Universidade de Coimbra

Sistemas de Gestão de Dados

Projeto 1

TPC-H benchmarking

Relatório

23 de Março de 2018

Autor	Número	Contacto
Daniel Azevedo	2014200607	dazevedo@student.dei.uc.pt
Miguel Arieiro	2014197166	marieiro@student.dei.uc.pt

Conteúdo

1 Introdução

Neste projeto foi-nos proposto desenvolver um benchmarking que permitisse avaliar o desempenho de três Sistemas de Gestão de Base de Dados (SGBD) diferentes. Para tal utilizámos o TPC-H (`www.tpc.org`) que é um benchmark de bases de dados de apoio à decisão. Usando o TPC-H é possível ter acesso a queries e registos para preencher as tabelas das bases de dados e com isso permite, com base nos tempos de execução das diferentes queries, comparar e tirar conclusões relativas às diferenças entre SGBD's. Com este projeto iremos também analisar as diferenças em termos de performance entre SGBD's relacionais (RDBMS) e não relacionais (NoSQL), neste caso será necessário criar queries que desempenhem funções semelhantes para SQL e para NoSQL de modo a poder avaliá-los de forma quantitativa e qualitativa.

Os testes e experiências vão ser em dois computadores com especificações diferentes:

- **PC1:** MacBook Pro 13-inch, processador 2 GHz Intel Core i5, 8 GB de Memória RAM e disco SSD de 256GB.
- **PC2:** i5 2320 @ 3.30 GHz, 16Gb RAM DDR3 @ 1600Mhz, Samsung 850 EVO (250Gb)

2 SGBDs escolhidos

Iremos analisar três SGBD's diferentes:

- O primeiro motor obrigatório será **MySQL**.
- O segundo motor escolhido será **PostgreSQL**, um SGBD relacional que permite correr as pesquisas do TPC-H.
- O último motor será **MongoDB**, este é bastante diferente dos anteriores por ser um motor NoSQL.

3 Especificações dos Motores de Base de Dados

- **MySQL**:
  - Versão: 5.7.19
  - Versão de Compilação: 64 bit
- **PostgreSQL**:
  - Versão: 9.6.4
  - Versão de Compilação: 64 bit
- **MongoDB**:
  - Versão: 3.4.7
  - Versão de Compilação: 64 bit

De notar que as configurações dos motores utilizadas foram as *default*, nao tendo havido alterações nas mesmas.

4 Setup

Antes de executar as queries tivemos que preparar o ambiente de testes, para tal executámos os seguintes passos:

1. **Instalação** do MySQL, PostgreSQL e MongoDB nos PC's.
2. **Criação das tabelas:** com base em código fornecido foi criado o ficheiro *create_tables.sql* onde está o código SQL para criação das tabelas para o **MySQL** e o **PostgreSQL** e corremos esses ficheiros pelo terminal nas respetivas shells. Para o **MongoDB** não foi necessário já que este além de não possuir o conceito de tabelas, cria as collections(equivalente de tabelas em NoSQL) automaticamente quando os registos são importados.
3. **Geração de registos para as tabelas:** para gerar os registos que iríamos colocar nas tabelas utilizámos o comando *dbgen* e gerámos 15GB de dados através do seguinte comando:

   ```
   ./dbgen -s 15
   ```

   Com este comando foram criados os ficheiros *.tbl* que iriam ser colocados nas tabelas.
4. **Colocação dos dados nas tabelas:** para a colocação dos dados nas tabelas criadas criámos os scripts *load_data.sql*:

   - Para o motor **MySQL** usámos como base o comando:

     ```
     load data local infile "table_name.tbl" into table table_name fields
                   terminated by "|" lines terminated by "\n";
     ```

   - Para o motor de **PostgreSQL** utilizámos como base o comando:

     ```
     COPY table_name FROM 'table_name.tbl' WITH DELIMITER AS '|';
     ```

     Para o PostgreSQL tivemos que fazer fazer uma alteração nos ficheiros *.tbl* de maneira a estes serem importados corretamente, como tal criámos o script *parse_tbls.sh* que retira o último "|" de cada linha, este script vai ser executado antes do *load_data.sql*.

   - Para o motor **MongoDB** utilizámos como base o comando:

     ```
     mongoimport -d tpch_test -c table_name --type csv --file
                   table_name.csv --headerline
     ```

     Neste passo tivemos dificuldades acrescidas, devido a não conseguirmos importar diretamente os ficheiros *.tbl* tivemos que converter estes ficheiros para *.csv* usando o script *create_csv.sh*. Após isso criámos o script *parse_csv.sh* que adiciona os headers necessários aos ficheiros, remove as "," dos *csv's* e por fim substitui as " | " por ",".
     Seguindo este processo vão ser criadas collections no MongoDB semelhantes às tabelas presentes nos motores MySQL e PostgreSQL, tendo os mesmos atributos e os mesmos registos.

5 Carregamento de dados em cada SGBD

5.1 Tempos de Carregamento

Na Figura 1 é possível ver graficamente os tempos de carregamento das tabelas de tamanho significativo(Partsupp, Orders e Lineitem).

Para as restantes tabelas, os tempos de carregamento são inferiores a 1 minuto, como tal estes não devem ser alvo de muita atenção visto que grande percentagem desses tempos se deve a warm-up dos motores das bases de dados.

Através de uma primeira análise destes gráficos conseguimos perceber que o motor PostgreSQL é aquele que apresenta melhores tempos, seguido do MySQL e terminando com MongoDB.

PC1

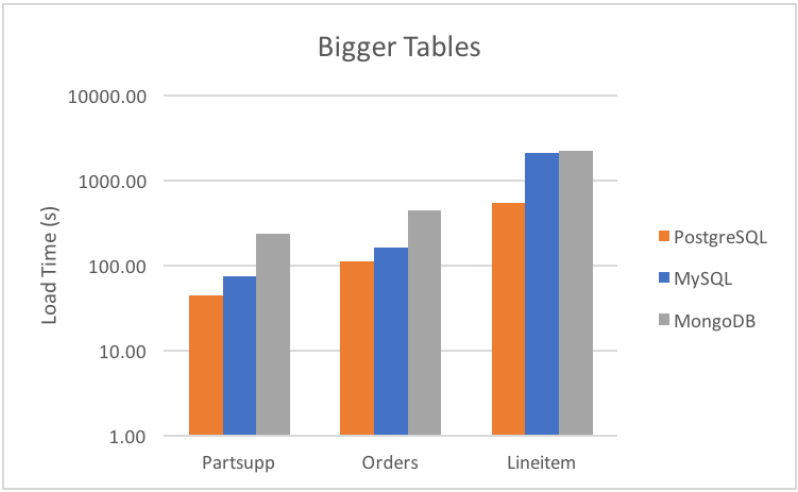


Figura 1: Carregamento das Tabelas - PC1

PC2

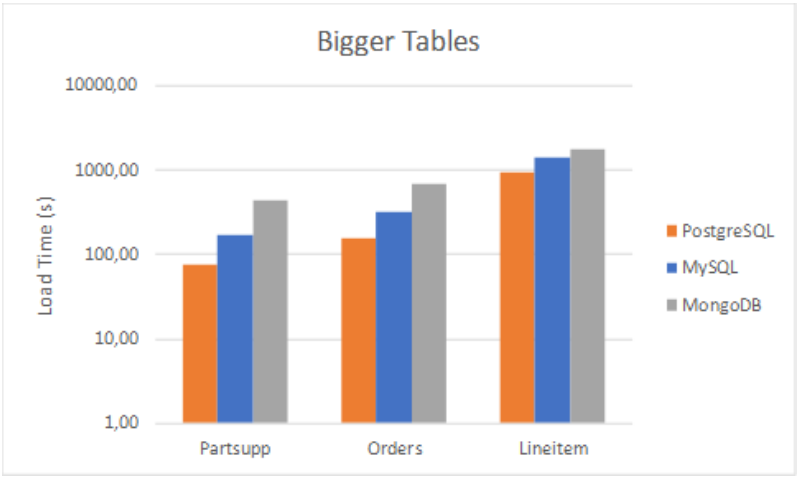


Figura 2: Carregamento das Tabelas - PC2

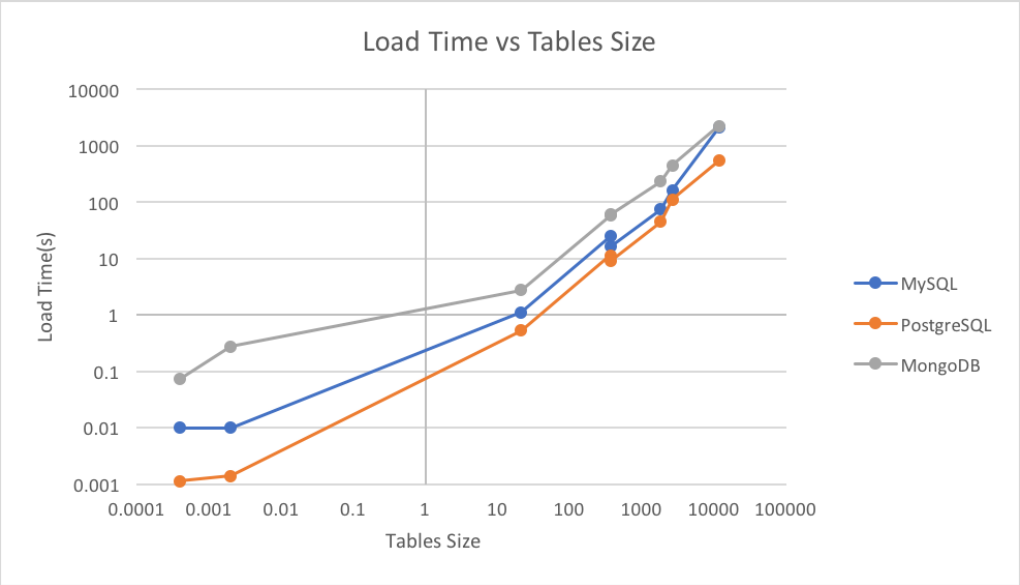
5.2 Tempos de Carregamento Vs Tamanho das tabelas

Na tabela seguinte podemos ver os tempos de carregamento dos registos para as diferentes tabelas e usando os 3 motores de buscar considerados.

