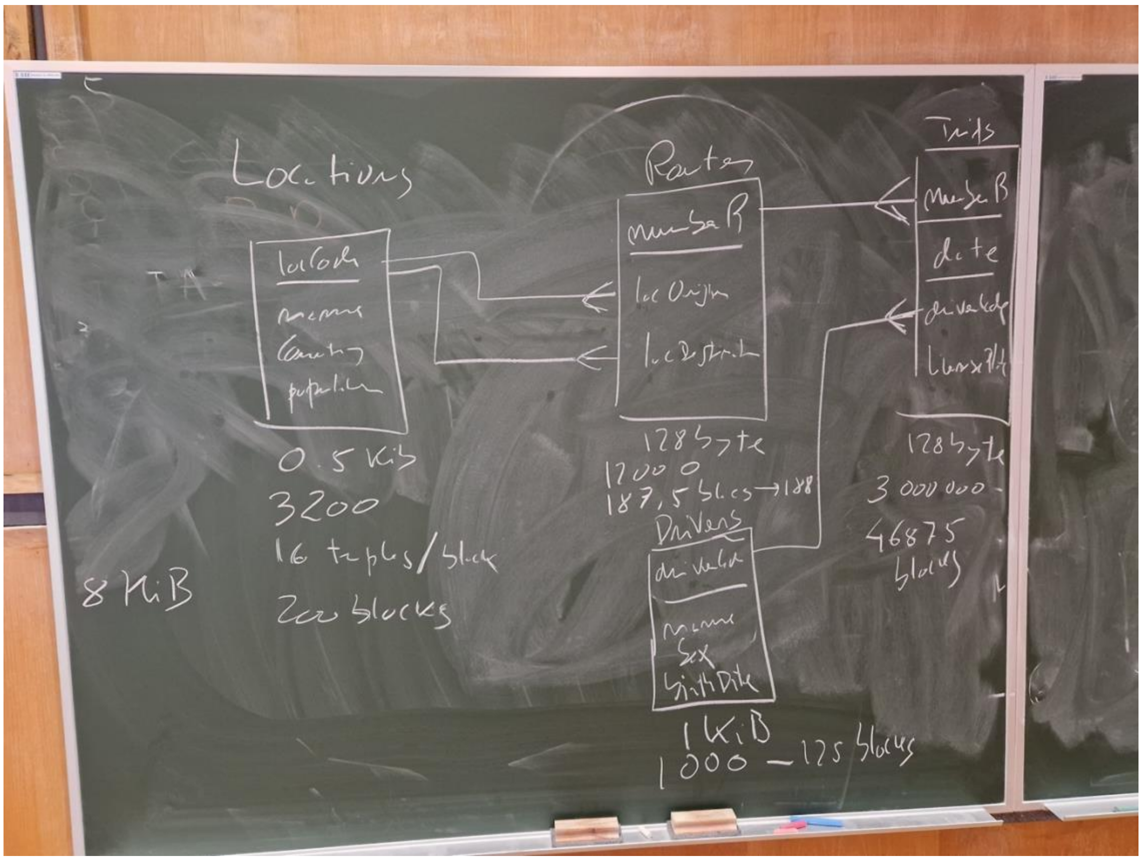
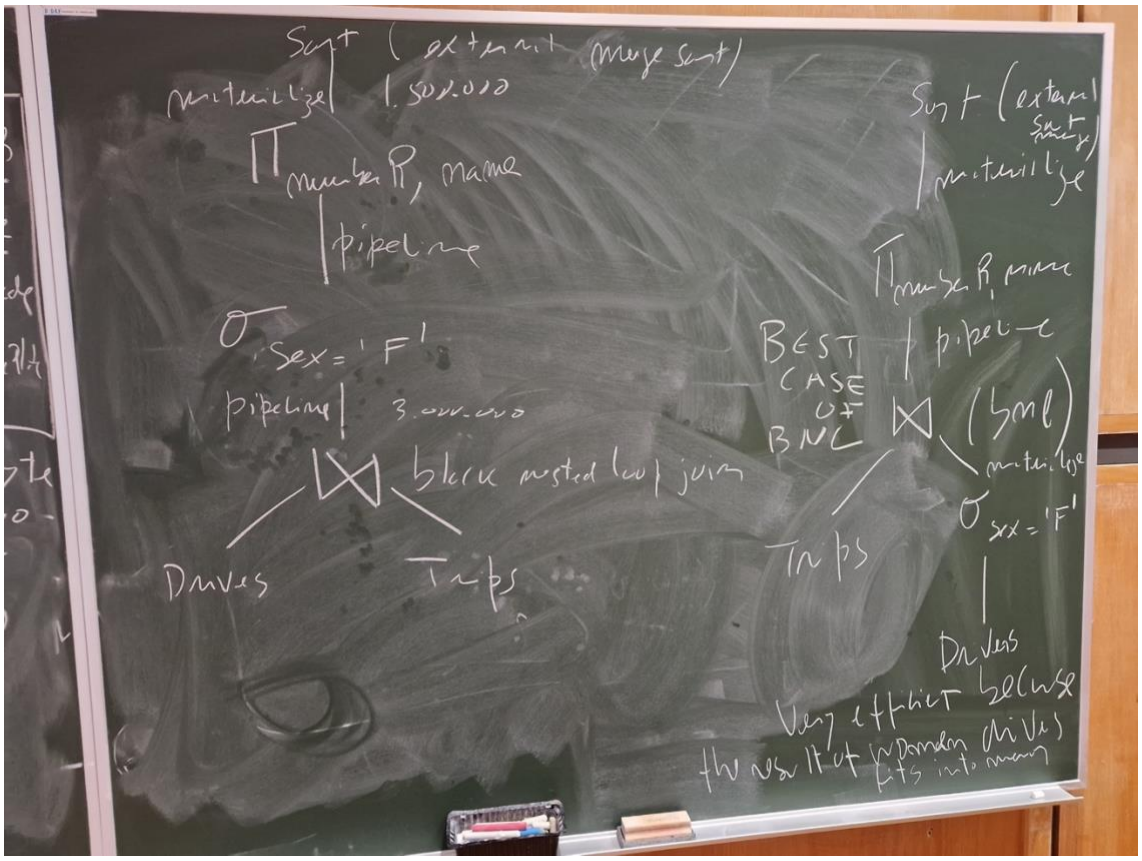
Notes to Selected Solutions of SBD Test 1 – 2022/23

