Data Management – AA 2016/17 – exam of 06/02/2017
Solutions $\boxed{\text{B}}$

Solution to problem 1

What we can do is to sort each table by means of the 2-way sorting algorithm, and then compute the result of the difference between the sorted table `R` and the sorted table `S` by means of a variant of the merge algorithm. In this variant, for each tuple t of `R`, we include in the result a number of copies of t obtained as the difference between the number of copies of t in `R` and the number of copies of t in `S` (not incuding the tuple in the case where such difference is negative).

We remind the reader that sorting a table with B pages by means of 2-way sorting costs $2 \times B \times (log_2 B + 1)$ page accesses, while merging two sorted tables of B_1 and B_2 pages respectively, requires $B_1 + B_2$ page accesses.

To derive the cost of the algorithm, we ignore as usual the cost of writing the final result, but obviously we do not ignore the cost of writing the two sorted tables, because we have to read them for computing the final result through the merging step.

Therefore, the whole algorithms costs

$$2 \times B_1 \times (log_2 B_1 + 1) + 2 \times B_2 \times (log_2 B_2 + 1) + 2 \times (B_1 + B_2)$$

page accesses.

Solution to problem 2

1. We disprove the proposition simply by exhibiting the following strange schedule that is not accepted by the timestamp-based scheduler:

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

2. We prove the proposition by defining a serial schedule S' starting from a strange schedule S, and then showing that S is conflict equivalent to S'. If S is a strange schedule, then define the serial S' as follows (with $h, k, m \geq 0$):

$$S' = T_s^1\, T_s^2\, \cdots\, T_s^h\, T_r^1\, T_r^2\, \cdots\, T_r^k\, T_f^1\, T_f^2\, \cdots\, T_f^m$$

where

- $T_s^1\, T_s^2\, \cdots\, T_s^h$ are all the "write only" transactions in S (in any order) that are not preceeded by any read action on the same element in S,
- $T_r^1\, T_r^2\, \cdots\, T_r^k$ are all the "read only" transactions in S (in any order), and
- $T_f^1\, T_f^2\, \cdots\, T_f^m$ are all the "write only" transactions in S (in any order) that are not followed by any read action on the same element in S.

To prove that S' is conflict equivalent to S, we proceed by showing that every pair of conflicting actions appearing in S appears in the same order in S'.

Since no element of the database is written more than once in S, we can concentrate only on conflicts of type (w, r) or (r, w) in S. Suppose that $w_i(x)$ appears before $r_j(x)$ in S; this implies that the transaction T_i constituted by the write action $w_i(x)$ is not proceeded by any read action on x (because no element of the database is read more than once in S), and therefore, by construction of S', T_i appears before the (only) transaction containing $r_j(x)$ in S'. Suppose that $w_i(x)$ appears after $r_j(x)$ in S; this implies that the (only) transaction T_i constituted by the write action $w_i(x)$ is not followed by any read action on x (because no element of the database is read more than once in S), and therefore, by construction of S', we have that the (only) transaction T_i constituted by the action $w_i(x)$ appears after the transaction containing $r_j(x)$ in S'. This shows that any pair of conflicting actions appearing in S appear in the same order in S', and therefore, S and S' are conflict serializable.

3. We prove the proposition simply by noticing that, since every strange schedule is conflict serializable, and every conflict serializable schedule is also view serializable, it follows that every strange schedule is view serializable.

Solution to problem 3

5.1 S is not a 2PL schedule, because transaction 2 should unlock x in order to allow transaction 3 to write on x. Therefore, in order for S to follow the 2PL protocol, transaction 2 should acquire the exclusive lock on u (needed for the last write of S) before unlocking x. But if this happens, transaction 4 will not be able to read u.

5.2 S is conflict-serializable, as can be easily seen from the fact that the precedence graph associated to S is acyclic. It follows that S is view serializable.

5.3 The behaviour of the timestamp-based scheduler when processing S is as follows:

$r_1(z) \rightarrow$ OK,	rts(z)=1
$r_1(y) \rightarrow$ OK,	rts(y)=1
$w_3(y) \rightarrow$ OK,	wts(y)=3, cb(y)=`false`
$r_1(x) \rightarrow$ OK,	rts(x)=1
$r_2(x) \rightarrow$ OK,	rts(x)=2
$c_1 \rightarrow$ OK,	
$w_4(z) \rightarrow$ OK,	wts(z)=4
$w_2(x) \rightarrow$ OK,	wts(x)=2, cb(x)=`false`
$w_3(x) \rightarrow$ OK,	transaction 3 suspended
$r_4(u) \rightarrow$ OK,	rts(u)=4
$c_4 \rightarrow$ OK,	wts-c(z)=4, cb(z)=`true`
$w_2(u) \rightarrow$ write too late,	transaction 2 rollbacks
$w_3(x) \rightarrow$ OK,	wts(x)=3
$c_3 \rightarrow$ OK,	wts-c(x)=3, cb(x)=`true`

5.4 S is ACR, because no transaction reads from another transaction in S.

Solution to problem 4

Each page has space for $600/60 = 10$ tuples of `R`, and therefore `R` is stored in a heap with $1.400.000/10 = 140.000$ pages. Similarly, each page has space for $600/100 = 6$ tuples of `Q`, and therefore `Q` is stored in a heap with $2.400.000/6 = 400.000$ pages.

