nesta secção são apresentados os resultados das execuções das queries disponibilizadas no enunciado do SSB. Trata-se de pesquisas a um conjunto de dados simples sem índices, isto é, sem nenhuma referência associada a PK e/ou FK. Depois do registo das execuções sem índices, executaram-se novamente as mesmas queries mas com uma estrutura de dados mais complexa, isto é, com índices que referenciam PK e/ou FK.

O objetivo destes registos é fazer uma comparação de performance de execução de queries com e sem índices na sua estrutura de dados.

3.2.1 Pesquisa de dados sem chaves

Como mencionado, a execução de cada query foi repetida 35 vezes permitindo obter resultados que seguem uma distribuição normal. Os resultados foram registados e computou-se a média para cada uma das diferentes pesquisas.

Como se pode observar no gráfico apresentado na Figura 4, na maioria das pesquisas o setup PC Ricardo apresenta um tempo médio de execução mais baixo que o setup PC David. Uma vez que há uma ligeira diferença entre setups é notável que existe uma ligeira diferença da performance. Contudo houve casos, como a execução de Q2.1 e Q2.2, em que tal não se verificou, havendo uma diferença bastante significativa de performance. Esta grande diferença deve-se a uma possível sobrecarga a que o processador esteve sujeito no momento de execução dessas 2 queries em específico e que eram impossíveis de controlar – como por exemplo outros processos do Windows a correr em simultâneo ou outros fatores desconhecidos.

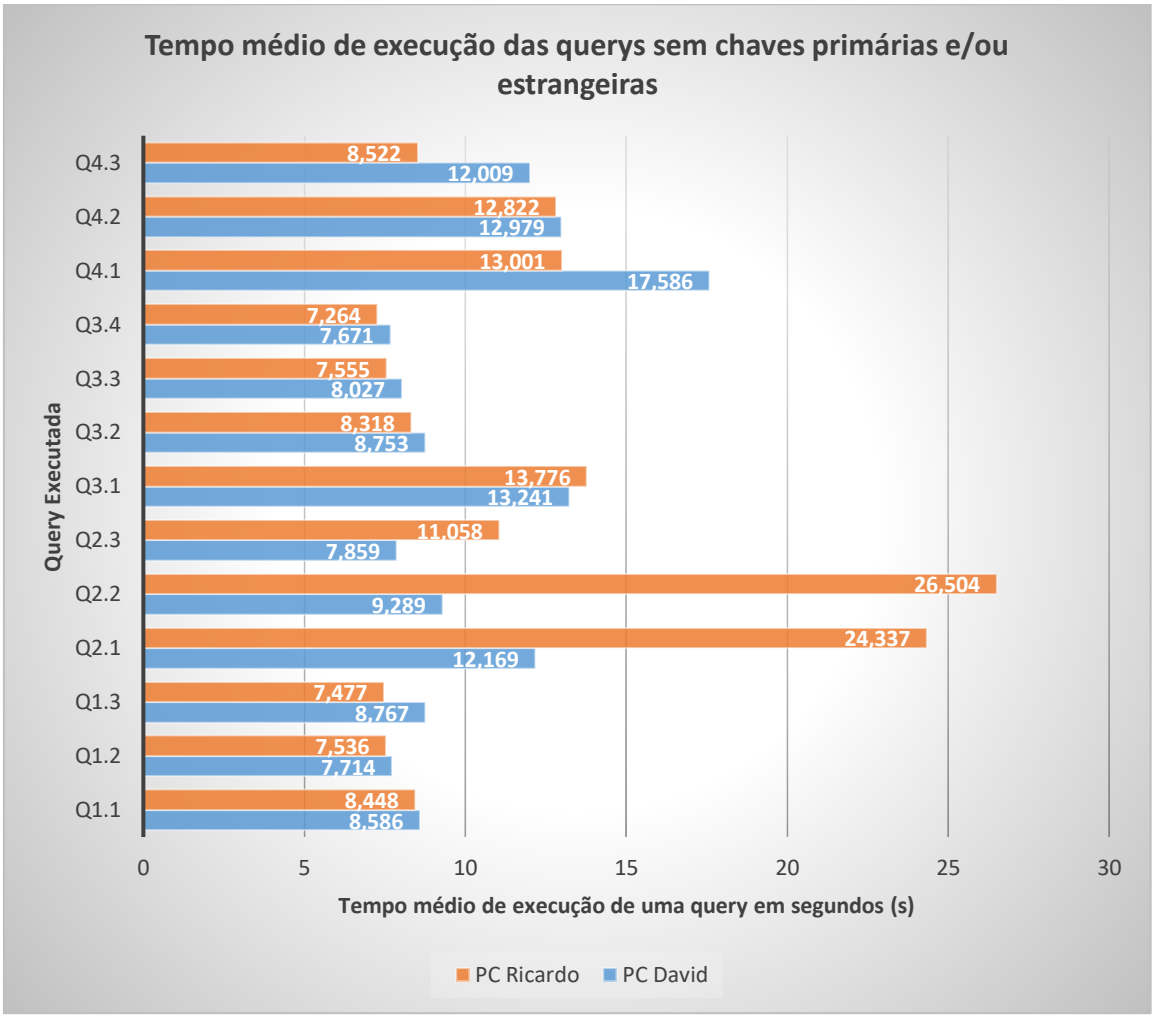


Figura 4 – Tempo médio de execução das querys sem chaves primárias e/ou estrangeiras

3.2.2 Criação das chaves

A criação das chaves não é um processo que possa ser repetido de forma fluída como foi no registo de performance das pesquisas sem índices. Para tal, foi necessário

realizar DROP após a criação dos índices e voltar a repetir o processo e assim, conseguir fazer os 35 registos de criação de chaves PK e/ou FK.

Como se pode observar no gráfico apresentado na Figura 5, uma vez mais o setup PC Ricardo foi ligeiramente mais eficiente que o setup PC David. Para além disso, é de notar que existe uma diferença de tempo na criação das PK's quando comparado com a criação de FK's. Esta diferença pode estar relacionada com o facto de no momento de criação de FK ser necessário aceder a informação de outras tabelas, aumentando um pouco a complexidade destas operações e consequentemente o tempo que demoram a executar.

A criação do PK lineorder apresenta uma média de tempo de execução bastante elevada uma vez que possuí uma quantidade de informação muito superior às outras tabelas (89987410 linhas registadas), sendo assim compreensível que os registos de tempo de execução da criação de chaves sejam maiores.

