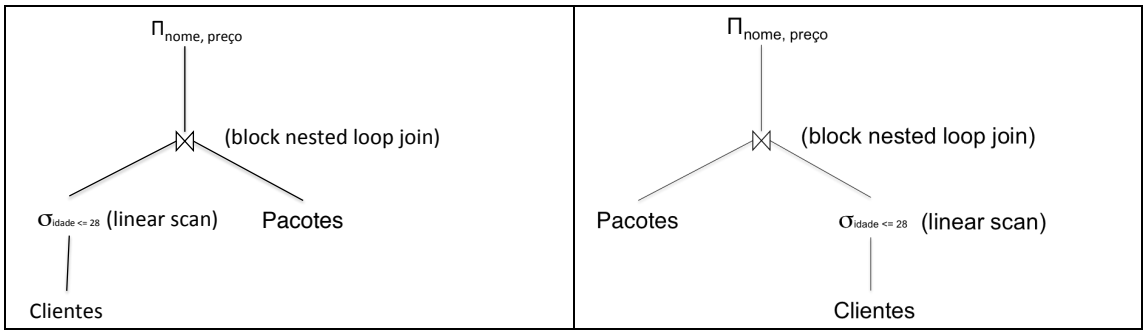
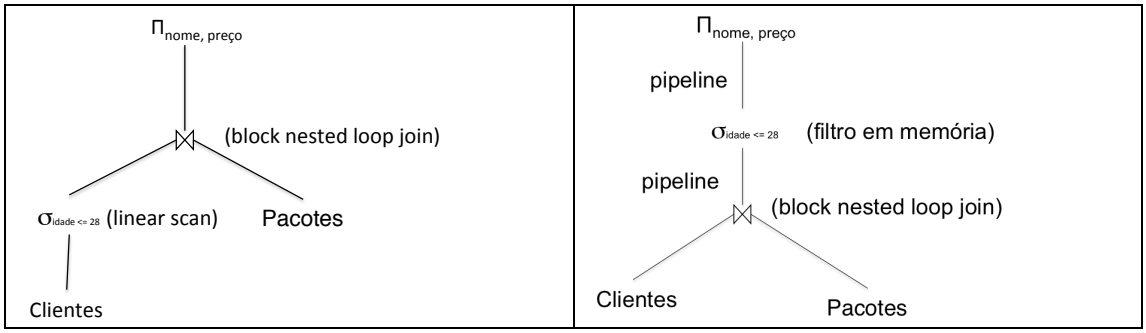
Resolução 1º teste SBD (2016/17)

1a) Sendo o ramo da esquerda a relação outer e o da direita a relação inner, o plano da esquerda seria mais eficiente pois Pacotes cabe em memória (só ocupa 1 bloco), pois o plano da direita obriga a fazer mais seeks.



Reparem que a resposta típica a este problema foi algo do género



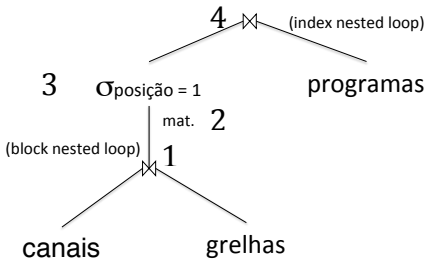
O plano da esquerda é ligeiramente mais eficiente do que o plano da direita, mas difícil de justificar. Se forem efetuados os cálculos verão que em termos de acesso a disco ambos os planos têm um custo idêntico pois nada é materializado e como pacotes cabe em memória (um bloco apenas...) efetuar o scan primeiro para filtrar ou fazer a posteriormente o custo será basicamente de 1 seek e de $b_{Clientes} + b_{Pacotes}$ transferências. Contudo, o tempo despendido em número de testes de tuplos a juntar no plano da direita é muito inferior e é isso que justifica ser mais eficiente mas justificável apenas em termos de tempo de CPU.

1b) SELECT * FROM Canais NATURAL INNER JOIN Programas
WHERE canal = '...';

A organização em disco a adoptar seria para cada registo de canal colocar nesse e blocos seguintes os programas desse canal. Reparem, se colocarem os programas com os canais "dentro" dos programas iremos ter duplicação de informação de canais.