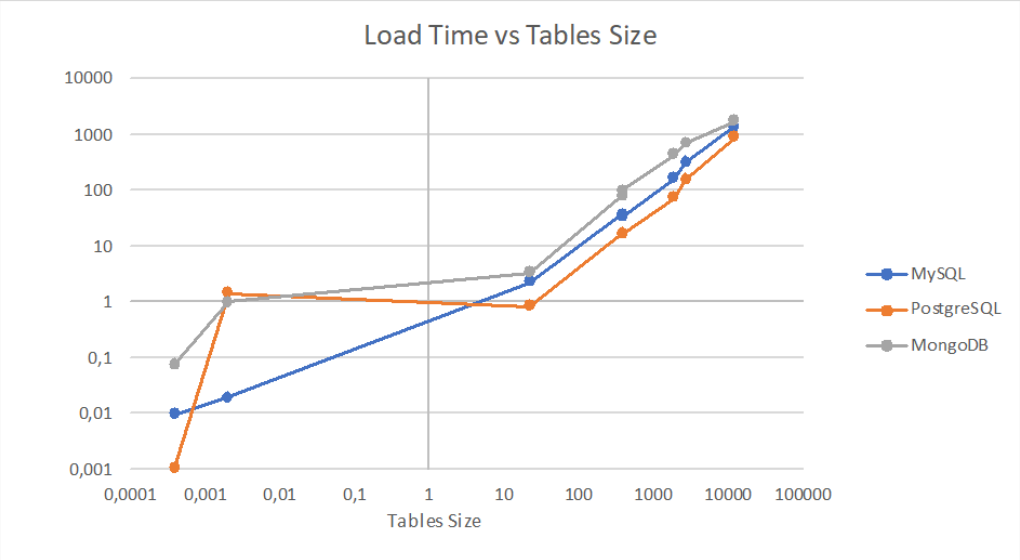
Size (MB)	PC1			PC2		
	MySQL (s)	PostgreSQL (s)	MongoDB (s)	MySQL (s)	PostgreSQL (s)	MongoDB (s)
Region - 0,000389	0,01	0,00	0,07	0,01	0,00	0,08
Nation - 0,002	0,01	0,00	0,28	0,02	1,46	1,02
Supplier - 21,3	1,12	0,53	2,79	2,31	0,89	3,43
Part - 365,5	25,02	11,48	58,42	37,89	17,66	84,29
Customer - 367,8	16,52	8,99	61,23	35,07	16,92	103,07
Partsupp - 1810	74,45	45,03	236,76	169,52	76,28	446,26
Orders - 2630	165,43	112,52	453,81	320,80	156,21	699,68
Lineitem - 11 720,00	2 112,41	551,09	2253,62	1 418,78	937,57	1753,23

Tabela 1: Tempos de Carregamento

PC1



PC2



Como podemos ver por estes dados o PostgreSQL é o motor que demora menos tempo a carregar os dados e também é o mais escalável, isto quer dizer que mesmo aumentando a carga de dados a carregar para a base de dados o tempo de carregamento não aumenta exponencialmente.

O motor MySQL e MongoDB apresentam piores resultados em termos de tempo de carregamento. Embora para poucos dados o MySQL apresente melhores tempos que o MongoDB, podemos prever, com base no gráfico acima, que para um elevado número de dados o MySQL apresentará um pior comportamento relativamente ao MongoDB.

No PC2 é possível observar alguma descrepância nos tempos da segunda tabela no MySQL, isto pode ter sido originado por algum ruído externo durante o carregamento dos dados. Como o MongoDB importa apenas ficheiros simples, sem verificar o seu tipo da dados ou conteúdo poderíamos pensar que seria mais rápido que o PostgreSQL e o MySQL, mas isso não é verificado a razão poderá ser a elevada otimização nos INSERTS nos motores relacionais.

5.3 Médias e Desvios Padrões

Nas tabelas 2 e 3 podemor ver a média e desvio padrão no carregamento de dados para ambos os PC's, como seria previsível o PostgreSQL é o motor que apresenta melhor média de carregamento(24,92 MB/s e 14,97MB/s) e cerca de linhas carregadas por segundo(180698 e 109145).

O MongoDB é aquele que apresenta pior média de carregamento(4,82/3,58 MB/s, e 35276/21164 linhas/s) sendo aproximadamente 5 vezes mais lento que o PostgreSQL, por outro lado é o que apresenta o menor desvio padrão significando que os valores oscilam pouco relativamente à média, sendo mais consistente.

	PC1			**PC2**		
	MySQL (MB/s)	**PostgreSQL (MB/s)**	**MongoDB (MB/s)**	**MySQL (MB/s)**	**PostgreSQL (MB/s)**	**MongoDB (MB/s)**
AVG	12,74	24,92	4,82	7,08	14,97	3,58
STDEV	9,62	16,63	3,09	4,42	9,88	2,48

Tabela 2: Média e Desvio Padrão (MB/s)

	PC1			**PC2**		
	MySQL (Lines/s)	**PostgreSQL (Lines/s)**	**MongoDB (Lines/s)**	**MySQL (Lines/s)**	**PostgreSQL (Lines/s)**	**MongoDB (Lines/s)**
AVG	91602,72	180698,24	35276,73	51952,52	109145,85	21164,21
STDEV	65503,29	111619,11	22500,22	31551,92	70071,67	19099,86

Tabela 3: Média e Desvio Padrão (linhas/s)

6 Construção das chaves em cada SGBD

Apenas os motores MySQL e PostegreSQL possuem os conceitos de chaves primárias e chaves estrangeiras, como consequência disto não existirão dados e tempos para a criação de chaves para o motor MongoDB.

6.1 Tempos de Carregamento

Pelos gráficos precebemos que o MySQL é aquele que apresenta valores mais elevados na criação de chaves, tanto primárias como estrangeiras. Mais uma vez apenas considerámos as tabelas de maior tamanho: Partsupp, Orders e Lineitem, já que as restantes apresentam tempos muito baixos o que não permite fazer uma análise correta devido à influência do tempo de warm-up das bases de dados.
Como podemos verificar pelos gráficos utilizámos uma escala logarítmica devido á elevada gama de tampos. No gráfico das chaves estrangeiras podemos ver que, especiamente no PC2, algumas chaves não foram criadas devido a erros/elevados tempos de processamento. Podemos também ver que no PC2 a chave primária não foi criada com sucesso.

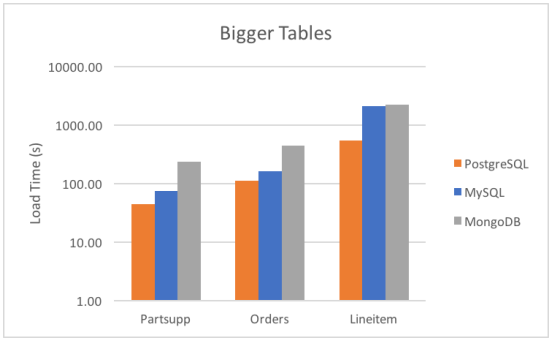


Figura 3: Chaves Primárias - PC1

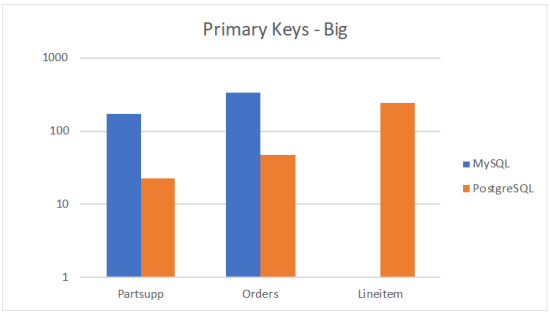


Figura 4: Chaves Estrangeiras - PC1

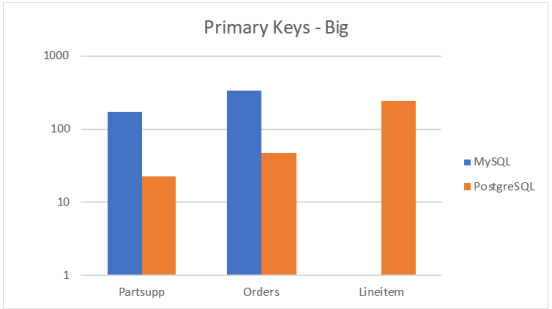


Figura 5: Chaves Primárias - PC2

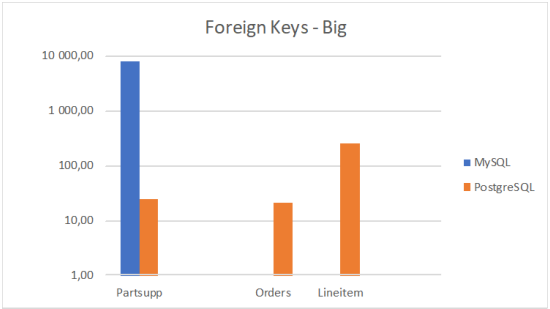


Figura 6: Chaves Estrangeiras - PC2

6.2 Tempos de Carregamento Vs Criação de chaves

Na Tabela 4 e 5 podemos ver os valores ilustrados acima mas agora em tabela e com referência às tabelas.
Na Tabela 4 vemos também que os tempos de criação de chaves primárias aumentam à medida que o tamanho das tabelas aumentam também.

Size (MB)	**PC1**		**PC2**	
	MySQL (s)	**PostgreSQL (s)**	**MySQL (s)**	**PostgreSQL (s)**
Region - 0,000389	0,06	0,01	0,09	0,01
Nation - 0,002	0,04	0,01	0,08	0,00
Supplier - 21,3	0,72	0,13	4,45	0,13
Part - 365,5	12,89	2,33	38,39	3,98
Customer - 367,8	10,46	2,07	24,08	3,25
Partsupp - 1810	94,45	12,88	171,01	23,01
Order - 2630	156,48	19,66	343,07	47,41
Lineitem - 11 720,00	787,43	86,86	-	242,14

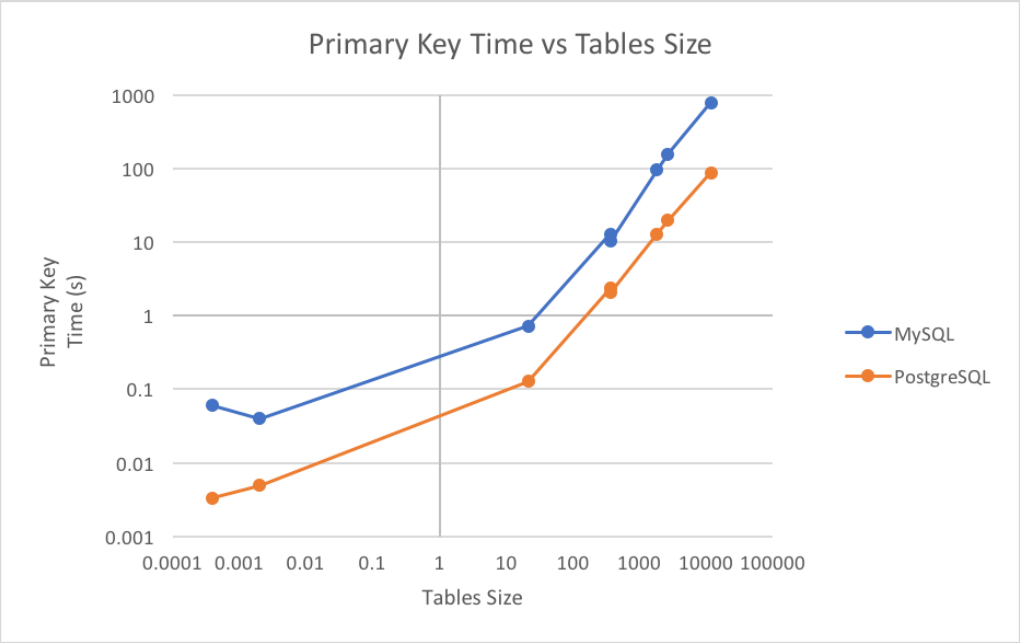
Tabela 4: Criação de chaves primárias

Na Tabela 5 podemos ver os tempos de criação de chaves estrangeiras, como podemos concluir a criação das chaves está dependente do tamanho da tabela onde vai ser criada a chave e não do tamanho da tabela à qual a chave primária corresponde. As células que possuem um "-" indicam que essa chave demorou mais que 2 horas a correr e devido a isso foi interrompido. De notar que o facto de uma determinada chave estrangeira não ter sido criada não tem muitas consequências práticas pois estas apenas servem para garantir integridade e consistência nos dados.