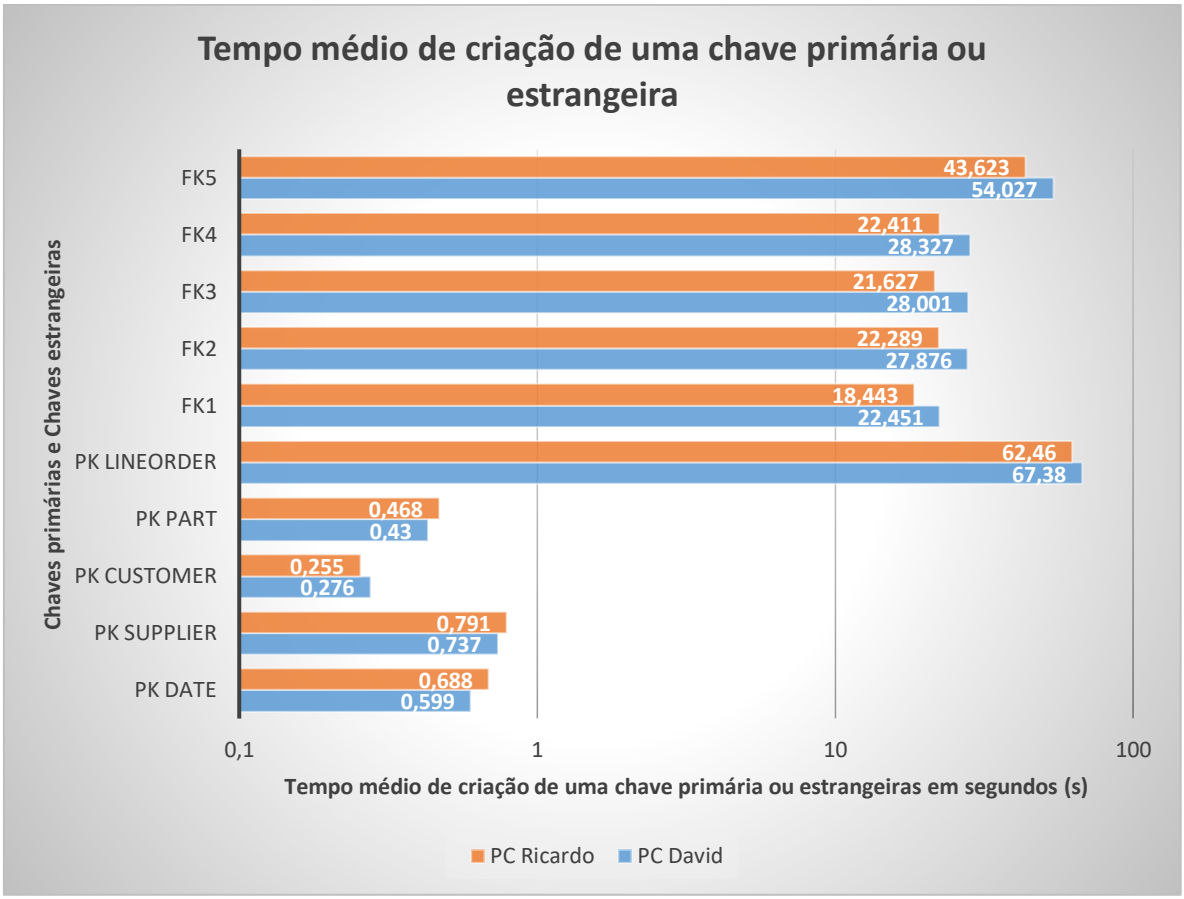


Figura 5 - Tempo médio de criação de uma chave primária ou estrangeira

3.2.3 Pesquisa de dados com chaves

Uma vez feita as criações de índices nas tabelas estavam reunidas as condições para fazer o último registo do tempo médio de execução das queries, mas desta vez com uma estrutura de dados na qual as chaves já estão inseridas nas tabelas. Era esperado que os tempos médios de execução fossem mais curtos, apresentando melhor performance, visto que a utilização de índices é feita para otimizar a localização de linhas da tabela de interesse quando é feita uma pesquisa.

Se for feita uma comparação entre o gráfico da Figura 4, acima apresentado, e o gráfico apresentado abaixo na Figura 6, a performance é claramente melhor quando não é usado o índice. Estamos a falar de uma rapidez cerca de 235% superior que a rapidez com o uso de índices. Estes resultados não foram de encontro ao espero provocando que fossem levantas algumas questões sobre os processos internos do motor da base de dados. Este problema será novamente abordado

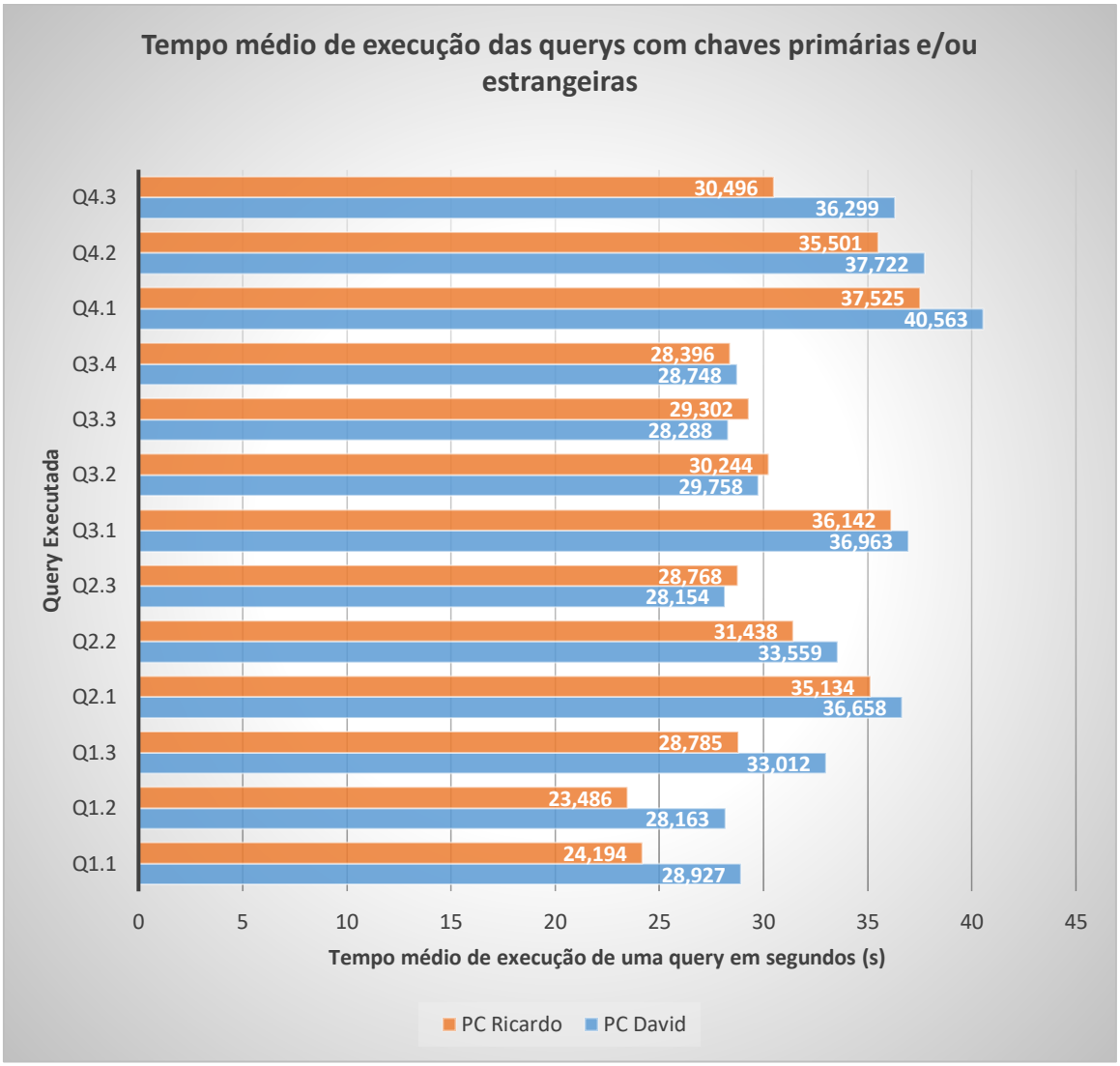


Figura 6 - Tempo médio de execução das querys com chaves primárias e/ou estrangeiras

Esta página foi deixada propositadamente em branco.

4. Planos de execução das Querys

Como mencionado anteriormente na secção 3.2.3, na qual se identificaram irregularidades nos registos de tempo obtidos pela execução das queries, foram levantadas algumas questões de como o motor fazia a gestão da execução da query e o porquê de estas irregularidades.

Desta forma, foi feita uma análise detalhada do Query Plan sem índices e com índices. O objetivo com está análise será identificar as maiores diferenças entre os planos de execução que o motor da base de dados realizou e conhecer as razões que levaram o mesmo a escolher uma opção ou outra.

4.1 Q1.2

O plano apresentado na Figura 7 é referente às pesquisas antes da criação das chaves e, por sua vez o plano apresentado na Figura 8 é referente às pesquisas depois da criação das chaves. Os planos são os mesmos, a única variante que alterada é o tempo de execução e planeamento que é bastante superior após a criação das chaves e, portanto, será apenas explicado um dos planos (visto que são iguais). Este fenómeno é extremamente difícil de explicar e entender, no entanto mais à frente neste documento serão abordadas algumas hipóteses que visam explicar este fenómeno.

Focando no plano de execução, a primeira operação realizada foi uma filtragem no acesso sequencial na tabela date – neste caso o "filter" verifica a condição WHERE (*where d_yearmonthnum = 199401*) para cada linha da tabela date, devolvendo apenas as que garantem condição. Os resultados devolvidos pelo acesso sequencial anterior são armazenados numa Hash Table em memória à medida que são lidos – "Hash". De seguida é feito um acesso sequencial paralelo à tabela lineorder – "Parallel Seq Scan on lineorder" - e são apenas devolvidas as linhas que respeitam as restantes condições WHERE – "filter". Simultaneamente, o Postgres compara estes valores com os valores da tabela date que estão armazenados em memória numa Hash Table – "Hash join" significa isso mesmo, as linhas da tabela lineorder vão para uma Hash Table em memória, criada neste momento, após terem sido comparadas com as linhas da tabela date. A condição de match entre a tabela lineorder e a tabela date pode ser vista na linha "Hash Cond". Por fim os valores são agregados no resultado pretendido – que neste caso é uma soma.

São apenas realizados acessos sequencias às tabelas e mesmo depois da criação das chaves, o Postgres decide continuar a realizar acessos sequenciais às tabelas.