1c) Sim, a pergunta pode ser eficientemente respondida pelos índices existentes na base de dados. O canal pode ser eficientemente obtido utilizando o índice de canais e obtendo o tuplo correspondente com muito poucos acessos a disco (árvore baixa e só devolvido um tuplo). Procurando o par ('RTP1','2016-12-30 00:00:00') no índice da chave de programas e obtendo todos os tuplos a partir das folhas da árvore B+ iremos também obter relativamente poucos valores (cerca de 40=2*(1825000/366)). Isto contrasta por exemplo com um varrimento sequencial da tabela Programas que tem 114063 blocos.

1d) Vamos numerar cada uma das operações no plano e obter a estimativa do custo para realizar cada uma delas:



1	(11+25)* t_T + 2* t_s = 56t_T	Ambas as relações cabem em memória logo é indiferente qual é inner ou outer. Devolvem-se 700 tuplos cada um deles com dimensão total de 700*(1024+128) bytes, necessitando de 99 blocos em disco.
2	2*99* t_T + 1* t_s = 208t_T	Necessitamos de escrever os 99 blocos em disco para materializar o resultado. Como temos de escrever repare-se no factor 2.
3	99* t_T + 1* t_s= 109t_T	Varrimento sequencial sobre o resultado materializado, devolvendo 700/250 tuplos, i.e. aproximadamente 3.
4	1*(t_T + t_s)+3*c em que c=(h+n_q+f)*(t_T + t_s) com h=4 pois 50*50*50*50>1825000, n = 9125 f = 182 = 307274t_T (o factor 1*(t_T + t_s) até poderia ser neglicenciado pois utilizamos pipeline evaluation)	Programas tem de ser a interior pois é a que tem o índice. A altura da árvore B+ no pior caso é 4 pois ceil($\log_{50}$(1825000)) = 4. Também podemos assumir que todas as folhas se encontram preenchidas e obtemos o mesmo valor para h (100*100*100*99) > 1825000 sendo este o melhor caso. Cada pesquisa ao índice devolve os programas de um canal que se estimamos serem n_q=1825000/200 = 9125. Mas necessitamos no pior caso de também aceder a f = 9125/50 = 182 folhas da árvore (1 acesso já está contabilizado no h)

1e) Criaria um índice bitmap no atributo tipo da tabela programas. O tipo tem um conjunto relativamente reduzido de valores necessitando de cerca de 28 (1825000/8/8192) blocos para cada valor possível para tipo. Assim, para se poder responder à consulta fornecida seriam necessários apenas ler 56 blocos de disco (+ no máximo 2 seeks). Este valor é muitíssimo inferior ao do varrimento da tabela programas (114063 blocos)

2a) Um índice esparso não possui entradas para todos os valores da chave de pesquisa sendo a estrutura habitual em índices primários. Num índice secundário a chave de pesquisa é diferente da chave de ordenação dos registos no ficheiro da tabela (ou pode mesmo não estar ordenada). Assim, não podemos recorrer à ordenação do ficheiro da tabela para reduzir o número de entradas no índice pois assim não seria possível saber a localização de um registo que não estivesse mencionado explicitamente no índice.

2b) O algoritmo de ordenação externa por fusão (external merge sort) permite a ordenação de ficheiros que não cabem memória. Suponhamos que dispomos de M blocos de memória para ordenação. O algoritmo divide-se em duas fases:

- Separa o ficheiro me N troços (runs) ordenados cada um deles com M blocos de dimensão;
- Funde M-1 troços de cada vez gerando um novo troço ordenado. Caso o número de troços originais N for maior ou igual que M esta fase tem de ser iterada mais do que uma vez.

Caso o ficheiro caiba totalmente em memória basta realizar a primeira fase. Caso contrário, ambas as fases terão de ser sempre realizadas.

2c) Em primeiro lugar teremos sempre de ler todos os blocos de *r*, a que corresponde o termo $b_r * t_T$ da expressão. Para cada um dos n_r tuplos de r teremos de utilizar o índice para obter o(s) correspondente(s) tuplos de s, que tem custo *c* para cada consulta, justificando o termo $n_r * c$. Como no pior caso só se tem um bloco para tuplos de r depois de esgotar todos os tuplos de um bloco de r teremos de fazer um seek para ir buscar um novo bloco de r (reparem que entretanto consultou-se o índice e a cabeça de leitura deslocou-se para outro local do disco), justificando o termo $b_r * t_S$ na fórmula. Caso tenhamos mais memória poderemos reescrever a expressão em $b_r * t_T + (b_r / M) * t_S + n_r * c$ pois só teremos de fazer *seeks* após tratar cada conjunto de M blocos de r.