Data Management – exam of 12/07/2021
Solutions

Problem 1
A schedule S is called "one-writer" if at most one transaction appearing in S has write actions. Prove or disprove the following two statements:

1. A "one-writer" schedule S is conflict-serializable if and only if it is view-serializable.
2. A "one-writer" schedule S is conflict-serializable if and only if it is a 2PL schedule.

The solution must be provided under the usual assumption that no transaction reads or writes the same element more than once.

Solution to problem 1

1. It is well-known that if S is conflict-serializable, then it is also view-serializable. So, it remains to show that if S is a view-serializable one-writer schedule, then it is also conflict-serializable. Assume that S is view-serializable, implying that there exists a serial schedule S' that is view-equivalent to S. Our aim is to prove that S and S' are conflict-equivalent. Suppose, by contradiction, that S and S' are not conflict-equivalent. This means that there is a pair of conflicting actions, say $a_i(X)$ and $a_j(X)$, appearing in different order in the two schedules. Since one of these actions is a write and one is a read, we conclude that S' has a different "read-from" relation wrt S, which implies that S and S' are not view-equivalent.

2. It is well-known that if S is a 2PL schedule, then it is conflict-serializable. So, it remains to check whether if S is a conflict-serializable one-writer schedule, then it is also a 2PL schedule. This is disproved by the following counterexample:

$$S = w_1(x)\ r_2(x)\ r_3(y)\ w_1(y)$$

Indeed, it is easy to see that S is conflict serializable, but it is not a 2PL schedule.

Problem 2
Consider the following schedule S:

$$B(T_0)\ \ w_0(A)\ \ c_0\ \ B(T_1)\ \ w_1(A)\ \ B(T_2)\ \ r_2(A)\ \ w_2(B)\ \ c_1\ \ B(T_3)\ \ r_3(A)\ \ w_3(C)\ \ w_3(B)\ \ w_2(C)\ \ c_3\ \ c_2$$

where the action B means "begin transaction", and every write action performed by transaction T_i writes the value i on the element corresponding to the argument. Suppose that S is executed by PostgreSQL with all the transactions defined with the isolation level "read committed", and for each action different from B tell what is the effect of the action and what is the behavior of the system when it executes the action.

Solution to problem 2
We should remember the following important points regarding SQL and PostgreSQL:

- The lock-based concurrency control implementation of SQL keeps write locks until the end of the transaction, but read locks are released as soon as the SELECT operation is performed.
- The concurrency control strategy of PostgreSQL is a sort of multiversion control, where reads are never blocked (they read the value written by the last committed transaction), and write actions are performed on a local store (snapshot), and their effects are transferred to the database at commit.

- PostgreSQL adopts a recognition approach to deadlock.

The effect of each action and the behavior of the system is specified as follows:

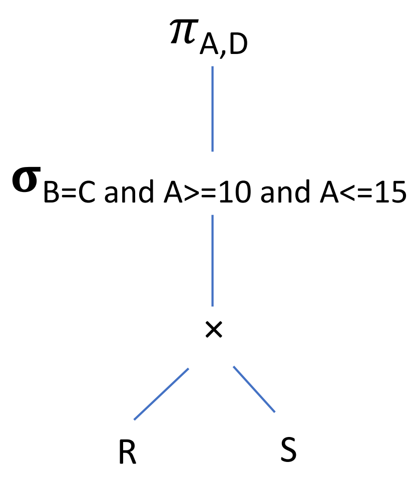
- $w_0(A)$ writes 0 on A in the local store associated to T_0
- c_0 commits T_0, and the value 0 for A is now in the database
- $w_1(A)$ writes 1 on A in the local store associated to T_1
- $r_2(A)$ reads the value 0, the value written by T_0, the last committed transaction that wrote on A
- $w_2(B)$ writes 2 on B in the local store associated to T_2
- c_1 commits T_1, and the value 1 for A is now in the database
- $r_3(A)$ reads the value 1, the value written by T_1, the last committed transaction that wrote on A
- $w_3(C)$ writes 3 on C in the local store associated to T_3
- when T_3 tries to execute the action $w_3(B)$, such action is not executed because T_2 holds the exclusive lock on B. So, T_3 is suspended waiting for the commit of T_2
- since C was written by T_3, when T_2 tries to execute the action $w_2(C)$, the system recognizes a deadlock, because T_3 is waiting for the end of T_2, and T_2 should now wait for the end of T_3. Therefore, action $w_2(C)$ is not executed, T_2 is aborted in order to resolve the deadlock, and T_3 becomes active again
- c_3 commits T_3, and the value 3 for C and the value 3 for B are now in the database
- c_2 is ignored, because T_2 was aborted