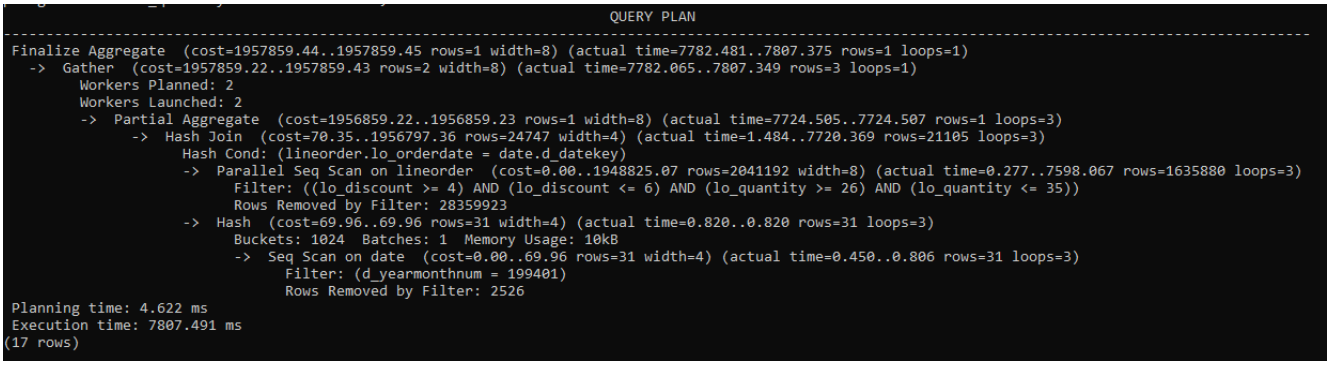


Figura 7 - Plano de Execução da Query Q1.2 antes da criação das chaves

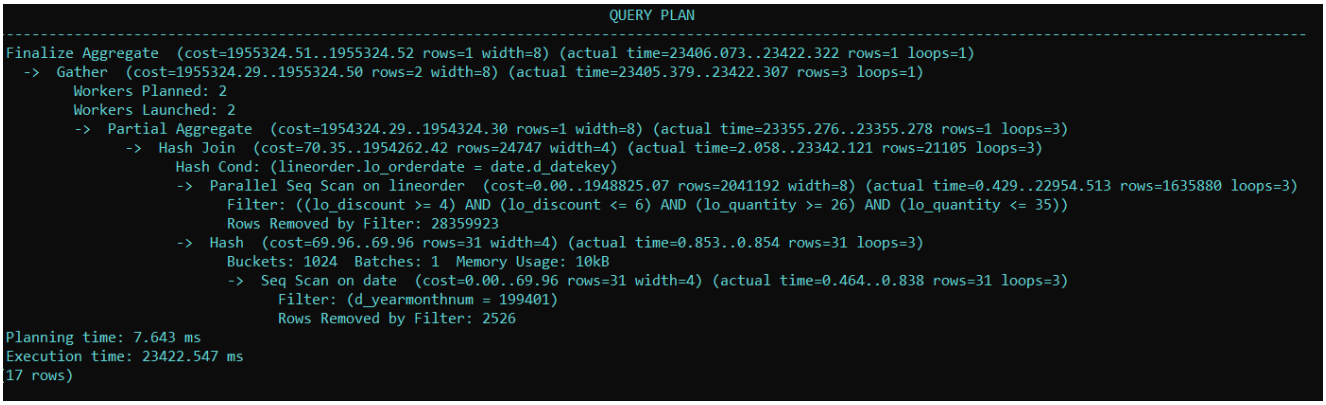


Figura 8 - Plano de Execução da Query Q1.2 depois da criação das chaves

4.2 Q2.2

Mais uma vez, não foi possível

O primeiro passo realizado foi um acesso sequencial na tabela. O Postgres faz uma estimativa de custos para esta primeira instrução na qual se pode observar em: (cost=0.00..63.57.04 rows=2557 width=8). Isto significa que para encontrar os valores da tabela, vai demorar cerca 63.57 unidades arbitrarias (UA) – unidade o qual se mede o custo computacional – e vai devolver cerca de 2557 linhas. De seguida, o resultado deste acesso é armazenado numa Hash Table em memória e que dará uso cerca de 132Kb como especificado no nó Hash.

De seguida é feito um segundo acesso sequencial, desta vez na tabela *supplier* para aplicar um filtro, ou mais conhecido como um *WHERE Clause* (s_*region* tem de ser igual a 'ASIA', isto é, uma condição que tem de ser realizada e que vai ter um custo

previsto de 798 UA e que retornou 6040 linhas das 30000 disponíveis (*Rows Removed by Filter : 23960*). Tal como aconteceu no acesso sequencial anterior mencionado, o resultado é armazenado novamente na Hash Table e indica que foi usado cerca de 277Kb de memoria.

O terceiro acesso sequencial é semelhante ao anterior, na qual é aplicado outro Filter (WHERE Clause) na tabela *part* no qual o atributo p_bran1 tem de estar entre a gama '*MFGR#2221*' e '*MFGR#2228*'. Foram devolvidas 6414 linhas das 800000 disponíveis sendo este resultado armazenado na Hash Table usando cerca de 365Kb de memoria.

Como quarto passo temos um acesso sequencial paralelo na tabela *lineorder* pois é são feitos Join's consecutivos, um Join entre as tabelas *lineorder* e *part*. Este Join é feito respeitando a condição *lineorder.lo_partkey = part.p_partkey*. Ainda, é feito um outro Join entre a tabela *lineorder* e *supplier* e ainda outro Join entre a tabela *lineorder* e date, respeitando as condições *linerorder.lo_suppkey = supplier.s_suppkey* e *lineorder.lo_orderdate = date.d_datekey*, respetivamente. O acesso paralelo é feito, pois ao mesmo tempo que esta tabela é acedida, a Hash Table que contem todos os resultados dos acessos anteriores servirá de intermédiario para a realização dos Joins.

O Postgres, como quinto e sexto passo irá ordenar o resultado por ano (*date.d_year*) e por marca (*part.p_brand1*) e agrupa-los usando os mesmos atributos (ano e marca) com os nós *Sort* e *Partial GroupAggreate*, respetivamente.

É de notar que o motor novamente não usou como recurso os índices das tabelas, optando sempre por acessos sequenciais.

```
                                                       QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Finalize GroupAggregate  (cost=1740033.22..1740039.09 rows=21 width=21) (actual time=9064.109..9079.558 rows=56 loops=1)
   Group Key: date.d_year, part.p_brand1
   ->  Gather Merge  (cost=1740033.22..1740038.56 rows=42 width=21) (actual time=9063.801..9079.523 rows=168 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial GroupAggregate  (cost=1739033.20..1739033.69 rows=21 width=21) (actual time=9035.826..9046.049 rows=56 loops=3)
               Group Key: date.d_year, part.p_brand1
               ->  Sort  (cost=1739033.20..1739033.27 rows=28 width=17) (actual time=9035.575..9040.006 rows=48360 loops=3)
                     Sort Key: date.d_year, part.p_brand1
                     Sort Method: external merge  Disk: 1432kB
                     ->  Hash Left Join  (cost=24547.07..1739032.53 rows=28 width=17) (actual time=429.975..8902.026 rows=48360 loops=3)
                           Hash Cond: (lineorder.lo_orderdate = date.d_datekey)
                           ->  Hash Join  (cost=24451.54..1738936.61 rows=28 width=17) (actual time=428.869..8873.270 rows=48360 loops=3)
                                 Hash Cond: (lineorder.lo_suppkey = supplier.s_suppkey)
                                 ->  Hash Join  (cost=23578.04..1738062.30 rows=140 width=21) (actual time=420.585..8785.873 rows=240385 loops=3)
                                       Hash Cond: (lineorder.lo_partkey = part.p_partkey)
                                       ->  Parallel Seq Scan on lineorder  (cost=0.00..1573877.53 rows=37494753 width=16) (actual time=0.333..5289.197 rows=29995803 loops=3)
                                       ->  Hash  (cost=23578.00..23578.00 rows=3 width=13) (actual time=420.178..420.179 rows=6414 loops=3)
                                             Buckets: 8192 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 365kB
                                             ->  Seq Scan on part  (cost=0.00..23578.00 rows=3 width=13) (actual time=0.464..418.195 rows=6414 loops=3)
                                                   Filter: (((p_brand1)::text >= 'MFGR#2221'::text) AND ((p_brand1)::text <= 'MFGR#2228'::text))
                                                   Rows Removed by Filter: 793586
                                 ->  Hash  (cost=798.00..798.00 rows=6040 width=4) (actual time=8.121..8.121 rows=6040 loops=3)
                                       Buckets: 8192  Batches: 1  Memory Usage: 277kB
                                       ->  Seq Scan on supplier  (cost=0.00..798.00 rows=6040 width=4) (actual time=0.265..7.042 rows=6040 loops=3)
                                             Filter: ((s_region)::text = 'ASIA'::text)
                                             Rows Removed by Filter: 23960
                           ->  Hash  (cost=63.57..63.57 rows=2557 width=8) (actual time=1.078..1.078 rows=2557 loops=3)
                                 Buckets: 4096  Batches: 1  Memory Usage: 132kB
                                 ->  Seq Scan on date  (cost=0.00..63.57 rows=2557 width=8) (actual time=0.420..0.789 rows=2557 loops=3)
 Planning time: 1.384 ms
 Execution time: 9081.232 ms
(32 rows)
```

