Data Management – AA 2018/19
Solutions for the exam of 12/07/2019

Problem 1

A *transition point* in a schedule S over transactions $T_1, \ldots, T_n$ is a sequence of two consecutive actions a_i, b_j in S such that a_i is a read or a write action of transaction T_i, b_j is a read or a write action of transaction T_j with $i \neq j$, and a_i is not the final read or write action of T_i. A schedule is said to be *quasi-serial*, if it contains exactly one transition point. By referring to the transactions $T_1 = r_1(x)\, w_1(y)$, $T_2 = w_2(x)\, w_2(y)$, and $T_3 = w_3(x)\, w_3(y)$, prove or disprove the following statements, motivating each of the answers in detail.

1.1 There is no quasi-serial schedule on transactions T_1, T_2, T_3 that is conflict-serializable.

1.2 There is no quasi-serial schedule on transactions T_1, T_2, T_3 that is view-serializable.

1.3 There is no quasi-serial schedule on transactions T_1, T_2, T_3 that is strict.

1.4 There is no quasi-serial schedule on transactions T_1, T_2, T_3 that is rigorous.

Solution to problem 1

1.1 The statement can be proven by observing that, since any schedule S on transactions T_1, T_2, T_3 that is quasi-serial has exactly one transition point, it contains two consecutive actions a_i, b_j such that a_i is a read or a write action of transaction T_i, b_i is a read or a write action of transaction T_j with $i \neq j$, and a_i is not the final read or write action of T_i. Now, let us consider all possible cases of transition point in S. If the transition point in S involves T_1 and T_2, then S contains the sequence $r_1(x)\, w_2(x)\, w_2(y)\, w_1(y)$ or the sequence $w_2(x)\, r_1(x)\, w_1(y)\, w_2(y)$, and therefore in both cases there is an edge from T_1 to T_2 and an edge from T_2 and T_1 in the precedence graph. Analogously, it can be seen that if the transition point in S involves T_1 and T_3, then there is an edge from T_1 to T_3 and an edge from T_3 and T_1 in the precedence graph, and, finally, if the transition point in S involves T_2 and T_3, then there is an edge from T_2 to T_3 and an edge from T_3 and T_2 in the precedence graph. In all cases, the precedence graph associated to S contains a cycle, and therefore S is not conflict-serializable.

1.2 The statement can be disproven by considering the following quasi-serial schedule

$$r_1(x)\, w_2(x)\, w_2(y)\, w_1(y)\, w_3(x)\, w_3(y)$$

and observing that it is view-equivalent to the serial schedule $r_1(x)\, w_1(y)\, w_2(x)\, w_2(y)\, w_3(x)\, w_3(y)$.

1.3 The statement can be disproven by observing that the above mentioned quasi-serial schedule $r_1(x)\, w_2(x)\, w_2(y)\, w_1(y)\, w_3(x)\, w_3(y)$ is strict.

1.3 We know that every rigorous schedule is conflict-serializable, and therefore the existence of a rigorous quasi-serial schedule on transactions T_1, T_2, T_3 would imply the existence of a quasi-serial schedule on transactions T_1, T_2, T_3 that is conflict-serializable. However, we showed in 1.1 that no quasi-serial schedule on transactions T_1, T_2, T_3 exists that is conflict-serializable. This proves that there is no quasi-serial schedule on transactions T_1, T_2, T_3 that is rigorous.

Problem 2

The relation `Student(id,school,genre,grade,year)` contains information about the final grades of a set of high-school students in the last 30 years. We know that there are 12.000 schools, each school has 200 students per year taking the final exam, 50% of the students are female, and 100 tuples of the relation fit in one page. The relation is stored in node 1 of a distributed system with 30 nodes in total, where each node has 420 frames available in its buffer. Consider the query Q shown on the right, and tell both which is the algorithm you would use to answer the query, and its cost in terms of number of page accesses.

