Data Management – AA 2018/19
Solutions for the exam of 13/06/2019

Problem 1

It is well-known that the schedulers based on locking can recognize deadlock situations when processing schedules. Provide the definition of deadlock, and prove or disprove the following statement: *If a schedule S is not conflict serializable, then the scheduler based on locking with both shared and exclusive locks recognizes a deadlock situation when processing S.*

Solution of problem 1

Definition of deadlock: a situation in which two or more transactions are waiting indefinitely for one another to give up locks.

To prove the statement we have to show that the presence of a cycle in the precedence graph $P(S)$ associated with S implies that the scheduler based on locking with both shared and exclusive locks recognizes a deadlock situation when processing S. Conversely, to disprove the statement we have to show that there is at least one schedule S that is not conflict serializable (i.e., with a cycle in $P(S)$) such that the scheduler based on locking with both shared and exclusive locks does not recognize a deadlock situation when processing S.

One might be tempted to prove the statement by trying to prove that the presence of a cycle in $P(S)$ implies the presence of a cycle in the wait-for graph associated to S. However, although it is easy to see that the presence of a cycle in the wait-for graph associated to S implies the presence of a cycle in $P(S)$, the converse is not true. Indeed, an edge from T_i to T_j in $P(S)$ means that there are conflicting actions in the two transactions, and the one of T_i comes before the one of T_j in S, whereas an edge from T_i to T_j in the wait-for graph associated to S means that T_i is waiting for T_j to release a lock. Now, the question is whether an edge from T_i to T_j in $P(S)$ implies an edge from T_j to T_i in the wait-for graph associated to S. Probably, the simplest example showing that the implication does not hold is the following schedule non-conflict-serializable schedule S:

$$r_1(x)\, w_2(x)\, w_1(x)$$

It can be observed that the edge from T_2 to T_1 in $P(S)$ due to the pair $\langle w_2(x), w_1(x)\rangle$ in S does not correspond to an edge from T_j to T_i in the wait-for graph associated to S, simply because T_2 has not acquired the lock on x. Indeed, the scheduler grants the shared lock on x to transaction 1, and then does not unlock x, because transaction 1 asks for an upgrade later on. So, the only possibility for the scheduler is to put transaction 2 in waiting stage, and to allow transaction 1 to proceed. When transaction 1 ends, transaction 2 is resumed by the scheduler.

Since the scheduler based on locking with both shared and exclusive locks does not recognize any deadlock situation when processing the above non-conflict-serializable schedule S, the statement has been disproved.

Problem 2

Relations R and S are stored in heap files with 3.000 and 20.000 pages, respectively, and we have to compute the set union of R and S by using k frames available in the buffer, where $150 < k < 1.500$. If we measure efficiency in terms of number of page accesses, tell which of the following sentences is true, explaining the answer in detail.

2.1 For no value of k the block-nested loop algorithm is more efficient than the two-pass algorithm.

2.2 For no value of k the two-pass algorithm is more efficient than the block-nested loop algorithm.

2.3 There are values of k for which the block-nested loop algorithm is more efficient than the two-pass algorithm, and there are values of k for which the converse holds.

Solution to problem 2

In the case described by the problem, the formula for the number of page accesses needed for set union in the case of block-nested loop is $3.000 + 3.000/k \times 20.000$, whereas the formula for the two-pass algorithm (i.e., the one based on sorting) is $3 \times 23.00 = 69.0000$.

Intuitively, the lower is k, the fewer are the chances for the block-nested loop algorithm to be more efficient than the two-pass algorithm.

Indeed, and we can immediately notice that $k = 153$ is the minimum vale for which we can use the two-pass algorithm (indeed, $153 \times 152 > 23.000$) and for such value the cost of the block-nested loop algorithm is 400.351, which is higher than 69.000, which the cost of the two-pass algorithm. The value 153 is therefore a value of k for which the two-pass algorithm is more efficient than the block-nested loop algorithm.

On the other hand, all values h of k such that $3.000 + 3.000/h \times 20.000 < 69.000$ correspond to cases where the block-nested loop algorithm is more efficient than the two-pass algorithm. The minimum of such values is 910, which is a value for k allowing the execution of the two-pass algorithm (because $23.000 < 910 \times 909$). Indeed, for $k = 910$ the cost of the block-nested loop is $3.000 + 3.000/910 \times 20.000 = 68.935$, that is lower than 69.000. Note that for $k = 1.499$ the cost of the block-nested loop is $3.000 + 3.000/1.499 \times 20.000 = 43.027$. So, for all values $910 <= k < 1.500$ the block-nested loop algorithm is more efficient than the two-pass algorithm.