Figura 9 - Plano de Execução da Query Q2.2 antes da criação das chaves

```
                                                       QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
Finalize GroupAggregate  (cost=1697882.38..1697990.11 rows=385 width=21) (actual time=26950.592..27000.201 rows=56 loops=1)
  Group Key: date.d_year, part.p_brand1
  ->  Gather Merge  (cost=1697882.38..1697980.49 rows=770 width=21) (actual time=26949.148..27000.094 rows=168 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial GroupAggregate  (cost=1696882.36..1696891.59 rows=385 width=21) (actual time=26842.796..26870.087 rows=56 loops=3)
              Group Key: date.d_year, part.p_brand1
              ->  Sort  (cost=1696882.36..1696883.70 rows=538 width=17) (actual time=26841.712..26853.894 rows=48360 loops=3)
                    Sort Key: date.d_year, part.p_brand1
                    Sort Method: external merge  Disk: 1432kB
                    ->  Hash Left Join  (cost=24547.75..1696857.95 rows=538 width=17) (actual time=1274.443..26377.438 rows=48360 loops=3)
                          Hash Cond: (lineorder.lo_orderdate = date.d_datekey)
                          ->  Hash Join  (cost=24452.21..1696761.01 rows=538 width=17) (actual time=1271.814..26316.123 rows=48360 loops=3)
                                Hash Cond: (lineorder.lo_suppkey = supplier.s_suppkey)
                                ->  Hash Join  (cost=23578.71..1695880.50 rows=2672 width=21) (actual time=1261.661..26151.621 rows=240385 loops=3)
                                      Hash Cond: (lineorder.lo_partkey = part.p_partkey)
                                      ->  Parallel Seq Scan on lineorder  (cost=0.00..1573877.53 rows=37494753 width=16) (actual time=0.353..14998.716 rows=29995803 loops=3)
                                      ->  Hash  (cost=23578.00..23578.00 rows=57 width=13) (actual time=1261.238..1261.239 rows=6414 loops=3)
                                            Buckets: 8192 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 365kB
                                            ->  Seq Scan on part  (cost=0.00..23578.00 rows=57 width=13) (actual time=0.411..1256.414 rows=6414 loops=3)
                                                  Filter: (((p_brand1)::text >= 'MFGR#2221'::text) AND ((p_brand1)::text <= 'MFGR#2228'::text))
                                                  Rows Removed by Filter: 793586
                                ->  Hash  (cost=798.00..798.00 rows=6040 width=4) (actual time=9.854..9.854 rows=6040 loops=3)
                                      Buckets: 8192  Batches: 1  Memory Usage: 277kB
                                      ->  Seq Scan on supplier  (cost=0.00..798.00 rows=6040 width=4) (actual time=0.347..8.612 rows=6040 loops=3)
                                            Filter: ((s_region)::text = 'ASIA'::text)
                                            Rows Removed by Filter: 23960
                          ->  Hash  (cost=63.57..63.57 rows=2557 width=8) (actual time=2.572..2.572 rows=2557 loops=3)
                                Buckets: 4096  Batches: 1  Memory Usage: 132kB
                                ->  Seq Scan on date  (cost=0.00..63.57 rows=2557 width=8) (actual time=0.665..1.784 rows=2557 loops=3)
Planning time: 11.036 ms
Execution time: 27002.893 ms
(32 rows)
```

Figura 10 - Plano de Execução da Query Q2.2 depois da criação das chaves

4.3 Q3.2

A Figura 11 apresenta o plano de execução da query Q3.2 antes da criação das chaves e a Figura 12 apresenta o plano de execução da mesma query depois da criação das chaves. Uma vez mais, o plano obtido é o mesmo, diferenciando apenas no tempo de planeamento e execução.

O primeiro passo passa por uma filtragem no acesso sequencial na tabela date – neste caso o "filter" verifica a condição WHERE (*d_year>= 1992 and d_year <= 1997*) para cada linha da tabela date devolvendo apenas as que garantem a condição, armazenando-as numa Hash Table em memória – "Hash" – que ocupa 118kB (2192 linhas) em memória. De seguida, é feita uma filtragem durante um acesso sequencial à

tabela customer – “filter” e “Seq Scan” respetivamente – de movo a garantir a cláusula WHERE (*c_nation = 'UNITED STATES'*), sendo que as linhas que a cumprem são armazenadas numa Hash Table em memória – “Hash” – que ocupa 1100kB (17987 linhas). Terceiro passo é novamente uma filtragem durante um acesso sequencial à tabela supplier de movo a garantir o cumprimento da condição WHERE (*s_nation = 'UNITED STATES'*) e as linhas devolvidas são também armazenadas numa nova Hash Table que ocupa 73kB (1201 linhas).

De seguida, são realizados todos os joins, recorrendo às Hash Tables anteriores – que armazenam os dados respetivos de cada tabela respeitando as condições WHERE. Desse mosso, é feito um acesso sequencial à tabela lineorder, onde cada linha da tabela é confrontada com os valores armazenados nas Hash Tables, pela seguinte ordem:

1. Join da tabela lineorder com a tabela suplier.
2. Join da tabela lineorder com a tabela costumer.
3. Join da tabela lineorder com a tabela date.

Agora que todas as linhas cumprem com as condições indicadas na query é realizado a sua ordenação necessitando de 1808kB de memória em disco. Esta ordenação é feita para as seguintes condições (*costumer.c_city, supplier.s_city, date.d_year*).

Por fim são realizados os Group Buy e os resultados finais são ordenados de forma decrescente recorrendo ao algoritmo de ordenamento quicksort – um dos mais conhecidos e eficazes. Esta ordenação é feita em memória o que é mais eficar do que se fosse feita em disco.

