Data Management – AA 2019/20
Solutions for the exam of 11/06/2020

Problem 1

If S is a schedule on transactions $T_1, \ldots, T_n$, then the *partial precedence graph* $\mathsf{PPG}(S)$ associated to S is a graph that has the transactions in S as nodes, and has an edge from T_i to T_j if and only if S contains two actions of different types (i.e., one read and one write) $a_i(x)$ in T_i and $a_j(x)$ in T_j on the same element x such that $a_i(x)$ precedes (not necessarily directly) $a_j(x)$ in S. Also, the *write-on graph* $\mathsf{WOG}(S)$ associated to S is a graph that has the transactions in S as nodes, and has an edge from T_i to T_j if and only if there is an x such that $w_j(x)$ is followed by $w_i(x)$ in S, and there is no write action on x in S between $w_j(x)$ and $w_i(x)$. Prove or disprove the following claims:

1. If both $\mathsf{PPG}(S)$ and $\mathsf{WOG}(S)$ are acyclic, then S is view-serializable.
2. If $\mathsf{PPG}(S)$ is acyclic, and $\mathsf{WOG}(S)$ has no edges, then S is conflict-serializable.

Solution to problem 1

1. The claim can be disproved, for instance, by means of the following counterexample S_1:

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

   S_1 is not view-serializable and is such that both $\mathsf{PPG}(S_1)$ and $\mathsf{WOG}(S_1)$ are acyclic. Observe that this example shows that even though both $\mathsf{PPG}(S_1)$ and $\mathsf{WOG}(S_1)$ are acyclic, their union may contain a cycle. Now, it is easy to see that the union of the partial precedence graph associated to S and the write-on graph associated to S is a subset of the precedence graph associated to S. This implies that if the union of the two graphs contains a cycle, such cycle appears also in the precedence graph associated to S, and therefore S is not conflict-serializable, and can even be non view-serializable, as the counterexample S_1 shows.

2. The claim can be proved by showing that, in the case where $\mathsf{WOG}(S)$ has no edges, the partial precedence graph $\mathsf{PPG}(S)$ associated to S coincides with the precedence graph associated to S. In order to show this property, we observe the following for S:

   - It is immediate to note that $\mathsf{PPG}(S)$ and the precedence graph associated to S have the same nodes.
   - It is also immediate to see that the set of edges in $\mathsf{PPG}(S)$ is a subset of the set of edges in the precedence graph associated to S.
   - In the case where $\mathsf{WOG}(S)$ has no edges, an edge from T_i to T_j exists in the precedence graph associated to S if and only if S contains two actions of different types (i.e., one read and one write) $a_i(x)$ in T_i and $a_j(x)$ in T_j on the same element x such that $a_i(x)$ precedes (not necessarily directly) $a_j(x)$ in S. This immediately implies that, if $\mathsf{WOG}(S)$ has no edges, then every edge in the precedence graph associated to S is also an edge in $\mathsf{PPG}(S)$.

   Being $\mathsf{PPG}(S)$ equal to the precedence graph associated to S, the acyclicity of $\mathsf{PPG}(S)$ implies that S is conflict-serializable.

Problem 2 Let R be a relation with 10.000.000 tuples, each with 50 attributes, occupying 1.000.000 pages, and let us consider the operation of searching for all the tuples of R with a given value for the non-key attribute A, knowing that A contains 100 values uniformly distributed over the tuples of R. We consider three methods for representing R in secondary storage: (1) R is stored as a sorted file with search key A, (2) R is stored as a heap file with an associated sorted index using alternative 2 with search key A, and (3) R is stored as a sorted file with search key A with an associated sorted index using alternative 2 with search key A. Under the assumption that each value and each pointer occupy the same space, tell which is the cost (in terms of number of page accesses) of the search operation in the cases corresponding to the three methods specified above.

Solution to problem 2

1. In the case of method 1, the operation can be performed by a simple binary search on the sorted file for accessing the first record with the appropriate value of A, plus the scan of the contiguous pages of the sorted file containing the tuples with that value in the attribute A. Since A contains 100 values uniformly distributed over the tuples of R, the number of pages of the sorted file containing the tuples with the same value in the attribute A is 1.000.000 / 100 = 10.000, and therefore the cost is $\log_2$ 1.000.000 + 10.000 = 10.020.