Table Schema

Always start by drawing a table schema, with metrics about the number of tuples and the number of blocks, as well as the tuple size. These are all relevant for answering appropriately to the questions.



1a) The most efficient execution plan is the one in the right-hand side, since by doing the selection first all the tuples of the drivers relation fit into memory, and therefore the block-nested-loop join can be efficiently executed (contrary to the case in the right-hand-side since both relations do not fit into memory).

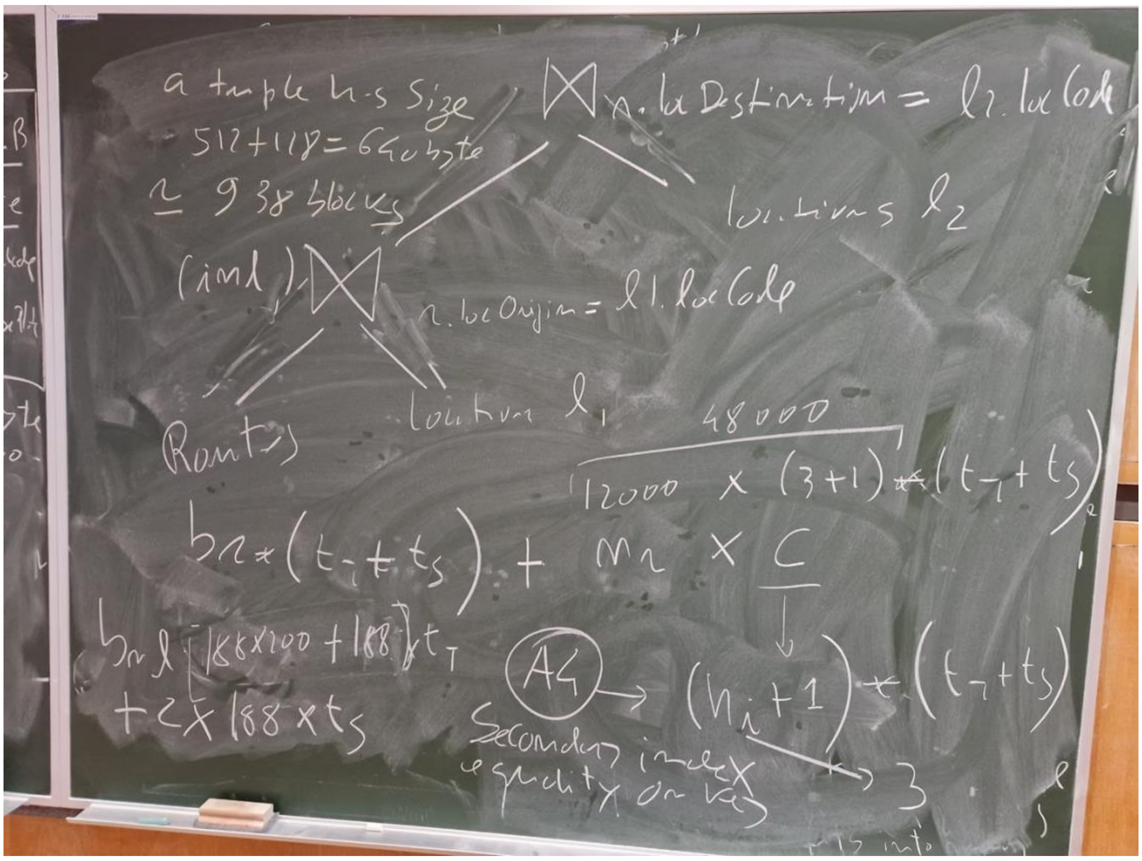


By convention, when you draw the diagram, the outer relation is on the left side of the join and the inner relation on the right side. Don't forget that an execution plan is the composition of operations PLUS the corresponding algorithm to execute each of the operations. Note that the projection operator in both plans does not need a special algorithm because it simply corresponds to dropping columns from the tuples incoming in the pipeline. Since we need to sort the result, we need to materialize the result of the projection operator because it does not fit into memory (circa of 1,500,000 tuples if we assume equal distribution among men and women drivers).

NOTE: in relational algebra the projection operation may drop duplicates if it uses set-semantics. Here we are assuming the version where we allow duplicates.

1b) Slotted pages cannot be used here because in average a tuple is bigger than a block. See the alternatives in the slides (e.g. TOAST mechanism of PostgreSQL).

1c) This is a quite interesting example showing that a lot of times the use of an index is not efficient at all! The only index that can be used here is the primary key of table Locations. So, the plan that can make use of the index is presented in the figure:



We have two theta-joins, and let us analyse the cost of the first index-nested-loop-join (INLJ) one (Routes with Locations l1) to convince that its cost is very high when compared to the block-nested-loop-join (BNLJ). The INLJ works has follows, for each tuple of routes (read block by block), we use the index to find the (starting) location. The index is a secondary index and we are searching using the primary key, getting exactly one answer. The cost formula in this case is:

$$
\begin{aligned}
& b_{routes} * (t_T+t_S) + n_{routes} * c_{lookup\text{-}key} \\
= \quad & 188 * (t_T+t_S) + 12000*(h_{index\_pk\_locations}+1)* (t_T+t_S) \\
= \quad & 188 * (t_T+t_S) + 12000*(3+1)* (t_T+t_S) \\
= \quad & 188 * (t_T+t_S) + 48000* (t_T+t_S) \\
= \quad & 188 * (1,1*t_S) + 48000* (1,1*t_S) \\
= \quad & 53006,8 * t_S
\end{aligned}
$$

This is a lot! Contrast with the block-nested-loop-join using the smaller relation (routes) as outer relation, with a cost of

$$
\begin{aligned}
& (b_{routes} * b_{locations} + b_{routes}) * t_T + 2* b_{routes} * t_S \\
= \quad & (188*200+188) * t_T + 2* 188 * t_S \\
= \quad & (188*200+188) * t_T + 2* 188 * t_S
\end{aligned}
$$