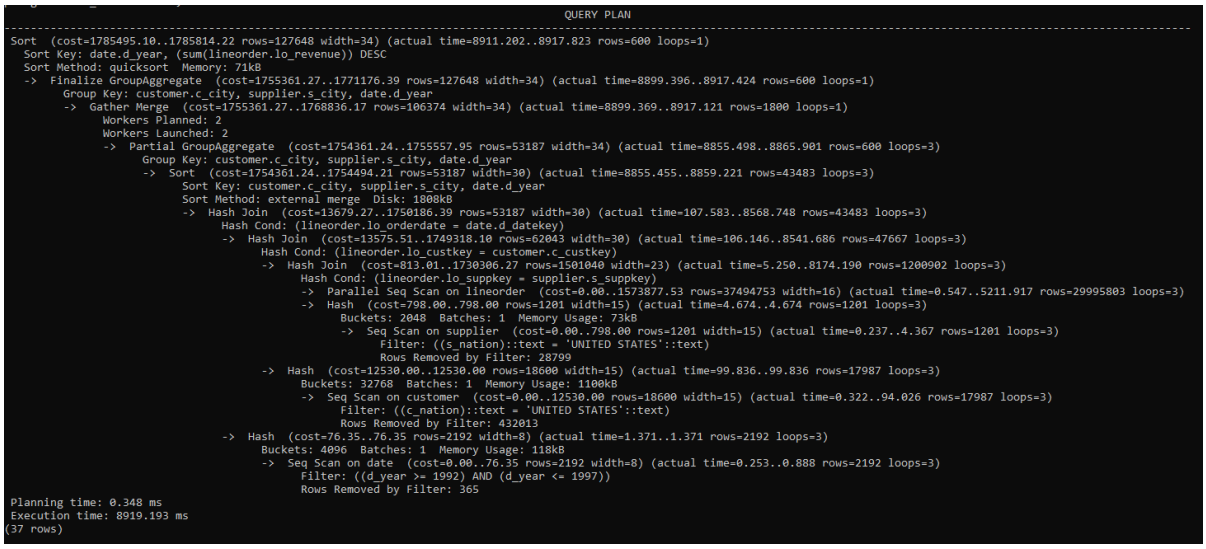
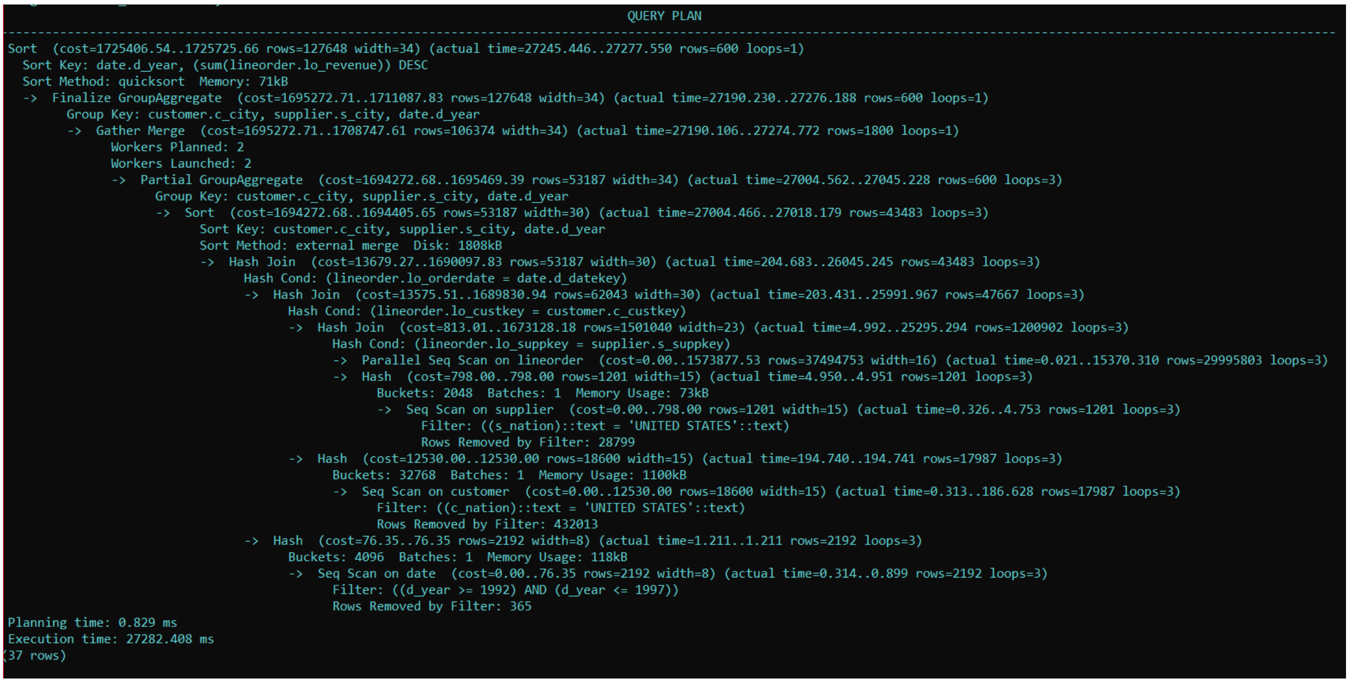


Figura 11 - Plano de Execução da Query Q3.2 antes da criação das chaves



```
                                   QUERY PLAN
--------------------------------------------------------------------------------------------
Sort  (cost=1725406.54..1725725.66 rows=127648 width=34) (actual time=27245.446..27277.550 rows=600 loops=1)
  Sort Key: date.d_year, (sum(lineorder.lo_revenue)) DESC
  Sort Method: quicksort  Memory: 71kB
  ->  Finalize GroupAggregate  (cost=1695272.71..1711087.83 rows=127648 width=34) (actual time=27190.230..27276.188 rows=600 loops=1)
        Group Key: customer.c_city, supplier.s_city, date.d_year
        ->  Gather Merge  (cost=1695272.71..1708747.61 rows=106374 width=34) (actual time=27190.106..27274.772 rows=1800 loops=1)
              Workers Planned: 2
              Workers Launched: 2
              ->  Partial GroupAggregate  (cost=1694272.68..1695469.39 rows=53187 width=34) (actual time=27004.562..27045.228 rows=600 loops=3)
                    Group Key: customer.c_city, supplier.s_city, date.d_year
                    ->  Sort  (cost=1694272.68..1694405.65 rows=53187 width=30) (actual time=27004.466..27018.179 rows=43483 loops=3)
                          Sort Key: customer.c_city, supplier.s_city, date.d_year
                          Sort Method: external merge  Disk: 1808kB
                          ->  Hash Join  (cost=13679.27..1690097.83 rows=53187 width=30) (actual time=204.683..26045.245 rows=43483 loops=3)
                                Hash Cond: (lineorder.lo_orderdate = date.d_datekey)
                                ->  Hash Join  (cost=13575.51..1689830.94 rows=62043 width=30) (actual time=203.431..25991.967 rows=47667 loops=3)
                                      Hash Cond: (lineorder.lo_custkey = customer.c_custkey)
                                      ->  Hash Join  (cost=813.01..1673128.18 rows=1501040 width=23) (actual time=4.992..25295.294 rows=1200902 loops=3)
                                            Hash Cond: (lineorder.lo_suppkey = supplier.s_suppkey)
                                            ->  Parallel Seq Scan on lineorder  (cost=0.00..1573877.53 rows=37494753 width=16) (actual time=0.021..15370.310 rows=29995803 loops=3)
                                            ->  Hash  (cost=798.00..798.00 rows=1201 width=15) (actual time=4.950..4.951 rows=1201 loops=3)
                                                  Buckets: 2048  Batches: 1  Memory Usage: 73kB
                                                  ->  Seq Scan on supplier  (cost=0.00..798.00 rows=1201 width=15) (actual time=0.326..4.753 rows=1201 loops=3)
                                                        Filter: ((s_nation)::text = 'UNITED STATES'::text)
                                                        Rows Removed by Filter: 28799
                                      ->  Hash  (cost=12530.00..12530.00 rows=18600 width=15) (actual time=194.740..194.741 rows=17987 loops=3)
                                            Buckets: 32768  Batches: 1  Memory Usage: 1100kB
                                            ->  Seq Scan on customer  (cost=0.00..12530.00 rows=18600 width=15) (actual time=0.313..186.628 rows=17987 loops=3)
                                                  Filter: ((c_nation)::text = 'UNITED STATES'::text)
                                                  Rows Removed by Filter: 432013
                                ->  Hash  (cost=76.35..76.35 rows=2192 width=8) (actual time=1.211..1.211 rows=2192 loops=3)
                                      Buckets: 4096  Batches: 1  Memory Usage: 118kB
                                      ->  Seq Scan on date  (cost=0.00..76.35 rows=2192 width=8) (actual time=0.314..0.899 rows=2192 loops=3)
                                            Filter: ((d_year >= 1992) AND (d_year <= 1997))
                                            Rows Removed by Filter: 365
Planning time: 0.829 ms
Execution time: 27282.408 ms
(37 rows)
```

Figura 12 - Plano de Execução da Query Q3.2 depois da criação das chaves

4.4 Q4.2

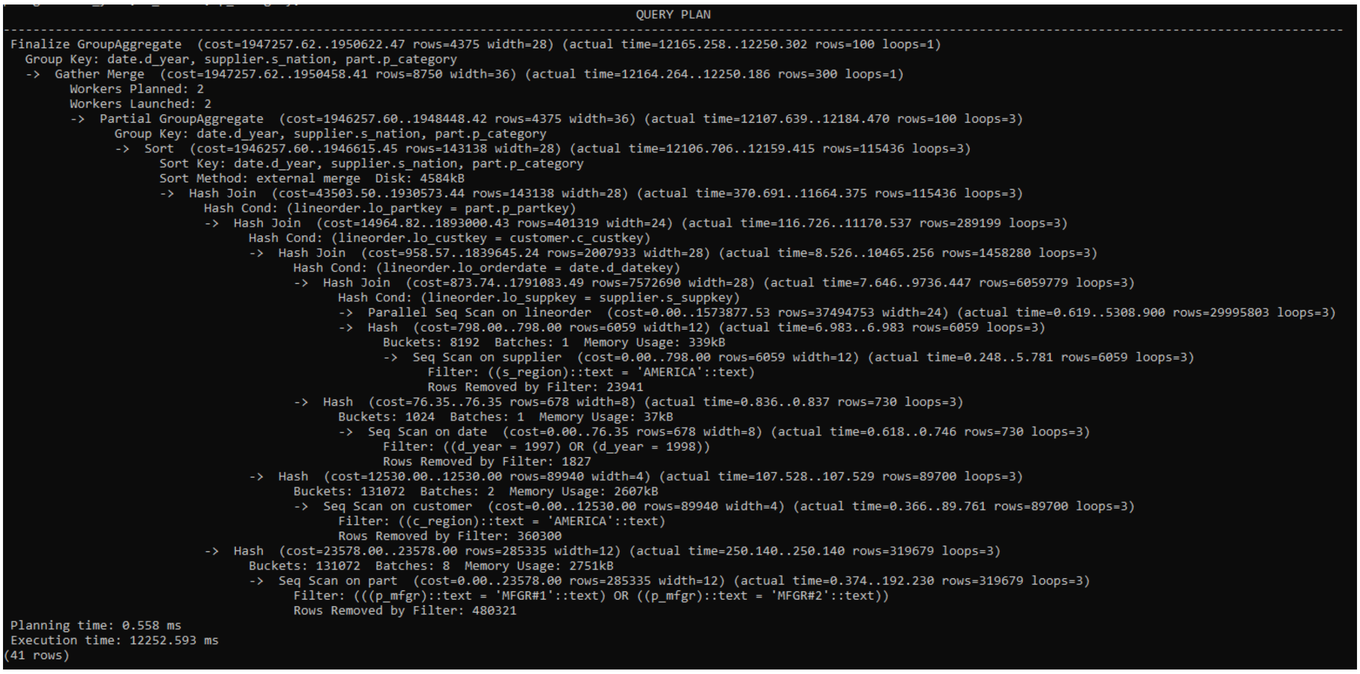
O plano de execução da query Q4.2 começa com um acesso em sequência na tabela part na qual se aplica um Filter em que o atributo p_mfgr tem de ser ‘MFGR#1’ ou ‘MFGR#2’.