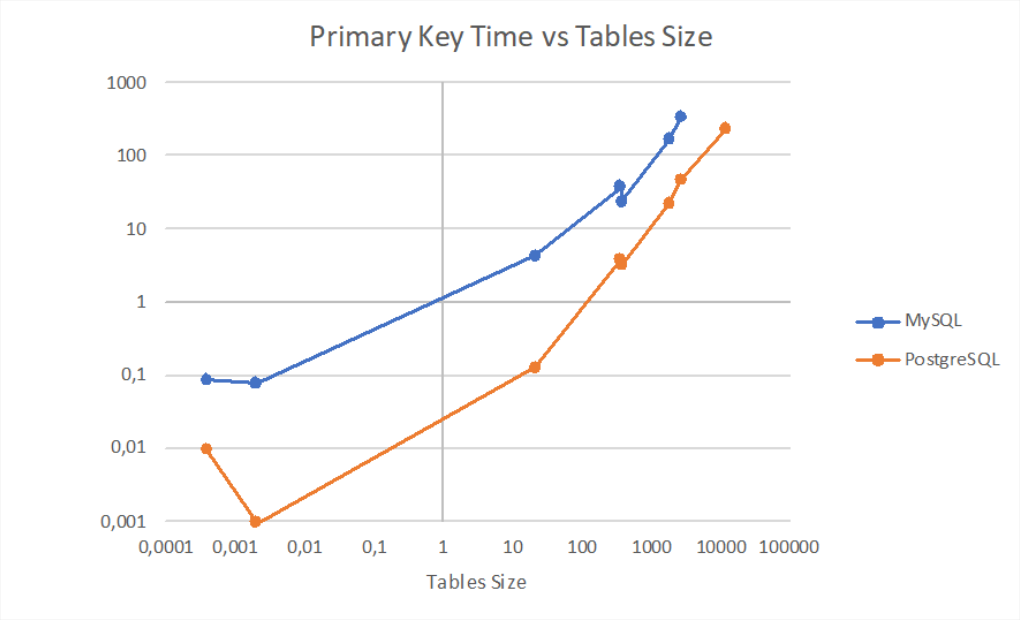
Size (MB)	**PC1**		**PC2**	
	MySQL (s)	**PostgreSQL (s)**	**MySQL (s)**	**PostgreSQL (s)**
Nation - Region	0,04	0,01	4,41	0,01
Supplier - Nation	1,32	0,10	52,96	0,05
Customer - Nation	18,65	0,75	1055,22	1,44
Partsupp - Supplier	2056,62	11,65	8116,84	24,82
Partsupp - Part				
Orders - Customer	9476,26	16,17	-	22,07
Lineitem - Orders	-	1769,08	-	256,95
Lineitem - Part, Supplier				

Tabela 5: Criação de Chaves Estrangeiras

PC1



PC2



Nestes dois gráficos podemos ver o tempo de criação de chaves primárias em função do tamanho das tabelas, como podemos perceber o motor PostgreSQL é aquele com melhores tempos na criação de chaves primárias. À medida que o tamanho das tabelas aumentam, os tempos para o MySQL aumentam consideravelmente o que demonstra que este não é muito escalável, ao contrário do PostgreSQL que demonstra ser ideal para tabelas como muitos dados.

6.3 Média e Desvio Padrão

Nea Tabela 6 podemos observar os valores da média e desvio padrão para esta operação no PC1 e no PC2, o PostgreSQL apresenta uma média muito superior ao MySQL, cerca de 5 vezes maior, o que seria de esperar tendo em conta os dados obtidos anteriormente.

	PC1		**PC2**	
	MySQL (MB/s)	**PostgreSQL (MB/s)**	**MySQL (MB/s)**	**PostgreSQL (MB/s)**
AVG	21,82	113,87	38,85	69,18
STDEV	53,96	71,78	86,18	55,31

Tabela 6: Média e Desvio Padrão

7 Resultados e Análise

Nesta secção é possível ver os resultados que obtivemos, para cada querie executada foi registado o seu tempo de execução no PC1 e no PC2, no Anexo B é possível ver estes valores na forma de gráfico.

7.1 MySQL

Queries	Time(s) PC1	Time(s) PC2
Q1	299,33	468,79
Q2	11,64	19,68
Q3	240,39	342,14
Q4	-	-
Q5	456,01	1002,33
Q6	155,82	105,94
Q7	492,22	775,47
Q8	1072,94	-
Q9	2 686,67	-
Q10	306,88	444,32
Q11	182,96	154,36
Q12	120,38	135,73
Q13	1 505,95	6200,88
Q14	121,84	425,50
Q15	219,08	225,25
Q16	120,97	126,48
Q17	-	-
Q18	1053,52	1544,49
Q19	216,13	717,75
Q20	-	-
Q21	-	-
Q22	18,33	15,67

Tabela 7: Tempos Queries MySQL

7.2 PostgreSQL

Queries	Time(s) PC1	Time(s) PC2
Q1	178,60	109,98
Q2	22,32	22,54
Q3	59,69	100,24
Q4	90,22	86,78
Q5	67,53	104,51
Q6	30,78	60,57
Q7	47,61	93,89
Q8	39,77	82,18
Q9	147,74	157,78
Q10	53,91	92,02
Q11	8,79	12,38
Q12	44,71	75,38
Q13	59,49	42,03
Q14	28,85	63,80
Q15	68,65	125,71
Q16	16,62	25,78
Q17	-	-
Q18	215,78	210,94
Q19	36,09	64,92
Q20	-	-
Q21	793,39	808,09
Q22	24,33	19,84

Tabela 8: Tempos Queries PostgreSQL

7.3 Análise dos tempos de execução das queries

Nesta secção vai ser feita uma análise dos tempos de execução obtidos para as 22 queries corridas em MySQL e PostgreSQL e vão ser tentadas responder as perguntas do **ponto 5** do enunciado.

Antes de tentar responder às perguntas tentámos perceber primeiro as diferenças em termos de estrutura e armazenamento entre o motor MySQL(mais especificamente InnoDB) e o motor PostgreSQL.

MySQL vs PostgreSQL

O motor **MySQL** possui uma estrutura B-tree, o que permite que os registos estejam de alguma forma ordenados de acordo com o index primário. Isto leva a que seja necessário menos I/O para aceder aos registos e o ordenamento poderá ser mais rápido com este motor. O index para a chave primária e o registo em si estão presentes na mesma estrutura, quando existem indexes secundários o acesso aos registos é feito através do index primário, o que pode tornar as queries mais lentas.

O motor **PostgreSQL** adopta uma estrutura Heap table para armazenar as tabelas, nesta forma os registos não estão ordenados e ficam armazenados em páginas em disco sem qualquer ordem. Para poderem ser acedidos é guardado numa estrutura a sua chave primária(index) e um ponteiro para o registo em disco, isto fará com que seja necessário mais acessos a disco, embora possa tornar mais rápido o acesso aos dados.

7.3.1 Pergunta 1 - Análise de queries dentro dos mesmos SGDB

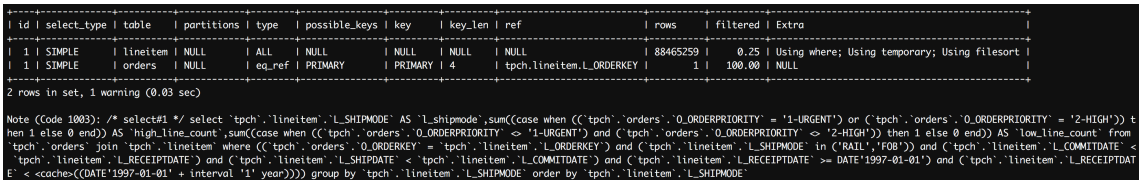
Porque é que os tempos de execução das queries variam, em particular como se justifica o tempo de execução de cada query (comparativamente com as restantes)?

7.3.1.1 MySQL

Com o objetivo de perceber melhor as variações dos tempos de execução entre certas queries corridas em MySQL, de seguida vão ser comparadas queries rápidas com queries lentas, mais especificamente a **Q12 vs Q9** e **Q14 vs Q13** e com a ajuda do plano de execução perceber as diferenças.

7.3.1.1.1 Q12 vs Q9

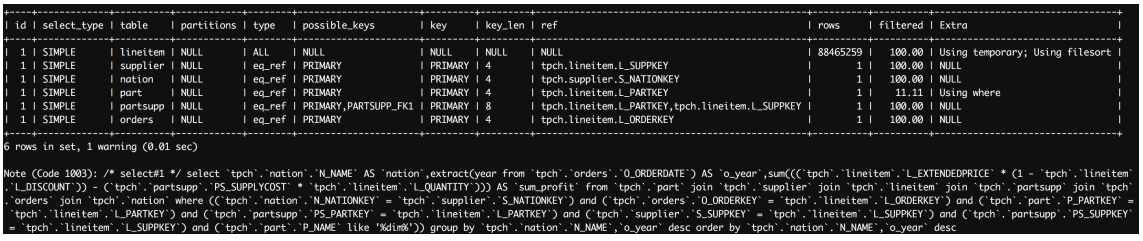
No plano da **Q12**, podemos ver que basicamente são acedidas duas tabelas: Lineitem e Orders - que são as maiores. Na leitura da primeira linha, que corresponde à tabela Lineitem, vemos que não são usados indexes e que a leitura é sequencial, já que o TYPE é ALL, por isso era de esperar a estimativa do número de rows(coluna ROWS) a examinar (88465259 linhas) seja semelhante ao número de linhas da tabela Lineitem. Uma possível otimização seria colocar um index no campo da tabela Lineitem sobre o qual é feita a pesquisa. Na segunda linha do plano vemos os detalhes sobre a pesquisa na tabela Orders, esta já utiliza um index para a pesquisa o que a tornará mais rápida, este index deverá ser o da chave primária, já que não foram criados indexes adicionais. Vendo o código é possível ver que a maioria das clásulas WHERE são simples e comparam um campo da tabela com um valor fixo, não o comparando com campos de outras tabelas.



```
+----+-------------+----------+------------+-------+---------------+---------+---------+--------------------------+----------+----------+----------------------------------------------+
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len | ref                      | rows     | filtered | Extra                                        |
+----+-------------+----------+------------+-------+---------------+---------+---------+--------------------------+----------+----------+----------------------------------------------+
|  1 | SIMPLE      | lineitem | NULL       | ALL   | NULL          | NULL    | NULL    | NULL                     | 88465259 |     0.25 | Using where; Using temporary; Using filesort |
|  1 | SIMPLE      | orders   | NULL       | eq_ref | PRIMARY      | PRIMARY | 4       | tpch.lineitem.L_ORDERKEY |        1 |   100.00 | NULL                                         |
+----+-------------+----------+------------+-------+---------------+---------+---------+--------------------------+----------+----------+----------------------------------------------+
2 rows in set, 1 warning (0.03 sec)

Note (Code 1003): /* select#1 */ select `tpch`.`lineitem`.`L_SHIPMODE` AS `l_shipmode`,sum((case when ((`tpch`.`orders`.`O_ORDERPRIORITY` = '1-URGENT') or (`tpch`.`orders`.`O_ORDERPRIORITY` = '2-HIGH')) t
hen 1 else 0 end)) AS `high_line_count`,sum((case when ((`tpch`.`orders`.`O_ORDERPRIORITY` <> '1-URGENT') and (`tpch`.`orders`.`O_ORDERPRIORITY` <> '2-HIGH')) then 1 else 0 end)) AS `low_line_count` from
 `tpch`.`orders` join `tpch`.`lineitem` where ((`tpch`.`orders`.`O_ORDERKEY` = `tpch`.`lineitem`.`L_ORDERKEY`) and (`tpch`.`lineitem`.`L_SHIPMODE` in ('RAIL','FOB')) and (`tpch`.`lineitem`.`L_COMMITDATE` <
 `tpch`.`lineitem`.`L_RECEIPTDATE`) and (`tpch`.`lineitem`.`L_SHIPDATE` < `tpch`.`lineitem`.`L_COMMITDATE`) and (`tpch`.`lineitem`.`L_RECEIPTDATE` >= DATE'1997-01-01') and (`tpch`.`lineitem`.`L_RECEIPTDAT
E` < <cache>((DATE'1997-01-01' + interval '1' year)))) group by `tpch`.`lineitem`.`L_SHIPMODE` order by `tpch`.`lineitem`.`L_SHIPMODE`
```

Figura 7: plano de execução Q12

No plano da **Q9** vemos que acede a praticamente todas as tabelas, numa querie são usadas todas as tabelas incluindo a Lineitem, Orders e Partsupp que são as maiores. Existem clásulas WHERE onde são comparadas chaves primárias destas tabelas com o campo respetivo na tabela Lineitem o que aumenta em muito o tempo. Embora os indexes das chaves primárias diminuam o tempo, este continua a ser muito grande devido a estarmos a comparar com campos normais na tabela Lineitem. O número de rows é quase sempre 1 porque é do TYPE *eq_ref*, que significa que em cada uma dessas tabelas apenas 1 linha será lida, ja que as chaves primárias, que é o que estamos a pesquisar, são únicas e por isso apenas existe uma linha retornada nessa tabela.