Problem 3
Let `R(A,B)` be a relation with 100.000 pages, and `S(C,D)` a relation with 500.000 pages. We know that (*i*) `R` has 2.000 values in `A`, uniformly distributed in the various tuples, (*ii*) there is a clustered, sparse B$^+$-tree index on `R` with search key `A`, (*iii*) 60 tuples of `R` fit in one page, (*iv*) we have 260 frames available in the buffer, and (*v*) every value and pointer occupy the same amount of space. Consider the query:

```
select A,D
from R, S
where B = C and A >= 10 and A <= 15
```

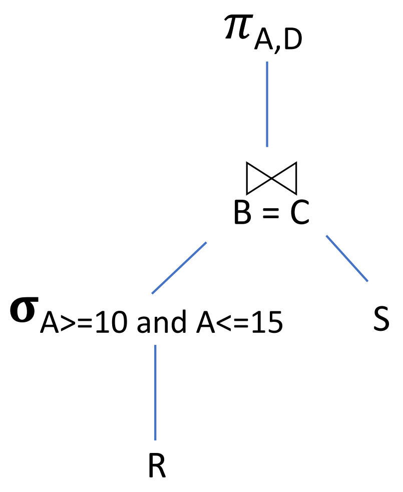
Show the logical query plan associated to the query, as well as the logical query plan and the physical query plan you would choose for executing the query efficiently. Also, tell which is the cost (in terms of number of page accesses) of executing the query according to the chosen physical query plan.

Solution to problem 3
The logical query plan associated to the query is shown below:



Logical query plan

The chosen logical query plan is shown below:



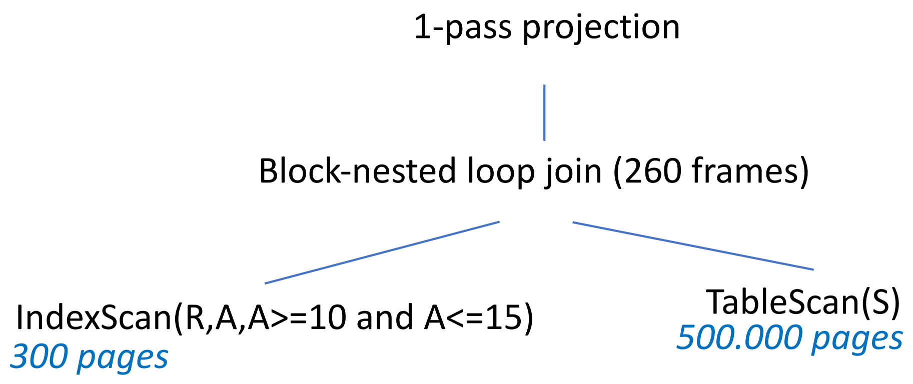
The chosen logical query plan

The selection operation can be performed by using the index. Indeed, this is a range selection, and can be well supported by a clustered index. So, let us evaluate the cost of computing all the pages of `R` satisfying the selction condition using the index, assuming that the index uses alternative 1. Since in every page we have space for 60 tuples of `R(A,B)`, it follows that we have space for 60 data entries, and taking into account the 67% occupancy rule, we have 40 data entries per page. Since the index is sparse, we have one data entry for each page of the relation, and so the number of leaves is 100.000/40 = 2.500. Since 60 tuples of `R` fit in one page, we have that the number of tuples of `R(A,B)` is 100.000 × 60 = 6.000.000, and the number of tuples with the same value for `A` is 6.000.000/2.000 = 3.000.

