Data Management – exam of 08/06/2022 (Compito A)

Problem 1

Given a schedule S, a serial schedule S_1 on the same transactions as S is said to be "begin-order preserving with respect to S" if it satisfies the following property: for every pair of transactions T_i, T_j in S, if the first action of T_i precedes the first action of T_j in S, then T_i precedes T_j in S_1. A schedule S is called *begin-order preserving conflict serializable* if there exists a serial schedule S_1 on the same transactions that is both conflict equivalent to S and begin-order preserving with respect to S.

1.1 Prove or disprove the following claim: every conflict serializable schedule is "begin-order preserving conflict serializable".

1.2 Is the problem of checking whether a schedule is "begin-order preserving conflict serializable" decidable? If the answer is negative, then motivate the answer; if the answer is positive, then exhibit an algorithm for the problem, provide evidence of the correctness of the algorithm and illustrate its computational conplexity.

Solution 1

1.1 The intuition is that begin-order preservation is independent from conflict serializability and therefore it should be easy to disprove the claim. Indeed, it is sufficient to consider the schedule $S : w_2(A)\ r_1(B)\ w_2(B)$ that is clearly conflict serializable. In particular, the serial schedule T_1, T_2 is the only serial schedule that is conflict equivalent to S, and is clearly not begin-order preserving with respect to S.

1.2 The problem of checking whether a schedule S is "begin-order preserving conflict serializable" is decidable. An algorithm solving the problem is based on the following property: by definition, the only serial schedule that is begin-order preserving with respect to S is the serial schedule S' that is coherent with the order imposed by the first actions of the transactions, i.e., the serial schedule where T_i comes immediately after T_j if the first action of T_i comes after the first action of T_j and no transaction starts in between. Therefore, an appropriate algorithm simply checks that such serial schedule S' is conflict equivalent to S. Clearly, constructing the serial schedule S' can be done in linear time with respect to the size of S, and checking if S and S' are conflict equivalent can be done in quadratic time with respect to the size of S and S'.

Problem 2

Consider the following schedule S (where we have relaxed the condition that no transaction contains more than one occurrence of the same action):

$$B(T_1)\ r_1(D)\ r_1(A)\ w_1(A)\ B(T_2)\ r_2(A)\ w_2(A)\ B(T_3)\ r_3(D)\ w_3(D)\ r_1(D)\ c_3\ r_1(D)\ c_1\ c_2$$

where the action B means "begin transaction", the initial values of A and D are 10 and 30, respectively, and every write action increases the value of the element on which it operates by 10. Suppose that S is executed by PostgreSQL, and describe what happens when the scheduler analyzes each action (illustrating also which are the values read and written by all the "read" and "write" actions) in both the following two cases: (1) all the transactions are defined with the isolation level "read committed"; (2) all the transactions are defined with the isolation level "repeatable read".

Solution 2

We first deal with the isolation level "read committed". We remind the reader that such isolation level does not prevent the unrepeatable read anomaly nor the lost update anomaly.

- $r_1(D)$: T_1 reads 30.
- $r_1(A)$: T_1 reads 10.

- $w_1(A)$: T_1 writes 20 on A in the local store.
- $r_2(A)$: T_1 reads 10.
- $w_2(A)$: not executed, because T_1 holds the write lock on A; so T_2 must wait for the end of T_1 and therefore is suspended.
- $r_3(D)$: T_3 reads 30.
- $w_3(D)$: T_3 writes 40 on D in the local store.
- $r_1(D)$: T_1 reads 30.
- c_3: T_3 commits and the value 40 for D is written in the database.
- $r_1(D)$: T_1 reads 40.
- c_1: T_1 commits and the value 20 for A is written in the database.
- $w_2(A)$: T_2 resumes and writes 30 on A in the local store.
- c_2: T_2 commits and the value 30 for A is written in the database.

We now deal with the isolation level "repeatable read" (in bold the difference with respect to the previous case). We remind the reader that such isolation level prevents both the unrepeatable read anomaly and the lost update anomaly.

