Universidade de São Paulo
Escola de Artes, Ciências e Humanidades
Disciplina: Laboratório de Banco de Dados
Profª Dra. Fátima Nunes.

Administração de Condomínio

Parte III - Artefato B

Fernando K. G. de Amorim – 10387644
João Guilherme da Costa Seike – 9784634
Lucas Pereira Castelo Branco – 10258772
Victor Gomes de O. M. Nicola – 9844881

São Paulo
2019

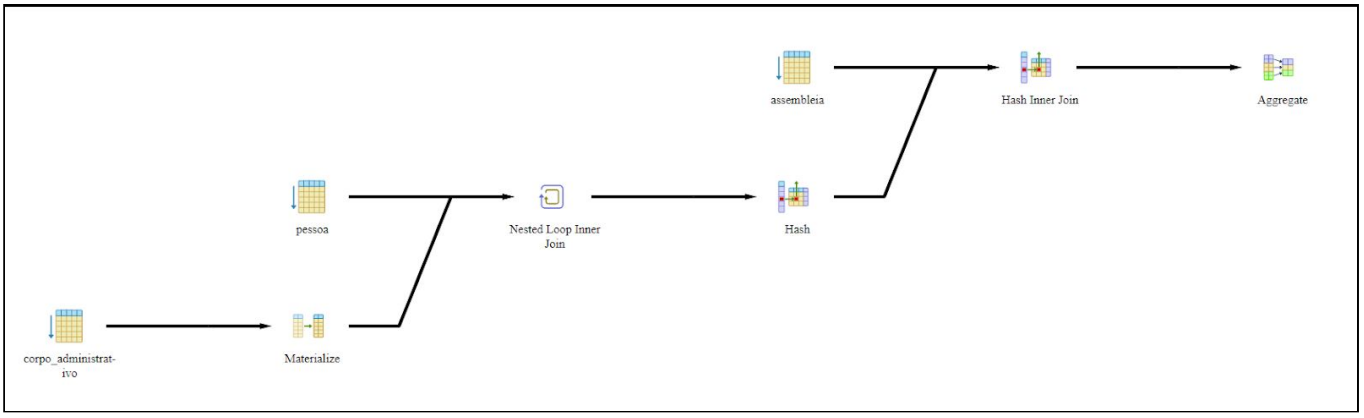
Artefato B

Todas as representações gráficas abaixo foram retiradas da ferramenta EXPLAIN disponível no software PGAdmin.

DISCLAIMER: Os tempos de planejamento e execução aqui demonstrados estão sob a política de *caching* do PostgreSQL, o que provoca uma mudança relativa nos tempos de execução se comparada a outras etapas demonstradas aqui em outros artefatos. Mais detalhes sobre a política do PostgreSQL podem ser encontraddas aqui: https://www.slideshare.net/uptimeforce/postgresql-query-cache-pgc

Query I

```
SELECT Pessoa.nome,
                MAX(assembleia.data) AS ultima_assembleia
        FROM adm_condominio.Pessoa AS Pessoa
        JOIN adm_condominio.Corpo_Administrativo AS Corpo_Administrativo
          ON Pessoa.id_pessoa IN (Corpo_Administrativo.id_sindico, Corpo_Administrativo.id_subsindico,
                                  Corpo_Administrativo.id_conselheiro_1,
Corpo_Administrativo.id_conselheiro_2,
                                  Corpo_Administrativo.id_conselheiro_3)
        JOIN adm_condominio.Assembleia AS Assembleia
          ON Corpo_Administrativo.id_corpo = Assembleia.fk_id_corpo_admin
        WHERE data_eleicao <= CAST(NOW() AS DATE) - interval '2 years'
        GROUP BY 1;
```



Nesse modelo, percebemos que a primeira tabela usada durante a construção fora a tabela 'corpo_administrativo', que possui os campos mais primitivos, e de fato, possui o menor número de linhas. Logo em seguida, a ação Materialize, realizada pelo banco, signifca que a tabela fora copiada para a memória para dar prosseguimento a execução da query.

Após disso, ocorre o JOIN com a tabela 'pessoa'. O sistema optou por realizar um JOIN aninhado em loop, o que significa que buscou otimizar o modelo de INNER exigido na query acima, e portanto, tratava-se de uma junção por condição simples. Após isso, somamos a utilização de *buckets* a partir do modelo de hash, de forma a usá-lo como índice na próxima etapa de JOIN, com a tabela 'assembleia', ao utilizar do que está sinalizado como 'hash inner join'. Após isso, ocorre a ação Aggregate, que nada mais é do que a ação que consolida as linhas resultantes da consulta.