= $(188*200+188) * t_T + 2* 188 * t_S$

= $37788 * t_T + 376 * t_S$

= $37788 * 0{,}1* t_S + 376 * t_S$

= $37788 * 0{,}1* t_S + 376 * t_S$

= $4154{,}8 * t_S$

This is more than 10x faster than the index-nested-loop join! A similar algorithm applies to the second join, but now we must have to take into account that the result of the first join is pipelined (and the number of results is 12000 since every tuple of routes joins with a tuple of locations l1). So, for the 2nd INLJ we can simplify the formula to since we do not need to read blocks from disk:

$\cancel{b_r * (t_T+t_S)} + n_r * c_{lookup\text{-}key}$

= $12000*(h_{index\_pk\_locations}+1)* (t_T+t_S)$

= $12000*(3+1)* (t_T+t_S)$

= $48000* (t_T+t_S)$

= $48000* (1{,}1*t_S)$

= $52800 * t_S$

Using a block-nested-loop join the formula is adapted to take into account pipelining as follows:

$(b_{routes} * b_{locations} \cancel{+ b_{routes}}) * t_T + \cancel{2}* b_{routes} * t_S$

= $(938*200) * t_T + 938 * t_S$

= $187600* 0{,}1 * t_S + 938* t_S$

= $19698 * t_S$

The adaptation of the formula is due to the fact that we do not need to transfer the blocks from disk and do seeks for the blocks of the outer relation. Note that now we need 938 blocks to store the result of the 1st join, since we need for each tuple of the join to store the information of a tuple of routes (128 byte) + tuple of locations (512 byte), occupying each tuple 640 byte. Thus we need 12000 * 640 / 8192 = 938 blocks. So we conclude that it is much more efficient to not use the index!

FINAL REMARK: if we use all available memory for the 2nd BNLJ to store the incoming tuples from the pipeline (98 blocks), then cost can be further reduced to 210 * t_S since in this case we only have to perform 10 iterations of the outer relation:

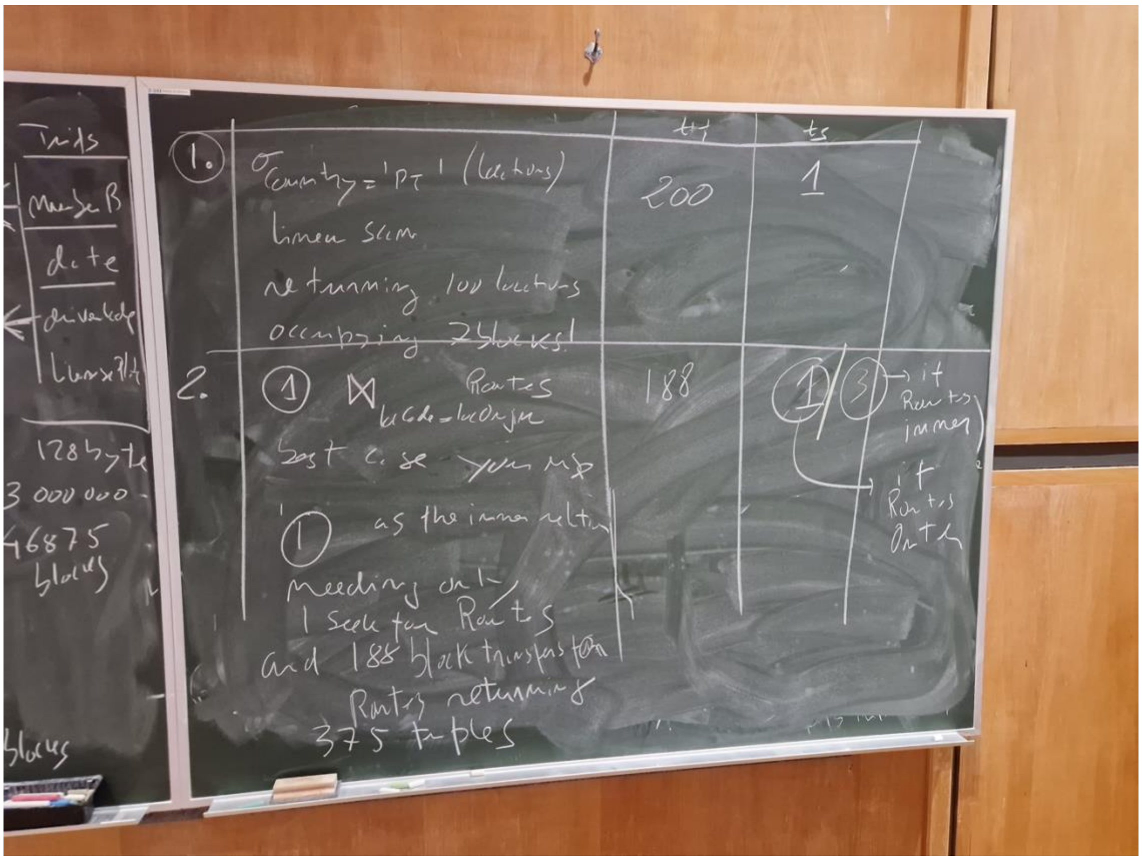
$(\lceil b_r/(M-2)\rceil * b_s \cancel{+ b_r}) * t_T + \cancel{2}*\lceil b_r / (M-2)\rceil * t_S$

