Data Management – AA 2015/16 – exam of 08/1/2016

Problem 1
Let R be a relation all of whose attributes have a fixed length, and such that no deletion is allowed on R. Which page organization method would you choose for storing the records within each of the pages of R? Please, explain the answer in detail, pointing out the advantages of the chosen page organization method with respect to other possibile methods.

Solution 1
The *packed organization*, one of the methods for organizing the pages in the case of fixed length records, is the best choice for organizing the pages of R, for the following reasons:

- Since all the attributes of R have the same size, a page organization method for fixed length records is perfectly adequate.
- Since there are no deletion from R, we know that tuples will never move or disappear within the pages. Therefore, the efficiency of the operations is maximized by the packed organization, where each tuple is stored in a slot of a page (and will never disappear from such slot), and the slots occupied by the tuples in the page are contiguous, from slot 1 to slot $N-1$. For example, once we have chosen the page P where to insert a tuple t, we can simply store t in the slot whose number is N, where $N-1$ is the current number of tuples in the page P, and increment the value representing the current number of tuples in P. In all other page organizations, insertion would be less efficient.
- In the case of relation R, the packed organization is better than the unpacked organization, because it allows us to avoid dealing with bit arrays within the pages.
- The packed organization is better than any organization method for pages with variable length records, simply because the latter are specific for the case where the tuples have varying length.

Problem 2
In general, a secondary, non-unique index contains duplicates (we remind the students that a duplicate is a pair of data entries with the same value for the search key). Are there cases where a secondary, non-unique index does not contain duplicates? If yes, which are those cases? Explain the answer in detail.

Solution 2
There are at least two cases where a secondary, non-unique index does not contain duplicates:

1. The index uses alternative (3), and therefore every relevant value of the search key is stored only once in the index, but with a list of rids associated to it.
2. The index uses alternative (2), and is clustered. Indeed, in this case, for each relevant value k of the search key, we can store in the index only one data entry, and we can make this data entry point to the first data record r with the value k for the search key. Since the index is clustered, the other data records with value k for the search key follow immediately r in the data file, and therefore, given k, we can easily access all of them after the access to r.

Problem 3
Prove or disprove each of the following claims.

1. If S is a schedule in ACR (Avoiding Casading Rollback), then S is view-serializable.
2. If S is a schedule containing at most one write action, then S is conflict-serializable.

Solution 3

1. We disprove the claim by simply exhibiting the following counterexample:

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

which is obviously in ACR (beacuse no transaction reads from another transaction), but is not view-serializable.

2. If S does not contain any write action, then it is obviously conflict-serializable, because the corresponding precedence graph is empty, and therefore trivially acyclic.

If S contains one write action, we can prove the claim in several ways. For example, one can prove that the precedence graph associated to S is acyclic.

We choose here another method to prove the claim. If S contains one write action, then let $w_i(X)$ of transaction T_i be such write action. Let $T_1, T_2, \ldots, T_n$ be any sequence of all the transactions containing an action reading from the action $w_i(X)$ in S, and let $T'_1, T'_2, \ldots, T'_m$ be any sequence of all the other transactions in S. Note that, since no transaction reads the same element more than once, every transaction in S is either in the set $\{T_1, T_2, \ldots, T_n\}$, or in the set $\{T'_1, T'_2, \ldots, T'_m\}$. We show that the serial schedule S'

$$T'_1, T'_2, \ldots, T'_m, T_i, T_1, T_2, \ldots, T_n$$

is conflict-equivalent to S. Indeed, consider any pair of conflicting actions in S. Since the only write action in S and S' is $w_i(X)$, we have that one of the actions in such conflicting pair is $w_i(X)$, and the other one is $r_j(X)$, for some transaction T_j. Now, there are two cases:

- T_j is before T_i in S': in this case, by construction of S', we can conclude that $r_j(X)$ does not read from $w_i(X)$ in S, and therefore $r_j(X)$ comes before $w_i(X)$ in S. Therefore, the actions forming the conficting pair appear in the same order in S and S'.
- T_j is after T_i in S': in this case, by construction of S', we can conclude that $r_j(X)$ does read from $w_i(X)$ in S, and therefore $r_j(X)$ comes after $w_i(X)$ in S. Also in this case, therefore, the actions forming the conficting pair appear in the same order in S and S'.

Since we have shown that for each pair of conflicting actions, such actions appear in the same order in S and S', we conclude that S and S' are conflict equivalent, and since S' is serial, this implies that S is conflict-serializable.

Problem 4
Consider the following schedule

$$S = r_2(x)\, r_1(x)\, w_3(t)\, w_1(x)\, r_3(y)\, r_4(t)\, r_2(y)\, w_2(z)\, w_5(y)\, w_4(z)$$

and answer the following questions:

- Tell whether S is conflict serializable, and, if so, exhibit one serial schedule which is conflict-equivalent to S.
- Tell whether S is in 2PL with shared and exclusive locks, motivating the answer.
- Tell whether S is strict, motivating the answer.

Solution 4

- It is easy to see that the precedence graph associated to S is acyclic, and that T_2, T_1, T_3, T_4, T_5 is a topological order of such graph. We can therefore conclude that T_2, T_1, T_3, T_4, T_5 is a serial schedule which is conflict-equivalent to S.

