Data Management – AA 2013/14 – exam of 21/2/2014

Compito B

Problem 1
A "transaction with single-ending-write" has a single write operation, and such write operation is the final action of the transaction. Let S be a schedule all of whose transactions are "transactions with single-ending-write". Prove or disprove each of the following two properties.

- S is conflict serializable.
- We can insert in S the "commit" operations of the various transactions appearing in S in such a way that the resulting schedule is ACR.

Solution 1
We disprove that if S is a schedule all of whose transactions are transactions with single-ending-write, then S is conflict serializable, by showing a counterexample. It is easy to see that the schedule

$$r_1(x)\, r_2(y)\, w_2(x)\, w_1(y)$$

is a schedule all of whose transactions are transactions with single-ending-write, but is not conflict serializable, as its precedence graph contains a cycle.

We prove that if S is a schedule all of whose transactions are transactions with single-ending-write, then we can insert in S the "commit" operations of the various transactions appearing in S in such a way that the resulting schedule is ACR. Indeed, consider the schedule S' obtained from S by adding, for each write action $w_i(z)$, the commit operation c_i of transaction T_i just after the action $w_i(z)$. Suppose S' is not ACR. Then there is a transaction T_i that reads from T_j, i.e., there is an action $r_i(x)$ reading from $w_j(x)$, and c_i is not between $w_j(x)$ and $r_j(x)$. But this contradicts the fact that c_i is just after $w_i(x)$. So, we have proved that assuming S' is not ACR leads to a contradiction, implying that S' is ACR.

Problem 2
Consider the following schedule

$$S = r_4(x)\, w_3(y)\, r_2(x)\, w_1(z)\, w_4(x)\, w_4(z)\, r_2(y)\, w_1(y)$$

2.1 Tell whether S is view serializable or not, explaining the a2nswer in detail. If the answer is yes, then tell if there is a single action that can be added to S in such a way that the resulting schedule is no longer view serializable. If the answer is no, then tell if there is a single action that can be deleted from S in such a way that the resulting schedule is view serializable.
2.2 Tell whether S is conflict serializable or not, explaining the answer in detail.
2.3 Tell whether S is accepted by the 2PL scheduler with exclusive and shared locks. If the answer is yes, then show the schedule obtained from S by adding suitable lock and unlock commands. If the answer is no, then explain the answer.
2.4 Tell whether S is ACR, whether it is strict, and whether it is rigorous, explaining the answer in detail.

Solution 2

4.1 S is view-serializable, since the serial schedule T_3, T_2, T_1, T_4 is clearly view-equivalent to S, because it has the same read-from relation and the same final-set as S. There is a single action that can be added to S in such a way that the resulting schedule is no longer view serializable: for example, if we add $r_1(x)$ at the end of the schedule, then the resulting schedule S' is no longer view serializable, because now T_1 reads from T_4 (implying that in every serial schedule view equivalent to S', T_4 preceeds T_1), and in S the final write on z is $w_4(z)$ (implying that in every serial schedule view equivalent to S', T_1 preceeds T_4).
4.2 S is conflict serializable because the precedence graph associated to S is clearly acyclic.

4.3 S is not accepted by the 2PL scheduler with exclusive and shared locks: in order to execute $w_4(z)$, T_1 must issue the command $u_1(z)$ before $w_4(z)$, and because of $w_1(y)$, T_1 should issue the command $xl_1(y)$ before $u_1(z)$; in order to execute $r_2(y)$, T_2 must issue the command $sl_2(y)$, but this command is is incompatible with $sl_1(y)$.

4.4 S is ACR, because T_2 reads from T_3, but T_3 can commit before $r_2(y)$. S is not strict, because T_4 writes on T_1, but T_1 cannot commit before $w_4(z)$, since $w_1(y)$ appears after $w_4(z)$. Obviously, since S is not strict, S is not rigorous.

Problem 3
Describe in detail the interpolation search algorithm that is used for key-based searching in the sorted file organization. Under which condition is this algorithm efficient?

Solution 3
See the slides of the course.

Problem 4
Suppose relations `R(A,B,C)` and `S(D,E,F)` are stored in 800 pages and 3000 pages, respectively, each without duplicates. Suppose also that we have 900 buffer frames available in main memory. Which is the algorithm you would use for computing the set intersection of `R` and `S` with a minimal number of page accesses? And which is the cost of such algorihtm in terms of page accesses?

Solution 4
Since the buffer has 900 frames free, we can use the one-pass algorithm: we read `R` into 800 buffer frames, and then we read `S` one page at a time, and for each tuple, we check whether the tuple is already in the buffer frames. If no, we ignore the tuple, otherwise, we write the tuple in the page devoted to the result (when such a page is full, we write it into the result), and we delete the tuple from the buffer frames. The number of page accesses (as usual, ignoring the pages used to write to result) is simply $800 + 3000 = 3.800$.

Problem 5
Consider the relation `TRAVEL(group,year,nation,cost,cities)` that stores information about travels, with the group of people participating in the travel, the year when the travel has taken place, the nation visited during the travel, the cost of the travel, and the number of people that participated in the travel. The relation occupies 600.000 pages, each of 1600 KB. We assume that all fields in every record have the same size of 20 KB, independently of the field type. There is a sparse B$^+$-tree index on `TRAVEL` with search key ⟨`group,year,nation`⟩, using alternative 2. Consider the query

```
select group,nation from TRAVEL;
```

that asks for the group and the nation of all the travels, and tell which algorithm you would use for executing the query, and how many page accesses such algorithm needs for computing the answer.

Solution 5
Since the index is sparse, we cannot use the index for answering the query, because we know that we have only one data entry for each page, and not for each value of the search key in the data records. Therefore, the only way to compute the result of the query is to use the algorithm for projection (possibly with duplicates), which is a one-pass algorithm. So, the number of page accesses is 600.000 (as usual, we ignore the cost of writing the result).
<u>Comment</u>:
Just as an observation, let us analyze the case of dense index also. In this case, since each page has size 1600 KB, and the size of each field in every record is 20 KB, independently of the field type, we conclude that in every page we have space for 80 values. Since the index uses alternative 2, and every search key value has 3 fields, we know that every data entry has 4 fields (the three values of the search key, plus the pointer to the data file), and therefore every leaf of the index has

room for 80/4 = 20 data entries. Taking into account the 67% occupancy rule, this means that every leaf of the index has 13 data entries. Since we are considering the case of dense index, we have one data entry for each tuple of the relation. How many tuples does the relation have? Since every tuple occupies 100KB, in every page we 1600/100=16 tuples, and therefore, we have 600.000 * 16 = 9.600.00 tuples in the relation. Now, 9.600.000/13 = 738.461 is the number of leaves of the tree. Since the query asks for the group and the nation of all the travels, the search key contains group and nation, and the index is now dense, we could in principle answer the query simply by an index-only scan. More precisely, we could use the one-pass algorithm computing the projection of the relation stored in the leaves of the index. However, the cost is 738.461 page accesses (as usual, we ignore the cost of writing the result), which is worse than just scanning the data file.