So, we conclude that sentences 2.1 and 2.2 are false, whereas sentence 2.3 is true.

Problem 3

Let R and S be two relations having A as common attribute. The *semijoin* between R and S, denoted as $R \ltimes S$, is defined as follows $R \ltimes S = \{\, t \in R \mid t.A \in S[A] \,\}$. Suppose that we have to compute the semijoin between R and S.

  3.1 Can we use the block-nested loop technique?

  3.2 Can we adapt the simple sort-based join technique?

  3.3 Can we adapt the sort-merge join technique?

In all the above three cases, if the answer is negative, then motivate the answer in detail, otherwise describe an appropriate algorithm for computing the semijoin between R and S using the corresponding technique. Also, among the three techniques, tell which is, in general, the best option in terms of efficiency, motivating the answer in detail.

Solution of problem 3

In what follows, we let k to be the number of buffer frames available, and we let B_R and B_S to be the number of pages of R and S, respectively. Also, we assume $B_R \leq B_S$.

  3.1 We can obviously use the block-nested loop technique for computing the semijoin of R and S, as follows. Read R in blocks of $k - 2$ pages, and for each page in the block, use one frame for reading the pages of S, and one frame for the output. When block b and page p of S are in the bufffer, consider every tuple of R appearing in b. For each such tuple t, check whether $t.A$ appears in the attribute A of the tuples of S in p. If the check in negative, then ignore t. If the check is positive, then put t in the output frame. As usual, when the output frame is full, write it in secondary storage.

  3.2 We can adapt the simple sort-based join technique for computing the semijoin of R and S, as follows. Sort R and S on attribute A using the multipass sorting algorithm, and then perform a "merge" step of the sorted relations, using two buffer frames of reading the two relations, and one buffer frame for the output. During the "merge" step, for every tuple t of R appearing in the page of R under consideration, we put t in the output frame if and only if $t.A$ appears in $S[A]$ As usual, when the output frame is full, write it in secondary storage. Note that the problem arising for the join of having too many tuples to consider simultaneously in the buffer

does not occur, because we just need to know if $t.A$ appears in $S[A]$, and we do not need to combine t with all the tuples of S "joining" t.

3.3 We can also adapt the sort-merge join technique for computing the semijoin of R and S, as follows. Using as many passes as needed, build the $k-1$ sorted sublists for R and S, sorted on attribute A, where k is the number of buffer frames available. The perform the "merge" step using $k-1$ frames for reading R and S, and one frame for the output. During the "merge" step, for every tuple t of R appearing in the pages of R under consideration, we put t in the output frame if and only there is a tuple in the pages for S having $t.A$ in the attribute A. As usual, when the output frame is full, write it in secondary storage. Again, note that the problem arising for the join of having too many tuples to consider simultaneously in the buffer does not occur, because we just need to know if $t.A$ appears in $S[A]$, and we do not need to combine t with all the tuples of S "joining" t.

The cost of the algorithm based on block-nested loop strategy is $B_R + B_R/k \times B_S$. The cost of the algorithms based on the simple sort-based join strategy can be characterized as $2 \times h_1 \times (B_R + B_s) + B_R + B_S$, where h_1 is the minimum number of passes needed to sort R and S using k buffer frames. The cost of the the algorithms based on the sort-merge join strategy is $2 \times h_2 \times (B_R + B_s) + B_R + B_S$, where h_2 is the minimum number of passes needed to produce the $k-1$ sorted sublists of R and S using k buffer frames. Since the problem of having too many tuples to consider simultaneously in the buffer does not occur, the technique that adapts the sort-merge join algorithm is clearly the best option in terms of efficiency.

.

Problem 4
Consider the following schedule

$$S = r_1(x)\, w_3(x)\, w_3(z)\, c_3\, w_2(x)\, w_2(y)\, c_2\, r_4(x)\, w_1(v)\, c_1 r_4(v)\, c_4.$$

4.1 Tell whether S is view-serializable or not, explaining the answer in detail.

4.2 Tell whether S is a 2PL schedule with both shared and exclusive locks or not, explaining the answer in detail.

4.3 Describe the behavior of the timestamp-based scheduler when processing the schedule S, assuming that, initially, for each element α of the database, we have rts(α)=wts(α)=wts-c(α)=0, and cb(α)=`true`, and assuming that the subscript of each action denotes the timestamp of the transaction executing such action.