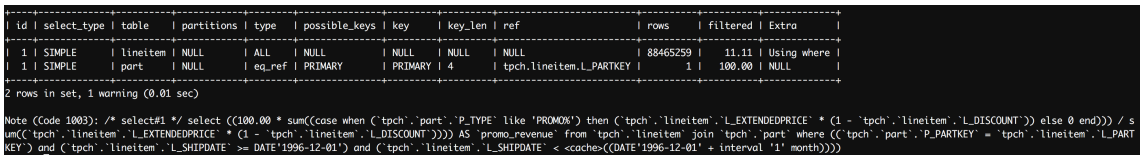
```
+----+-------------+----------+------------+--------+---------------------+---------+---------+-------------------------------------------------+----------+----------+--------------------------------+
| id | select_type | table    | partitions | type   | possible_keys       | key     | key_len | ref                                             | rows     | filtered | Extra                          |
+----+-------------+----------+------------+--------+---------------------+---------+---------+-------------------------------------------------+----------+----------+--------------------------------+
|  1 | SIMPLE      | lineitem | NULL       | ALL    | NULL                | NULL    | NULL    | NULL                                            | 88465259 |   100.00 | Using temporary; Using filesort |
|  1 | SIMPLE      | supplier | NULL       | eq_ref | PRIMARY             | PRIMARY | 4       | tpch.lineitem.L_SUPPKEY                         |        1 |   100.00 | NULL                           |
|  1 | SIMPLE      | nation   | NULL       | eq_ref | PRIMARY             | PRIMARY | 4       | tpch.supplier.S_NATIONKEY                       |        1 |   100.00 | NULL                           |
|  1 | SIMPLE      | part     | NULL       | eq_ref | PRIMARY             | PRIMARY | 4       | tpch.lineitem.L_PARTKEY                         |        1 |    11.11 | Using where                    |
|  1 | SIMPLE      | partsupp | NULL       | eq_ref | PRIMARY,PARTSUPP_FK1 | PRIMARY | 8       | tpch.lineitem.L_PARTKEY,tpch.lineitem.L_SUPPKEY |        1 |   100.00 | NULL                           |
|  1 | SIMPLE      | orders   | NULL       | eq_ref | PRIMARY             | PRIMARY | 4       | tpch.lineitem.L_ORDERKEY                        |        1 |   100.00 | NULL                           |
+----+-------------+----------+------------+--------+---------------------+---------+---------+-------------------------------------------------+----------+----------+--------------------------------+
6 rows in set, 1 warning (0.01 sec)

Note (Code 1003): /* select#1 */ select `tpch`.`nation`.`N_NAME` AS `nation`,extract(year from `tpch`.`orders`.`O_ORDERDATE`) AS `o_year`,sum(((`tpch`.`lineitem`.`L_EXTENDEDPRICE` * (1 - `tpch`.`lineitem`
.`L_DISCOUNT`)) - (`tpch`.`partsupp`.`PS_SUPPLYCOST` * `tpch`.`lineitem`.`L_QUANTITY`))) AS `sum_profit` from `tpch`.`part` join `tpch`.`supplier` join `tpch`.`lineitem` join `tpch`.`partsupp` join `tpch`
.`orders` join `tpch`.`nation` where ((`tpch`.`nation`.`N_NATIONKEY` = `tpch`.`supplier`.`S_NATIONKEY`) and (`tpch`.`orders`.`O_ORDERKEY` = `tpch`.`lineitem`.`L_ORDERKEY`) and (`tpch`.`part`.`P_PARTKEY` =
 `tpch`.`lineitem`.`L_PARTKEY`) and (`tpch`.`partsupp`.`PS_PARTKEY` = `tpch`.`lineitem`.`L_PARTKEY`) and (`tpch`.`supplier`.`S_SUPPKEY` = `tpch`.`lineitem`.`L_SUPPKEY`) and (`tpch`.`partsupp`.`PS_SUPPKEY`
 = `tpch`.`lineitem`.`L_SUPPKEY`) and (`tpch`.`part`.`P_NAME` like '%dim%')) group by `tpch`.`nation`.`N_NAME`,`o_year` desc order by `tpch`.`nation`.`N_NAME`,`o_year` desc
```

Figura 8: plano de execução Q9

Como era de esperar, o facto de na **Q9** serem acedidas e comparadas muitas tabelas com a Lineitem, faz com que o tempo da Q9 seja maior que o da Q12.

7.3.1.1.2 Q14 vs Q13

No plano da **Q14**, podemos ver que são acedidas duas tabelas: a Lineitem e a Part. A pesquisa na tabela Lineitem é sequencial e não usa indexes, daí fazer sentido que o número de linhas estimados para sere analisadas seja semelhante ao número de linhas da tabela: 88465259. Na tabela Part, a pesquisa não é sequencial, sendo usado o index da chave primária(p_partkey). Embora estejam a ser feitas pesquisas na tabela Lineitem o tempo de execução não foi muito grande, isto pode-se dever ao facto de as pesquisas nas restrições WHERE serem simples e a pesquisa sequencial.



```
+----+-------------+----------+------------+--------+---------------+---------+---------+-------------------------+----------+----------+-------------+
| id | select_type | table    | partitions | type   | possible_keys | key     | key_len | ref                     | rows     | filtered | Extra       |
+----+-------------+----------+------------+--------+---------------+---------+---------+-------------------------+----------+----------+-------------+
|  1 | SIMPLE      | lineitem | NULL       | ALL    | NULL          | NULL    | NULL    | NULL                    | 88465259 |    11.11 | Using where |
|  1 | SIMPLE      | part     | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | tpch.lineitem.L_PARTKEY |        1 |   100.00 | NULL        |
+----+-------------+----------+------------+--------+---------------+---------+---------+-------------------------+----------+----------+-------------+
2 rows in set, 1 warning (0.01 sec)

Note (Code 1003): /* select#1 */ select ((100.00 * sum((case when (`tpch`.`part`.`P_TYPE` like 'PROMO%') then (`tpch`.`lineitem`.`L_EXTENDEDPRICE` * (1 - `tpch`.`lineitem`.`L_DISCOUNT`)) else 0 end))) / sum((`tpch`.`lineitem`.`L_EXTENDEDPRICE` * (1 - `tpch`.`lineitem`.`L_DISCOUNT`)))) AS `promo_revenue` from `tpch`.`lineitem` join `tpch`.`part` where ((`tpch`.`part`.`P_PARTKEY` = `tpch`.`lineitem`.`L_PARTKEY`) and (`tpch`.`lineitem`.`L_SHIPDATE` >= DATE'1996-12-01') and (`tpch`.`lineitem`.`L_SHIPDATE` < <cache>((DATE'1996-12-01' + interval '1' month))))
```

Figura 9: plano de execução Q14

No plano da **Q13**, observamos que foram acedidas 3 tabelas: uma tabela derivada de uma subquery de junção das tabelas Customer e Orders e naturalmente a tabela Customer e tabela Orders, que são lidas na subquery anterior. Estas tabelas sempre são mais pequenas que a Lineitem, mesmo assim a tabela da junção de ambas ficou com muitas linhas: 49450114425088, este número tem lógica já que é o resultado da multiplicação do número de linhas estimadas para as tabelas Customers e Orders(2212288 * 22353476 = 49452326713088). Os valores das ROWS são parecidos aos da tabela Customer e Orders. Na pesquisa da tabela Customer é usado index da chave primária, o que podemos ver pela coluna 'key', o que faz sentido já que este campo é usado nas condições da subquery.

```
+----+-------------+-------------+------------+-------+---------------+---------+---------+------+----------------+----------+----------------------------------------------+
| id | select_type | table       | partitions | type  | possible_keys | key     | key_len | ref  | rows           | filtered | Extra                                        |
+----+-------------+-------------+------------+-------+---------------+---------+---------+------+----------------+----------+----------------------------------------------+
|  1 | PRIMARY     | <derived2>  | NULL       | ALL   | NULL          | NULL    | NULL    | NULL | 49450114425088 |   100.00 | Using temporary; Using filesort              |
|  2 | DERIVED     | customer    | NULL       | index | PRIMARY       | PRIMARY | 4       | NULL |        2212288 |   100.00 | Using index; Using temporary; Using filesort |
|  2 | DERIVED     | orders      | NULL       | ALL   | NULL          | NULL    | NULL    | NULL |       22352476 |   100.00 | Using where; Using join buffer (Block Nested Loop) |
+----+-------------+-------------+------------+-------+---------------+---------+---------+------+----------------+----------+----------------------------------------------+
3 rows in set, 1 warning (0.01 sec)

Note (Code 1003): /* select#1 */ select `c_orders`.`c_count` AS `c_count`,count(0) AS `custdist` from (/* select#2 */ select `tpch`.`customer`.`C_CUSTKEY` AS `c_custkey`,count(`tpch`.`orders`.`O_ORDERKEY`) AS `c_count` from `tpch`.`customer` left join `tpch`.`orders` on(((`tpch`.`orders`.`O_CUSTKEY` = `tpch`.`customer`.`C_CUSTKEY`) and (not((`tpch`.`orders`.`O_COMMENT` like '%pending%deposits%')))))) where 1 group by `tpch`.`customer`.`C_CUSTKEY`) `c_orders` group by `c_orders`.`c_count` order by `custdist` desc,`c_orders`.`c_count` desc
```

Figura 10: plano de execução Q13

Como podíamos prever pelo número de linhas estimadas em cada tabela no plano de execução, a **Q13** apresenta maior tempo de execução que a **Q14**.

7.3.1.2 PostgreSQL

No caso do postgreSQL, os tempos de execução variam menos, podemos ver que as queries mais lentas são Q11, Q16 e Q22 e as mais rápidas são Q18, Q1 e Q21. Ao analisar o custo final em cada plano já dá para ter uma ideia das queries mais rápidas e mais lentas, embora estes custos sejam apenas previsões já que a querie não chegou a correr. No geral, aquelas com um maior custo serão mais lentas e as com menor custo serão mais rápidas.
De maneira a perceber com mais detalhes as variações nos tempos de execução vai ser feita uma análise da **Q11 vs Q18**, da **Q16 vs Q1** e da **Q22 vs Q17**.