	QUERY PLAN text
1	HashAggregate (cost=172.41..173.41 rows=100 width=17) (actual time=1.645..1.652 rows=39 loops=1)
2	Group Key: pessoa.nome
3	-> Hash Join (cost=148.56..168.30 rows=822 width=17) (actual time=1.568..1.599 rows=125 loops=1)
4	Hash Cond: (assembleia.fk_id_corpo_admin = corpo_administrativo.id_corpo)
5	-> Seq Scan on assembleia (cost=0.00..13.00 rows=300 width=8) (actual time=0.024..0.027 rows=40 loops=1)
6	-> Hash (cost=145.14..145.14 rows=274 width=17) (actual time=1.525..1.525 rows=280 loops=1)
7	Buckets: 1024 Batches: 1 Memory Usage: 22kB
8	-> Nested Loop (cost=0.00..145.14 rows=274 width=17) (actual time=0.055..1.471 rows=280 loops=1)
9	Join Filter: ((pessoa.id_pessoa = corpo_administrativo.id_sindico) OR (pessoa.id_pessoa = corpo_administrativo.id_subsindico) OR (pess...
10	Rows Removed by Join Filter: 5320
11	-> Seq Scan on pessoa (cost=0.00..2.00 rows=100 width=17) (actual time=0.014..0.022 rows=100 loops=1)
12	-> Materialize (cost=0.00..3.28 rows=56 width=24) (actual time=0.000..0.003 rows=56 loops=100)
13	-> Seq Scan on corpo_administrativo (cost=0.00..3.00 rows=56 width=24) (actual time=0.017..0.061 rows=56 loops=1)
14	Filter: (data_eleicao <= ((now())::date - '2 years'::interval))
15	Rows Removed by Filter: 44
16	Planning Time: 0.340 ms
17	Execution Time: 1.725 ms

Entrando em maiores detalhes, podemos verificar algumas outras constatações feitas durante a execução. Vale observar que, até a linha 7, o custo inicial de todas as operações havia sido 0, o que muito é justificado pelo tamanho relativamente pequeno do banco a ser analisado. Porém, justamente nessa linha, conseguimos observar quão custosa é a operação de junção aninhada, gerando um custo ao final de 145.14, que possui um peso significativo até o final da execução.

