Data Management – solutions of the exam of 31/01/2013

Problem 1 An *extended schedule* is a schedule expressed in an extended notation, that allows the schedule to contain assignment statements involving local variables, and enriched read and write statements of the form $read_i(B,x)$ and $write_i(B,x)$, where the statement $read_i(B,x)$ means that transaction i reads the database element B and stores its value in the local variable x, and the statement $write_i(B,x)$ means that transaction i writes the value of the local variable x in the database element B. Consider the following extended schedule S:

$$v_1 := 0; read_1(A,w_1);\ w_1 := v_1 * w_1;\ read_2(A,w_2);\ w_2 := w_2 - w_2;\ write_2(A,w_2);\ write_1(A,w_1);.$$

1.1 Tell whether S is serializable or not. Explain the answer in detail.

1.2 Tell whether S is view-serializable or not. Explain the answer in detail.

Solution 1

1.1 We remind the reader that two schedule S_1 and S_2 on the same transactions are said to be equivalent if, for each database state s_0, the effect of executing S_1 starting from s_0 is the same as the effect of executing S_2 starting from s_0. Also, a schedule S is said to be serializable if there is a serial schedule on the same transactions that is equivalent to S.

It is easy to see that S is serializable. Indeed, consider the serial schedule S' where transaction T_1 is followed by transaction T_2:

$$v_1 := 0; read_1(A,w_1);\ w_1 := v_1 * w_1;\ write_1(A,w_1);$$
$$read_2(A,w_2);\ w_2 := w_2 - w_2;\ write_2(A,w_2);$$

No matter which is the initial state s_0 of the database, the effect of the execution of S on s_0 is the same as the effect of the execution of S' on s_0. Indeed, after both executions, the value of A in the database, as well as the value of each local variable is 0.

1.2 S is clearly not view-serializable, because S has an empty "read-from" relation, whereas both serial schedules T_1, T_2 and T_2, T_1 have a non-empty "read-from" relation (in the schedule T_1, T_2, $read_2(A,w_2)$ reads from $write_1(A,w_1)$, whereas in the schedule T_2, T_1, $read_1(A,w_1)$ reads from $write_2(A,w_2)$).

Problem 2 A *1-write schedule* is a schedule containing exactly one write action. Give the definition of view-serializable schedule, and, using only such definition, prove or disprove that every 1-write schedule is view-serializable.

Solution 2 A schedule is view-serializable if it is view-equivalent to a serial schedule on the same transactions, where two schedules are view-equivalent if the have the same final-write set, and the same read-from relation. By using only the above definition, we now prove that every 1-write schedule is view-serializable.

Let S be a 1-write schedule, and let $w_i(X)$ the only write action in S. Now, let S' be the serial schedule

$$T_{j_1}, T_{j_2}, \ldots, T_{j_k}, T_i, T_{h_1}, \ldots, T_{h_m}$$

where $T_{j_1}, T_{j_2}, \ldots, T_{j_k}$ is a sequence (in any order) of all the transactions different from T_i that do not read from $w_i(X)$ in S, and $T_{h_1}, \ldots, T_{h_m}$ is a sequence (in any order) of all the transactions reading from $w_i(X)$ in S. In other words, S' contains first all the transactions (in any order) that do not read from $w_i(X)$ in S, then T_i, and then all the transactions (in any order) reading from $w_i(X)$

in S. We now show that S' is view-equivalent to S. Indeed, since $w_i(X)$ is the only write action in both schedules, it is immediate to verify that the final-write set of S is equal to the final-write set of S'. Also, since the set of transactions reading from $w_i(X)$ in S is exactly the set of transactions appearing after T_i in S', it is immediate to verify that the read-from relation of S is equal to the read-from relation of S'.

Problem 3 Consider the following schedule

$$S = r_1(A)\, r_3(C)\, w_3(B)\, r_2(A)\, w_1(B)\, r_2(C)\, w_3(C)\, r_3(A)\, w_2(D).$$

3.1 Tell whether S is a 2PL schedule with exclusive and shared locks, explaining the answer in detail.

3.2 Tell whether S is conflict-serializable or not. If the answer is yes, then show a serial schedule that is conflict-equivalent to S. If the answer is no, then motivate the answer in detail.