2. In the case of method 2, the index is unclustering, and the operation can be performed by a binary search on the sorted index for accessing the first occurrence of the appropriate value of A, followed by the access to all the pages of R where the records pointed by the data entries are located. From the fact that the tuples of R have 50 attributes and occupy 1.000.000 pages, and from the assumption that each value and each pointer occupy the same space, we infer that the data entries of the sorted index associated to R, each data entry being constituted by two values, occupy 40.000 pages, where each page contains 250 data entries. How many pages of the index we have to access after the binary search? Since A contains 100 values uniformly distributed over the tuples of R, the number of the index pages with the same value of the search key in the data entries is 40.000 / 100 = 400. How many pointer should we follow to get to the pages of R? Since there are 10.000.000 tuples in R, for each value of A there are 10.000.000 / 100 = 100.000 tuples with that value in the attribute A, and therefore we have to follow 100.000 pointers. We conclude that the cost is $\log_2$ 40.000 + 400 + 100.000 = 100.416.

3. In the case of method 3, the index is clustering, and therefore the number of data entries is 1.000.000 (one data entry per page of R), stored in 1.000.000 / 250 = 4.000 pages. The operation can be performed by a binary search on the sorted index for accessing the first occurrence of the appropriate value of A, followed by one access to the heap file storing R, plus the scan of the contiguous pages of the sorted file containing the tuples with the appropriate value in the attribute A. So, the cost is $\log_2$ 4.000 + 1 + 10.000 = 10.013.

Problem 3

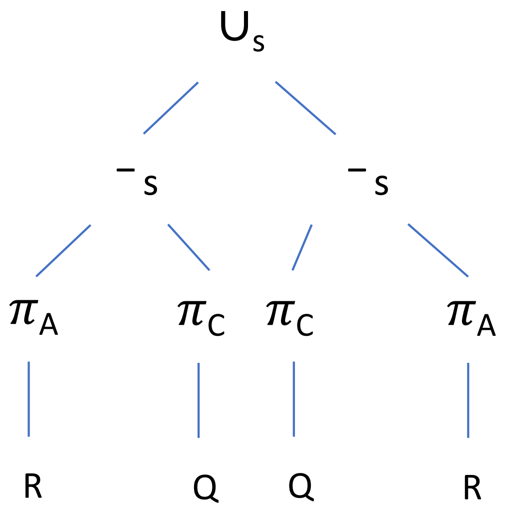
Consider the relations R($\underline{\texttt{A}}$,$\underline{\texttt{B}}$,C,D,E,F) and Q($\underline{\texttt{C}}$,D), where R is stored in 20.000 pages of a heap file with an associated B$^+$-tree index with search key ⟨A,B⟩, Q is stored in 600 pages of a heap file, each page contains 20 tuples of R, each attribute and each pointer occupy the same space, and we know that there are 150 available frames in the buffer. Consider the following query

```
select A from R where A not in (select C from Q)
  union
select C from Q where C not in (select A from R)
```

Show the logical query plan associated to the query, as well as the logical query plan and the physical query plan you would choose for executing the query efficiently. Also, tell which is the cost (in terms of number of page accesses) of executing the query according to the chosen physical query plan.

Solution to problem 3

The logical query plan associated to the query, which is also the logical query plan chosen, is shown below.