Query Q:

```
select school, year, avg(grade)
from Student
where genre = 'female'
group by school, year
order by school
```

Solution to problem 2

Obviously, one possibility is to evaluate the query at one node, say node 1. Note that the number of tuples corresponding to the groups computed by the "group-by" is $12.000 \times 30 = 360.000$ (number of schools times the number of years), each with three values. Since 100 tuples with 6 values fit in one page, we infer that 200 tuples with 3 values fit in one page, and therefore, we need $360.000/200 = 1.800$ pages to store the information about the tuples constituting the result. Since the buffer has 420 frames available at node 1, we conclude that we cannot answer the query in one pass if we just compute the answer at node 1.

Let us therefore consider the algorithm based on the parallel evaluation of the query, and see if we can evaluate the query in one pass using this method. In node 1 we read the $(12.000 \times 200 \times 30)/100 = 720.000$ pages of `Student` using one buffer frame, and for each tuple with `genre = 'female'` of each of such pages, we use a hash function based on `school,year`, so as to compute the bucket (among the 30 buckets, each one corresponding to one node) where the tuple should go. Obviously, we do that using 30 frames in the buffer for the buckets, and when a frame is full, we send its content to the corresponding node. We assume that the hash function distributes uniformly the tuples in the 30 buckets. When a page arrives at node i, every tuple of the page is analyzed, and the group to which the tuple belongs is created (if it is the first), or updated so as to store the sum of the grades of the students belonging to that school and that year. We rely on the fact that, by virtue of the hash function, the tuples belonging to the same group are sent to the same bucket (i.e., to the same node), and we also keep the tuples arrived at each node sorted according to `school`. At the end of this process, each node has a subset of the final result, sorted by `school`.

Note that at each node we need $1.800/30 = 60$ pages to store the tuples corresponding to the groups. Since each node has 420 buffer frames available, we conclude that we can store the tuples corresponding to the groups in the buffer of each node, and also carry out the sorting on school of such tuples in the buffer. Note that this is true also for node 1, where one frame is used for reading the relation and filter the students, 30 frames are used for the buckets, and the remaining frames (much more than 60) are available for the tuples corresponding to the groups to be stored in node 1.

The last step aims at sorting the whole result. Since the tuples at each node are already sorted, it is sufficient to send all the tuples to one node, say node 1 again, and execute the merge step on the various tuples so as produce the final result. The final merge can again be done using the buffer, in particular the 420 buffer frames at node 1. Actually, besides the output frame we only need 30 frames again: the pages sent by node i will be received in the frame i of node 1, for $i = 1..30$. The result of the merge will be written in a file at node 1.

It follows that, if we ignore, as usual, both the cost of writing the final result, and the communication cost for sending the tuples, the overall cost is simply the one of reading the relation `Student`, i.e., 720.000 page accesses.

Problem 3

The *quotient* between two relations, denoted by ÷, is a binary operator of the extended relational algebra, defined as follows: if `R(A,B,C,D)` and `S(C,D)` are two relations, the result of `R` ÷ `S` is the set

$$\{\ \langle a,b\rangle\ \in\ \texttt{R[A,B]}\mid \forall c,d:\ \langle c,d\rangle\ \in\ \texttt{S}\ \rightarrow\ \langle a,b,c,d\rangle\ \in\ \texttt{R}\ \}$$

In other words, `R` ÷ `S` contains all the tuples of `R[A,B]`, i.e., the projection of `R` on attributes `A,B`, that are combined in `R` with all the tuples of `S`. Let `R` and `S` be stored as heap files with 160.000 pages, and 450 pages, respectively, and let the buffer have 500 frames available. Describe the algorithm you would use to compute the result of `R` ÷ `S`, and tell which is the cost of the algorithm in terms of number of page accesses.