3.2 Tell whether S is strict or not, and whether S is ACR or not, explaining the answer in detail.

Solution 3

3.1 Let us try to add lock and unlock commands to S coherently with the 2PL protocol:

$$sl_1(A)\, r_1(A)\, sl_3(C)\, r_3(C)\, xl_3(B)\, w_3(B)\, sl_2(A)\, r_2(A)$$

At this point, T_3 should release the lock on B, since T_1 wants to write on B. If we want to follow the 2PL protocol, however, T_2, before unlocking B, should acquire all the locks it needs in the future, namely: $xl_3(C)$ and $sl_3(A)$. If T_3 does so, however, then T_2 will not be able to read element C, because T_3 would hold the exclusive lock on such element. So, we conclude that S is not a 2PL schedule.

3.2 If we build the precedence graph associated to S, we immediately observe that it does not contain any cycle. Indeed, the only edges of the graph are the edge from the node corresponding to transaction T_3 to the node correspnding to transaction T_1, and the edge from the node corresponding to transaction T_2 to the node corresponding to transaction T_3. From this observation, we can immediately verify that the serial schedule

$$S' = r_2(A)\, r_2(C)\, w_2(D)\, r_3(C)\, w_3(B)\, w_3(C)\, r_3(A)\, r_1(A)\, w_1(B)$$

is conflict-equivalent to S.

3.2 S is not strict. Indeed, transaction T_1 (with action $w_1(B)$) writes on transaction T_3 (action $w_3(B)$) before the commit of T_3 (which cannot occur before the action $r_3(A)$). On the other hand, S is ACR, because no transaction reads from any other transaction.

Problem 4 Suppose that page P in our Data Base Management System is a page with fixed-length records containing 100 slots, and suppose we ask for the deletion of the record with *rid*=$\langle P, 10\rangle$. Illustrate the various actions that the system performs in the two cases of packed and unpacked organization for P, respectively.

Solution 4 We distinguish between the case of packed and unpacked organization.

- In the case of packed organization, the last slot in the page P contains the value M of records actually stored in the page (in particular, such records are stored in the first M positions of the page). Since we are deleting the record with *rid*=$\langle P, 10\rangle$, we have $M \geq 10$. To delete the record, the system simply does the following: for each $11 \leq i \leq M$, it moves the record that is currently in position i from position i to position $i-1$. Finally, the system changes the value stored in the last slot from M to $M-1$, to reflect the fact that the number of records stored in the page is decreased by one.

- In the case of unpacked organization, the last slot in the page P contains both the value M of slots that can be used for the records in the page, and an array of M bits, telling whether the various slots in the page are occupied or not. Again, since we are deleting the record with *rid*=$\langle P, 10\rangle$, we have surely $M \geq 10$. In this case, to delete the record with *rid*=$\langle P, 10\rangle$, it is sufficient to set the bit in position 10 of the array to 0, so as to reflect the fact that the slot in position 10 is now free.

Problem 5 Consider the relation `TENNISCHOOL(code,city,cost,numstud)` that stores 200.000 tuples, where each tuple $\langle d, t, c, n\rangle$ means that the tennis school d is in city t, has a monthly cost of c, and has n students. We assume that (*i*) in the average, for every value v of `cost`, there are 20 schools whose monthly cost is v, (*ii*) in the average, for every value w of `numstud`, there are 10 schools whose number of student is w, (*iii*) every page in secondary storage has space for 10 records of the relation `TENNISCHOOL`, (*iv*) every attribute and every record id occupies the same space in the page, and (*v*) the following are the most important queries on `TENNISCHOOL` (where Query 1 is even more important than Query 2):

Query 1	**Query 2**
select `code`	select `code`, `city`
from `TENNISCHOOL`	from `TENNISCHOOL`
where `cost` $\geq \alpha$ and `cost` $\leq \alpha + 10$	where `numstud` $\geq \beta$ and `numstud` $\leq \beta + 2$

Tell which is the method for representing the relation `TENNISCHOOL` you would choose in order to optimize the computation of the above queries, explaining in detail your answer. Also, tell how many pages are accessed during the execution of Query 1, and how many pages are accessed during the execution of Query 2.

Solution 5 Both queries are range queries, and therefore we can think of two B$^+$ tree indexes, one with search key `cost`, and the other with search key `numstud`. The best would be if both indexes were clustered. However, it is well known that only one index on the same relation can be clustered. Since the most important query is Query 1, we decide that the index with search key `cost` is clustered, so as to provide the best support to Query 1. Therefore, here is the decision on the method for representing the relation `TENNISCHOOL`:

- We store the data file in a sorted file with search key `cost`.
- We define a clustered B$^+$ tree sparse index with search key `cost`, using alternative 2. The index is sparse because we do not need to have one data entry for each data record in the data file; rather, it is sufficient to have one data entry for each value of `cost` appearing in the data file.
- We define an unclustered B$^+$ tree dense index with search key `numstud`, using alternative 2. Such index is dense, because an unclustered index cannot be sparse.