Note that the B$^+$-tree index on (`E,F`), where (`E,F`) is the key of `Q`, is unclustering, and therefore dense. Since each page has space for $600/40 = 15$ data entries of such index, taking into account the 67% occupancy rule, we know that each page contains 10 data entries, implying that the B$^+$-tree index has $2.400.000/10 = 240.000$ leaf pages.

Notice that $400 \times 400 = 160.000$, and that $140.000 < 160.000$. Notice also that the query requires to compute the union (without duplicates) between the sorted projection of `R` on `A,B` and the sorted projection `Q` on `E,F`. Also, observe that the sorted projection `Q` on `E,F` is directly available in the leaves of the B$^+$-tree index on `Q` with search key (`E,F`). It follows that in order to compute the result of the query, we can use a variant of the two-pass algorithm for set union based on sorting, by first producing the sorted sublists for `R`, and then applying the second pass to compute the union without duplicates using directly the leaves of the B$^+$-tree index on `Q`) with search key (`E,F`).

More precisely, the algorithm and the corresponding cost is as follows:

Pass 1 Read `R`, and whenever we have 399 pages of `R` in the buffer, sort such pages, and using one buffer frame, write the projection on `A,B` of the tuples contained in such pages, thus producing a sorted sublist of `R`. Note that the size of the projection of `R` on attributes `A,B` is 1.400.000/ (600/40) = 93.334. Therefore pass 1 requires to access 140.000 pages for reading `R`, and 93.334 pages for writing the 140.000/399 = 359 sorted sublists containing the projection of `R` on attributes `A,B`.

Pass 2 Use the 400 free buffer frames for loading one page at a time of the sorted sublists of the projection of `R` on attributes `A,B`, and one page at a time of the leaves of the B$^+$-tree index on `Q` with search key (`E,F`). At each stage, we analyze the first (according to the sorting) tuple stored in the pages devoted to the projection of `R`, and we write one copy (ignoring the other copies) of such a tuple in the output frame only if it appears in the page devoted to the leaves of the B$^+$-tree index. If, on the contrary, such a tuple does not appear in the page devoted to the leaves of the B$^+$-tree index, then we ignore all its copies without writing any of them in the output frame. During such a process, whenever the output frame is full, we copy it in the file corresponding to the final result. Pass 2 requires to read 93.334 + 240.000 pages.

The total cost of the algorithm is 140.000 + 93.334 + 93.334 + 240.000 = 566.668 page accesses, where, as usual, we have ignored the cost of writing the final result.

Solution to problem 5

Since `Q` has 10.000 tuples and two attributes, each of 20 Bytes, and the size of each page is 400 Bytes, the number of pages of `Q` is 10.000 $\times$ 2 $\times$ 20 / 400 = 1.000. Since `R` has 400.000 tuples and four attributes, each of 20 Bytes, the number of pages of `R` is 400.000 $\times$ 4 $\times$ 20 / 400 = 80.000.

1. If `R` is represented as a heap file, then the query can be answered by means of a block nested-loop algorithm, where we load relation `Q` in blocks, each of 250 pages, and for each block b we scan relation `Q` to find the tuples in b that satisfy the `where` condition (we use one buffer frame among the 252 free frames available for reading `R`, and one for producing the output). The cost is then 1.000 + (1.000 / 250) $\times$ 80.000 = 321.000 page accesses.

2. If `R` is represented as a sorted file, then the query can be answered by scanning the tuples of `Q`, and for each tuple t_1 of `Q`, using binary search for checking whether there exists a tuple t_2 in `R` such that $t_1.F = t_2.B$, and including t_1 in the result if such check fails. The cost is then 1.000 + 10.000 $\times$ $\log_2$ 80.000 = 171.000 page accesses.

3. If `R` is represented as a heap file with unclustering, dense sorted index with duplicates (i.e., strongly dense, which means that we have one data entry per data record) with search key `B`, then the query can be answered by means of an index-based index algorithm that scans the tuples of `Q`, and for each tuple t_1 of `Q` uses the sorted index for checking whether there exists a tuple t_2 in `R` such that $t_1.F = t_2.B$, including t_1 in the result if such check fails. Since the index is unclustering, is dense, and has duplicates, it has one data entry for each tuple in `R`, i.e., it has 400.000 data entries. Each data entry requires 2 $\times$ 20 = 40 Bytes, and therefore each page has 10 data entries, and the number of pages of the index is 40.000. It follows that the cost is 1.000 + 10.000 $\times$ $\log_2$ 40.000 = 161.000 page accesses.

4. If `R` has a clustering, dense sorted index without duplicates (i.e., we have one data entry for each vale of the search key) with search key `B`, then the query can be answered by

means of an index-based algorithm, as before. Since the index is clustering and dense, it can avoid duplicates, and therefore it has one data entry for each value in B, i.e., 2.000. Since each page has 10 data entries, as we saw before, the number of pages of the index is 2.000/ 10 = 200. It follows that the cost is $1.000 + 10.000 \times \log_2 200 = 81.000$ page accesses.

5. If R has a clustering, sparse sorted index with search key B, then the query can be answered again by means of an index-based index algorithm, as before. Since the index is clustering and sparse, it has one data entry for each page of R. Since each page has 10 data entries, as we saw before, the number of pages of the index is 80.000 / 10 = 8.000. Note that in this case, after accessing the index, we have to follow the pointer to the data file, since the index is sparse. It follows that the cost is $1.000 + 10.000 \times (\log_2 8.000 + 1) = 141.000$ page accesses.