4.4 Tell whether S is strict or not, and whether S is rigorous or not, explaining the answer in detail.

Solution of problem 4
It is immediate to verify that the precedence graph associated to S is acyclic, and thus S is conflict-serializable, and therefore also view serializable. It is also easy to see that S is a 2PL schedule with both shared and exclusive locks. Also, it is possiblr to verify the behavior of the timestamp-based scheduler when processing the schedule S, assuming that, initially, for each element α of the database, we have rts(α)=wts(α)=wts-c(α)=0, and cb(α)=`true`, and assuming that the subscript of each action denotes the timestamp of the transaction executing such action. The only notable thing is that $w_2(x)$ is ignored by means of the Thomas rule. Finally, one can easily see that S is strict, but not rigorous (indeed, the first two actions $r_1(x), w_3(x)$ are conflicting, and there is no commit action of transaction 1 between them.

.

Problem 5
Assume that the relation Player(code,name,yearOfBirth,prizelevel) (where code is a key), has 325.000 tuples, each attribute and each pointer in the system occupies 100 Bytes, each page has space for

2.000 Bytes, the values for **prizelevel** are in the range [1..1000], equally distributed among the various tuples of the relation, and the most frequent query on **Player** asks for all players whose **prizelevel** falls into a given range.

5.1 Tell which method would you use to represent the relation in secondary storage.

5.2 Tell which logical plan and physical plan (assume that 40 free buffer frames are available) would you use for the following query Q:

       select t.code, t.yearOfBirth, t.prizelevel

       from (select code, yearOfBirth, prizelevel from Player order by yearOfBirth) as t

       where t.prizelevel $>=$ 925 and t.prizelevel $<$ 950

5.3 Tell which is the cost of executing the physical query plan you have defined for item 5.2 in terms of the number of page accesses.
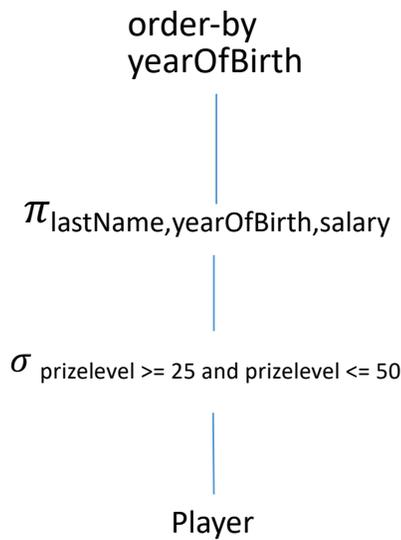
Solution of problem 5

The logical plan associated to the query code is as follows.

$$\pi_{\text{code,yearOfBirth,prizelevel}}$$

$$\text{order-by}$$
$$\text{yearOfBirth}$$

$$\sigma_{\text{prizelevel} >= 25 \text{ and prizelevel} <= 50}$$
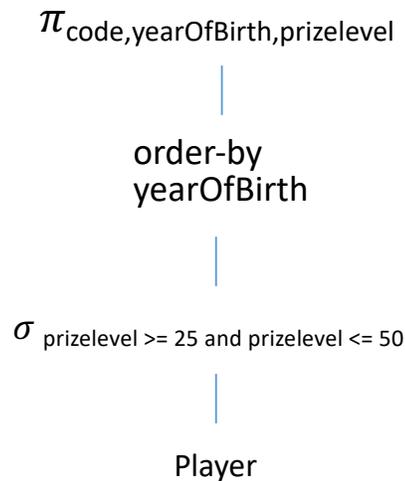
$$\text{Player}$$

Since the most frequent query on **Player** asks for all players whose **prizelevel** falls into a given range, we focus our attention on the method that stores the relation in a sorted file with sorting key **prizelevel**, with an associated clustering, sparse tree-based index on search key **prizelevel**. Indeed, it is well known that range queries are well supported by a clustering B$^+$-tree index. Having the clustering B$^+$-tree index, we can use the index for the selection operator, thus finding the first page of **Player** with the tuples satisfying the where condition. We then exploit the fact that the index is clustering, and sort on **yearOfBirth** all pages with tuples satisfying the condition, so as to get the final result. Note that we do not materialize the result of the selection operator. Rather, a pipeline approach is used to pass the result of the selection operator to the sorting operator. Note that it is advantageous to push projection under the order-by clause, because the size of the operand will decrease. Note also that we cannot push projection under selection, because otherwise we cannot use the index.