Solution to problem 3

Probably the first technique that may come to our mind is the block-nested loop. We immediately note that the smallest relation, i.e., `S`, fits in the buffer, and therefore can act as the outer relation in the bock-nested loop algorithm. So, one could think of storing the whole relation `S` in the buffer, and then reading `R` one page at a time to check whether its tuples (or rather, the tuples in the projection `R[A,B]` of `R` on attributes `A,B`) belong to the result. However, it is not clear how we can check whether one tuple $\langle a,b\rangle$ in such projection is included or not in the result, because this check amounts to verify whether for every tuple $\langle c,d\rangle$ in `S`, the tuple $\langle a,b,c,d\rangle$ is in `R`, and we do not have all the tuples of `R` in the buffer. We observe that such check would be facilitated if the relation `R` was sorted with sorting key `A,B`. Indeed, in this case, it is sufficient to count the number N of tuples of `S` when loading them into the buffer, then load one page at a time the relation `R`, and for each pair $\langle a,b\rangle$ in `R[A,B]`, count the number $N_{a,b}$ of tuples $\langle c,d\rangle$ in `S` such that $\langle a,b,c,d\rangle \in$ `R`. The tuple $\langle a,b\rangle$ will be in the result of the quotient (and therefore will be put in the frame of the buffer devoted to the output) if and only if $N_{a,b} = N$. Note that, since `R` is sorted with sorting key `A,B`, all the tuples with $\langle a,b\rangle$ in the attributes `A,B` are stored contiguously in `R`. It follows that the whole algorithm is as follows:

- Sort the relation `R` on search key `A,B`.
- Load the relation `S` in the buffer, at the same time counting the number N of its tuples.
- Load the relation `R` previously sorted one page at a time, and for each pair $\langle a,b\rangle$ in `R[A,B]`, count the number $N_{a,b}$ of $\langle c,d\rangle$ such that $\langle a,b,c,d\rangle \in$ `R`, by exploiting the fact that all such tuples are contigous. If $N_{a,b} = N$, then put the pair $\langle a,b\rangle$ in the output frame. As usual, write the output frame into secondary storage whenever it is full.

As for the cost of the above algorithm, we observe the following.

- To sort the relation `R` we can use the two-pass algorithm because we have 500 buffer frames available, and $500 \times 500 > 160.000$, and therefore we spend $3 \times 160.000 = 480.000$ page accesses, plus 160.000 page accesses to write the sorted relation.
- To load the relation `S` in the buffer, and count the number of its tuples we spend 450 page accesses.
- To load the relation `R` previously sorted one page at a time, we spend 160.000 page accesses.

The total cost will be $480.000 + 160.000 + 450 + 160.000 = 800.450$ page acceses.

Problem 4

Consider the following schedule on transactions $\{T_1, T_2, T_3\}$:

$$S = r_1(x)\, w_3(x)\, r_2(x)\, r_1(y)\, w_1(x)\, r_2(y)\, w_2(y)\, w_3(y)$$

4.1 Show the precedence graph associated to S, and tell whether S is conflict-serializable or not, explaining the answer in detail.

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

4.3 If the answer to 4.2 is positive, then show a serial schedule on transactions $\{T_1, T_2, T_3\}$ that is view-equivalent to S. If the answer to 4.2 is negative, then tell whether there is a way to add to S suitable operations on local variables carried out by the transactions $\{T_1, T_2, T_3\}$ in such a way that the resulting schedule is serializable. We remind the student that when we consider the operations on local variables carried out by transactions, the read and write actions are expressed in the notation $r_i(x, Z)$ – meaning that the value of the database element x is read by transaction i and then stored in the local variable Z – and $w_i(x, Z)$ – meaning that the value of the local variable Z is written by transaction i in x.

4.4 Tell whether S is ACR (Avoiding Cascading Rollbacks) or not, explaining the answer in detail.

Solution to problem 4

4.1 The pair of conflicting actions $w_3(x), r_2(x)$ and the pairs of conflicting actions $r_2(y), w_3(y)$ that appear in S imply that there is a cycle in the precedence graph associated to S. It follows that S is not conflict-serializable.

4.2 Since $w_3(y)$ is the final write on y in S, and $w_2(y)$ is in S, we observe that T_3 must appear after T_2 in every serial schedule S' on T_1, T_2, T_3 that is view-equivalent to S, because otherwise S' would have a different "final write se" with respect to S. On the other hand, the sequence $w_3(x)\, r_2(x)$ in S implies that in any serial schedule S' on T_1, T_2, T_3 that is view-equivalent to S transaction T_2 appears immediately after transaction T_3, because otherwise S' would have a different "read from" relation with respect to S. We conclude that no serial schedule on T_1, T_2, T_3 exists that is view-equivalent to S, i.e., S is not view-serializable.