Query II

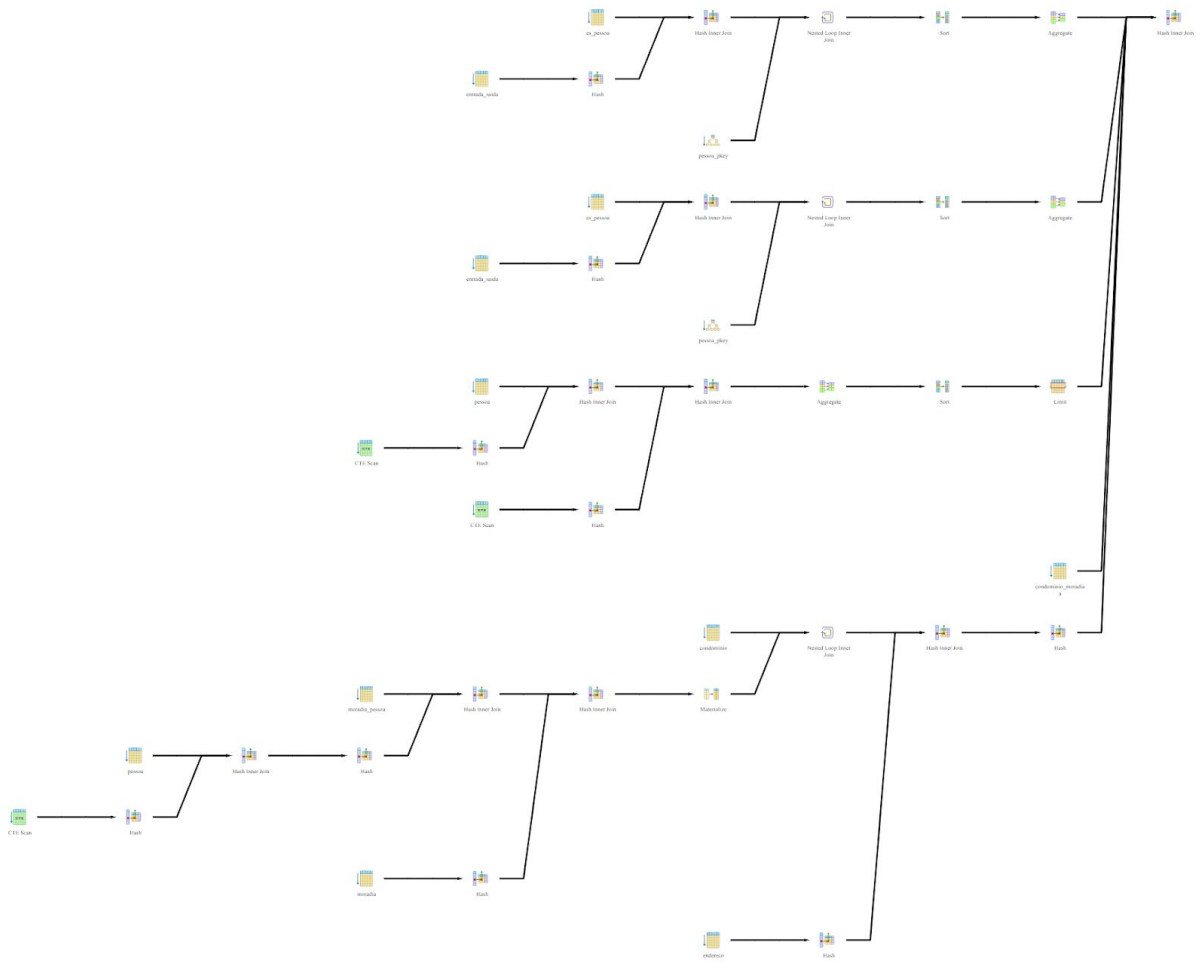
```
WITH sum_entradas AS (
		SELECT pessoa.id_pessoa AS id_pessoa, SUM(DATE_PART('epoch', Entrada_Saida.data_hora)) AS soma
		FROM adm_condominio.Pessoa AS Pessoa
		JOIN adm_condominio.Es_Pessoa AS Es_Pessoa ON Pessoa.id_pessoa = Es_Pessoa.fk_id_pessoa
		JOIN adm_condominio.Entrada_Saida AS Entrada_Saida ON Entrada_Saida.id_es = Es_Pessoa.fk_id_es
		WHERE Entrada_Saida.acao = 'e'
		GROUP BY 1
),

sum_saidas AS (
		SELECT pessoa.id_pessoa AS id_pessoa, SUM(DATE_PART('epoch', Entrada_Saida.data_hora)) AS soma
		FROM adm_condominio.Pessoa AS Pessoa
		JOIN adm_condominio.Es_Pessoa AS Es_Pessoa ON Pessoa.id_pessoa = Es_Pessoa.fk_id_pessoa
		JOIN adm_condominio.Entrada_Saida AS Entrada_Saida ON Entrada_Saida.id_es = Es_Pessoa.fk_id_es
		WHERE Entrada_Saida.acao = 's'
		GROUP BY 1
```

```
),

top_five AS (
	SELECT Pessoa.id_pessoa, SUM(sum_entradas.soma - sum_saidas.soma)
		FROM adm_condominio.Pessoa AS Pessoa
		JOIN sum_entradas ON sum_entradas.id_pessoa = Pessoa.id_pessoa
		JOIN sum_saidas ON sum_saidas.id_pessoa = Pessoa.id_pessoa
		GROUP BY 1 ORDER BY 2 DESC
		LIMIT 5
)

SELECT Pessoa.nome, Endereco.cidade
	FROM adm_condominio.Pessoa AS Pessoa
	JOIN adm_condominio.Moradia_Pessoa AS Moradia_Pessoa
	  ON Moradia_Pessoa.fk_id_pessoa = Pessoa.id_pessoa
	JOIN adm_condominio.Moradia AS Moradia
	  ON Moradia.id_moradia = Moradia_Pessoa.fk_id_moradia
	JOIN adm_condominio.Condominio_Moradia AS Condominio_Moradia
	JOIN adm_condominio.Condominio AS Condominio
	  ON Condominio.id_condominio = Condominio_Moradia.fk_id_condominio
	  ON Condominio_Moradia.fk_id_condominio = Condominio.id_condominio
	JOIN adm_condominio.Endereco AS Endereco
	  ON Endereco.id_endereco = Condominio.fk_id_endereco
	JOIN top_five
	  ON top_five.id_pessoa = Pessoa.id_pessoa;
```



Essa consulta é extremamente custosa, e é muito bem demonstrada pela esquematização acima. Isso é claramente afetado pela presença de diversas instâncias de *subqueries* externas à query principal, criadas pelas ocorrências de WITH TABLE AS. Isso já é demonstrado exatamente pelos 'nós filhos' da árvore de execução criada, sinalizada por CTE Scan.

Indo de cima para baixo, da esquerda para a direita, dividimos em quatro partes:

1) A primeira parte equivale a subquery 'sum_entradas'. As tabelas 'entrada_saida' e 'es_pessoa' são as que formam a primeira junção. Para isso, a tabela 'entrada_saida' passa a ser compartimentada pelo uso de hashing, e dessa forma, cria-se a junção por condição simples com a tabela 'es_pessoa' (que é um mero intermediário', com o uso de Hash Inner Join. Após isso, o sistema cria um índice em árvore para 'pessoa', que será utilizada em diversas outras oportunidades nessa query. A ela, é dada o nome de 'pessoa_pkey', ou seja, a chave primária da tabela em questão. Com a criação da tabela da operação anterior, é feito um JOIN aninhado simples com INNER JOIN entre o índice e a tabela em hash. Por fim, e feita a ordenação e agregação, encerrando essa tabela resultante.
2) A segunda parte, simbolizada na query pela *label* 'sum_saidas', é extremamente análogo a 'sum_entradas' (apresentado na parte 1), e portanto, a única diferença prática é a sua saída enquanto coluna resultante.
3) Esta parte equivale ao trecho como 'top_five'. Perceba que o início dessa subquery se inicia pelo CTE Scan, o que na verdade, significa que a subquey em questão se utiliza de duas outras subqueries (no caso, as duas citadas acima) para gerar seus resultados. Ambas se utilizam de Hash INNER Join para ocorrer, e a primeira delas realiza esta junção simples com a tabela 'pessoa'. Note que, ao final da primeira junção, soma-se a outra CTE, e realiza-se a operação de junção por hash novamente. O final representa, na ordem, a agregação (dado por GROUP BY), ordenação (dado por ORDER BY) e limite (que é, como o *label* sugere, as cinco primeiras tuplas da relação criada).