= $\lceil 938/(100-2)\rceil * 200 * t_T + \lceil 938/(100-2)\rceil * t_S$

= $10 * 200 * 0{,}1 * t_S + 10* t_S$

= $210 * t_S$

1d) Let us compute the cost step by step:



Trips

[illegible]

date

[illegible]

[illegible]

128 bytes

3 000 000

46875 blocks

blocks

		t_T	t_S	
1.	$\sigma_{country='PT'}$ (lectures) linear scan returning 100 lectures occupying 7 blocks!	200	1	
2.	① $\bowtie_{lecCode=lecOrigin}$ Routes best case you use ① as the inner relation needing only 1 seek for Routes and 188 block transfers for Routes returning 375 tuples	188	① / ③	→ if Routes inner) → if Routes outer

1st step: perform a liner scan.

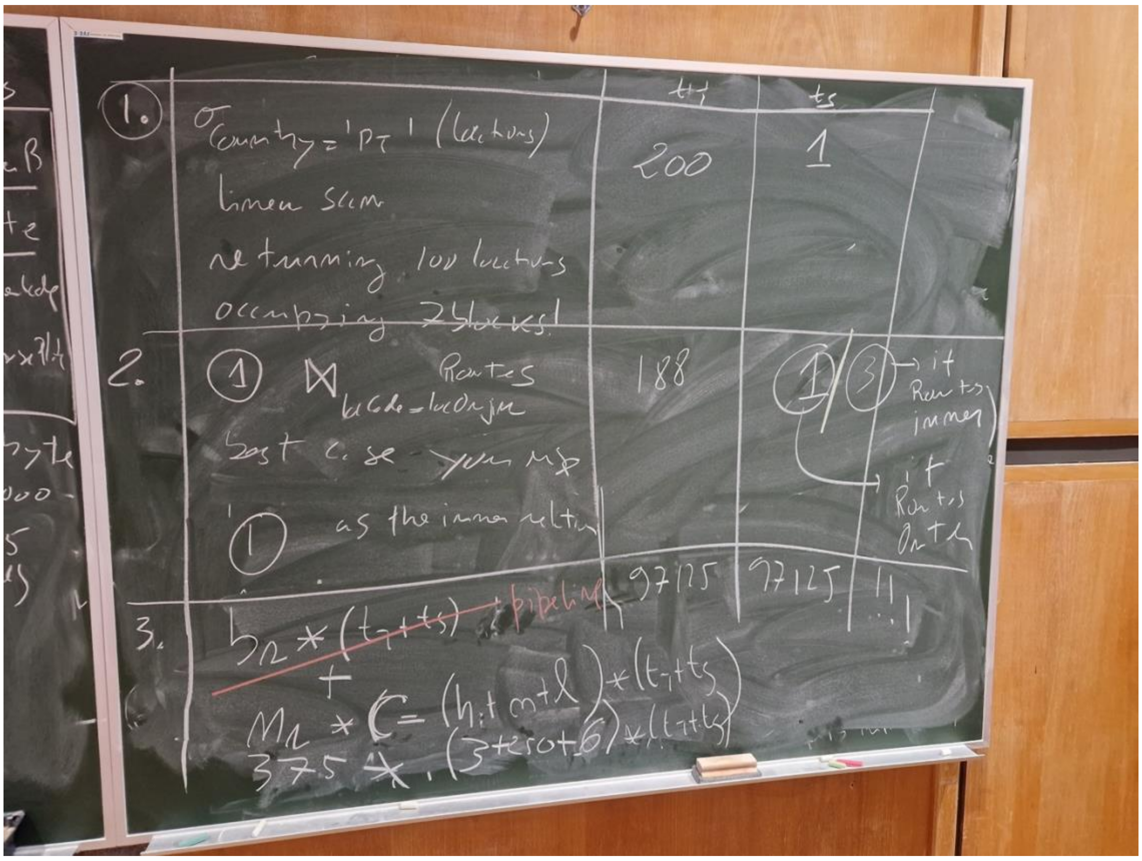
The cost is 1 seek and 200 block transfers, returning 100 results (see question), and occupying 7 disk blocks.