7.3.1.2.1 Q11 vs Q18

Vendo o plano de execução podemos ver que no caso da **Q11** esta executa várias operações(select, where, group by, order by) sendo que o custo acaba por ser repartido igualmente nas áreas assinaladas do plano. Nestas áreas os tempos são semelhantes o que seria de esperar já que o código respetivo é semelhante. Os custos em si são baixos o que pode ser justificado por terem sido feitos maioritariamente acessos sequenciais (sem uso de indexes) em tabelas mais pequenas: Nation, Supplier e Partsupp.

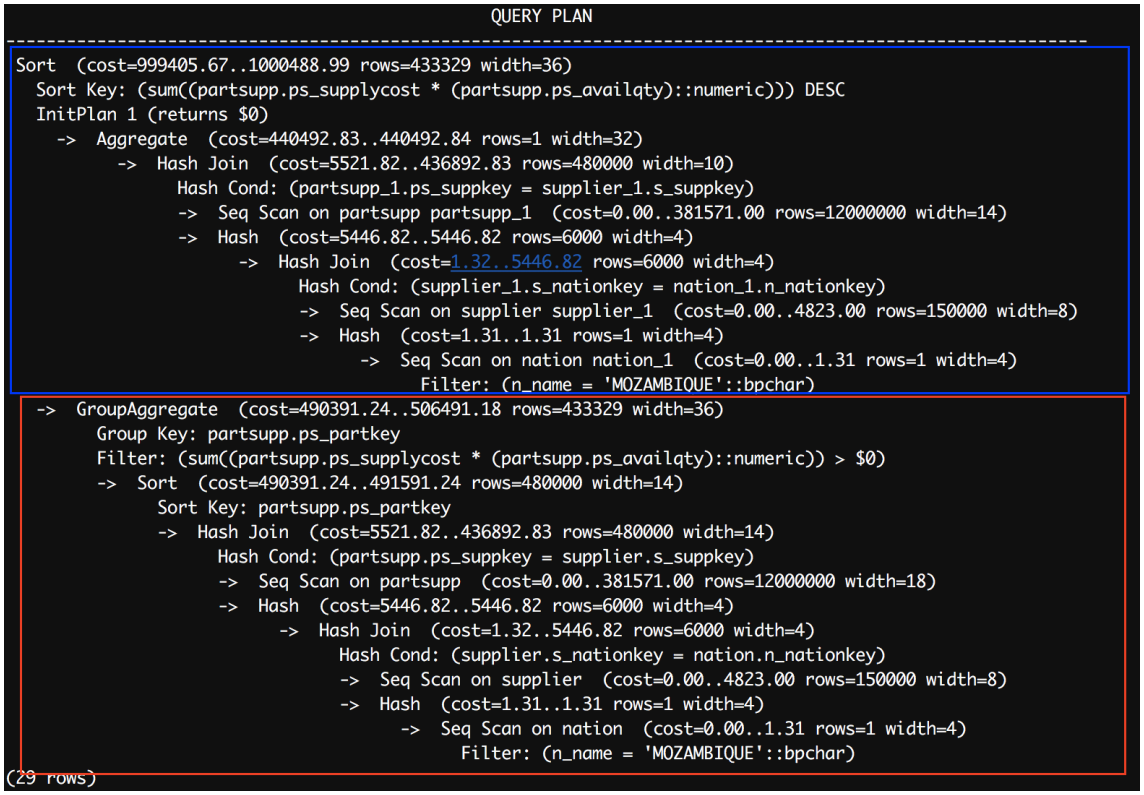


Figura 11: plano de execução Q11

Na **Q18** continuam a ser usados vários tipos de operações, no plano de execução podemos ver duas zonas destacadas que contribuíram muito para o valor do custo

final. A vermelho podemos ver a subquery, nesta é corrida sequencialmente a tabela Lineitem para realizar o *sum(l_quantity) >314* e é feito sort pela lineitem.orderkey o que justifica o elevado custo. Na zona a azul podemos ver outro *sequencial scan* à tabela Lineitem de maneira a poder ser feito a operação *o_orderkey = l_orderkey*, o que também justifica o custo significativo associado.

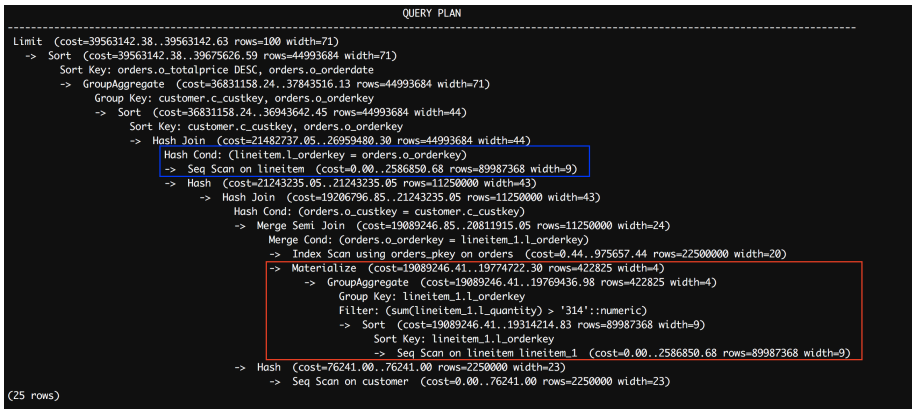


Figura 12: plano de execução Q18

Como podemos concluir, ambas as **Q11** e **Q18** executam diversas operações não sendo esse o prinicipal motivo da diferença nos tempos de execução, o principal motivo será o facto de na **Q18** serem feitos varrimentos a tabelas significativamente maiores como a Lineitem e a Orders o que terá impacto no tempo e custo final.

7.3.1.2.2 Q16 vs Q1

Mais uma vez na **Q16** obtivemos valores de execução pequenos, analisando o plano de execução podemos perceber que as zonas destacadas tem um papel significativo no custo total. Na zona a vermelho podemos ver um *Hash Join* que faz join dos registos da Hash(que são o resultado da cláusula WHERE que envolve a tabela Part) com os registos resultantes do *Seq scan* na tabela Supplier. Na zona a azul podemos ver um varrimento da tabela Partsupp, que envolve uma comparação(Filter) com a tabela Part e também uma comparação(Filter) com uma subquery. Nesta tabela os acessos são feitos a tabelas pequenas: Part, Partsupp, Supplier.

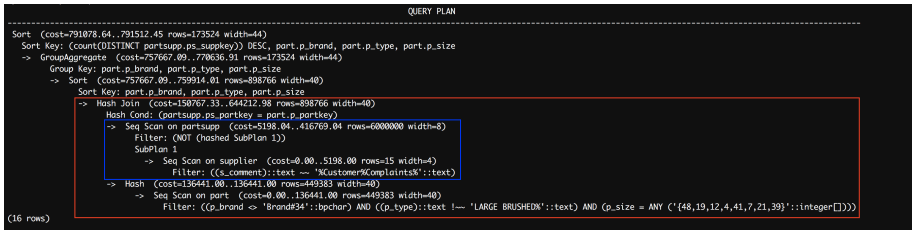


Figura 13: plano de execução Q16

No plano da **Q1**, podemos observar que o plano é mais simples de perceber, a operação que mais custo produz é o GROUP BY por l_returnflag e l_linestatus

(área vermelha do plano). Embora esta querie execute operações relativamente simples como esta a ler a tabela Lineitem o seu tempo de execução vai aumentar significativamente, nomeadamente na operação de group by.

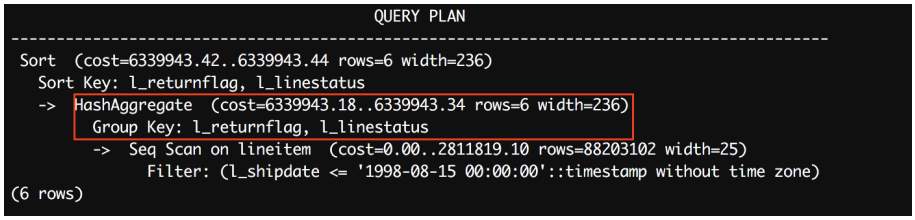


Figura 14: plano de execução Q1

Em conclusão podemos ver que embora a **Q1** seja mais simples em termos de SQL que a **Q16**, esta possuí um tempo de execução muito mais elevado que a **Q16**, justamente por ler e fazer sort na tabela Lineitem que é muito maior que as tabelas acedidas nas **Q16**.

7.3.1.2.3 Q22 vs Q17

Analisando o plano de execução podemos ver que no caso da **Q22** a query é relativamente complexa, analisando os valores dos custos previstos conseguimos concluir que a zona a vermelho tem grande importância no custo total, isto deve-se a ser feito um varrimento da tabela orders, que é a segunda maior tabela do tpch, como se trata apenas de um varrimento simples, isto é sem operações e filters complexos associados, e sequencial o tempo de execução não será muito grande.

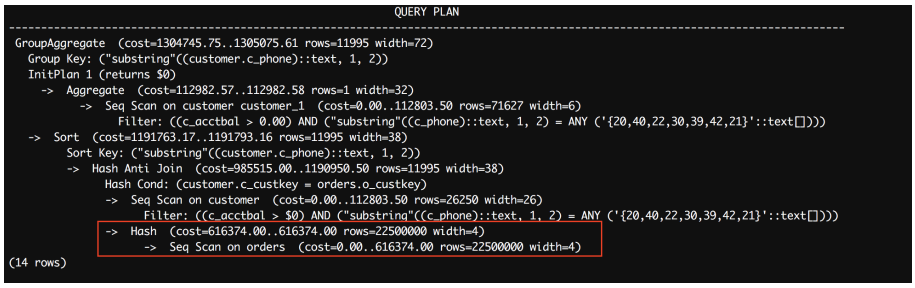


Figura 15: plano de execução Q22

Vendo agora o plano da **Q17**, percebemos que a querie não é muito complexa. O Hash Join, na zona vermelha do plano, é o grande responsável pelo custo estimado, este faz join dos registos resultantes da leitura sequencial da tabela Lineitem com os registos resultantes da leitura sequencial e comparação(filters) da tabela Part. Mais especificamente, nesse join o Join Filter(zona azul) irá ter grande influência já que faz com que em cada linha da tabela Lineitem corra o subplan(que irá correr a tabela Lineitem sequencialmente) para depois comparar a l_quantity, este é a principal razão para a querie não correr em tempo útil.

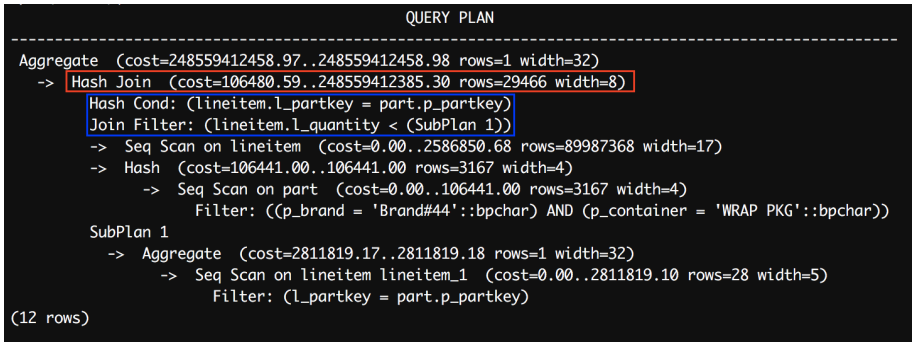


Figura 16: plano de execução Q17

Concluindo embora a **Q22** seja mais complexa esta irá executar operações não muito dispendiosas em termos computacionais(são feitas leituras simples) enquanto que na **Q17** embora também sejam feitas leituras simples, estas são feitas na tabela Lineitem e são feitas muitas vezes.