4) Por fim, a *query* principal é apenas a aglutinação final demonstradas pelos JOINs. Para isso, é realizado a princípio um CTE Scan, para obter as informações de 'top_five', e que por sua vez, é colocado através de uma função hash, que será utilizado na próxima junção, dessa vez com a tabela 'Pessoa'. Após isso, é feita a mesma função de hash com 'moradia_pessoa', mostrando como é o método mais recomendado para sistemas que usam de múltiplas tabelas e subqueries. O mesmo é feito por 'moradia', aplicando-o ao hash, e fazendo sua junção com o restante. É importante notar que, ao final desses passos, temos o caso de Materialize, para que tais dados sejam salvos em memória, de forma a permitir que as outras junções ocorram.
   Logo, essa relação resultante se une primeiramente a 'condominio', através de uma junção aninhada simples, e após isso, a tabela 'endereco' junta-se, dessa vez repetindo o uso de hash, e terminando o processo com 'condominio_moradias'.

Ao final de tudo, todas as partes se unem através de um Hash Inner Join (o que é justificado como o uso constante de hashs em torno de toda a função).

Em mais detalhes, a listagem do plano de execução é:

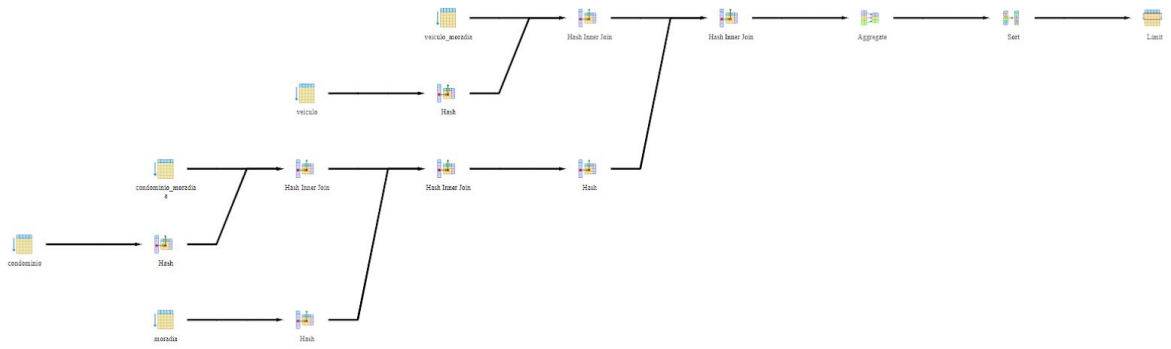
	QUERY PLAN text
1	Hash Join (cost=278.78..918.19 rows=126560 width=24) (actual time=0.887..0.887 rows=0 loops=1)
2	Hash Cond: (condominio_moradia.fk_id_condominio = condominio.id_condominio)
3	CTE sum_entradas
4	-> GroupAggregate (cost=56.37..56.55 rows=9 width=12) (actual time=0.194..0.202 rows=6 loops=1)
5	Group Key: pessoa_1.id_pessoa
6	-> Sort (cost=56.37..56.39 rows=9 width=12) (actual time=0.181..0.182 rows=6 loops=1)
7	Sort Key: pessoa_1.id_pessoa
8	Sort Method: quicksort Memory: 25kB
9	-> Nested Loop (cost=16.29..56.23 rows=9 width=12) (actual time=0.126..0.169 rows=6 loops=1)
10	-> Hash Join (cost=16.15..54.74 rows=9 width=12) (actual time=0.098..0.106 rows=6 loops=1)
11	Hash Cond: (es_pessoa.fk_id_es = entrada_saida.id_es)
12	-> Seq Scan on es_pessoa (cost=0.00..32.60 rows=2260 width=8) (actual time=0.020..0.022 rows=10 loops=1)
13	-> Hash (cost=16.13..16.13 rows=2 width=12) (actual time=0.051..0.052 rows=11 loops=1)
14	Buckets: 1024 Batches: 1 Memory Usage: 9kB
15	-> Seq Scan on entrada_saida (cost=0.00..16.13 rows=2 width=12) (actual time=0.022..0.027 rows=11 loops=1)
16	Filter: (acao = 'e'::bpchar)
17	Rows Removed by Filter: 9
18	-> Index Only Scan using pessoa_pkey on pessoa pessoa_1 (cost=0.14..0.17 rows=1 width=4) (actual time=0.009..0.009 rows=1 loops=6)
19	Index Cond: (id_pessoa = es_pessoa.fk_id_pessoa)
20	Heap Fetches: 6
21	CTE sum_saidas
22	-> GroupAggregate (cost=56.37..56.55 rows=9 width=12) (actual time=0.111..0.115 rows=4 loops=1)
23	Group Key: pessoa_2.id_pessoa
24	-> Sort (cost=56.37..56.39 rows=9 width=12) (actual time=0.107..0.108 rows=4 loops=1)
25	Sort Key: pessoa_2.id_pessoa

```
26          Sort Method: quicksort  Memory: 25kB
27          ->  Nested Loop  (cost=16.29..56.23 rows=9 width=12) (actual time=0.083..0.098 rows=4 loops=1)
28                ->  Hash Join  (cost=16.15..54.74 rows=9 width=12) (actual time=0.069..0.075 rows=4 loops=1)
29                      Hash Cond: (es_pessoa_1.fk_id_es = entrada_saida_1.id_es)
30                      ->  Seq Scan on es_pessoa es_pessoa_1  (cost=0.00..32.60 rows=2260 width=8) (actual time=0.016..0.018 rows=10 loops=1)
31                      ->  Hash  (cost=16.13..16.13 rows=2 width=12) (actual time=0.032..0.032 rows=9 loops=1)
32                            Buckets: 1024  Batches: 1  Memory Usage: 9kB
33                            ->  Seq Scan on entrada_saida entrada_saida_1  (cost=0.00..16.13 rows=2 width=12) (actual time=0.017..0.023 rows=9 loops=1)
34                                  Filter: (acao = 's'::bpchar)
35                                  Rows Removed by Filter: 11
36                ->  Index Only Scan using pessoa_pkey on pessoa pessoa_2  (cost=0.14..0.17 rows=1 width=4) (actual time=0.005..0.005 rows=1 loops=4)
37                      Index Cond: (id_pessoa = es_pessoa_1.fk_id_pessoa)
38                      Heap Fetches: 4
39    CTE top_five
40      ->  Limit  (cost=3.39..3.41 rows=5 width=12) (actual time=0.447..0.447 rows=0 loops=1)
41            ->  Sort  (cost=3.39..3.42 rows=9 width=12) (actual time=0.446..0.446 rows=0 loops=1)
42                  Sort Key: (sum((sum_entradas.soma - sum_saidas.soma))) DESC
43                  Sort Method: quicksort  Memory: 25kB
44                  ->  HashAggregate  (cost=3.16..3.25 rows=9 width=12) (actual time=0.435..0.435 rows=0 loops=1)