2nd step: perform a block nested loop join.

Since the result of the 1st step fits into memory we can use that relation as inner relation, and we do not need to read it from disk since the result of the 1st step is pipelined. Therefore the cost instead of being *br* + *bs* block transfers + 2 seeks it is reduced to *br* block transfers + 1 seek, i.e. 188 block transfers and 1 seek.

Note that if we use as inner relation the Routes relation the cost is much higher. If we do this by reading all the blocks of result of the selection, and storing them in memory, we can read use the remaining 100 – 7 – 1 (for output) blocks = 92 blocks to reduce the number of seeks resulting in 3 seeks ($\lceil 188/92 \rceil = 3$) plus 188 block transfers to read the Routes relation.

3rd step: perform a index nested loop join



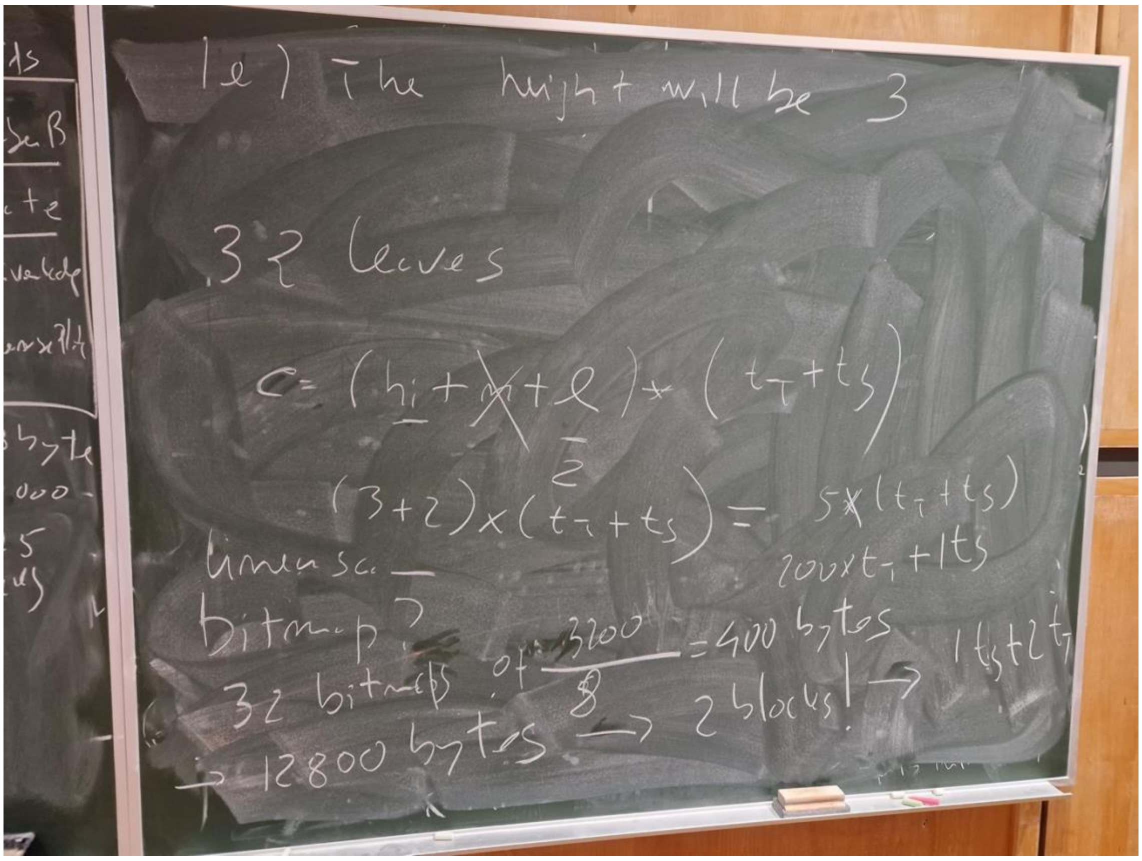
		t_T	t_S	
1.	$\sigma_{country='PT'}$ (lectors) linear scan returning 100 lectors occupying 2 blocks!	200	1	
2.	① $\bowtie_{lcCode=lecOrigin}$ Routes best case - you use ① as the inner relation	188	① / ③	→ if Routes inner; → if Routes outer
3.	~~$b_R * (t_T+t_S)$~~ pipeline + $M_R * C = (h_i + n + l) * (t_T+t_S)$ 375 * (3+250+6) * (t_T+t_S)	97125	97125	!!!