So, the number of pages with the same value of `A` is 3.000/60 = 50. Since we have 6 values for `A` to consider, we have to access $50 \times 6 = 300$ pages of `R(A,B)`. To access them, we use the index searching for the value 10 of `A` in `R(A,B)` and then we move to the data file in order to access the 300 pages of `R(A,B)`. The fan-out of the tree is (60+30)/2=45, and therefore the cost of accessing the first data record is $\log_{45} 2.500 + 1$.

The set of tuples in the pages of the data file accessed by the index form one operand of the join with `S(C,D)`. Since $300 > 260$, the join cannot be executed in one pass. One possibility is using a two pass algorithm, but the cost would be $(300 + 500.000) \times 3$. Instead, it is no difficult to see that it is more efficient to choose a block-nested loop algorithm for the join, requiring to scan the relation `S(C,D)` only twice.

The resulting physical query plan is shown below:


1-pass projection
Block-nested loop join (260 frames)
IndexScan(R,A,A>=10 and A<=15)
300 pages
TableScan(S)
500.000 pages

Physical query plan

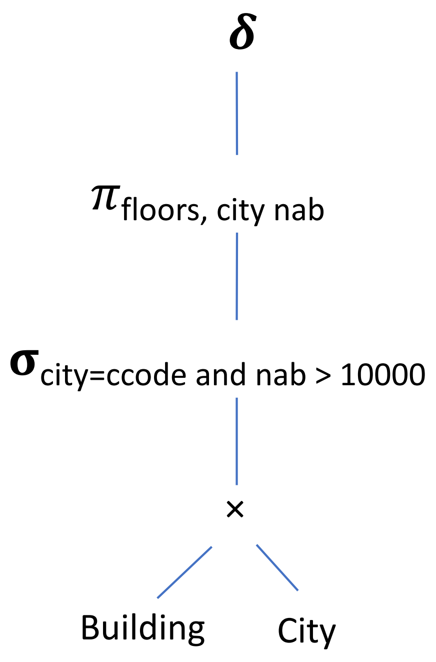
The cost of the execution of the whole physical query plan is $\log_{45} 2.500 + 1 + 300 + 2 \times 500.000 =$ 1.000.304 page accesses.

Problem 4
Let `Building(bcode,floors,area,value,cat,city)` be stored in a heap file with 840.000 tuples, and `City(ccode,nation,nab)` be stored in a heap file with 9.000.000 tuples. We assume that every value has the same size, that every page has room for 600 values, that $V($`Building,floors`$) = 100$, $V($`Building,city`$) = 300$, and that we have 85 free buffer frames available. Consider the query:

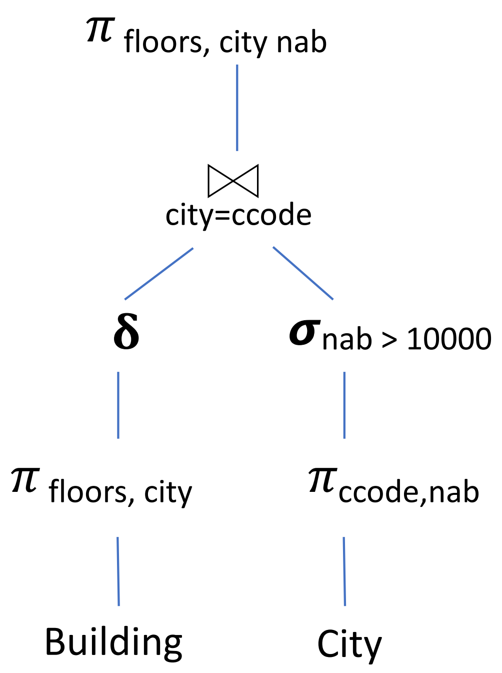
```
select distinct floors, city, nab
from Building, City
where city = ccode and nab > 10000
```

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.