45                        Group Key: pessoa_3.id_pessoa
46                        ->  Hash Join  (cost=0.58..3.09 rows=9 width=20) (actual time=0.434..0.434 rows=0 loops=1)
47                              Hash Cond: (pessoa_3.id_pessoa = sum_entradas.id_pessoa)
48                              ->  Hash Join  (cost=0.29..2.76 rows=9 width=16) (actual time=0.178..0.200 rows=4 loops=1)
49                                    Hash Cond: (pessoa_3.id_pessoa = sum_saidas.id_pessoa)
50                                    ->  Seq Scan on pessoa pessoa_3  (cost=0.00..2.00 rows=100 width=4) (actual time=0.025..0.036 rows=100 loops=1)
51                                    ->  Hash  (cost=0.18..0.18 rows=9 width=12) (actual time=0.131..0.131 rows=4 loops=1)
52                                          Buckets: 1024  Batches: 1  Memory Usage: 9kB
53                                          ->  CTE Scan on sum_saidas  (cost=0.00..0.18 rows=9 width=12) (actual time=0.113..0.120 rows=4 loops=1)
54                              ->  Hash  (cost=0.18..0.18 rows=9 width=12) (actual time=0.216..0.216 rows=6 loops=1)
55                                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
56                                    ->  CTE Scan on sum_entradas  (cost=0.00..0.18 rows=9 width=12) (actual time=0.196..0.207 rows=6 loops=1)
57    ->  Seq Scan on condominio_moradia  (cost=0.00..32.60 rows=2260 width=4) (actual time=0.038..0.038 rows=1 loops=1)
58    ->  Hash  (cost=113.27..113.27 rows=3920 width=28) (actual time=0.794..0.794 rows=0 loops=1)
59          Buckets: 4096  Batches: 1  Memory Usage: 32kB
60          ->  Hash Join  (cost=9.93..113.27 rows=3920 width=28) (actual time=0.793..0.793 rows=0 loops=1)
61                Hash Cond: (condominio.fk_id_endereco = endereco.id_endereco)
62                ->  Nested Loop  (cost=5.67..98.30 rows=3920 width=21) (actual time=0.681..0.681 rows=0 loops=1)
63                      ->  Seq Scan on condominio  (cost=0.00..1.70 rows=70 width=8) (actual time=0.023..0.029 rows=70 loops=1)
64                      ->  Materialize  (cost=5.67..47.74 rows=56 width=13) (actual time=0.009..0.009 rows=0 loops=70)
65                            ->  Hash Join  (cost=5.67..47.46 rows=56 width=13) (actual time=0.622..0.622 rows=0 loops=1)
66                                  Hash Cond: (moradia_pessoa.fk_id_moradia = moradia.id_moradia)
67                                  ->  Hash Join  (cost=2.65..44.29 rows=56 width=17) (actual time=0.537..0.537 rows=0 loops=1)
68                                        Hash Cond: (moradia_pessoa.fk_id_pessoa = pessoa.id_pessoa)
69                                        ->  Seq Scan on moradia_pessoa  (cost=0.00..32.60 rows=2260 width=8) (actual time=0.020..0.020 rows=1 loops=1)
70                                        ->  Hash  (cost=2.59..2.59 rows=5 width=21) (actual time=0.492..0.492 rows=0 loops=1)
71                                              Buckets: 1024  Batches: 1  Memory Usage: 8kB
72                                              ->  Hash Join  (cost=0.16..2.59 rows=5 width=21) (actual time=0.492..0.492 rows=0 loops=1)
73                                                    Hash Cond: (pessoa.id_pessoa = top_five.id_pessoa)
74                                                    ->  Seq Scan on pessoa  (cost=0.00..2.00 rows=100 width=17) (actual time=0.023..0.023 rows=1 loops=1)
75                                                    ->  Hash  (cost=0.10..0.10 rows=5 width=4) (actual time=0.449..0.449 rows=0 loops=1)
76                                                          Buckets: 1024  Batches: 1  Memory Usage: 8kB
77                                                          ->  CTE Scan on top_five  (cost=0.00..0.10 rows=5 width=4) (actual time=0.449..0.449 rows=0 loops=1)
78                                  ->  Hash  (cost=1.90..1.90 rows=90 width=4) (actual time=0.066..0.066 rows=90 loops=1)
79                                        Buckets: 1024  Batches: 1  Memory Usage: 12kB
80                                        ->  Seq Scan on moradia  (cost=0.00..1.90 rows=90 width=4) (actual time=0.023..0.035 rows=90 loops=1)
81                ->  Hash  (cost=3.00..3.00 rows=100 width=15) (actual time=0.092..0.092 rows=100 loops=1)
82                      Buckets: 1024  Batches: 1  Memory Usage: 13kB
83                      ->  Seq Scan on endereco  (cost=0.00..3.00 rows=100 width=15) (actual time=0.027..0.051 rows=100 loops=1)
84    Planning Time: 2.393 ms
85    Execution Time: 1.533 ms
```

