Data Management – exam of 12/07/2023 **(A)**

Problem 1
The *separation degree* between researcher A and researcher B (different from A) is 1 if they have collaborated, and is 2 if both of them have collaborated with a third researcher C. Let `R(A,B)` be a relation (therefore, without duplicates) storing all the tuples $\langle r, s\rangle$ such that researchers r and s have collaborated. Obviously, `R` is symmetric. We know that (i) no researcher has more than 300 collaborations, (ii) associated to `R` there is a clustering B$^+$-tree index on attribute `A` whose cost for retrieving the first tuple of `R` with a certain value for `A` is 4 page accesses, (iii) every page has space for 500 tuples of `R`, and (iv) the buffer has 100 frames available. Considering the query Q that, given a specific researcher x, computes the set (therefore, without duplicates) of researchers with whom x has separation degree less than or equal to 2, illustrate the most efficient algorithm you can think of for answering Q, and tell which is the cost of the algorithm in terms of number of page accesses.

Solution 1
We can use the index for retrieving the tuples of `R` whose first component is x, telling us which are the collaborators of x and store them in one frame F of the buffer, relying on the fact that their maximum number is 300 (500 tuples fit in one frame). We can then use the index again for each of the collaborators of x to retrieve all the collaborators of the collaborators of x, whose maximum number is $300 \times 300 = 90.000$. Since 500 tuples of two values fit in one page, we have that 1.000 values fit in one page and therefore we can store the codes of all the collaborators of x and the codes of the collaborators of the collaborators of x using $90.300/1.000 = 91$ frames in the buffer, eliminate all duplicates by working in the buffer, and then write all the pages of the result. As for the cost, we count 4 page accesses for retrieving the first collaborator of x, plus one page for retrieving, in the worst case, all the collaborators of x. Similarly, we need $(4 + 1) \times 300$ page accesses for retrieving all the collaborators of the collaborators of x. Therefore, the cost is simply 1.505 page accesses.

Problem 2
Consider the following schedule S:

$$r_1(X)\ w_4(Z)\ w_2(X)\ r_2(Y)\ w_3(Y)\ w_3(X)\ r_1(Y)\ r_4(Y)\ w_1(Z)$$

and answer (with suitable motivations) the following questions: (i) Is S conflict serializable? (ii) Is S a 2PL schedule (with shared and exclusive locks)? (iii) Can we insert the commit commands in such a way that the resulting schedule is accepted by the timestamp-based scheduler (assuming that the timestamp associated to a transaction coincides with the physical time when the transaction starts)? (iv) In all the three cases above where the answer is no, provide the answer to the following further question: is there any action α in S such that, is we delete α fron S, then the answer to the question would switch to yes?

Solution 2
(i) Since the precedence graph associated to S is cyclic, S is not conflict serializable.

(ii) Since transaction T_1 should enter the shrinking phase when $w_2(x)$ is executed, and we cannot anticipate all the future locks required by T_1 (in particular, we cannot anticipate the shared lock on Y required for $r_1(Y)$), we have that S is not a 2PL schedule with both shared and exclusive locks.

(iii) Since $r_1(Y)$ comes after $w_3(Y)$ and the timestamp of T_3 is higher than the one of T_1, we have that $r_1(Y)$ is a "read too late action". Analogously, since $r_4(Y)$ comes after $w_3(Y)$ and the timestamp of T_3 is higher than the one of T_4, we have that $r_4(Y)$ is a "read too late action". Therefore, we cannot insert the commit commands in such a way that the resulting schedule is accepted by the timestamp-based scheduler.

(iv) We analyze the three cases separately.

- Looking at the precedence graph, we can realize that all cycles involve node T_1 and deleting action $r_1(X)$ eliminates both the edge from T_1 to T_2 and the edge from T_1 to T_3, thus eliminating all cycles. So, if we delete $r_1(X)$ from S, then the answer to question (i) would switch to yes. Analogously, we could eliminate $w_3(Y)$ and have the answer switched to yes.
- It can be verified that if we delete $r_1(X)$ from S, then the answer to question (ii) would switch to yes. And the same applies to action $w_3(Y)$.

- Finally, if we delete action $w_3(Y)$, insert the commit commands in S as follows:

$$r_1(X)\ w_4(Z)\ w_2(X)\ r_2(Y)\ c_2\ \ w_3(X)\ c_3\ r_1(Y)\ r_4(Y)\ c_4\ w_1(Z)\ c_1$$

  and use the Thomas rule when executing $w_1(Z)$, the resulting schedule is accepted by the timestamp-based scheduler.

Problem 3
A schedule S is called *parsimonious* if it contains only write actions and no element of the database is used by more than two actions in S. Prove or disprove that every parsimonious schedule is conflict serializable if and only if it is view serializable.