O resultado vai para uma Hash Table na qual é usada cerca de 2751Kb. De seguida é feito um novo acesso sequencial, desta vez na tabela customers e da mesma maneira é aplicado um Filter em que o atributo c_region tem de ser igual a ‘AMERICA’ – o resultado é armazenado na Hash Table. Repetindo a mesma lógica, é feito outro acesso sequencial a tabela date e aplicado um novo Filter sobre o atributo d_year o qual tem de ser igual a ‘1997’ ou ‘1998’. Este resultado é também armazenado numa Hash Table. Um último acesso sequencial é realizado na tabela supplier aplicando-se um Filter sobre o atributo s_region o qual tem de ser igual a ‘AMERICA’.

Tendo-se aplicado todos os filtros (WHERE Clauses) o motor da base de dados aplica uma série de Join’s. Para tal é realizado um acesso sequencial paralelo na tabela lineorder e junto com a Hash Table irá realizar-se o join entre as tabelas lineorder e date (lineorder.lo_orderdate = date.d_datekey), entre as tabelas lineorder e customers (lineorder.lo_custkey = customers.c_custkey), entre as tabelas lineorder e part (lineorder.lo_partkey = part.p_partkey), respetivamente. Uma vez feito todos os Join’s o

motor da base de dados ordena o resultado por ano, por nacionalidade e por categoria (d_year, s_nation e p_category).

É também feito um agrupamento dos dados (Group By) com o Partial GroupAggregate por ano, nacionalidade e categoria (d_year, s_nation e p_category, respetivamente).



```
                                                                                     QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize GroupAggregate  (cost=1947257.62..1950622.47 rows=4375 width=28) (actual time=12165.258..12250.302 rows=100 loops=1)
   Group Key: date.d_year, supplier.s_nation, part.p_category
   ->  Gather Merge  (cost=1947257.62..1950458.41 rows=8750 width=36) (actual time=12164.264..12250.186 rows=300 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial GroupAggregate  (cost=1946257.60..1948448.42 rows=4375 width=36) (actual time=12107.639..12184.470 rows=100 loops=3)
               Group Key: date.d_year, supplier.s_nation, part.p_category
               ->  Sort  (cost=1946257.60..1946615.45 rows=143138 width=28) (actual time=12106.706..12159.415 rows=115436 loops=3)
                     Sort Key: date.d_year, supplier.s_nation, part.p_category
                     Sort Method: external merge  Disk: 4584kB
                     ->  Hash Join  (cost=43503.50..1930573.44 rows=143138 width=28) (actual time=370.691..11664.375 rows=115436 loops=3)
                           Hash Cond: (lineorder.lo_partkey = part.p_partkey)
                           ->  Hash Join  (cost=14964.82..1893000.43 rows=401319 width=24) (actual time=116.726..11170.537 rows=289199 loops=3)
                                 Hash Cond: (lineorder.lo_custkey = customer.c_custkey)
                                 ->  Hash Join  (cost=958.57..1839645.24 rows=2007933 width=28) (actual time=8.526..10465.256 rows=1458280 loops=3)
                                       Hash Cond: (lineorder.lo_orderdate = date.d_datekey)
                                       ->  Hash Join  (cost=873.74..1791083.49 rows=7572690 width=28) (actual time=7.646..9736.447 rows=6059779 loops=3)
                                             Hash Cond: (lineorder.lo_suppkey = supplier.s_suppkey)
                                             ->  Parallel Seq Scan on lineorder  (cost=0.00..1573877.53 rows=37494753 width=24) (actual time=0.619..5308.900 rows=29995803 loops=3)
                                             ->  Hash  (cost=798.00..798.00 rows=6059 width=12) (actual time=6.983..6.983 rows=6059 loops=3)
                                                   Buckets: 8192  Batches: 1  Memory Usage: 339kB
                                                   ->  Seq Scan on supplier  (cost=0.00..798.00 rows=6059 width=12) (actual time=0.248..5.781 rows=6059 loops=3)
                                                         Filter: ((s_region)::text = 'AMERICA'::text)
                                                         Rows Removed by Filter: 23941
                                       ->  Hash  (cost=76.35..76.35 rows=678 width=8) (actual time=0.836..0.837 rows=730 loops=3)
                                             Buckets: 1024  Batches: 1  Memory Usage: 37kB
                                             ->  Seq Scan on date  (cost=0.00..76.35 rows=678 width=8) (actual time=0.618..0.746 rows=730 loops=3)
                                                   Filter: ((d_year = 1997) OR (d_year = 1998))
                                                   Rows Removed by Filter: 1827
                                 ->  Hash  (cost=12530.00..12530.00 rows=89940 width=4) (actual time=107.528..107.529 rows=89700 loops=3)
                                       Buckets: 131072  Batches: 2  Memory Usage: 2607kB
                                       ->  Seq Scan on customer  (cost=0.00..12530.00 rows=89940 width=4) (actual time=0.366..89.761 rows=89700 loops=3)
                                             Filter: ((c_region)::text = 'AMERICA'::text)
                                             Rows Removed by Filter: 360300
                           ->  Hash  (cost=23578.00..23578.00 rows=285335 width=12) (actual time=250.140..250.140 rows=319679 loops=3)
                                 Buckets: 131072  Batches: 8  Memory Usage: 2751kB
                                 ->  Seq Scan on part  (cost=0.00..23578.00 rows=285335 width=12) (actual time=0.374..192.230 rows=319679 loops=3)
                                       Filter: (((p_mfgr)::text = 'MFGR#1'::text) OR ((p_mfgr)::text = 'MFGR#2'::text))
                                       Rows Removed by Filter: 480321
 Planning time: 0.558 ms
 Execution time: 12252.593 ms
(41 rows)
```