Desta listagem, vale citar o custo altíssimo, apesar de estarmos lidando com uma quantidade razoavelmente baixa de entradas em nossas relações. É muito válido citar nominalmente a presença de 126560 linhas que foram consultadas durante toda a execução, como mostrado na linha 1, reflexo do grande número de subqueries do tipo CTE (porém, vale ressaltar que a consulta dessas CTEs em si é extremamente barata, visto o princípio de proximidade na memória, o que causa um custo final praticamente nulo). É notável também o aumento abrupto no custo de execução nesse último JOIN por hash (também demonstrado na linha 1): de 278.78 como inicial para 918.19. O motivo disso é fácil de identificar, visto que estamos lidando com a junção de três relações. Vale citar que o custo relativo de 'sum_entradas' (da linha 3 até a 20) e 'sum_saidas' (da linha 21 até 38) são exatamente iguais, visto que as operações internas são as mesmas, com a diferenciação apenas no tipo de ação que elas monitoram.

Query III

```
SELECT v.marca, COUNT(v.*) FROM adm_condominio.Veiculo AS v
        INNER JOIN adm_condominio.Veiculo_Moradia AS vM ON vM.fk_id_veiculo = v.id_veiculo
        INNER JOIN adm_condominio.Moradia AS m ON m.id_moradia = vM.fk_id_moradia AND
m.tipo_moradia = 'a'
        INNER JOIN adm_condominio.Condominio_Moradia AS cM ON cM.fk_id_moradia = m.id_moradia
        INNER JOIN adm_condominio.Condominio AS c ON c.id_condominio = cM.fk_id_condominio AND
c.tipo_condominio = 'e'
        GROUP BY v.marca
        ORDER BY 2
        LIMIT 3;
```



Neste modelo, basicamente há uma quantidade razoável de joins por hash, como segue-se nas outras queries anteriores. Inicia-se em 'condominio' juntamente com 'condominio_moradia', e logo em seguida, juntando-se com 'moradia', o que forma a ligação entre elas de forma geral. Após isso, 'veiculo' e 'veiculo_moradia' fazem uma nova junção em uma relação em paralelo, e acaba por juntar-se com a relação citada na primeira parte. No final, ocorre a função Aggregate (que leva a memória para operações), Sort, e por fim, limita-se, que equivale a função LIMIT 3

Olhando para a análise mais minunciosa:

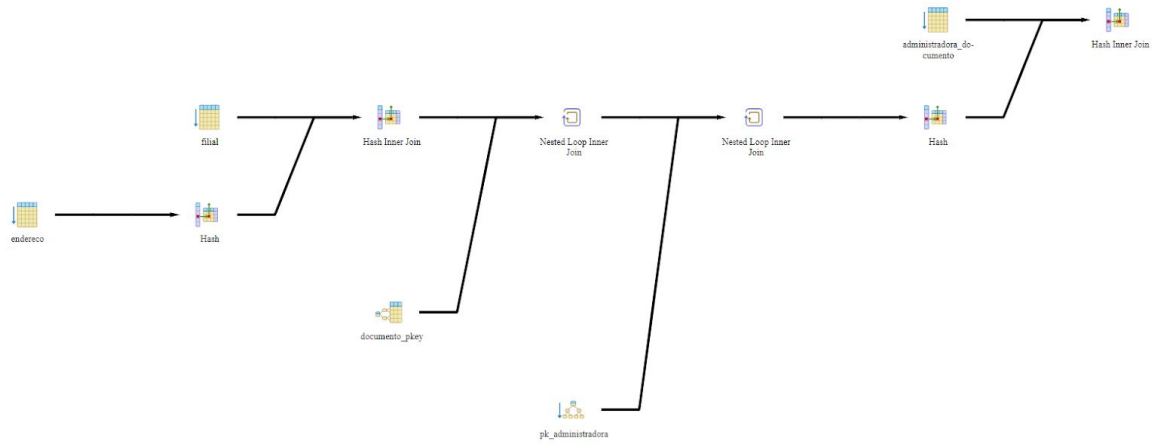
	QUERY PLAN text
1	Limit (cost=327.84..327.85 rows=3 width=16) (actual time=0.315..0.318 rows=3 loops=1)
2	-> Sort (cost=327.84..327.93 rows=34 width=16) (actual time=0.314..0.314 rows=3 loops=1)
3	Sort Key: (count(v.*))
4	Sort Method: quicksort Memory: 25kB
5	-> HashAggregate (cost=327.06..327.40 rows=34 width=16) (actual time=0.291..0.293 rows=4 loops=1)
6	Group Key: v.marca
7	-> Hash Join (cost=59.20..228.71 rows=19671 width=76) (actual time=0.274..0.284 rows=4 loops=1)
8	Hash Cond: (vm.fk_id_moradia = m.id_moradia)
9	-> Hash Join (cost=2.35..41.00 rows=2260 width=80) (actual time=0.129..0.138 rows=20 loops=1)
10	Hash Cond: (vm.fk_id_veiculo = v.id_veiculo)
11	-> Seq Scan on veiculo_moradia vm (cost=0.00..32.60 rows=2260 width=8) (actual time=0.029..0.030 rows=20 loops=1)
12	-> Hash (cost=1.60..1.60 rows=60 width=80) (actual time=0.082..0.082 rows=60 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 15kB
14	-> Seq Scan on veiculo v (cost=0.00..1.60 rows=60 width=80) (actual time=0.030..0.059 rows=60 loops=1)
15	-> Hash (cost=47.06..47.06 rows=783 width=8) (actual time=0.132..0.132 rows=9 loops=1)
16	Buckets: 1024 Batches: 1 Memory Usage: 9kB
17	-> Hash Join (cost=5.20..47.06 rows=783 width=8) (actual time=0.114..0.126 rows=9 loops=1)
18	Hash Cond: (cm.fk_id_moradia = m.id_moradia)
19	-> Hash Join (cost=2.34..40.99 rows=1195 width=4) (actual time=0.061..0.070 rows=19 loops=1)
20	Hash Cond: (cm.fk_id_condominio = c.id_condominio)
21	-> Seq Scan on condominio_moradia cm (cost=0.00..32.60 rows=2260 width=8) (actual time=0.013..0.015 rows=40 loops=1)
22	-> Hash (cost=1.88..1.88 rows=37 width=4) (actual time=0.034..0.034 rows=37 loops=1)
23	Buckets: 1024 Batches: 1 Memory Usage: 10kB
24	-> Seq Scan on condominio c (cost=0.00..1.88 rows=37 width=4) (actual time=0.015..0.027 rows=37 loops=1)
25	Filter: (tipo_condominio = 'e'::bpchar)
26	Rows Removed by Filter: 33
27	-> Hash (cost=2.13..2.13 rows=59 width=4) (actual time=0.042..0.043 rows=59 loops=1)
28	Buckets: 1024 Batches: 1 Memory Usage: 11kB
29	-> Seq Scan on moradia m (cost=0.00..2.13 rows=59 width=4) (actual time=0.017..0.029 rows=59 loops=1)
30	Filter: (tipo_moradia = 'a'::bpchar)
31	Rows Removed by Filter: 31
32	Planning Time: 0.772 ms
33	Execution Time: 0.489 ms

É curioso notar o escalonamento dos custos conforme a query vai utilizando de JOINs por hash. Na linha 17, por exemplo, o custo de criação, de 5.20, passa para 47.06; já na linha 7, próximo do fim da execução, conseguimos notar outro grande salto de custos, de 59.20 para 228.71. Outra característica curiosa dessa execução é que conseguimos observar o comportamento do SGBD ao lidar com comparativos de char, usando da operação especial "::bpchar"