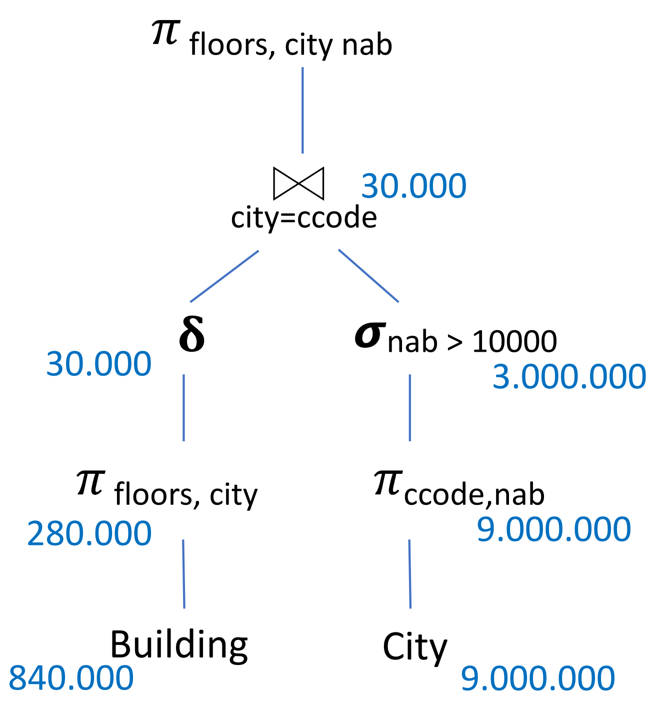
Solution to problem 4
The logical query plan associated to the query is shown below.



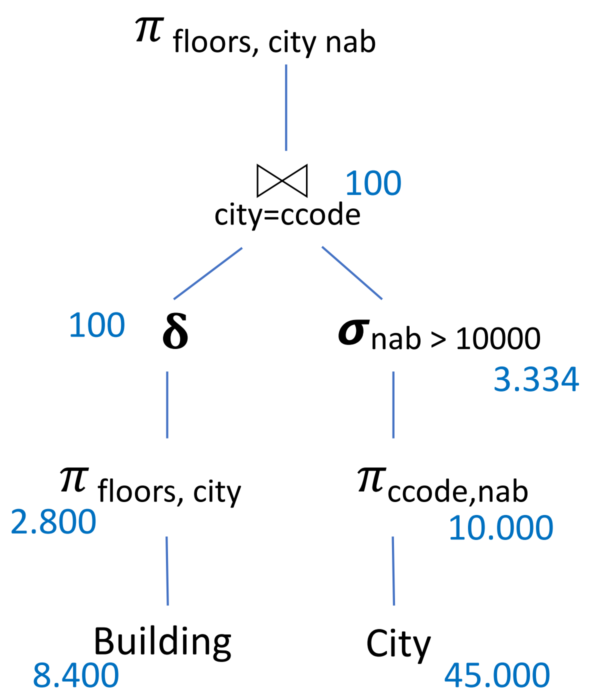
After pushing selection, duplicate elimination and projections we have the following logical query plan:



Note that duplicate elimination has not been pushed through the branch of `City`, because the right-hand side operand of the join contains a key of the relation, and therefore does not have duplicates. Note also that after duplicate elimination we cannot have more that $V(\texttt{Building,floors}) \times V(\texttt{Building,city}) = 30.000$ tuples. The number of tuples of the various nodes of the plan is shown here:

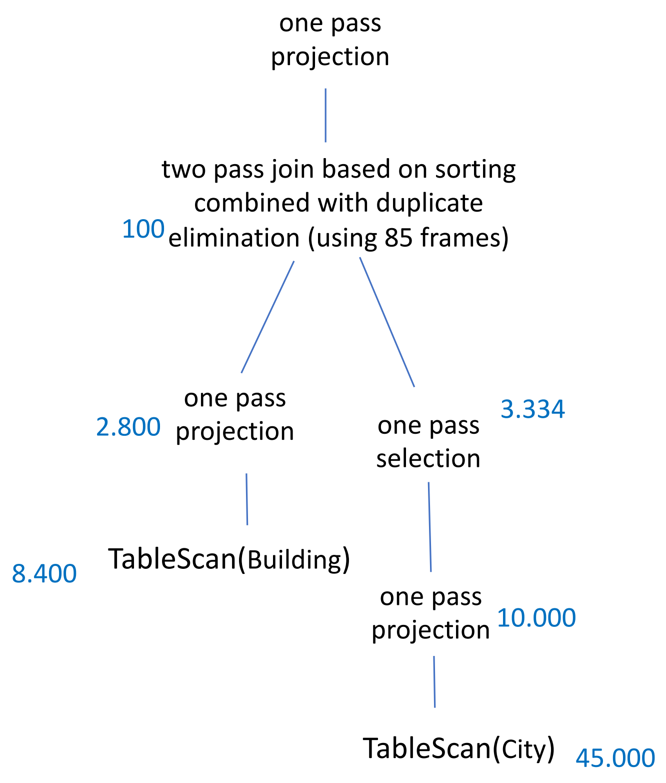


The number of pages is shown here:



Note that $2.800 + 3.334 < 85 \times 84$, and therefore we can use a two pass algorithm for the join. Also, we can combine the join with the duplicate elimination for `Building` if we use an algorithm based on sorting. Indeed, we can produce the sorted sublists for `Building` based on a sorting on $\langle$`city,floors`$\rangle$, and we can produce the sorted sublists for `City` based on a sorting on `ccode`, so that we can carry out the merging phase based on `city` and `ccode`, and simply eliminate duplicates during such merging phase of the two pass join algorithm.

Here is the physical query plan:



The cost is: 8.400 (reading of `Building`) + 45.000 (reading of `City`) + 2.800 (writing of the 2.800 / 85 = 33 sublists for `Building` during the first phase of the two pass join algorithm) + 3.334 (writing of the 3.334/85 = 40 sublists for `City` during the first phase of the two pass join algorithm) + 2.800 (reading of the 33 sublists during the merging phase of the two pass join algorithm) + 3.334 (reading of the 40 sublists during the merging phase of the two pass join algorithm) = 65.668 page accesses.

Problem 5

If $\mathtt{R}_1(\underline{\mathtt{A}},\mathtt{B})$ and $\mathtt{R}_2(\underline{\mathtt{A}},\mathtt{B})$ are two relations each with key A, then the *disjoint left-union* of $\mathtt{R}_1$ and $\mathtt{R}_2$, indicated as $\mathtt{R}_1 \oslash \mathtt{R}_2$, is the relation with attributes A,B and with the set of tuples specified as follows: (*i*) for each tuple $t \in \mathtt{R}_1$ such that $t.\mathtt{B}$ is not null, we have $t \in \mathtt{R}_1 \oslash \mathtt{R}_2$; (*ii*) for each tuple $t \in \mathtt{R}_1$ such that $t.\mathtt{B}$ is null and such that there exists $t' \in \mathtt{R}_2$ with $t.\mathtt{A} = t'.\mathtt{A}$, we have $t' \in \mathtt{R}_1 \oslash \mathtt{R}_2$. Suppose that we have a multiprocessor system with N nodes $n_1, \ldots, n_N$, each of them with $M > N$ free frames available, and that the two relations $\mathtt{R}_1$ and $\mathtt{R}_2$ are stored in node n_1.