Figura 13 - Plano de Execução da Query Q4.2 antes da criação das chaves

```
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
Finalize GroupAggregate  (cost=1792778.27..1796155.71 rows=4375 width=28) (actual time=36376.577..36691.817 rows=100 loops=1)
  Group Key: date.d_year, supplier.s_nation, part.p_category
  ->  Gather Merge  (cost=1792778.27..1795991.65 rows=8750 width=36) (actual time=36372.016..36691.426 rows=300 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial GroupAggregate  (cost=1791778.25..1793981.66 rows=4375 width=36) (actual time=36100.405..36386.919 rows=100 loops=3)
              Group Key: date.d_year, supplier.s_nation, part.p_category
              ->  Sort  (cost=1791778.25..1792138.19 rows=143977 width=28) (actual time=36097.772..36295.832 rows=115436 loops=3)
                    Sort Key: date.d_year, supplier.s_nation, part.p_category
                    Sort Method: external merge  Disk: 4568kB
                    ->  Hash Join  (cost=43532.40..1775995.18 rows=143977 width=28) (actual time=1190.535..34941.960 rows=115436 loops=3)
                          Hash Cond: (lineorder.lo_partkey = part.p_partkey)
                          ->  Hash Join  (cost=14964.82..1740268.12 rows=401319 width=24) (actual time=302.549..33479.094 rows=289199 loops=3)
                                Hash Cond: (lineorder.lo_custkey = customer.c_custkey)
                                ->  Hash Join  (cost=958.57..1693185.00 rows=2007933 width=28) (actual time=8.069..31692.570 rows=1458280 loops=3)
                                      Hash Cond: (lineorder.lo_orderdate = date.d_datekey)
                                      ->  Hash Join  (cost=873.74..1673188.90 rows=7572690 width=28) (actual time=7.087..29507.115 rows=6059779 loops=3)
                                            Hash Cond: (lineorder.lo_suppkey = supplier.s_suppkey)
                                            ->  Parallel Seq Scan on lineorder  (cost=0.00..1573877.53 rows=37494753 width=24) (actual time=0.016..15302.682 rows=29995803 loops=3)
                                            ->  Hash  (cost=798.00..798.00 rows=6059 width=12) (actual time=7.021..7.022 rows=6059 loops=3)
                                                  Buckets: 8192  Batches: 1  Memory Usage: 339kB
                                                  ->  Seq Scan on supplier  (cost=0.00..798.00 rows=6059 width=12) (actual time=0.302..5.926 rows=6059 loops=3)
                                                        Filter: ((s_region)::text = 'AMERICA'::text)
                                                        Rows Removed by Filter: 23941
                                      ->  Hash  (cost=76.35..76.35 rows=678 width=8) (actual time=0.950..0.950 rows=730 loops=3)
                                            Buckets: 1024  Batches: 1  Memory Usage: 37kB
                                            ->  Seq Scan on date  (cost=0.00..76.35 rows=678 width=8) (actual time=0.665..0.840 rows=730 loops=3)
                                                  Filter: ((d_year = 1997) OR (d_year = 1998))
                                                  Rows Removed by Filter: 1827
                                ->  Hash  (cost=12530.00..12530.00 rows=89940 width=4) (actual time=292.878..292.878 rows=89700 loops=3)
                                      Buckets: 131072  Batches: 2  Memory Usage: 2607kB
                                      ->  Seq Scan on customer  (cost=0.00..12530.00 rows=89940 width=4) (actual time=0.325..251.183 rows=89700 loops=3)
                                            Filter: ((c_region)::text = 'AMERICA'::text)
                                            Rows Removed by Filter: 360300
                          ->  Hash  (cost=23578.00..23578.00 rows=287007 width=12) (actual time=862.576..862.577 rows=319679 loops=3)
                                Buckets: 131072  Batches: 8  Memory Usage: 2751kB
                                ->  Seq Scan on part  (cost=0.00..23578.00 rows=287007 width=12) (actual time=0.670..673.223 rows=319679 loops=3)
                                      Filter: (((p_mfgr)::text = 'MFGR#1'::text) OR ((p_mfgr)::text = 'MFGR#2'::text))
                                      Rows Removed by Filter: 480321
Planning time: 1.053 ms
Execution time: 36700.077 ms
(41 rows)
```

Figura 14 - Plano de Execução da Query Q4.2 depois da criação das chaves

Apesar dos índices terem sido criados o Postgres nunca os utilizou e o comportamento do motor ficou bastante prejudicado. Este acontecimento não está relacionado com falha da nossa parte durante a realização dos testes, mas sim com algo que não conseguimos controlar e, portanto, não conseguimos encontrar uma justificação plausível que explique o problema.

Este problema poderá ter várias origens, inicialmente equacionamos a possibilidade do Postgres versão 10 ter algum tipo de bug. No entanto, após uma conversa com o docente, surgiu a hipótese estar relacionada com o *File System Cache* do Windows – que é automatizado e controlado pelo sistema operativo. Com isso em mente, o aumento do tempo de pesquisas, após a criação das chaves, pode estar relacionado com o facto das chaves poderem estarem em vários ficheiros – como mostra na Figura 15 – e o sistema não os carrega para o sistema de ficheiros mencionado, obrigando o motor de base de dados a carregar vários ficheiros e consequentemente, aumentar o tempo de pesquisa. No entanto, não temos como o provar e investigar mais o porquê do problema foge ao âmbito da unidade curricular e do trabalho – apesar de ser um tema bastante interessante de se explorar.

Fica a tentativa de uma explicação para o que achamos anormal, uma vez que um sistema como o Postgres deveria ser capaz de ser mais inteligente. Num trabalho futuro,

seria interessante gerar uma carga mais pesada de dados, por exemplo 100 ou 200 GB, e analisar se a criação das chaves tem realmente um enorme impacto nas pesquisas no mesmo ambiente utilizado para este trabalho.

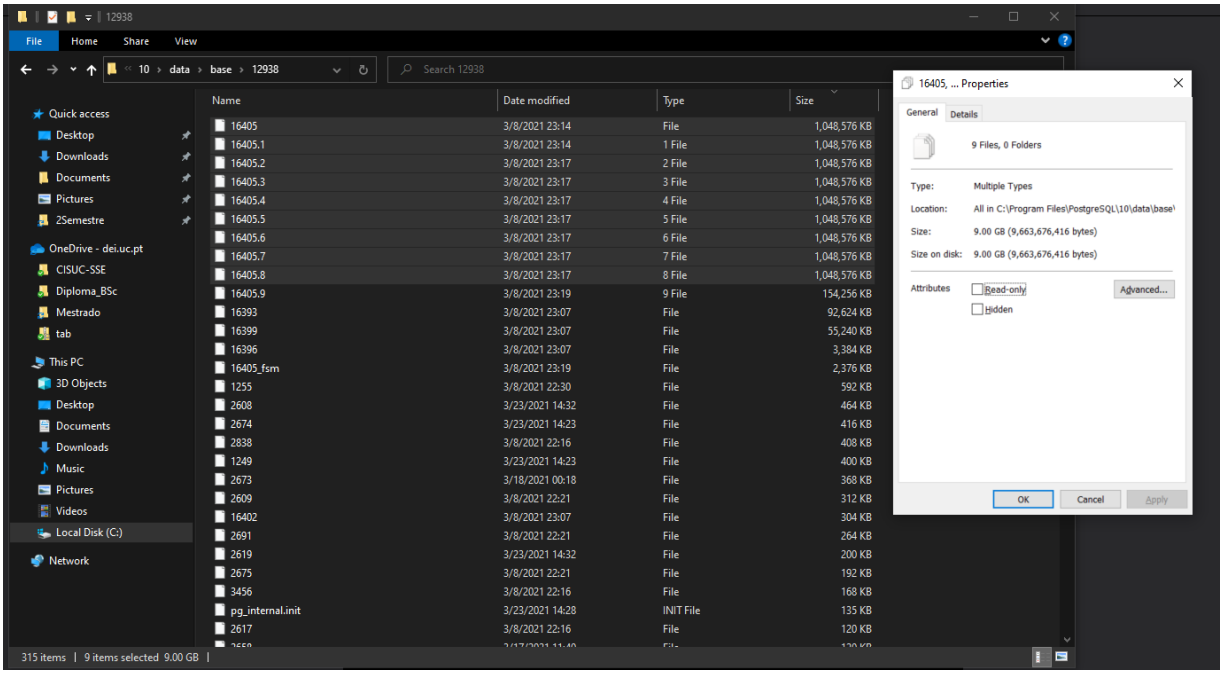


Figura 15 -Ficheiros do Postgres relativos à tabela lineorder

De forma a tentar explorar um pouco o problema, forçamos o motor de base de dados Postgres a utilizar os índices durante a pesquisa. A Figura 16 mostra o exemplo de um plano de execução que utiliza índices e que nos mostra que o tempo de execução baixa um pouco face aos apresentados anteriormente. No entanto, o resultado continua a ser superior aos valores registados antes da criação das chaves.