7.3.2 Pergunta 2 - Análise de queries entre SGDB

Porque e como é que os tempos de execução das queries variam entre SGBDs testadas? Em particular, o que poderá fazer um SGBD mais eficiente do que o outro?

7.3.2.1 Q2 - MySQL vs Q2 - PostgreSQL

No MySQL a **Q2** executa em 11.64s e 19.68s e em PostgreSQL em 22.32s e 22.54s. Observando os planos de execução conseguimos perceber que grande parte das comparações feitas nos WHERE utilizam os indexes das chaves primárias, juntando isto ao facto das tabelas acedidas serem de tamanhos pequenos consegue-se concluir que as queries vão ser executadas em pouco tempo. Como os tempos de execução são muito baixos estes podem variar de execução em execução, sendo por isso que os valores obtidos podem não corresponder totalmente ao tempo útil de processamento já que algum do tempo foi utilizado pela BD como por exemplo para warm-up.

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	region	NULL	ALL	PRIMARY	NULL	NULL	NULL	5	20.00	Using where; Using temporary; Using filesort
1	PRIMARY	nation	NULL	ALL	PRIMARY	NULL	NULL	NULL	25	10.00	Using where; Using join buffer (Block Nested Loop)
1	PRIMARY	part	NULL	ALL	PRIMARY	NULL	NULL	NULL	2971228	1.11	Using where; Using join buffer (Block Nested Loop)
1	PRIMARY	partsupp	NULL	ref	PRIMARY,PARTSUPP_FK1	PRIMARY	4	tpch.part.P_PARTKEY	4	100.00	Using where
1	PRIMARY	supplier	NULL	eq_ref	PRIMARY	PRIMARY	4	tpch.partsupp.PS_SUPPKEY	1	10.00	Using where
2	DEPENDENT SUBQUERY	region	NULL	ALL	PRIMARY	NULL	NULL	NULL	5	20.00	Using where
2	DEPENDENT SUBQUERY	partsupp	NULL	ref	PRIMARY,PARTSUPP_FK1	PRIMARY	4	tpch.part.P_PARTKEY	4	100.00	NULL
2	DEPENDENT SUBQUERY	supplier	NULL	eq_ref	PRIMARY	PRIMARY	4	tpch.partsupp.PS_SUPPKEY	1	100.00	NULL
2	DEPENDENT SUBQUERY	nation	NULL	eq_ref	PRIMARY	PRIMARY	4	tpch.supplier.S_NATIONKEY	1	10.00	Using where

9 rows in set, 2 warnings (0.06 sec)

Figura 17: plano de execução Q2 - MySQL

```
                                   QUERY PLAN
------------------------------------------------------------------------------------------------
 Limit  (cost=4695576.66..4695576.66 rows=1 width=271)
   ->  Sort  (cost=4695576.66..4695576.66 rows=1 width=271)
         Sort Key: supplier.s_acctbal DESC, nation.n_name, supplier.s_name, part.p_partkey
         ->  Merge Join  (cost=1404376.56..4695576.65 rows=1 width=271)
               Merge Cond: (part.p_partkey = partsupp.ps_partkey)
               Join Filter: (partsupp.ps_supplycost = (SubPlan 1))
               ->  Index Scan using part_pkey on part  (cost=0.43..154356.43 rows=13111 width=30)
                     Filter: (((p_type)::text ~~ '%STEEL'::text) AND (p_size = 30))
               ->  Materialize  (cost=1404375.23..1416375.23 rows=2400000 width=251)
                     ->  Sort  (cost=1404375.23..1410375.23 rows=2400000 width=251)
                           Sort Key: partsupp.ps_partkey
                           ->  Hash Join  (cost=7059.99..575815.99 rows=2400000 width=251)
                                 Hash Cond: (partsupp.ps_suppkey = supplier.s_suppkey)
                                 ->  Seq Scan on partsupp  (cost=0.00..381571.00 rows=12000000 width=14)
                                 ->  Hash  (cost=5687.99..5687.99 rows=30000 width=245)
                                       ->  Hash Join  (cost=2.49..5687.99 rows=30000 width=245)
                                             Hash Cond: (supplier.s_nationkey = nation.n_nationkey)
                                             ->  Seq Scan on supplier  (cost=0.00..4823.00 rows=150000 width=145)
                                             ->  Hash  (cost=2.43..2.43 rows=5 width=108)
                                                   ->  Hash Join  (cost=1.07..2.43 rows=5 width=108)
                                                         Hash Cond: (nation.n_regionkey = region.r_regionkey)
                                                         ->  Seq Scan on nation  (cost=0.00..1.25 rows=25 width=112)
                                                         ->  Hash  (cost=1.06..1.06 rows=1 width=4)
                                                               ->  Seq Scan on region  (cost=0.00..1.06 rows=1 width=4)
                                                                     Filter: (r_name = 'ASIA'::bpchar)
               SubPlan 1
                 ->  Aggregate  (cost=297.32..297.33 rows=1 width=32)
                       ->  Nested Loop  (cost=1.93..297.30 rows=6 width=6)
                             Join Filter: (supplier_1.s_nationkey = nation_1.n_nationkey)
                             ->  Hash Join  (cost=1.07..2.43 rows=5 width=4)
                                   Hash Cond: (nation_1.n_regionkey = region_1.r_regionkey)
                                   ->  Seq Scan on nation nation_1  (cost=0.00..1.25 rows=25 width=8)
                                   ->  Hash  (cost=1.06..1.06 rows=1 width=4)
                                         ->  Seq Scan on region region_1  (cost=0.00..1.06 rows=1 width=4)
                                               Filter: (r_name = 'ASIA'::bpchar)
                             ->  Materialize  (cost=0.85..292.84 rows=28 width=10)
                                   ->  Nested Loop  (cost=0.85..292.70 rows=28 width=10)
                                         ->  Index Scan using partsupp_pkey on partsupp partsupp_1  (cost=0.43..56.18 rows=28 width=10)
                                               Index Cond: (part.p_partkey = ps_partkey)
                                         ->  Index Scan using supplier_pkey on supplier supplier_1  (cost=0.42..8.44 rows=1 width=8)
                                               Index Cond: (s_suppkey = partsupp_1.ps_suppkey)
(41 rows)
```

Figura 18: plano de execução Q2 - PostgreSQL

7.3.2.2 Q18 - MySQL vs Q18 - PostgreSQL

Para a **Q18** já vemos diferenças significativas entre os tempos de execução no MySQL(1053,52s e 1544,49s) e no PostgreSQL(215,78s e 210.94). Ao observar os planos de execução conseguimos perceber que existem pesquisas sobre tabelas como a Lineitem e a Orders(por exemplo: *c_custkey* = ***o_custkey*** e *o_orderkey* = ***l_orderkey***) que implicam que estas sejam lidas sequenciamente já que esses campos das tabelas não possuem indexes. Pode não ser fácil através dos planos de execução perceber por que razão o PostgreSQL é mais rápido que o MySQL, no entanto uma das razões encontradas para tal poderá ser o facto de, segundo apurado na Internet(ver referências), o MySQL apresentar grandes limitações em operar com subqueries o que dá uma grande vantagem ao PostgreSQL neste tipo de pesquisas. Outra razão poderá ser o facto de o PostgreSQL ser muito mais eficiente em leituras sequenciais de tabelas, já que possui ponteiros em memória para as linhas da tabela, enquanto que o MySQL é mais lento nesta operação.

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	lineitem	NULL	ALL	NULL	NULL	NULL	NULL	88465259	100.00	Using where; Using temporary; Using filesort
1	PRIMARY	orders	NULL	eq_ref	PRIMARY	PRIMARY	4	tpch.lineitem.L_ORDERKEY	1	100.00	NULL
1	PRIMARY	customer	NULL	eq_ref	PRIMARY	PRIMARY	4	tpch.orders.O_CUSTKEY	1	100.00	NULL
2	SUBQUERY	lineitem	NULL	ALL	NULL	NULL	NULL	NULL	88465259	100.00	Using temporary; Using filesort

4 rows in set, 1 warning (0.01 sec)

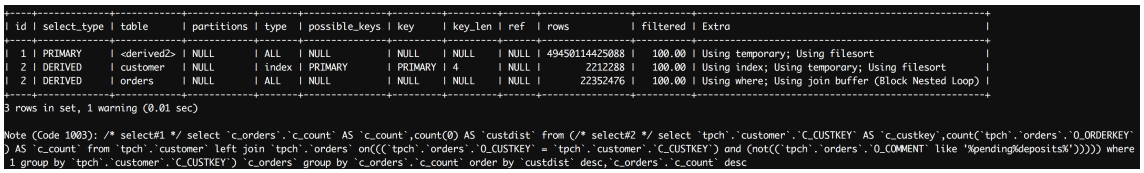
Figura 19: plano de execução Q18 - MySQL

```
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=39563142.38..39563142.63 rows=100 width=71)
   ->  Sort  (cost=39563142.38..39675626.59 rows=44993684 width=71)
         Sort Key: orders.o_totalprice DESC, orders.o_orderdate
         ->  GroupAggregate  (cost=36831158.24..37843516.13 rows=44993684 width=71)
               Group Key: customer.c_custkey, orders.o_orderkey
               ->  Sort  (cost=36831158.24..36943642.45 rows=44993684 width=44)
                     Sort Key: customer.c_custkey, orders.o_orderkey
                     ->  Hash Join  (cost=21482737.05..26959480.30 rows=44993684 width=44)
                           Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
                           ->  Seq Scan on lineitem  (cost=0.00..2586850.68 rows=89987368 width=9)
                           ->  Hash  (cost=21243235.05..21243235.05 rows=11250000 width=43)
                                 ->  Hash Join  (cost=19206796.85..21243235.05 rows=11250000 width=43)
                                       Hash Cond: (orders.o_custkey = customer.c_custkey)
                                       ->  Merge Semi Join  (cost=19089246.85..20811915.05 rows=11250000 width=24)
                                             Merge Cond: (orders.o_orderkey = lineitem_1.l_orderkey)
                                             ->  Index Scan using orders_pkey on orders  (cost=0.44..975657.44 rows=22500000 width=20)
                                             ->  Materialize  (cost=19089246.41..19774722.30 rows=422825 width=4)
                                                   ->  GroupAggregate  (cost=19089246.41..19769436.98 rows=422825 width=4)
                                                         Group Key: lineitem_1.l_orderkey
                                                         Filter: (sum(lineitem_1.l_quantity) > '314'::numeric)
                                                         ->  Sort  (cost=19089246.41..19314214.83 rows=89987368 width=9)
                                                               Sort Key: lineitem_1.l_orderkey
                                                               ->  Seq Scan on lineitem lineitem_1  (cost=0.00..2586850.68 rows=89987368 width=9)
                                       ->  Hash  (cost=76241.00..76241.00 rows=2250000 width=23)
                                             ->  Seq Scan on customer  (cost=0.00..76241.00 rows=2250000 width=23)
(25 rows)
```

