Data Management – some solutions of the exam of 22/02/2013

Problem 2 A schedule S on trasactions $T_1, \ldots, T_n$ is called *soft* if (i) the commit command c_i of every transaction in $\{T_1, \ldots, T_n\}$ appears in S, (ii) each read action in S reads only from transactions that have already committed, and (iii) no write action in S writes on another transaction in S. Prove or disprove that every soft schedule is a 2PL schedule with both shared and exclusive locks.

Solution 2 We remind the reader that a 2PL schedule (with shared and exclusive locks) is a legal schedule with shared and exclusive locks constituted by well-formed transactions following the 2PL protocol.

Looking at the definition of soft schedule, it is not hard to see that a soft schedule is a strict schedule. So the question is related to checking whether every strict schedule is 2PL schedule (with shared and exclusive locks). It is well known that not every strict schedule is a 2PL schedule. One of the reasons is that a transaction T_1 can write on an element that has been read by another transaction T_2 before the commit of T_2.

Based on this observation, we can disprove the claim by showing a soft schedule that is not in 2P (with shared and exclusive locks). Consider the following schedule:

$$S = r_1(x)\, w_2(x)\, w_2(y)\, c_2\, r_1(y)\, c_1$$

It is immediate to verify that S is a soft schedule. However, S is not in 2PL, because transaction T_1 must release the lock on x and cannot acquire the lock on y before entering the shrinking phase.

Problem 5 Consider the relation `DEPT(deptcode,chair)` that stores information about which is the chair of the various departments, and the relation `PROF(profcode,name,age)`, that stores name and age of each professor. Note that `deptcode` is the key of `DEPT`, and `profcode` is the key of `PROF`. We assume that the number of departments is 2000, the number of professors is 10.000, the size of every page is 120K, and the size of every attribute, every record id, and every pointer is 2K. Also, we assume that the following are the most important queries on the two relations:

Query 1	**Query 2**
select *	select deptcode
from PROF	from DEPT
where age $>$ 60	where chair not in (select profcode
order by profcode	from PROF
	where age $<$ 40)

Query 1 asks for name and age of all professors whose age is greater than 60, ordered by the value of attributes `profcode`. Query 2 asks for the `deptcode` of every department whose chair is not a professor of less than 40 years old. We assume that when executing Query 2, the system scans the relation `DEPT`, and for each of its tuples, it checks whether the value of the attribute `chair` appears among the `profcode` of the tuples of `PROF` whose value of `age` is less than 40.

Tell which method for representing the two relations you would choose in order to optimize the computation of the above queries, explaining in detail your answer. Also, based on the method chosen, tell how many pages are accessed both during the execution of Query 1, and during the execution of Query 2.

Solution 5 The crucial issue is that Query 1 requires the result to be sorted by `profcode`. One possibility would be to represent the relation `PROF` by means of a sorted file. But with this method, Query 2 would not be well supported, because it would require to scan the file for every tuple of the relation `DEPT`.

So, instead of using a sorted file, we use tree-based index associated to the relation `PROF` on search key $\langle$`profcode,age`$\rangle$. This allow us to answer Query 1 efficiently, because the records of

relation PROF will be stored in the right order in the leaves of the tree. So, to answer Query 1 we simply scan the leaves of the tree, and select only those data entries for which age is greater than 60. Also, the index supports Query 2 well, because for every tuple of the relation DEPT, we can use the index to access the leaf containing the data entry whose value of profcode is equal to chair, and check whether the value of age is less than 40.

Each index entry is constituted by 3 items (two attribute values and one pointer), and therefore the number of index entries in each page is 120K / 6K = 20. So, the fan out of the tree is 20.

Analogously, each data entry is constituted by 3 items (two attribute values and one pointer), and therefore the number of data entries in each page is also 120K / 6K = 20. It follows that we need 10.000 / 20 = 500 pages to store all the data entries of the tree. Taking into account the 67% occupancy rule, we have that the number of leaves of the tree is 750.

As for relation DEPT, we use a heap. Since every tuple needs 4K, every page contains 120K / 4K = 30 tuples, and the pages needed to store the relation is 2000 / 30 = 67.

Taking into account the above observations, we have that:

- For Query 1, we simply scan the leaves of the tree, and therefore need 750 page accesses to answer the query.
- For Query 2, the algorithm scans the relation DEPT, and for each of its tuples, it uses the index to check whether the value of the attribute chair appears among the profcode of the tuples of PROF whose value of age is less than 40. The cost is therefore

$$67 + 2000 \times log_{20}750 = 6067$$