1^a) Logical query optimisation has no knowledge of the algorithms that are used by the DMS in its SQL implementation, nor information about the machine it runs on. An advantage of this is that logical query implementations are portable between different machines and DBMS’ that adhere to relational algebra.

Physical query optimisations use knowledge of the algorithms in use to implement operators as well as the machine’s specs. An advantage of this is that better optimisations can be made that are tailored to the specific implementation and machine.

1b) First Join Input : orders + customers = 10,000,000 + 500,000 = 10,500,000

First Join Output : orders (because of fk between orders/customers) = 10,000,000

Second Join Input : First Join Output + nations = 10,000,000 + 193 = 10,000,193

Second Join Output : orders (because of fk between customers/nations) = 10,000,000

Groupby Input : Second Join Output \* 2 (one for read and one for aggregate function) = 10,000,000 \* 2 = 20,000,000

Groupby Output : 1 per group (distinct region) = 5

Read Result : 5

Total Function Calls = First Join Input + Second Join Input + Groupby Input + Read Result = 40,500,198

(please verify) +9000 agree

1c) assume customers/orders pre-sorted for sort merge join. Then:

We only need the following columns, all ints, this will reduce the input IO because we are using decomposed storage:

Orders: id

Customers: id, nations\_id

Nations: id, region\_id

First join is sequential IO for both sides because of sort-merge.

Accesses to read orders :-

Orders x Size(Order.customer\_id) / PageSize = 10’000’000 x 4 / 16 = 2’500’000

Accesses to read customers :-

Customers x ( Size(Customer.id) + Size(pointer to Customer.nation\_id) ) / PageSize

= 500’000 x (4 + 4) / 16 = 250’000

Accesses to read nations = |nations| \* (size(id) + size(pointer to region\_id)) / pageSize

= (193 \* (4+4) ) / 16 = 97

There are two pipeline breakers:

* Join with nations needs a hash table.   
  Table size: Nations x ( Size(pointer to Nation.id) + Size(Nation.region\_id) ) x OverAllocationFactor / PageSize =   
  193 x (4+2) x 2 / 16 = 145 pages > buffer pool, So this will generate random access IO
* Grouping, needs hash table. 5 groups, store two integers each, 2x allocation factor, 4 bytes per int: 5 x 2 x 4 x 2 / 16 = 5 pages < buffer pool, so this does not generate random IO and can be ignored

Total: read order + read customer + read nation + random IO for each tuple to join with nation = 2’500’000 + 250’000 + 97 + 10’000’000 = 12’750’097 page IO operations

(please verify)

------ Alternative solution



Total is 2.5M + 250k + 193 \* 2.5M + 193 \* 10M + 2.5M = 6,805,250,000 page faults

------

**Another Alternative (pls verify):**

Assumption: identical strings are stored out-of-place under the same reference, hence equi-joins on the nation\_id string can be computed on the reference alone, so there is no I/O cost from reading from the dictionary.

Nations scan IO = |nations| \* ( sizeof(pointer to id) + sizeof(pointer to region\_id) ) / pagesize

= 193 \* (4 + 4) / 16 = 97 page faults

Orders scan IO = |orders| \* sizeof(customer\_id) / pagesize

= 10 Mill \* 4 / 16 = 2,500,000 page faults

Customers scan IO = |customer| \* ( sizeof(id) + sizeof(pointer to nation\_id) ) / pagesize

= 500K \* (4 + 4) / 16 = 250,000 page faults

first join output = |orders| \* sizeof(pointer to nation\_id) / 16

= 10000000 \* 4 / 16 = 2,500,000 pages > 128 page buffer pool

second join output = |nations| \* sizeof(pointer to region\_id) \* over-allocation / pagesize

= 193 \* 4 \* 2 / 16 = 97 pages < 128 page buffer pool!!!!

Since it fits in the buffer pool, no I/O cost.

aggregate output = |grouping cardinality| \* ( sizeof(pointer to region\_id) + sizeof(INT count) ) / pagesize

= 5 \* (4 + 4) / 16 = 3 pages < 128 page buffer pool!!!

So, no I/O cost here either!