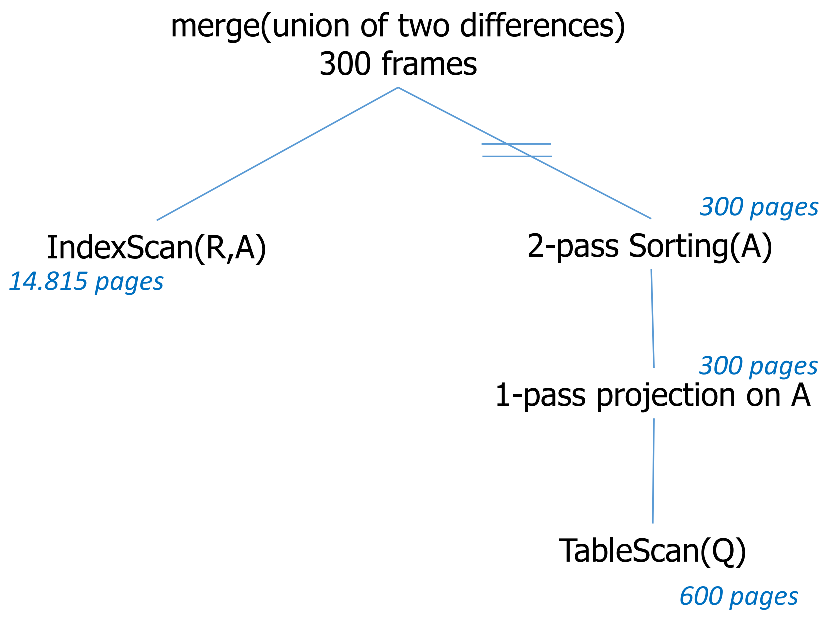


Logical query plan

If a page contains 20 tuples, each with 6 attributes, then it has space for 40 data entries. Taking into account the 67% rule, we know that each leaf contains 27 data entries. This means that the index has 400.000 / 27 = 14.815 leaves.

To decide the physical query plan, we observe that what we have to compute is the set V_1 of values that are in $\pi_A(\texttt{R})$ but not in $\pi_C(\texttt{Q})$, union the set V_2 of values that are in $\pi_C(\texttt{Q})$, but not in $\pi_A(\texttt{R})$. Note that V_1 and V_2 are disjoint, and therefore we can ignore duplicate elimination during the union.

Observe that having the values of $\pi_A(\texttt{R})$ sorted, and having the values of $\pi_C(\texttt{Q})$ sorted would definitely help in computing the union of the two differences. Indeed, one could simply perform a kind of merge step, by scanning the sorted files, copying to the output both the values in $\pi_A(\texttt{R})$ that are not in $\pi_C(\texttt{Q})$, and the values in $\pi_C(\texttt{Q})$ that are not in $\pi_A(\texttt{R})$. Note also that the values in $\pi_A(\texttt{R})$ already sorted are exactly in the leaves of the index. One could then decide to sort $\pi_C(\texttt{Q})$ and then perform the merging step. However, this implies to sort the 300 pages of $\pi_C(\texttt{Q})$ in two passes (since the buffer has 150 frames) and to materialize the results. The corresponding physical query plan is as follows:

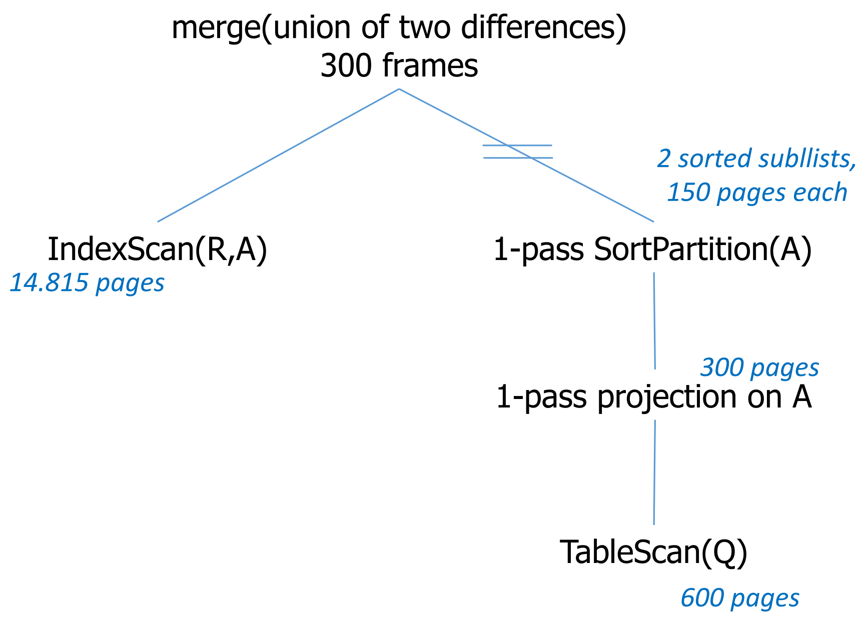


Physical query plan

The cost in terms of the number of page accesses is 14.815 (index scan) + 600 (reading of Q) + 900 (sorting of π_C(Q)) + 300 (reading of the sorted π_C(Q)) = 16.615.

We can actually do better, by avoiding the materialization step as follows. We produce the sorted sublists of π_C(Q), sorted on the basis of C, in one pass (2 sorted sublists), and then we perform a merge step using only 4 buffer frames (1 for the leaves of the index, 2 for the sublists of π_C(Q), and 1 for the output) for computing the result. In such a merge step, a value that is both in π_A(R) and in π_C(Q) is ignored, whilst all other values are copied to the output.