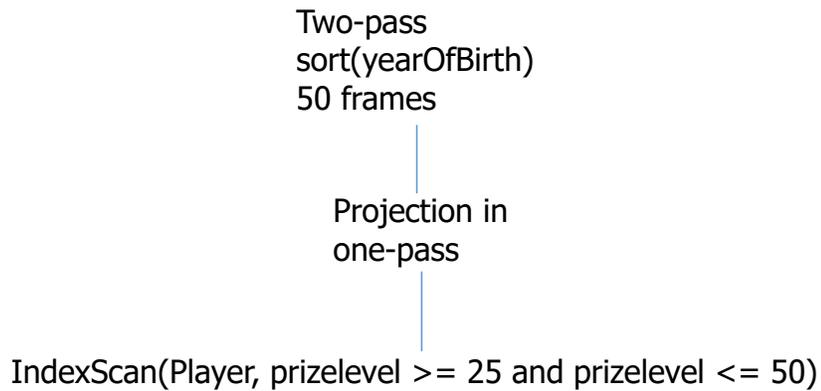The logical plan associated to the query is now as follows.

order-by
yearOfBirth

|

$\pi_{\text{lastName,yearOfBirth,salary}}$

|

$\sigma_{\text{prizelevel >= 25 and prizelevel <= 50}}$

|

Player

Let us now turn our attention to the physical query plan.

Algorithm for selection. Since each value or pointer occupies 100 Bytes, it follows that every tuple has 4 values, and since each page has space for 2.000 Byte, we conclude that every tuple occupies 400 Bytes, which means that each page has space for 5 tuples of Player, and the relation is stored in 65.000 pages. Also, we have that each data entry requires 200 Bytes, and therefore each page has space for 10 data entries. Taking into account the 67% occupancy rule for the leaves of the tree-based index, we have that each leaf contains 7 data entries. Also, since each page has space for 10 index entries, we can assume the value $(10 + 5)/2 = 7$ for the fan-out. Note that, since the index is sparse, it stores one value of prizelevel for each page of Player, and therefore the number of leaves is 65.000 / 7 = 9.286. It follows that reaching the right leaf requires $\log_7 9.286 = 5$ page accesses.

Algorithm for projection. After reaching the right leaf, we have to compute the projection of 3 over 4 attributes of all the tuples satisfying the condition on prizelevel. Since we have to access the tuples with prizelevel in the given range, and we know that the values in prizelevel are equally distributed on the tuples of Players, the number of qualifying records are $(325.000/1000) \times 25 = 8.125$, stored in $8.125/5 + 1 = 1.626$ pages. We then read such 1.626 pages, and, since we are interested in 3/4 of the attributes, we count $(3/4) \times 1.626 = 1.220$ pages to pass to the sorting operator.

Sorting. The records of such pages must be sorted having 40 buffer frames available. Since $1.220 < 40 \times 39$ (i.e., $1.220 < 1.560$), we can use the two-pass sorting algorithm. The additional cost for sorting is therefore the cost of reading the 1.220 pages, sort them in blocks, and writing the sorted sublists, i.e., $2 \times 1.220 = 3.660$ page accesses. The total cost is $5 + 1.626 + 3.660 = 4.071$.

The resulting physical plan is as follows.

```
                    Two-pass
                    sort(yearOfBirth)
                    50 frames
                         |
                         |
                    Projection in
                    one-pass
                         |
                         |
    IndexScan(Player, prizelevel >= 25 and prizelevel <= 50)
```

Notice that if we had not pushed projection under order-by, we could not have used the two-pass algorithm for sorting, because $1.626 > 1.560$.

One final observation about the order-by operator. One might argue that the semantics of the query does not really require to apply the order-by operator in the inline view in the **from** clause, because the application of the selection and the projection operator does not force to maintain the sorting. Indeed, the solution where the order-by operator is not applied is also acceptable as a correct solution to the exercise. In this case, the cost would be simply $5 + 1.626$.