Total = 97 + 2,500,000 + 250,000 + 2,500,000 = 5250097 page faults

----------------------------------------------------

THE OTHER ALTERNATIVE ANSWER

***Magic happened***

2.5M is from reading Order

250K is from reading Customers

2.5M \* 2 is for the merge where you have to access the out of place varchar and so it is double

+1 is from loading left hand side of hash-join

193 \* 1.5 is from the right hand of the join and to access the out of place varchar and so it is double

2.5M is from the output of the hash join

2.5M + 250K + (((2.5M \* 2) + 1)\*193 \* 2) + 2.5M = 1.935Billion

1d) 17 is not found, takes 2 lookups (4 -> 7)

9 is not found , takes 2 lookups (4 -> 7)

4 is not found, takes 1 lookup (7)

3 is found, takes 3 lookups (6 -> 1 -> 4)

14 is found, takes 1 lookup (1)

7 is not found, takes 1 lookup (2)

**5** slots are probed (1, 2, 4, 6, 7)/\*-.

Thinks it’s **10**, doesn’t say how many unique/different slots are probed

1e)

Hash Join : Partition the hash table and compute the join on separate cores.

You could probably make the hash table probe use multiple cores. But this might interfere with the volcano pipelining?

Merge Join : You can also parallelise the merge join since you can assume the ids are sorted.

2a) Copied data vs pointers

Can only have one clustered index per table, many for unclustered

In clustered, data rows are sorted based on key vals, in unclustered the structure is separate from data rows.

2b)i) Assume no null terminator needed. Then the longest string is 6 bytes. We need to store id, name, nation: 4 + 6 + 2 bytes x 500’000 entries. 12 x 500’000 / 16 = 375’000 pages

**Alternative answer:**

(4 + 27 + 2) \* 500000 / 16 =

2bii) The strings are stored separately. So the strings for the name need 6 null terminators + 3 x len 6 + 1 x len 5 + 1 x len 4 + 1 x len 3 = 36 bytes, so 3 pages

The country codes are 9 bytes so clearly fit one page.

Customer now has id, pointer, pointer: (4 + 4 + 4) x 500’000 / 16 = 375’000 pages. We add 3 string storage pages = 375’003 pages total.

2c) Not sure if he wants decomposed or nary storage....

Assuming decomposed:

Page1: 0 0 0 9 0 0 0 8 0 0 0 1 0 0 0 5

Page2: sam000 thomas holg

Page3: FR FR UK UK FR FR DE FR

Assuming nary (which we assume for the rest of question):

Page1: 0009 Sam0 0000 0000

Page2: 0000 0000 0000 000F

Page3: R000 8Tho mas0 0000

--------------------------------------------------------------------------------------------------------------------------

ALTERNATIVE

Using n-ary, and VARCHAR(27) for name:

Page1: 0009Samxxxxxxxxx

Page2: xxxxxxxxxxxxxxxF

Page3: R0008Thomasxxxxx

Using decomposed:

Page1: 0 0 0 9 0 0 0 8 0 0 0 1 0 0 0 5

Page2: 0 0 0 10 0 0 0 4 0 0 0 6 0 0 0 2

Page3: 0 0 0 7 0 0 0 3 S a m x x x x x

--------------------------------------------------------------------------------------------------------------------------

ALTERNATIVE ALTERNATIVE (using slots)

Slots mean it must be stored n-ary, page length is 16 bytes so only need 1 byte for slot

Assuming slots at bottom of page, growing upward:

Page 1 (header indicates 1 slot/record)

[ 0 0 0 9 | S a m x | x x x x | x x x 0 ]

Page 2 (header indicates 0 slots/records)

[ x x x x | x x x x | x x x x | x x x x ]

Page 3 (header indicates 1 slot/record)

[ F R 0 0 | 0 8 T h | o m a s | x x x 2 ]

ALTERNATIVE FORMAT

<1st byte = header (tuple count), last byte(s) is the offsets to the tuples>

2 0 0 0 || 9 S a m || ‘\0’ F R ’\0’ || 0 0 11 0

1 0 8 T || h o m a || s ‘\0’ F R || ‘\0’ 0 0 12

1 0 1 H || o l g e || r ‘\0’ U K || ‘\0’ 0 0 12

2d)i) 4 customer ids per page

4 prices per page

Chance that id = 5: 1/1000. (assuming uniform distribution)

Read 2’500’000 pages looking for id=5. Probability that a page contains id=5: 4/1000. So 10’000 times we need to read another page to get the price of the order. Total: 2’520’000

2dii) Bitmap size: 1 bit corresponds to 1 entry. 1 byte -> 8 entries. There are 10 buckets per entry. So 8 entries take 10 bytes.

There are 10’000’000 orders / 8 \* 10 = 12’500’000 bytes.

12.5M/16 = 781250 page faults generated reading bitmap sequentially.