- $r_1(D)$: T_1 reads 30.
- $r_1(A)$: T_1 reads 10.
- $w_1(A)$: T_1 writes 20 on A in the local store.
- $r_2(A)$: T_1 reads 10.
- $w_2(A)$: not executed, because T_1 holds the write lock on A; so T_2 must wait for the end of T_1 and therefore is suspended.
- $r_3(D)$: T_3 reads 30.
- $w_3(D)$: T_3 writes 40 oin D in the local store.
- $r_1(D)$: T_1 reads 30.
- c_3: T_3 commits and the value 40 for D is written in the database.
- $r_1(D)$: T_1 **reads 30**.
- c_1: T_1 commits and the value 20 for A is written in the database.
- $w_2(A)$: T_2 **aborted with message: "ERROR: could not serialize access due to concurrent update"**.
- c_2: **ignored, because T_2 has aborted**.

Problem 3

Let τ indicate the ternary operator such that $\tau(R, S, T) = R \cup_s (S - T)$, where R, S and T are three relations with the same schema and without duplicates, $\cup_s$ indicates set union and $-$ indicates set difference.

3.1 Design and describe in detail a two pass algorithm that, given R, S, T, each one stored as a heap, computes $\tau(R, S, T)$.

3.2 Tell under which condition the algorithm can be used and illustrate the cost of the algorithm in terms of number of page accesses.

Solution 3

Assuming that the buffer has M frames available, a two pass algorithm based on sorting can be defined as follows (similarly, we could design a two pass algorithm based on hashing, but we will not illustrate such an algorithm).

- First pass: produce $M - 1$ sorted sublists for R, S and T (each one with M pages).
- Second pass: reserve one buffer frame for the output and one buffer frame for each of the sublists produced in the first pass, and perfom the "merge" phase of algorithm, based on the fact that all the sublists are sorted and we know exactly which is the relation associated to each buffer frame. As usual, whenever an input frame is exhausted, we read in that frame the next page of the corresponding sublist, if it exists, and whenever the output frame is full, we write its content in the result. At each step of the merge phase we do the following (tup(X) denotes the least tuple of relation X that we have in the buffer or NULL if no tuple of X exists in the buffer; we assume that advancing X when tup(X) is NULL has no effect, and we assume that any condition on NULL is always false):
  - if both tup(R) and tup(S) are NULL, then stop
  - if tup(S) is NULL and tup(R) is not NULL, then copy R in the output and stop
  - If tup(T) $<$ tup(S), then advance T and go the next iteration
  - If tup(T) $=$ tup(S), then advance S and T and go the next iteration
  - If tup(R) $<$ tup(S), then put tup(R) in the output, advance R and go to the next iteration
  - If tup(R) $=$ tup(S), then put tup(R) in the output, advance R and S and go to the next iteration
  - If tup(R) is NULL or tup(R) $>$ tup(S), then put tup(S) in the output, advance S and go to the next iteration

The above algorithm can be used under the following condition:

$$\lceil (B(R)/M \rceil + \lceil B(S)/M \rceil + \lceil B(T)/M) \rceil \leq M - 1$$

where $B(X)$ denotes the number of pages of relation X. The cost of the algorithm is, as usual for two pass algorithms (we obviousy ignore the cost of writing the result):

$$3 \times (B(R) + B(S) + B(T)).$$

Problem 4

Consider the relations `Restaurant(code,citycode,seats)` with 18.000.000 tuples, and `City(citycode,region,country)` occupying 195 pages, each page with 100 tuples (keys are underlined). We assume that 150 values are in the attribute `country`, that each value (regardless of the type) requires the same number of bytes and that we have 200 frames available in the buffer. Consider the query

```
select country, sum(seats)
from Restaurant r, City c where r.citycode = c.citycode
group by country
```

and describe the algorithm you would use to execute the query, illustrating the number of page accesses required by the execution of the algorithm.