We obtain 375 results after the 2nd step (see question), and now we perform an INLJ but each lookup returns 250 results. Since the outer relation is pipelined the cost is:

$$
\begin{aligned}
& \cancel{b_r * (t_T+t_S) +}\ n_r * c_{lookup\text{-}nonkey} \\
= \quad & 375 * (h_{index\_pk\_trips}+n+l) * (t_T+t_S) \\
= \quad & 375 * (4+250+6) * (t_T+t_S) \\
= \quad & 97500 * (t_T+t_S)
\end{aligned}
$$

The height of the tree is at most $\log_{51}(3{,}000{,}000) = 4$ (this is wrong in the picture), we need to retrieve n=250 records from each lookup, and consult at most 6 leaves of the B^+-tree index to find the pointers to those 250 records (a leaf in worst-case is half full thus we may need 5 leaves + 1 extra because of alignment in blocks)

1e)



1e) The height will be 3

32 leaves

$c = (h_i + \cancel{m} + \underset{2}{\ell}) * (t_T + t_S)$

$(3+2) \times (t_T + t_S) = 5 \times (t_T + t_S)$

linear sc... $200 \times t_T + 1 t_S$

bitmap?

32 bitmaps of $\frac{3200}{8} = 400$ bytes $\quad 1 t_S + 2 t_T$

$\rightarrow 12800$ bytes $\rightarrow$ 2 blocks! $\rightarrow$

The height of the index is 3, and the index has between 32 and 64 leaves. To do the counting we do not need to retrieve the records. So, we expect to obtain 100 results for each country that at most occupy 2 leaves. We need to do the query twice (this is not in the figure), so the cost of using the index is

$2 * (3 + 2) * (t_T+t_S) = 2 * 5 * (t_T+t_S) = 10 * (t_T+t_S) = 11 * t_S$

To perform a linear scan, we spend $200 * t_T + t_S = 21 * t_S$

If we use a bitmap index we require 32 bitmaps (one for each country), each of length 3200 bits = 400 byte. So to store the index, we just require 32 * 400 / 8192 = 2 disk blocks. By using the bitmap index, the cost is reduced to $2 * t_T + 1 * t_S = 1{,}2 * t_S$ So, the ideal in this case is to use a bitmap index being 10x faster than the B^+-tree index.