- S is in 2PL with shared and exclusive locks, as shown by the following legal schedules with well-formed transactions, all of which follow the 2PL protocol:
  $sl_2(x)\, r_2(x)\, sl_1(x) r_1(x)\, xl_3(t)\, w_3(t)\, sl_2(y)\, xl_2(z)\, u_2(x)\, xl_1(x)\, w_1(x)\, sl_3(y) r_3(y)\, u_3(t)\, sl_4(t)\, r_4(t)\, r_2(y)$
  $w_2(z)\, u_2(y)\, u_3(y)\, xl_5(y)\, w_5(y)\, u_2(z) xl_4(z)\, w_4(z)\, u_1(x)\, u_5(y)\, u_4(z)$
- S is strict, because
  - $w_4(z)$ writes on $w_2(z)$ when all actions of T_2 have finished, and therefore we can make T_2 commit before $w_4(z)$ writes on $w_3(t)$.
  - $r_4(t)$ reads from $w_3(t)$ when all actions of T_3 have finished, and therefore we can make T_3 commit before $r_4(t)$ reads from $w_3(t)$.

Problem 5
Suppose we have to compute the set intersection between the relation `R` with 200.000 pages and the relation `S` with 150.000 pages, having 600 free frames in the buffer. Tell which algorithm you would choose among the following three alternatives: 1 pass, 2 pass, and 3 pass. Also, tell which is the cost of the algorithm you have chosen for computing the result.

Solution 5
Obviously, we should choose the algorithm with the minimum number of passes. Unfortunately, we cannot compute the result by means of the 1-pass algorithm, because 600, i.e., the number of free frames in the buffer, is less than the number of pages of both relations. However, since $600^2 \geq 200.000+150.000$, we can choose the 2-pass algorithm for computing the set intersection between the relation `R` and the relation `S`. In particular, we can choose the 2-pass algorithm based on sorting:

1. In pass 1, we build the sorted sublists for the two relations, `R` and `S`;
2. In pass 2, we use one buffer frame for each sublist of `R` and `S` and repeatedly find the first remaining tuple t among all the frames. We then copy t to the output if and only if it appears in both `R` and `S`, and we remove from the frames all copies of t. During the process, if a buffer becomes empty, then we reload it with the next page from the corresponding sublist.

The cost of the algorithm in terms of number of page accesses (ignoring as usual the handling of the output) is $3(B(\texttt{R}) + B(\texttt{S})) = 3(200.000 + 150.000) = 1.050.000$.

Problem 6
Consider the relations `Tournament(name,year,city,winner)`, which contains 600 pages storing information about 30.000 tennis tournaments, and `Player(code,name,yearOfBirth,cityOfBirth)`, which stores information about 200.000 tennis players. Assume that every attribute of every relation and every pointer has the same size, and that there is a tree-based, primary, unclustered index on `Player` with search key `code`. Consider the query

```
select *
from Tournament, Player
where winner = code
```

and, assuming that you can only use one buffer frame (besides what is needed for handling the output), tell

1. which algorithm would you use for computing the answer to the above query;
2. which is the cost of computing the answer to the above query using the chosen algorithm.

Solution 6
The most appropriate algorithm is the index-based algorithm: we scan the pages of `Tournament`, and for each tuple `t` of `Tournament` that we find in such pages, we use the index to find the tuple `p` of

`Player` such that `t.winner = p.code`, and to build the corresponding tuple in the output. Note that the algorithm uses only one buffer frame, besides what is needed for handling the output.

To come up with the cost of computing the answer to the above query using the chosen algorithm, we have to compute the number of page accesses we need to find a tuple of `Player` given a value of the attribute `Code`, using the tree index.

In order to do so, we need to compute the number of leaves in the tree. We obviously assume that the index uses alternative 2. The leaves of the tree must store 200.000 data entries of the form $< v, p >$, where v is a value of the search key, and p is a pointer. How many pages do we need to store 200.000 data entries of such form? Since 600 pages are sufficient to store 30.000 tuples of `Tournament` (each one with 4 fields), we infer that 50 tuples of `Tournament` fit in one page, and therefore 100 data entries (each one with 2 fields) of the tree index fit in one page. At this point we know that we need 200.000 / 100 = 1000 pages for the leaves of the tree, and taking into account the 67% rule occupancy, we conclude that the leaf pages of the tree index are 3000. Also, we know that 100 is the number of index entries (each one with 2 fields) that fit in one page, and therefore we can assume that the fan-out of the tree index is 75 (the average value between 50 and 100). This means that the number of page accesses we need to reach the correct data entry of the index, given a value of the attribute `Code`, is $log_{75} 300 = 2$, and the number of page accesses we need to find the corresponding tuple of `Player` is $log_{75} 3000 + 1 = 3$ (because we have to reach the data record in the data file).

We are now ready to conclude that the cost of the algorithm, measured in terms of the number of pages accessed during its execution (ignoring as usual the cost for handling the output), is

$$600 + 30.000 \times 3 = 90.600.$$