The probability that a bitmap entry is not 0 in the bucket we are interested in (1-100) is 8/1000. When this happens, we need to check in the best case 2 pages (id, order) or in the worst case 4 (id, order). So maybe just guess 3 page faults on average when we hit a non-0 entry in the bitmap (this is probably wrong).

So in total, 781250 \* 8/1000 = 6250 times that we find a bitmap entry not 0.

Then for 3 pages on average thats 18750 page faults.

Total: 781250 + 18750 = 800’000 page faults.

----------------------------------------------------------------------------------------

Page Faults for Bitmap: 781,250 (As prior)

\*\* Potential other number of Page Faults for Bitmap:  
Only look at pages for the bitvector Bin, 1-100. So:  
1bit \* 10,000,000 / 8 == 1250000 bytes, 1250000/16 = 78125 pages

Chance that an Order has Bin1 flagged: 1/10

Chance that the Order with Bin1 flagged actually has id==5: 1/100

1. We must first check the customer\_ids of all Orders with Bin1 flagged: v b

10,000,000 \* 1/10 = 1,000,000

IF ASSUME spanned:

2. Since it is N-ary storage,

For (1,000,000 \* 1/100) of those entries, we need to also read their price and orders. This might occur on 2 pages.

ASSUME 50% of the correct ones are spread across 2 pages

TOTAL:

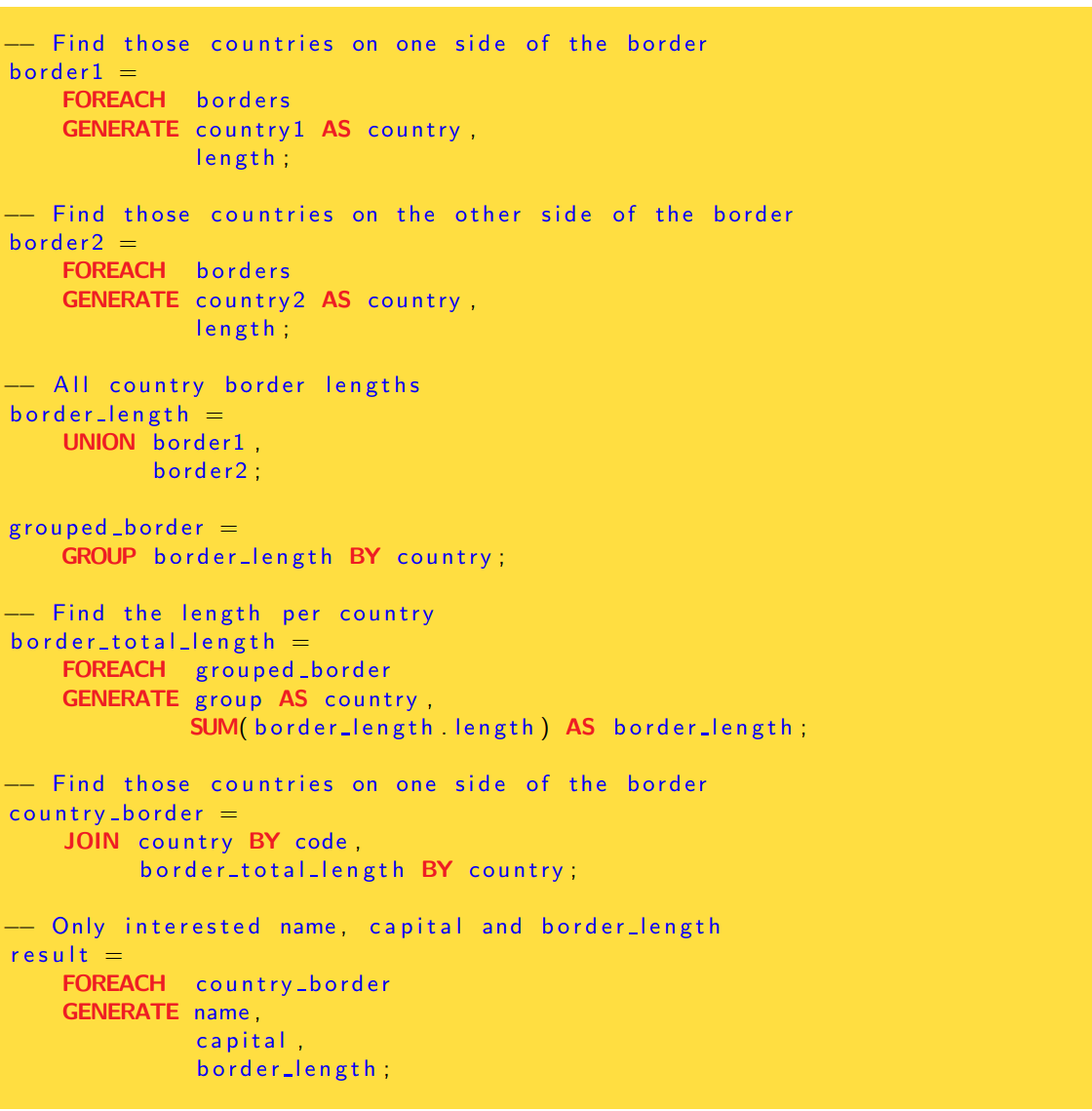
781,250 + (1,000,000 - 10,000) + 10,000\*0.5 + 10,000\*2\*0.5 = 1,786,250  
  