The physical query plan is as follows:



Physical query plan

The cost in terms of the number of page accesses is 14.815 (index scan) + 600 (reading of Q) + 300 (writing of the sorted sublists of π_C(Q)) + 300 (reading of the sorted sublists of π_C(Q)) = 16.015.

Problem 4
Given the two relations $\texttt{R}_1\texttt{(A,B,C)}$ and $\texttt{R}_2\texttt{(C,D)}$, the following equivalences were intended to be used during the optimization of logical query plans involving $\texttt{R}_1$ and $\texttt{R}_2$:

1. If $\texttt{R}_1$ or $\texttt{R}_2$ (or both) is a bag, i.e., may contain duplicates, then $\delta(\texttt{R}_1 \bowtie \texttt{R}_2) = \delta(\texttt{R}_1) \bowtie \delta(\texttt{R}_2)$.
2. If $\texttt{R}_1$ and $\texttt{R}_2$ are sets, then $\delta(\pi_{\texttt{A,B,C}}(\texttt{R}_1 \bowtie \texttt{R}_2)) = \pi_{\texttt{A,B,C}}(\texttt{R}_1 \bowtie \texttt{R}_2)$.

For each of the above equivalences, prove or disprove, explaining your answer in details, that it is valid, and can indeed be used in query optimization. We remind the students that δ denotes duplicate elimination, π denotes projection (without duplicate eliminations) and $\bowtie$ denotes natural join, i.e., the join of two relations based on equality on common attributes.

Solution to problem 4

1. This equivalence is valid. To show validity, we will prove that, for any $\texttt{R}_1$ (set or bag) and $\texttt{R}_2$ (set or bag), (i) $t \in \delta(\texttt{R}_1 \bowtie \texttt{R}_2)$ implies $t \in \delta(\texttt{R}_1) \bowtie \delta(\texttt{R}_2)$, and (ii) $t \in \delta(\texttt{R}_1) \bowtie \delta(\texttt{R}_2)$ implies $t \in \delta(\texttt{R}_1 \bowtie \texttt{R}_2)$.

   (i) If a tuple $\langle x_1, x_2, x_3, x_4\rangle$ is in $\delta(\texttt{R}_1 \bowtie \texttt{R}_2)$, then at least one occurrence of $\langle x_1, x_2, x_3, x_4\rangle$ is in $R_1 \bowtie R_2$. By definition of natural join, we know that there exist $\langle x_1, x_2, x_3\rangle \in R_1$, and $\langle x_3, x_4\rangle \in R_2$. This in turn implies that there exist at least one occurrence of the tuple $\langle x_1, x_2, x_3\rangle$ in $\delta(R_1)$ and one occurrence of the tuple $\langle x_3, x_4\rangle$ in $\delta(R_2)$, and therefore $\langle x_1, x_2, x_3, x_4\rangle \in \delta(\texttt{R}_1) \bowtie \delta(\texttt{R}_2)$.

   (ii) Consider a tuple $\langle x_1, x_2, x_3, x_4\rangle \in \delta(\texttt{R}_1) \bowtie \delta(\texttt{R}_2)$; by definition of natural join, we know that there exist $\langle x_1, x_2, x_3\rangle \in \delta(R_1)$, and $\langle x_3, x_4\rangle \in \delta(R_2)$, which implies that there exist at least one occurrence of the tuple $\langle x_1, x_2, x_3\rangle$ in R_1, and at least one occurrence of the tuple $\langle x_3, x_4\rangle$ in R_2. This in turn implies that at least one occurrence of $\langle x_1, x_2, x_3, x_4\rangle$ is in $\texttt{R}_1 \bowtie \texttt{R}_2$, and therefore $\langle x_1, x_2, x_3, x_4\rangle \in \delta(\texttt{R}_1 \bowtie \texttt{R}_2)$.