Let us now see how many pages are accessed during the execution of the two queries, by analyzing each of them separately.

1. Let us consider Query 1. We first compute the number of leaves in the tree index with search key `cost`. Since the relation has 200.000 tuples, and in the average, for every value v of `cost`, there are 20 schools whose monthly cost is v, we have 200.000 / 20 = 10.000 different values of `cost` that have to be stored in the leaves. Since every page in secondary storage has space for 10 records of the relation, we know that every page has space for 40 values. Since every data entry is constituted by 2 values (one value for `cost`, and one value of the same size for the record id), we know that every page has space for 20 data entries or index entry. We can also consider 20 to be the fan-out of the tree, since each index entry has the same structure of a data entry, being constituted by one value of `cost` and one pointer (and we are assuming that the size of such pointer is the same as the size of a record id). Now, the number of

leaves needed to keep the 10.000 data entries is 10.000 / 20 = 500. Taking into account the 67% occupancy property for the leaves, we have that 500 * 1.5 = 750 is the real number of leaves. This means that we need $log_{20}750 = 3$ page accesses to get to the leaf with first value α of the range. Since we need the `code` attribute in the data file, and we have our data file stored in a sorted file with search key `cost`, we now have to go to the data file, and then scan the data file to find all other qualifying records. How many qualifying records do we have? We have 10 values of `cost` to consider in the range, which means 200 schools to retrieve in the average. Since we have 10 records per page for the relation `TENNISCHOOL`, this means 20 accesses to the pages of the data file. We then conclude that the number of page accesses needed for the execution of Query 1 is:

$$log_{20}750 + 20 = 3 + 20 = 23.$$

Note that, if we decided to use alternative 1 for the clustered index, then the index would be dense, and the number of page accesses needed for Query 1 would be as follows. The number of leaf pages needed to store the relation would be 200.000 / 10 = 20.000. Taking into account the 67% occupancy property for the leaves, we have that 20.000 * 1.5 = 30.000 would be the real number of leaves. This means that we would need $log_{20}30.000 = 4$ page accesses to get to the leaf with first value α of the range. How many further page accesses do we need for accessing the qualifying records? The qualifying records are 200, and, if the pages were full, then we would need to access 20 pages for the qualifying records. Taking into account the 67% occupancy property for the leaves, we conclude that we would need to access 30 pages for further leaf pages. Thus, in this case, the number of page accesses needed for the execution of Query 1 would be:

$$log_{20}30.000 + 30 = 4 + 30 = 34.$$

2. Let us consider Query 2. We first compute the number of leaves in the tree index with search key `numstud`. Since the relation has 200.000 tuples, and the index is dense, we have 200.000 data entries that have to be stored in the leaves. Since every page in secondary storage has space for 10 records of the relation, we know that every page has space for 40 values. Since every data entry is constituted by 2 values (one value for `numstud`, and one value of the same size for the record id), we know that every page has space for 20 data entries or index entry. We can also consider 20 to be the fan-out of the tree, since each index entry has the same structure of a data entry, being constituted by one value of `numstud` and one pointer (and we are assuming that the size of such pointer is the same as the size of a record id). Now, the number of leaves needed to keep the 200.000 data entries is 200.000 / 20 = 10.000. Taking into account the 67% occupancy property for the leaves, we have that 10.000 * 1.5 = 15.000 is the real number of leaves. This means that we need $log_{20}15.000 = 4$ page accesses to get to the leaf with first value β of the range. There are 3 values of `numstud` in the range to consider, and for each such value of `numstud`, we have 10 schools with that value. This means that we need to analyze 30 data entries, which means that we need to access 3 more leaf pages in the worst case. Finally, since we need the attributes `code` and `city` in the data file, and we have our data file stored in a sorted file with search key `cost`, we now have to go to the data file for each of the data entry. Since the index is unclustered, this means that we need 30 more page accesses to the data file.

   We then conclude that the number of page accesses needed for the execution of Query 2 is:

$$log_{20}15.000 + 3 + 30 = 4 + 3 + 30 = 37.$$