Figura 20: plano de execução Q18 - PostgreSQL

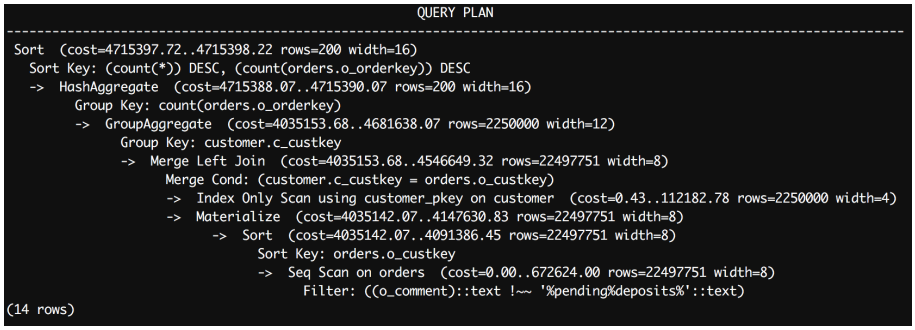
7.3.2.3 Q13 - MySQL vs Q13 - PostgreSQL

Na **Q13** também é notório diferenças entre os tempos de execução no MySQL(1505,95s e 6200,88s) e no PostgreSQL(59,49s e 42,03s). Na query SQL é possível feito um JOIN dentro de uma subquerie no FROM, esta operação também é visível nos planos de execução: no MySQL é possível ver que a tabela de primeira linha corresponde à tabela gerada pela subquery já que esta é *<derived2>* e o número de linhas previsto para esta tabela é muito elevado(49450114425088 linhas) precisamente por resultar do JOIN das tabelas Customer e Orders(2212288*22352476 linhas). No PostgreSQL conseguimos ver a presença da operação de JOIN pela presença no plano de *Merge Left Join*. Ao pesquisar na Internet(ver referências) e perante o número de linhas envolvidos na operação de JOIN(embora seja complicado comparar as linhas de um plano com outro) concluímos que o MySQL é limitado na realização de JOIN's e também de subqueries, sendo estas as duas razões pensadas para a diferença entre os tempos de execução da querie em MySQL e PostgreSQL



```
+----+-------------+------------+------------+------+---------------+---------+---------+------+----------------+----------+----------------------------------------------+
| id | select_type | table      | partitions | type | possible_keys | key     | key_len | ref  | rows           | filtered | Extra                                        |
+----+-------------+------------+------------+------+---------------+---------+---------+------+----------------+----------+----------------------------------------------+
|  1 | PRIMARY     | <derived2> | NULL       | ALL  | NULL          | NULL    | NULL    | NULL | 49450114425088 |   100.00 | Using temporary; Using filesort              |
|  2 | DERIVED     | customer   | NULL       | index | PRIMARY      | PRIMARY | 4       | NULL |        2212288 |   100.00 | Using index; Using temporary; Using filesort |
|  2 | DERIVED     | orders     | NULL       | ALL  | NULL          | NULL    | NULL    | NULL |       22352476 |   100.00 | Using where; Using join buffer (Block Nested Loop) |
+----+-------------+------------+------------+------+---------------+---------+---------+------+----------------+----------+----------------------------------------------+
3 rows in set, 1 warning (0.01 sec)

Note (Code 1003): /* select#1 */ select `c_orders`.`c_count` AS `c_count`,count(0) AS `custdist` from (/* select#2 */ select `tpch`.`customer`.`C_CUSTKEY` AS `c_custkey`,count(`tpch`.`orders`.`O_ORDERKEY`
) AS `c_count` from `tpch`.`customer` left join `tpch`.`orders` on((`tpch`.`orders`.`O_CUSTKEY` = `tpch`.`customer`.`C_CUSTKEY`) and (not((`tpch`.`orders`.`O_COMMENT` like '%pending%deposits%'))))) where
 1 group by `tpch`.`customer`.`C_CUSTKEY`) `c_orders` group by `c_orders`.`c_count` order by `custdist` desc, `c_orders`.`c_count` desc
```

Figura 21: plano de execução Q13 - MySQL



```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=4715397.72..4715398.22 rows=200 width=16)
   Sort Key: (count(*)) DESC, (count(orders.o_orderkey)) DESC
   ->  HashAggregate  (cost=4715388.07..4715390.07 rows=200 width=16)
         Group Key: count(orders.o_orderkey)
         ->  GroupAggregate  (cost=4035153.68..4681638.07 rows=2250000 width=12)
               Group Key: customer.c_custkey
               ->  Merge Left Join  (cost=4035153.68..4546649.32 rows=22497751 width=8)
                     Merge Cond: (customer.c_custkey = orders.o_custkey)
                     ->  Index Only Scan using customer_pkey on customer  (cost=0.43..112182.78 rows=2250000 width=4)
                     ->  Materialize  (cost=4035142.07..4147630.83 rows=22497751 width=8)
                           ->  Sort  (cost=4035142.07..4091386.45 rows=22497751 width=8)
                                 Sort Key: orders.o_custkey
                                 ->  Seq Scan on orders  (cost=0.00..672624.00 rows=22497751 width=8)
                                       Filter: ((o_comment)::text !~~ '%pending%deposits%'::text)
(14 rows)
```

Figura 22: plano de execução Q13 - PostgreSQL

7.4 MongoDB

De modo a poderem ser feitas comparações entre motores de busca relacionais e NoSQL criámos 4 queries semelhantes em SQL e NoSQL, e corremos e registámos os tempos de execução para cada um dos três motores.

7.4.1 Q1

SQL

```
SELECT * FROM orders
WHERE O_ORDERSTATUS = "O"
ORDER BY O_TOTALPRICE ASC
LIMIT 100000
```

NoSQL

```
db.orders.find(
{ orderstatus: "O" } ).sort(
{ totalprice: 1 } ).limit(100000)
```

	MySQL (s)	**PostgreSQL (s)**	**MongoDB (s)**
Q1 - **PC1**	61,85	50,67	87,03
Q1 - **PC2**	87,32	42,00	98,05

Tabela 9: Tempos de execução Q1

Como podemos ver pelos tempos obtidos, o MySQL e o PostgreSQL acabam por obter melhores tempos que o MongoDB, embora os três valores estejam muito perto uns dos outros. Isto deve-se ao facto de só serem devolvidos poucas linhas(100000) o que não evidencia com clareza o motor mais eficiente.

7.4.2 Q2

SQL

```
SELECT SUM(l_quantity)
FROM lineitem
GROUP BY l_suppkey;
```

NoSQL

```
db.lineitem.aggregate([
{ $group: { _id: "$suppkey",
     total: { $sum: "$quantity" } } }
], { allowDiskUse: true })
```

	MySQL (s)	**PostgreSQL (s)**	**MongoDB (s)**
Q2 - **PC1**	179,87	169,33	545,12
Q2 - **PC2**	252,06	89,15	460,03

Tabela 10: Tempos de execução Q2

Analizando os valores percebemos que mais uma vez o MongoDB é mais lento que os restantes motores, neste caso em particular isto pode dever-se ao facto de, por defeito, o MongoDB apenas usar 100MB de RAM na operação de *group*. Nos casos que necessita de mais RAM, como é o caso desta querie, é necessário adicionar a opção *'allowDiskUse: true'*, esta opção permite que a querie seja executada escrevendo para ficheiros temporários, o uso destes ficheiros temporários poderá acrescentar overhead e aumentar o tempo de execução da querie. No motores de base de dados relacionais, como o MySQL e PostgreSQL, não existe esta limitação em termos de RAM.

7.4.3 Q3

SQL

```
SELECT * FROM lineitem
WHERE quantity < 24 AND
discount between 0.06-0.01
AND 0.06+0.01
```

NoSQL

```
db.lineitem.find(
    { quantity:
        { "$lt": 24},
     discount:
        { $gt: 0.06-0.01, $lte: 0.06+0.01 }
    })
```

	MySQL (s)	**PostgreSQL (s)**	**MongoDB (s)**
Q3 - **PC1**	136,34	57,93	356,29
Q3 - **PC2**	198,71	97,41	410,91

Tabela 11: Tempos de execução Q3

Com base nos valores apresentados podemos perceber que o PostgreSQL é o motor mais rápido, seguido pelo MySQL. O MongoDB é o que demora mais tempo, a razão para isto acontecer pode ser o facto de o MongoDB não ter tipos de dados pré definidos, isto implica que em cada registo tenha que processar cada atributo para perceber se se trata de um número ou uma String, este processo ao fim de muitas linhas gera um overhead significativo. Nesta querie em particular são feitas muitas comparações entre números o que comprova o que foi dito acima.

7.4.4 Q4

SQL

```
SELECT l_linestatus, COUNT(*),
AVG(l_discount) FROM lineitem
GROUP BY l_linestatus
ORDER BY COUNT(*) DESC
LIMIT 10
```

NoSQL

```
db.lineitem.aggregate(
   [{ $group : {
           _id : "$linestatus",
           count: { $sum: 1 },
           avg: { $avg: "$discount" }
         } },
      { $sort: { count: -1 } },
      { $limit : 10 } ])
```

	MySQL (s)	**PostgreSQL (s)**	**MongoDB (s)**
Q4 - **PC1**	120,19	52,26	479,12
Q4 - **PC2**	141,89	68,11	516,66

Tabela 12: Tempos de execução Q4

Das queries criadas esta é a mais complexa em termos de código. Pelos tempos observados vemos novamente que o MongoDB apresenta tempos muito piores que os restantes motores, mais uma vez a razão pode ser o facto de a operação de *group* ser limitada em termos de RAM e necessitar de escrever em ficheiros temporários para ultrapassar essa limitação, podendo causar overhead extra. Outra possivel razão poderá ser o facto de o PostgreSQL e o MySQL serem muito mais otimizados que o MongoDB, o que faz com que consiga realizar certas pesquisas mais rapidamente.

7.4.5 Conclusão - MongoDB

Nas queries apresentadas ficou claro que o MongoDB apresentou piores resultados, mas não podemos assumir que isto se irá passar em todos os casos. Como referido inicialmente, na inicialização da BD de MongoDB foram criadas collections isoladas para cada tabela, tendo no final ficado com o mesmo número de collections/documets e tabelas/linhas. Esta abordagem pode não ter beneficiado 100% o MongoDB já que o limita em alguns aspetos, caso tivesse optado por uma abordagem com nested/embedded Documents, em que por exemplo em cada documento Order estaria incluído os documentos Lineitem respetivos dessa Order, aumentaria a performance em certos casos como em queries com Joins de tabelas, por outro lado a leitura de apenas Lineitem poderia ter um custo maior já que os documents Lineitem estariam mais "fundos" pois teríamos primeiro que aceder às Orders, neste caso.
Ao usar o MongoDB, o essencial será fazer o armazenamento/estrutura correto das collections de forma a que as queries benefeciem dessa estrutura, otimizando as suas pesquisas.

8 Comparação de Resultados entre PC1 e PC2

Ao analisar de forma geral os tempos de execução das queries e do carregamento de dados para os SGDB's entre o **PC1** e **PC2** podemos concluir que embora os valores não sejam iguais, estes que variam pouco, tendo valores semelhantes na maioria das queries e na maioria dos cerregamentos de dados, tanto com o MySQL, como PostgreSQL e MongoDB. Este comportamento seria de esperar visto que ambos os PC's possuem SSD e estão a correr 15GB de dados.

9 Casos especiais

Com o objetivo de melhorar a performance e rapidez nas pesquisas vamos agora testar possíveis otimizações.

9.1 Index

Por default, são criados apenas indexes para as chaves primárias, no entanto, criar indexes sobre outros campos das BD pode aumentar o acesso aos registos e por isso aumentar a rapidez de execução das queries. Por outro lado, nem sempre compensa o uso de indexes já que estes podem aumentar o tempo de execução, isto na prática não deverá acontecer já que o motor de busca consegue prever e impedir esses casos.