1. Illustrate a parallel algorithm for computing $\mathtt{R}_1 \oslash \mathtt{R}_2$.
2. Assuming $B(\mathtt{R}_1) = 10.000$, $B(\mathtt{R}_2) = 15.000$, $N = 10$ and $M = 40$, describe the cost of the algorithm both in terms of the elapsed time, and in terms of number of page accesses.

Solution to problem 5

1. We use a hash function on the values of A for distributing the tuples of both $\mathtt{R}_1$ and $\mathtt{R}_2$ among the various nodes, in such a way that each node n_i will handle one bucket of $\mathtt{R}_1$ and one bucket of $\mathtt{R}_2$ resulted from the same value of the hash function, and therefore containing all the tuples with values of A that have given the result i for the hash function. This implies that if a tuple t of $\mathtt{R}_1$ has the same A-value as the tuple t' of $\mathtt{R}_2$, t and t' will be stored in the same node.

   - The distribution of the tuples is done at n_1, following the usual mechanism: we reserve one frame for each node (this is possible because $M > N$), and one frame for the input. We load each page of $\mathtt{R}_1$ in the input frame, one by one. For each tuple t in the input frame such that $t.\mathtt{B}$ is not null, we write t in the output of n_1, because it has to be part of the result. For each tuple t in the input frame such that $t.\mathtt{B}$ is null, we apply the hash function so as to obtain the bucket (or, the processor) where the tuple should go, and we store the tuple in the frame corresponding to the "correct" node. Obviously, when a frame is full, we ship the tuples to the corresponding node. The same algorithm is carried out for $\mathtt{R}_2$. If the hash function is good, each node receives at most $B(\mathtt{R}_1)/N$ pages with tuples of $\mathtt{R}_1$ and at most $B(\mathtt{R}_2)/N$ pages with tuples of $\mathtt{R}_2$, where $B(R)$, as usual, denotes the number of pages of R.
   - The computation at each node n_i is done as follows: the pages of $\mathtt{R}_1$ received are stored in a file F_1 and the pages of $\mathtt{R}_2$ received are stored in a different file F_2. Then, using a multi-pass strategy, we iteratively produce a suitable number of sorted sublists for F_1 and a suitable number of sorted sublists for F_2. Obviously, the sorting is done based on the attribute A. In particular, the goal is to reach a situation where the number of sorted sublists for F_1 is less than $(M \times B(\mathtt{R}_1)/(B(\mathtt{R}_1)+B(\mathtt{R}_2))$, and for F_2 is less than $(M \times B(\mathtt{R}_2)/(B(\mathtt{R}_1)+B(\mathtt{R}_2))$. At this point, we carry out a final "merge-like" phase using M frames, and we compute the final result for the buckets in n_i. For doing that, we simply apply the definition of $\mathtt{R}_1 \oslash \mathtt{R}_2$.

2. The worst-case cost of the algorithm in terms of total number page accesses is determined as follows.

   - We have the cost of reading the pages of both relations in the tuple distribution phase: 10.000 + 15.000 = 25.000;
   - For each of the 10 nodes $n_1, \ldots, n_{10}$, we have the following:
     - the cost 10.000/10 = 1.000 of writing the bucket for $\mathtt{R}_1$, and the cost 15.000/10 = 1.500 of writing the bucket for $\mathtt{R}_2$. Since 2.500 > 40, 2.500 > 40 × 39, and 2.500 < 40 × 40 × 39, we need 3 passes to solve the problem, and therefore the additional cost is 5 × 2.500 (as usual, we ignore the cost of writing the result). The total cost at each node is 15.000.

Therefore the cost in terms of number of page accesses is: $25.000 + 15.000 \times 10 = 175.000$ page accesses, and the cost in terms of elapsed time is $25.000 + 15.000 = 40.000$ page accesses.

Notice that the above strategy could be made more efficient, by avoiding storing the pages shipped to nodes $n_2, \ldots, n_{10}$. However, we do not discuss such an improvement here.