Solution 3
We immediately notice that each page can contain 300 values (100 tuples of three values each). Also, we immediately notice that the whole relation `City` fits in the buffer, and that the 150 tuples of the form (`country, sum(seats)`) in the result of the query fits in one frame F of the buffer (because 300 values fit in one page, and therefore also in one frame of the buffer). It follows that we can use a one pass algorithm as follows:

- load the relation `City` in 195 frames of the buffer;
- load the relation `Restaurant` one page P at a time in one frame of the buffer and for each tuple t of `Restaurant` in P single out the tuple t' of `City` (at most one) joining with t. Consider the value c that the tuple t' has in the attribute `country`, and update in F the value of `sum(seats)` corresponding to c, making use of the value that t has in the attribute `seats`.

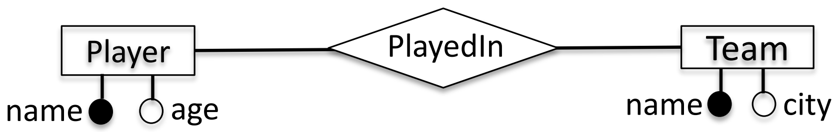
The number of page accesses required by the algorithm is simply $B(\texttt{Restaurant}) + B(\texttt{City})$. Since each page has space for 100 tuples of `City`, and since `Restaurant` and `City` have the same number of attributes, and since we assume that all values requires the same number of bytes, we conclude that each page has space for 100 tuples of `Restaurant` as well, and therefore $B(\texttt{Restaurant}) = 18.000.000/100 = 180.000$. Therefore, the number of page accesses required by the algorithm is $180.000 + 195 = 180.195$.

Problem 5
Consider a graph database with nodes of type `Player` with properties `name` (identifying the player) and `age`, nodes of type `Team` with properties `name` (identifying the team) and `city`, and edges of type `PlayedIn`, connecting each player with the teams they have played in. In such database, for example, one may represent the node `p1` of type `Player` with `name`:"Totti" and `age`:39 connected to node `t1` of type `Team` with `name`:"Roma" and `city`:"Roma" by means of an edge of type `PlayedIn`.

5.1 Illustrate how you would represent the above database as a schema in the relational model.

5.2 Assuming that the three most relevant queries are (1) given the name of a player, compute the teams where the player played, (2) produce the sorted list of ⟨name of player, name of team⟩ for players and the teams they have played in, and (3) compute all the names of the teams of a given city, describe the file organizations you would choose for each of relations in the relational database mentioned above and then describe the algorithms for executing the three queries on the basis of such file organizations.

Solution 5
The conceptual schema representing the graph database is as follows:



Consequently, the correct relational schema to represent such database is as follows:

```
Player(name, age)
Team(name, city)
PlayedIn(pname, tname)
  foreign key PlayedIn(pname) ⊆ Player(name)
```

`foreign key PlayedIn(tname)` $\subseteq$ `Team(name)`

We immediately notice that in order to support the second query it is essential to have the sorted list of ⟨name of player, name of team⟩ in secondary storage. Notice that we get such a list simply by building a B^+-tree index on `PlayedIn` with search key ⟨`pname`,`tname`⟩ using alternative 1: the leaves of the tree will be exactly the sorted list of interest. The nice feature of such an index is that it also supports the first query, because a B^+-tree index well supports all queries searching for a value corresponding to a prefix of the search key. As for the third query, we can simply build a hash-based index on `Team` with search key `city`. Based on the above decisions, the algorithms for the various queries are as follows:

- Query 1: given the name n of a player, use the B^+-tree index on `PlayedIn` to search for the value n in the attribute `pname`, and then return all the values in the attribute `tname` associated to n.
- Query 2: simply return all the records in the leaves of the B^+-tree index.
- Query 3: given a city c, use the hash index to retrieve all the records of `Team` having c in the attribute `city`, and return all the values in the attribute `name` of such records.