A **Q17** podemos ver que possuí elevados tempos de execução para MySQL e PostgreSQL, superior a 2 horas, e ao analisar os comandos SQL na query vemos que esta possuí várias pesquisas WHERE sobre o campo *l_partkey* e portanto criar um index sobre este campo poderá melhor o tempo de execução da querie. Para criar o index sobre o campo *l_partkey* utilizámos o seguinte comando:

```
CREATE INDEX index_name on lineitem(l_partkey)
```

	MySQL (s)	**PostgreSQL (s)**
Q17 - Sem index	>7200	>7200
Q17 - Com index	79.18	2.52

Tabela 13: Tempos de execução com/sem index

Neste caso vemos claramente que o uso de index melhora em muito o tempo de execução da querie, mais de 90 vezes mais rápido. Ao observarmos a zona a vermelho nos planos de execução da querie antes e depois da criação do index conseguimos perceber o porquê de ser mais rápido, o custo na mesma operação passa a ser muito menor com index.

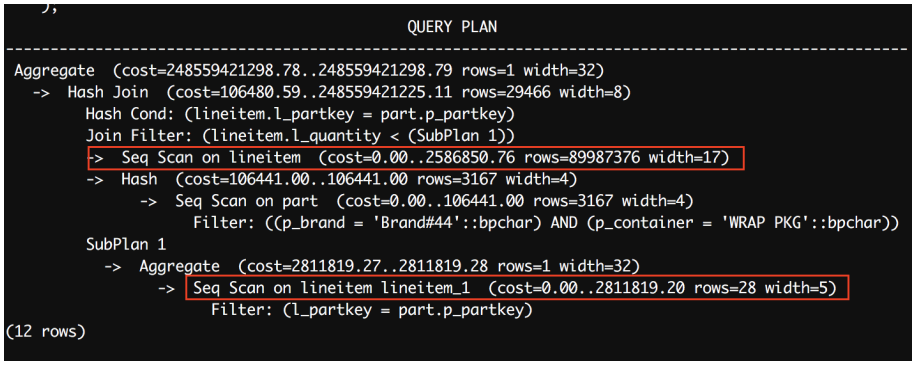


Figura 23: plano de execução Q17 sem index

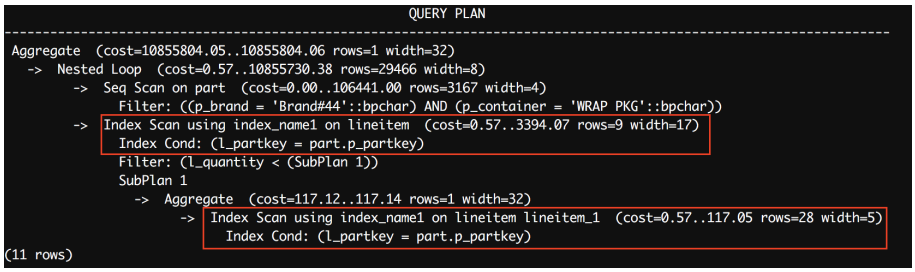


Figura 24: plano de execução Q17 com index

9.2 Buffer Cache

Outra possível solução seria aumentar o **data/buffer cache** no motor de base de dados, isto permite aumentar a quantidade de memória que o motor de busca poderá usar o que poderá, em teoria, reduzir os tempos de execução das queries.

Para testar a influência do tamanho do Data Buffer vamos testar a seguinte query com diresntes valores de data/cache nos motores MySQL e PostgreSQL:

```
select * from lineitem
 where l_quantity < ( select AVG(l_quantity) from lineitem);
```

	MySQL (s)	PostgreSQL (s)
1GB/128MB	360,3	38,4
4GB	300,2	37,1

Tabela 14: Influência Buffer Cache

Como podemos constatar não houve grande diferença ao aumentar a Buffer Cache, tanto no MySQL como no PostgreSQL, este não era o resultado esperado já que com mais memória disponível e pesquisando duas vezes a mesma tabela estaríamos à espera que alocasse mais linhas da tabela em memória e por isso reduzisse o tempo de execução. Uma possível explicação seria o facto de não haver mais espaço em memória para alocar ao motor(ver Anexo C) e por isso, embora estivesse definido um valor maior este não correspondiam ao valor real.

10 Conclusão

Neste assignment comparámos os três motores: MySQL, PostgreSQL e MongoDB em diferentes perspetivas, cada um dos motores possuí diferentes estruturas, modo de armazenamentos e técnicas de compressão diferentes o que faz com que certos motores sejam melhores em determinadas tarefas. No geral o motor PostgreSQL foi aquele que possuiu melhores tempos, embora não possamos dizer que este motor é melhor que os restantes dois, já que cada motor tem as suas vantagens e desvantagens.

11 Referências

- Performance difference between MySQL and PostgreSQL for the same schema[closed]
- Reading a Postgres EXPLAIN ANALYZE Query Plan
- MySQL - EXPLAIN SELECT
- SQL to MongoDB Mapping Chart
- TPCH-dbgen
- Choose PostgreSQL over MySQL

A Tamanho das tabelas

Na tabela em baixo é possível ver o tamanho dos ficheiros com as tabelas gerados pelo *dbgen*.

Tabela	Size(MB)
Region	0,000389
Nation	0,002
Supplier	21,3
Part	365,5
Customer	367,8
Partsupp	1810
Orders	2630
Lineitem	11 720,00

Tabela 15: Tamanho das tabelas

B Gráfico Tempo de Execução das Queries

B.1 PC1

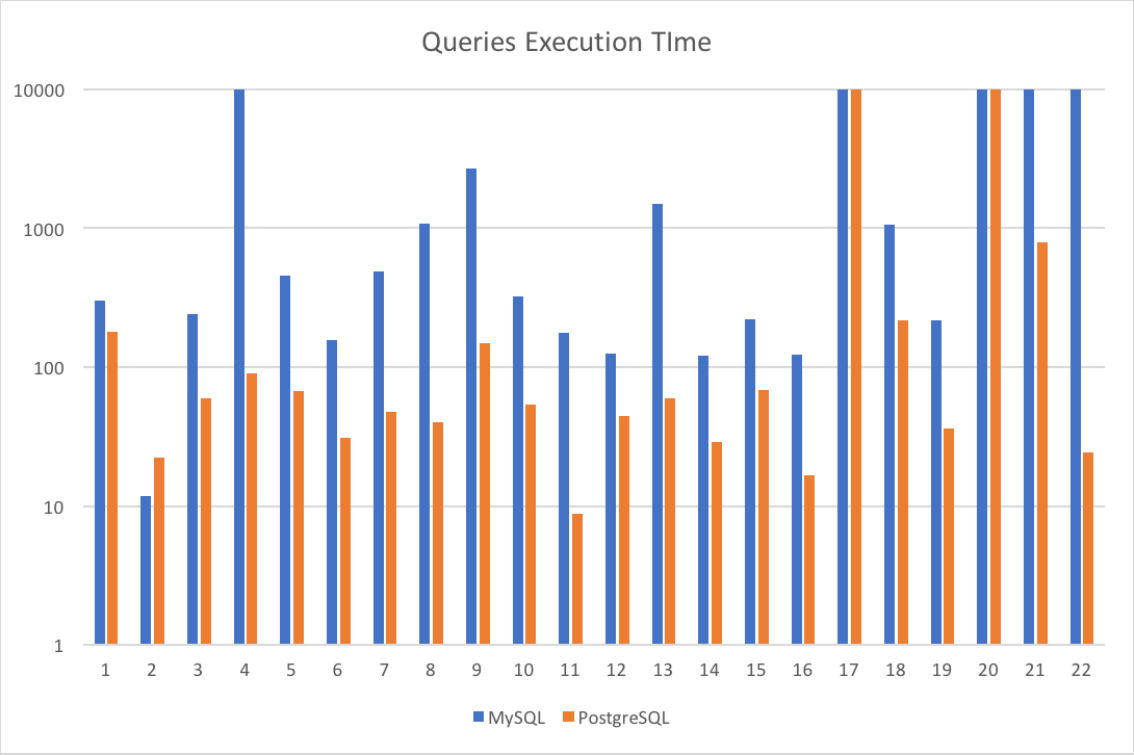


Figura 25: Tempo de execução Queries

B.2 PC2

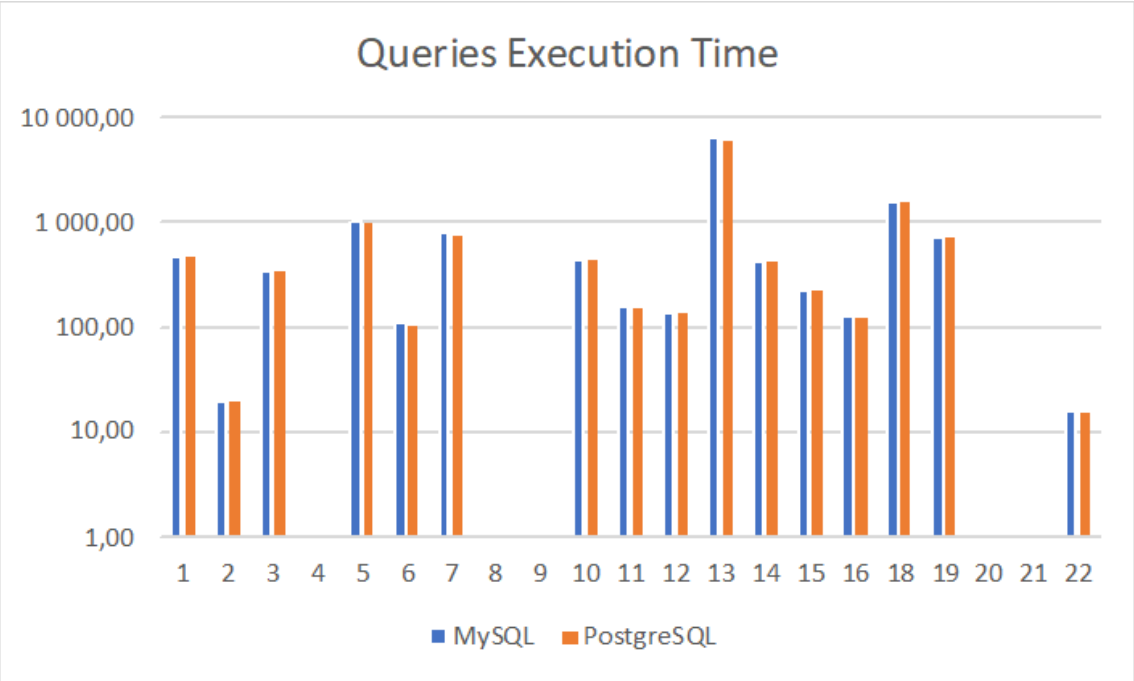


Figura 26: Tempo de execução Queries

C Buffer Cache

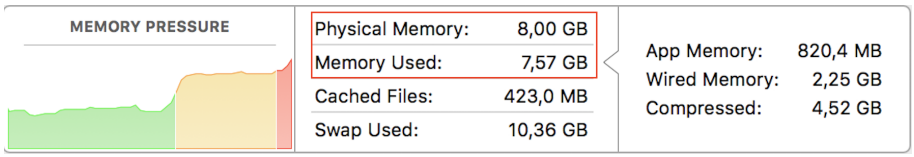


Figura 27: Memória usada