Como referido anteriormente, não conseguimos encontrar uma explicação certa para isto, no entanto fica registado o esforço. Queremos realçar de que nos apercebemos desta anomalia e não a quisemos ignorar.

```
                                                                QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=10002581099.53..10002586337.37 rows=4375 width=28) (actual time=22104.731..22256.515 rows=100 loops=1)
   Group Key: date.d_year, supplier.s_nation, part.p_category
   ->  Sort  (cost=10002581099.53..10002581963.39 rows=345544 width=28) (actual time=22102.765..22206.471 rows=346308 loops=1)
         Sort Key: date.d_year, supplier.s_nation, part.p_category
         Sort Method: external merge  Disk: 13720kB
         ->  Hash Join  (cost=10000064093.71..10002541041.55 rows=345544 width=28) (actual time=335.053..21588.964 rows=346308 loops=1)
               Hash Cond: (lineorder.lo_partkey = part.p_partkey)
               ->  Hash Join  (cost=10000022738.69..10002484468.22 rows=963165 width=24) (actual time=127.962..21063.925 rows=867597 loops=1)
                     Hash Cond: (lineorder.lo_custkey = customer.c_custkey)
                     ->  Hash Join  (cost=10000001531.02..10002384372.45 rows=4819039 width=28) (actual time=6.891..19886.741 rows=4374841 loops=1)
                           Hash Cond: (lineorder.lo_orderdate = date.d_datekey)
                           ->  Hash Join  (cost=10000001394.13..10002336448.53 rows=18174457 width=28) (actual time=5.723..18431.582 rows=18179337 loops=1)
                                 Hash Cond: (lineorder.lo_suppkey = supplier.s_suppkey)
                                 ->  Seq Scan on lineorder  (cost=10000000000.00..10002098804.08 rows=89987408 width=24) (actual time=0.254..8963.631 rows=89987410 loops=1)
                                 ->  Hash  (cost=1318.39..1318.39 rows=6059 width=12) (actual time=5.435..5.435 rows=6059 loops=1)
                                       Buckets: 8192  Batches: 1  Memory Usage: 339kB
                                       ->  Index Scan using supplier_const on supplier  (cost=0.29..1318.39 rows=6059 width=12) (actual time=0.007..4.745 rows=6059 loops=1)
                                             Filter: ((s_region)::text = 'AMERICA'::text)
                                             Rows Removed by Filter: 23941
                           ->  Hash  (cost=128.42..128.42 rows=678 width=8) (actual time=1.146..1.146 rows=730 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 37kB
                                 ->  Index Scan using date_const on date  (cost=0.28..128.42 rows=678 width=8) (actual time=0.864..1.075 rows=730 loops=1)
                                       Filter: ((d_year = 1997) OR (d_year = 1998))
                                       Rows Removed by Filter: 1827
                     ->  Hash  (cost=19731.42..19731.42 rows=89940 width=4) (actual time=120.158..120.158 rows=89700 loops=1)
                           Buckets: 131072  Batches: 2  Memory Usage: 2607kB
                           ->  Index Scan using customer_const on customer  (cost=0.42..19731.42 rows=89940 width=4) (actual time=0.009..107.825 rows=89700 loops=1)
                                 Filter: ((c_region)::text = 'AMERICA'::text)
                                 Rows Removed by Filter: 360300
               ->  Hash  (cost=36365.43..36365.43 rows=287007 width=12) (actual time=204.941..204.941 rows=319679 loops=1)
                     Buckets: 131072  Batches: 8  Memory Usage: 2751kB
                     ->  Index Scan using part_const on part  (cost=0.42..36365.43 rows=287007 width=12) (actual time=0.013..160.796 rows=319679 loops=1)
                           Filter: (((p_mfgr)::text = 'MFGR#1'::text) OR ((p_mfgr)::text = 'MFGR#2'::text))
                           Rows Removed by Filter: 480321
 Planning time: 1.011 ms
 Execution time: 22259.928 ms
(36 rows)
```

Figura 16 - Plano de Execução após forçar o motor a usar os índices

5. Considerações Finais

Dado por terminado o projeto é necessário fazer um balanço dos objetivos alcançados, assim como as competências adquiridas.

De uma forma geral, toda a execução de queries e produção de gráficos solicitados no enunciado foram implementadas com sucesso. Com isto em mente, é importante realçar que os objetivos de aprendizagem foram alcançados. Nomeadamente, foram conhecidos um conjunto de opções que o motor de base de dados pode escolher para melhorar a performance de execução, assim como técnicas para ajudar o motor a melhorar o seu desempenho.

O desenvolvimento do trabalho prático e as aulas práticas, no seu conjunto foram de grande suporte para compreendermos o processo interno que um motor de base de dados realiza antes da devolução do resultado. Assim sendo, no futuro, caso seja necessário, estaremos preparados para desenvolver soluções e analisar como um motor de base de dados faz internamente a sua escolha de execução de queries e dessa forma construir soluções mais eficientes de pesquisas à base de dados.

Consideramos, ainda, que os resultados apresentados podem ser influenciados pelo tipo de hardware (setup) como mencionado em secções anteriores, pois existe, hoje em dia, hardware muito avançado o que faz com que os resultados sejam mais eficientes em máquinas mais recentes. Contudo, tentamos procurar fazer uma comparação dos nossos setups e encontrar soluções ajustadas aquilo que nos é exigido e esperado no âmbito da disciplina de Sistema de Gestão de Dados.

Anexo A. Querys

Q1.1:

```
select sum(lo_revenue) as revenue
from lineorder
left join date on lo_orderdate = d_datekey
where d_year = 1993
and lo_discount between 1 and 3
and lo_quantity < 25;
```

Q1.2:

```
select sum(lo_revenue) as revenue
from lineorder left join date on lo_orderdate = d_datekey
where d_yearmonthnum = 199401
and lo_discount between 4 and 6
and lo_quantity between 26 and 35;
```

Q1.3:

```
select sum(lo_revenue) as revenue
from p_lineorder left join date on lo_orderdate = d_datekey
where d_weeknuminyear = 6 and d_year = 1994 and
lo_discount between 5 and 7 and lo_quantity between 26 and
35;
```

Q2.1

```
select sum(lo_revenue) as lo_revenue, d_year, p_brand1 from lineorder left join date on
lo_orderdate = d_datekey left join part on lo_partkey = p_partkey left join supplier on
lo_suppkey = s_suppkey where p_category = 'MFGR#12' and s_region = 'AMERICA'
group by
d_year, p_brand1 order by d_year, p_brand1;
```

Q2.2

```
select sum(lo_revenue) as lo_revenue, d_year, p_brand1 from lineorder left join date on
lo_orderdate = d_datekey left join part on lo_partkey = p_partkey left join supplier on
lo_suppkey = s_suppkey where p_brand1 between 'MFGR#2221' and 'MFGR#2228' and s_region
= 'ASIA' group by d_year, p_brand1 order by d_year, p_brand1;
```

Q2.3

```
select sum(lo_revenue) as lo_revenue, d_year, p_brand1 from lineorder left join date on
lo_orderdate = d_datekey left join part on lo_partkey = p_partkey left join supplier on
lo_suppkey = s_suppkey where p_brand1 = 'MFGR#2239' and s_region = 'EUROPE' group by
d_year, p_brand1 order by d_year, p_brand1;
```

Q3.1

```
select c_nation, s_nation, d_year, sum(lo_revenue) as lo_revenue from lineorder left join
date on lo_orderdate = d_datekey left join customer on lo_custkey = c_custkey left join
supplier on lo_suppkey = s_suppkey where c_region = 'ASIA' and s_region = 'ASIA'and
d_year
>= 1992 and d_year <= 1997 group by c_nation, s_nation, d_year order by d_year asc,
lo_revenue desc;
```

Q3.2

```
select c_city, s_city, d_year, sum(lo_revenue) as lo_revenue from lineorder left join date
on
lo_orderdate = d_datekey left join customer on lo_custkey = c_custkey left join supplier
on
lo_suppkey = s_suppkey where c_nation = 'UNITED STATES' and s_nation = 'UNITED
STATES' and
d_year >= 1992 and d_year <= 1997 group by c_city, s_city, d_year order by d_year asc,
lo_revenue desc;
```

Q3.3

```
select c_city, s_city, d_year, sum(lo_revenue) as lo_revenue from lineorder left join date
on
lo_orderdate = d_datekey left join customer on lo_custkey = c_custkey left join supplier
on
lo_suppkey = s_suppkey where (c_city='UNITED KI1' or c_city='UNITED KI5') and
(s_city='UNITED KI1' or s_city='UNITED KI5') and d_year >= 1992 and d_year <=
1997 group by
c_city, s_city, d_year order by d_year asc, lo_revenue desc;
```

Q4.1

```
select d_year, c_nation, sum(lo_revenue) - sum(lo_supplycost) as profit from lineorder
left
join date on lo_orderdate = d_datekey left join customer on lo_custkey = c_custkey left
join
supplier on lo_suppkey = s_suppkey left join part on lo_partkey = p_partkey where
c_region =
'AMERICA' and s_region = 'AMERICA' and (p_mfgr = 'MFGR#1' or p_mfgr =
'MFGR#2') group by
d_year, c_nation order by d_year, c_nation;
```

Q4.2

```
select d_year, s_nation, p_category, sum(lo_revenue) - sum(lo_supplycost) as profit from
lineorder left join date on lo_orderdate = d_datekey left join customer on lo_custkey =
c_custkey left join supplier on lo_suppkey = s_suppkey left join part on lo_partkey =
p_partkey
where c_region = 'AMERICA'and s_region = 'AMERICA' and (d_year = 1997 or d_year
= 1998)
and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2') group by d_year, s_nation, p_category
order by
d_year, s_nation, p_category;
```

Q4.3

```
 select d_year, s_city, p_brand1, sum(lo_revenue) - sum(lo_supplycost) as profit from lineorder

left join date on lo_orderdate = d_datekey left join customer on lo_custkey = c_custkey left

join supplier on lo_suppkey = s_suppkey left join part on lo_partkey = p_partkey where

c_region = 'AMERICA'and s_nation = 'UNITED STATES' and (d_year = 1997 or d_year = 1998) and

p_category = 'MFGR#14' group by d_year, s_city, p_brand1 order by d_year, s_city, p_brand1;
```