Query IV

```
SELECT Administradora.id_administradora,
                    CASE WHEN Documento.status_documento = '1'
                                        THEN 'Aprovado'
                                WHEN Documento.status_documento = '2'
                                        THEN 'Recusado'
                                WHEN Documento.status_documento = '3'
                                        THEN 'Em Aprovação'
                                WHEN Documento.status_documento = '4'
                                        THEN 'Em Processamento'
                                WHEN Documento.status_documento = '5'
                                        THEN 'Não Disponível'
                                END AS status_documento,
                                Documento.data AS data_criacao
        FROM adm_condominio.Administradora AS Administradora
```

```
JOIN adm_condominio.Administradora_Documento AS Administradora_Documento
  ON Administradora_Documento.fk_id_documento = Administradora.id_administradora
JOIN adm_condominio.Documento AS Documento
  ON Documento.id_documento = Administradora_Documento.fk_id_documento
JOIN adm_condominio.Filial AS Filial
  ON Filial.fk_id_administradora = Administradora.id_administradora
JOIN adm_condominio.Endereco AS Endereco
  ON Endereco.id_endereco = Filial.fk_id_endereco
WHERE NOT Documento.status_documento = '1'
          AND Endereco.estado IN ('ES', 'MG',  'RJ', 'SP');
```



Esta query, ao contrário das outras, tem uma profundidade relativamente menor, porém usa de recursos pouco explorados nas anteriores. A princípio, é realizado uma junção via função hash entre ‘endereco’ e ‘filial’. Para que seja realizado o join com ‘documento’, o SGBD apenas utiliza de sua primary key, de forma a usá-la para executar a função de junção aninhada com a relação citada anteriormente. Após isso, restando apenas mais uma junção com a tabela ‘administradora_documento’, ocorre o uso de chave primária de administradora, que está organizada em índice de árvore, que serve para mais uma função aninhada; com essa total relação montada, faz-se finalmente com a tabela final, através da função hash.

	QUERY PLAN text
1	Hash Join (cost=26.75..69.26 rows=83 width=44) (actual time=0.182..0.182 rows=0 loops=1)
2	Hash Cond: (administradora_documento.fk_id_documento = administradora.id_administradora)
3	-> Seq Scan on administradora_documento (cost=0.00..32.60 rows=2260 width=4) (actual time=0.043..0.043 rows=1 loops=1)
4	-> Hash (cost=26.66..26.66 rows=7 width=78) (actual time=0.113..0.113 rows=0 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 8kB
6	-> Nested Loop (cost=3.84..26.66 rows=7 width=78) (actual time=0.113..0.113 rows=0 loops=1)
7	Join Filter: (documento.id_documento = administradora.id_administradora)
8	-> Nested Loop (cost=3.69..22.67 rows=13 width=74) (actual time=0.113..0.113 rows=0 loops=1)
9	-> Hash Join (cost=3.55..18.24 rows=15 width=4) (actual time=0.112..0.112 rows=0 loops=1)
10	Hash Cond: (filial.fk_id_endereco = endereco.id_endereco)
11	-> Seq Scan on filial (cost=0.00..13.70 rows=370 width=8) (actual time=0.021..0.021 rows=1 loops=1)
12	-> Hash (cost=3.50..3.50 rows=4 width=4) (actual time=0.064..0.064 rows=0 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 8kB
14	-> Seq Scan on endereco (cost=0.00..3.50 rows=4 width=4) (actual time=0.063..0.063 rows=0 loops=1)
15	Filter: ((estado)::text = ANY ('{ES,MG,RJ,SP}'::text[]))
16	Rows Removed by Filter: 100
17	-> Index Scan using documento_pkey on documento (cost=0.14..0.29 rows=1 width=70) (never executed)
18	Index Cond: (id_documento = filial.fk_id_administradora)
19	Filter: ((status_documento)::text <> '1'::text)
20	-> Index Only Scan using pk_administradora on administradora (cost=0.15..0.29 rows=1 width=4) (never executed)
21	Index Cond: (id_administradora = filial.fk_id_administradora)
22	Heap Fetches: 0
23	Planning Time: 1.249 ms
24	Execution Time: 0.264 ms

No plano descrito, podemos observar que o maior custo da operação deriva da parte final da query, onde há um escaneamento sequencial da tabela ‘administradora_documento’ com a relação resultante das operações anteriores. Podemos observar um custo final de 32.60 ao percorrer 2260 tuplas geradas para comparativos. Por fim, ao final disso, o JOIN por hash que resulta nas tuplas finais aumenta o custo de 26.75 para 69.26; curioso notar como a filtragem é feita na linha 15, onde ele deve lidar com múltiplos valores em texto para comparação. O custo fora relativamente baixo (3.50), porém altíssimo se considerarmos o número de tuplas envolvidas (no caso, apenas 4).