2. Clearly, this equivalence is not valid, as shown by the following counterexample: $\texttt{R}_1\texttt{(A,B,C)} = \{\langle a, b, c\rangle\}$, $\texttt{R}_2\texttt{(C,D)} = \{\langle c, d_1\rangle, \langle c, d_2\rangle\}$, for which we have $R_1 \bowtie R_2 = \{\langle a, b, c, d_1\rangle, \langle a, b, c, d_2\rangle\}$, and $\pi_{\texttt{A,B,C}}(R_1 \bowtie R_2) = \{\langle a, b, c\rangle, \langle a, b, c\rangle\}$, $\delta(\pi_{\texttt{A,B,C}}(R_1 \bowtie R_2)) = \{\langle a, b, c\rangle\}$.

Problem 5
Let $\texttt{R}_1\texttt{(A,B,C,D)}$ and $\texttt{R}_2\texttt{(A,B,C,D)}$ be two relations stored in two heap files with $B(\texttt{R}_1)$ and $B(\texttt{R}_2)$ pages, respectively. We know that $B(\texttt{R}_1) < B(\texttt{R}_2)$, $B(\texttt{R}_1) > K$, and $B(\texttt{R}_2) > K$, where K is the number of available frames in the buffer. We have to compute the intersection of $\texttt{R}_1$ and $\texttt{R}_2$, in four different scenarios: (a) both $\texttt{R}_1$ and $\texttt{R}_2$ are sets; (b) $\texttt{R}_1$ is a set and $\texttt{R}_2$ is a bag; (c) $\texttt{R}_1$ is a bag and $\texttt{R}_2$ is a set; (d) both $\texttt{R}_1$ and $\texttt{R}_2$ are bags. For each of the above scenarios, tell whether the "classical block-nested loop algorithm" can be used or not; if the answer is negative, then motivate the answer in detail, and if the answer is positive, then briefly describe the algorithm and its cost (in terms of number of page accesses). We remind the students that the "classical block-nested loop algorithm" reads all the pages of the outer relation in blocks, and for each block, it reads all the pages of the inner relation, and while doing this, it does not execute any write operation other than the writes of the pages of the result.

Solution to problem 5
We analyze the various cases.

(a) If both $\mathtt{R}_1$ and $\mathtt{R}_2$ are sets, then we can surely use the classical block-nested loop algorithm: we simply choose the smallest relation $\mathtt{R}_1$ as the outer relation, we load the pages of such relation in blocks of $K-2$ pages, and for each block we load all the pages of the inner relation $\mathtt{R}_2$ one at a time, doing the following: we mark the tuples in the block that we encounter when scanning $\mathtt{R}_2$, and at the end of the scan we copy to the output the marked tuples. This algorithm is correct because, since $\mathtt{R}_1$ has no duplicates, if a tuple t of $\mathtt{R}_1$ appears in a certain block L, then it appears *only* in that block. This implies that after the scan of $\mathtt{R}_2$ done for the block L, we know exactly what to do with the tuple t. It is well known that the cost is $B(\mathtt{R}_1) + B(\mathtt{R}_1)/(K-2) \times B(\mathtt{R}_2)$.

(b) If $\mathtt{R}_1$ is a set and $\mathtt{R}_2$ is a bag, then we can still use the classical block-nested loop algorithm: indeed, the algorithm for case (a) is still valid, because the same argument used in (a) applies. Obviously, the cost is still $B(\mathtt{R}_1) + B(\mathtt{R}_1)/(K-2) \times B(\mathtt{R}_2)$.

(c) If $\mathtt{R}_1$ is a bag and $\mathtt{R}_2$ is a set, then we can still use the classical block-nested loop algorithm: indeed, we can use the same idea as the algorithm for case (a), but we have to consider $\mathtt{R}_2$ as the outer relation, and $\mathtt{R}_1$ as the inner relation, even if $\mathtt{R}_2$ is the largest relation. So, the cost is $B(\mathtt{R}_2) + B(\mathtt{R}_2)/(K-2) \times B(\mathtt{R}_1)$.

(d) If both $\mathtt{R}_1$ and $\mathtt{R}_2$ are bags, then the classical block-nested loop algorithm cannot be used. Indeed, when we have a block of the outer relation (say, $\mathtt{R}_1$) in the buffer, and we consider a tuple t in such block, we do not know how many other occurrences of t exist in the other blocks of $\mathtt{R}_1$, and therefore we cannot decide what to do with t by simply scanning the inner relation (say, $\mathtt{R}_2$). In principle, we could try to modify the algorithm so as to keep track of the properties of the tuples in the block, but this would imply writing some information in secondary storage, thus deviating from the classical block-nested loop technique.