IF ASSUME unspanned:  
TOTAL:  
781,250 + 1,000,000= 1,781,250

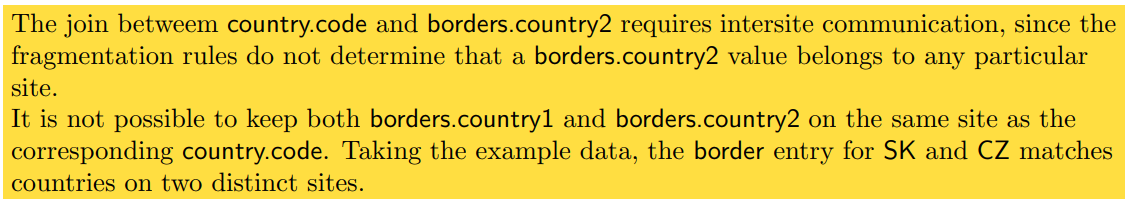
2e) Probably a B+ tree because its ea sy to maintain under updates, and allows fast access for queries like 2d) using the leaf pointer chain. But not sure.

FK Index?

Use Delta-Main?

Q3a)

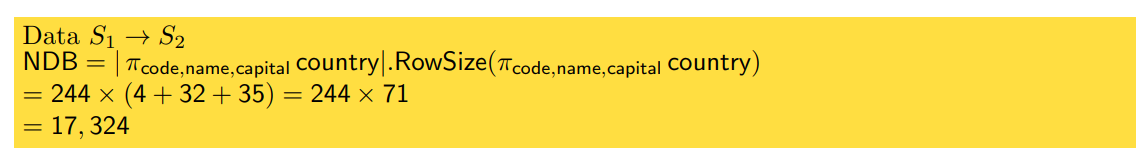


3b) 

3c)



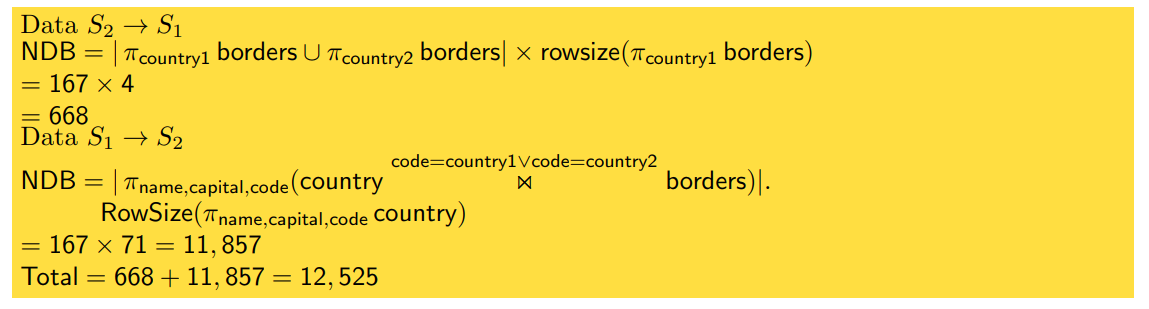
4ai



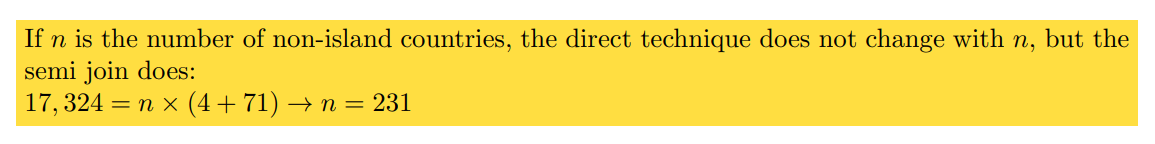
ii)