Solution 3
As usual, we rule out the possibility that a schedule contains two different write actions on the same element issued by the same transaction. Therefore, if an element of the database is written by two actions in a parsimonious schedule S, such actions belong to different transactions. Also, since it is well known that conflict serializability implies view serializability, in order to prove the statement it is sufficient to prove that for parsimonious schedules, view serializability implies conflict serializability. We proceed by showing that if a serial schedule S' is view equivalent to a parsimonious schedule S, then S' is also conflict equivalent to S. Let S' be a serial schedule view equivalent to S. Obviously, since S is parsimonius, S' is parsimonious too, and the read-from relations for both is empty. Also, they have the same final-write set, and this allows us to show that S' is conflict equivalent to S. Indeed, consider any pair of conflicting actions $w_i(x), w_j(x)$ such that $w_j(x)$ comes after $w_i(x)$ in S. Since S is parsimonious, there is no other write action on x in S, and therefore, $w_j(x)$ is a final write in S. It follows that $w_j(x)$ is also a final write in S' (otherwise S' would not be view equivalnent to S), which means that $w_i(x), w_j(x)$ appear in the same order in S and S'. We then conclude that S' is conflict equivalent to the serial schedule S', implying that S is conflict serializable.

Problem 4
Let `R(A,B,C)` and `S(D,E)` be two relations, each stored in a heap. A tuple t of `R` is said to be a *lower k-match* with `S` if there are at least k tuples of `S` whose value in the attribute `D` is equal to the value of t in the attribute `C`. Let M be the number of buffer frames available, let Q be the query that, given a value for k, computes all the tuples in `R` that are lower k-matches with `S`, and consider the following questions.

4.1 Describe the conditions under which we can process Q using the one-pass, the two-passes, and the block-nested loop method, respectively.

4.2 For each of the methods mentioned above, illustrate the corresponding algorithm and tell which is the cost of such algorithm in terms of the number of pages of `R` and `S`.

Solution 4
In order to process Q in one pass, a possible condition is that the relation `S` fits in $M-2$ buffer frames. Under this condition the algorithms would be as follows: we load `S` in the buffer, and then use one buffer frame F for reading `R` one page at a time, and for each tuple t in F, we check whether there are at least k tuples of `S` in the buffer whose value in the attribute `D` is equal to the value of t in the attribute `C`. If yes, we write t in the output frame, otherwise, we ignore t. As usual, we write the output frame in secondary storage when full. The cost is simply $B(\texttt{S}) + B(\texttt{R})$. Note that another condition that would allow the one pass strategy is that the relation `R` fit in the buffer, but in this case we would need another field to associate to each tuple of `R` a counter and this must be taken into account in order to establish the condition under which we can apply the one pass strategy (see the paragraph below on the block-nested loop technique).

In order to process Q in two passes, a possible condition is that $B(\texttt{S}) + B(\texttt{R}) \leq M \times (M-1)$. Under this condition the algorithm would be as follows: we produce the sorted sublists (each of size M) of `R`, sorted on the sorting key `C`; we produce the sorted sublists (each of size M) of `S`, sorted on the sorting key `D`; we do the merging phase by simply checking for each tuple t of `R`, if there are at least k tuples of `S` whose value in the attribute `D` is equal to the value of t in the attribute `C`. If yes, we write t in the output frame, otherwise, we ignore t. As usual, we write the output frame in secondary storage when full. The cost is simply $3 \times (B(\mathtt{S}) + B(\mathtt{R}))$.

In order to process Q using the block nested loop technique, we just need 3 buffer frames ($M \geq 3$) and we proceed as follows: we read R in block of $M-2$ pages, by also reserving in the buffer enough space for a new integer field N storing for each tuple of R the count of joining tuples of R, we read one by one the pages of S in one frame, and while reading such pages we register, for each tuple t of R in the $M-2$ pages, in the field N associated to t the number of tuples of S joining with t. After reading all the pages of S we use the field N to figure out which are the tuples to be written in the output frame, and then we consider the next block of R. As usual, we write the output frame in secondary storage when full. If we consider the space used to store N as irrelevant, the cost is simply $B(\mathtt{R}) + B(\mathtt{R}) \times B(\mathtt{S})/(M-2)$. If we want to be more precise, then we can assume that the size of N is the same as the size of the other fields of R, so that we consider the size of the outer relation of the block nested loop algorithm $4 \times B(\mathtt{R})/3$ instead of $B(\mathtt{R})$, and obtain the total cost of $4/3 \times (B(\mathtt{R}) + B(\mathtt{R}) \times B(\mathtt{S})/(M-2))$.

Problem 5 (only for students enrolled in an A.Y. before 2021/22 who do **not** do the project)
Parallel algorithms for the implementation of relational operators are based on the idea of *horizontal data partitioning*, by which we can split a relation in various chunks, each one stored at a different node.

5.1 Describe the techniques that can be used for partitioning.

5.2 Among the described techniques, tell which is the most common one and explain why.

5.3 Illustrate the idea for designing a parallel algorithm that, given a table S, based on the partitioning technique mentioned for item 2) above, and depending on the number of buffer frames available at the various nodes, computes the relation obtained from S by eliminating duplicates. Also, discuss the cost of the algorithm.

5.4 Illustrate the idea for designing a parallel algorithm that, based on the partitioning technique mentioned for item 2) above, and depending on the number of buffer frames available at the various nodes, computes the set difference of two relations given in input, also discussing the cost of the algorithm.

Solution 5
Please, refer to the slides of the course.