4.3 It is well-known that a schedule S on $T_1, T_2, \ldots, T_n$ is serializable if there exists a serial schedule on $T_1, T_2, \ldots, T_n$ that is *equivalent* to S, where two schedules S_1 and S_2 on the the same set of transactions are *equivalent* if, for each database state D, the execution of S_1 starting from the database state D produces the same outcome (i.e., leaves the database in the same state) as the execution of S_2 starting from the same database state D. It follows that there is a simple method to show that there is a way to add to S suitable operations on local variables carried out by the transactions T_1, T_2, T_3 in such a way that the resulting schedule is serializable: it is sufficient to insert operations on local variables in such a way that any write operations in S will write the same value (e.g., the value 1) on the corresponding database element. Following this idea, S is transformed into the following schedule S':
$r_1(x, z_{1,0}), z_{3,1} := 1, w_3(x, z_{3,1}), r_2(x, z_{2,0}), r_1(y, z_{1,1}), z_{1,2} := 1, w_1(x, z_{1,2}), r_2(y, z_{2,1}), z_{2,2} := 1,$
$w_2(y, z_{2,2}), z_{3,2} := 1, w_3(y, z_{3,2})$
that leaves the database in a state where the value of both the element x and the element y of the database is 1. It is not difficult to see that any serial schedule built on the transactions of S' is equivalent to S'. For example, the following serial schedule S'':
$r_1(x, z_{1,0}), r_1(y, z_{1,1}), z_{1,2} := 1, w_1(x, z_{1,2}),$
$r_2(x, z_{2,0}), r_2(y, z_{2,1}), z_{2,2} := 1, w_2(y, z_{2,2}),$
$z_{3,1} := 1, w_3(x, z_{3,1}), , z_{3,2} := 1, w_3(y, z_{3,2})$
buolt on the actions of S' leaves the database in a state where the value of both the element x and the element y of the database is 1, and therefore is equivalent to S'. Since there is a serial schedule that is equivalent oto S', we can conclude that S' is serializable.

4.4 S is clearly not ACR, because $r_2(x)$ reads element x from transaction T_3 (action $w_3(x)$) while transaction T_3 is not committed yet (T_3 cannot commit until the end, which coincides with action $w_3(y)$).

.

Problem 5

Assume that the relation R(A,B,C,E,F,G) has 960.000 tuples, each attribute and each pointer in the system occupy 40 Bytes, each page has space for 240 Bytes, B has 1.000 values uniformly distributed among the tuples of R, and there is an unclustering B$^+$-tree index on R with search key B using alternative 2. Consider the query Q_2 shown on the right, describe the algorithm you would use to compute the result of evaluating the query, and tell which is the cost of the algorithm in terms of number of page accesses.

Query Q_2:
select A,E
from R
where B = 10 and C = 20

Solution of problem 5

The query is so simple that we can avoid to show the logical and the physical plan. So, we turn pur attention directly to the algorithm. We rely on an index-based algorithm for implementing the "selection" operator of relational algebra. We use the index on the condition B = 10 for reaching the "right" leaf of the tree, and then we scan all the leaves with the value 10 of the search key, and follow the corresponding pointers to retrieve the data records and select only those satisfying the condition C = 20.

Let us evaluate the cost of the algorithm. The B$^+$-tree index is unclustering, and therefore dense. So, the number of search key values that are stored in the leaves of the B$^+$-tree index is 960.000. Each data entry occupies 80 Bytes, and therefore 3 data entries fits in one page, and the fan-out of the tree is 2. Taking into account the 67% occupancy rule, we know that 2 data entries are stored in each leaf of the B$^+$-tree index. It follows that the tree has 960.000 / 3 = 320.000 leaves. This means that the cost of reaching the right leaf is $log_2\, 320.000 = 19$. Once we have reached the right leaf, we still have to scan all the leaves containing the data entries associated to the search key value 10. The number of such data entries is equal to the number of data records satisying teh condition B = 10, i.e., 960.000/1.000 = 960; they are packed into 960/2 = 480 pages. Finally, we have to follow the 960 pointers and access the corresponding pages of the data file. Therefore, the total cost is 19 + 480 + 960 = 1459.