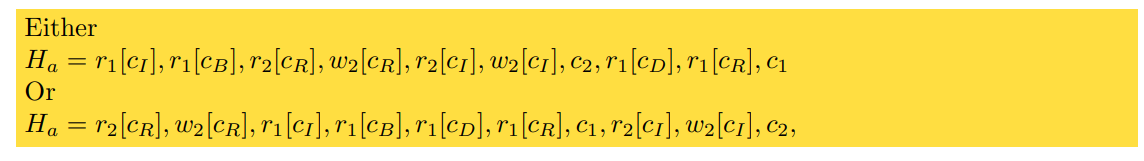
iii)



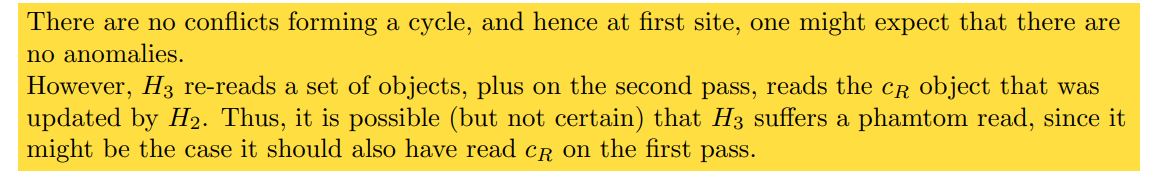
iv)

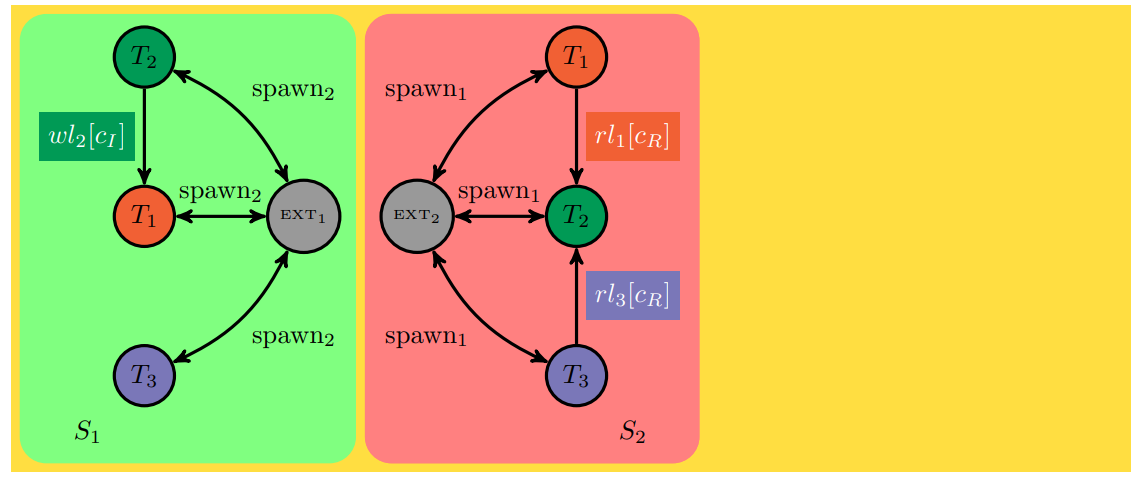


bi)

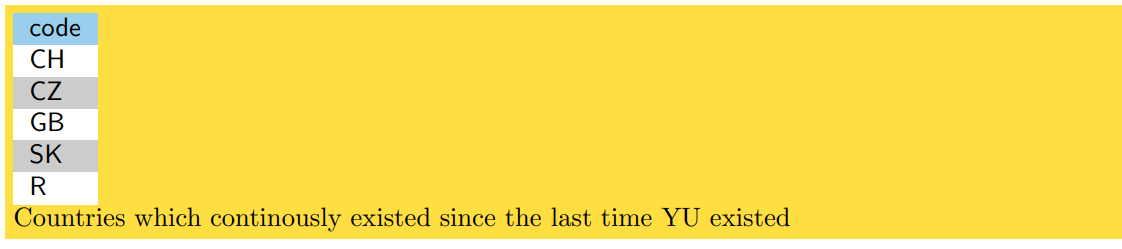


ii)



iii) 

ci)



ii)

