Data Management – exam of 08/06/2023 **(B)**

Problem 1

Let `Trainer(id,nation)` be a relation with 500 pages stored in a file sorted on ⟨`id,nation`⟩, `Team(name,city)` a relation with 100 pages sorted on ⟨`name,city`⟩ and `Coached(id,nation,name,city)` a relation with 35.000 pages sorted on ⟨`id,nation,name,city`⟩. With the goal of knowing, for each trainer, the teams that (s)he has not coached, we want to compute the set difference between the cartesian product of `Trainer` and `Team` and the relation `Coached`. Tell which is the best algorithm that a query engine with 107 free buffer frames should use for this task, indicating also the cost of executing such algorithm.

Solution 1

An obvious and acceptable solution would be to compute the whole cartesian product sorted on the basis of the same sorting criteria as `Coached`, store it in a temporary relation and then compute in one pass the set difference between such temporary relation and the relation `Coached`. However, it is worth looking for a more efficient method, that does not require to materialize the cartesian product. For this purpose, we can notice that, since `Trainer(id,nation)` is sorted on ⟨`id,nation`⟩ and `Team(name,city)` is sorted on ⟨`name,city`⟩, we can easily compute the cartesian product of `Trainer` and `Team` in such a way that the resulting relation is sorted on ⟨`id,nation,name,city`⟩: it is sufficient to combine the tuples of the two relation by respecting the order of each of them. Also, notice that the whole relation `Team` fits in the buffer. This means that if we read `Team` in 100 buffer frames and then we read `Trainer` one page at a time using the 101th buffer frame, we can produce the cartesian product sorted on ⟨`id,nation,name,city`⟩ one page at a time using the 102th buffer fame. In turn this means that, based on the sorting on ⟨`id,nation,name,city`⟩ of both the cartesian product of `Trainer` and `Team` and the relation `Coached`, we can compute on the fly their difference by reading the relation `Coached` one page at a time in the 103th buffer frame and using the 104th buffer frame as the output frame. Since we access each page of all the relations only once, the whole algorithm is one-pass, and its cost is 100 + 500 + 35.000 = 35.600 page accesses.

Problem 2

Answer the following two questions, providing a suitable motivation for each answer.

3.1 Give an example of three transactions which have the following properties: (*i*) there exists a schedule S on the three transactions such that when S is given in input to a timestamp-based scheduler, a deadlock occurs; (*ii*) for each pair of the three transactions and for any schedule S on such pair, no deadlock occurs when S is given in input to a timestamp-based scheduler.

3.2 Try to generalize the above example to the case of n transactions. More precisely, try to come up with an example of n transactions having the following properties: (*i*) There exists a schedule S on the n transactions such that when S is given in input to a timestamp-based scheduler, a deadlock occurs. (*ii*) For each $n-1$ transactions of the n chosen transactions and for any schedule S on such $n-1$ transactions, no deadlock occurs when S is given in input to a timestamp-based scheduler.

Solution 2

3.1 Here is the example:

$T_1 = w_1(x_1)\ w_1(x_2)$,
$T_2 = w_2(x_2)\ w_2(x_3)$,
$T_3 = w_3(x_3)\ w_3(x_1)$.

Consider now the schedule S: $w_1(x_1)\ w_2(x_2)\ w_3(x_3)\ w_1(x_2)\ w_2(x_3)\ w_3(x_1)$. It is easy to see that:

- when S is given in input to a timestamp-based scheduler, a deadlock occurs;
- for every pair of transactions T_1, T_2, T_3 and for every schedule S' on such pair, no deadlock occurs when S' is given in input to a timestamp-based scheduler.

3.2 The generalization of the above example to n transactions is immediate:

$T_1 = w_1(x_1)\ w_1(x_2)$,
$T_2 = w_2(x_2)\ w_2(x_3)$,
...
$T_{n-1} = w_{n-1}(x_{n-1})\ w_{n-1}(x_n)$,
$T_n = w_n(x_n)\ w_n(x_1)$.

Consider now the schedule S: $w_1(x_1)\ w_2(x_2) \dots w_{n-1}(x_{n-1})\ w_n(x_n)\ w_1(x_2)\ w_2(x_3) \dots\ w_{n-1}(x_n) w_n(x_1)$. It is easy to see that:

– when S is given in input to a timestamp-based scheduler, a deadlock occurs;

– for every pair of transactions $T_1, T_2, T_3, \dots, T_n$ and for every schedule S' on such pair, no deadlock occurs when S' is given in input to a timestamp-based scheduler.

Problem 3

Let `R(A,B,C,D)` (with 52.000 pages) and `S(E,F,G,H)` (with 20.000 pages) be two relations stored in a heap at processor P_0. We know that 300 free buffer frames are available at P_0 and that 100 tuples of each relation fit in one page. Also, we know that attribute `C` contains 500 values uniformly distributed in the tuples of `R`, and the same holds for attribute `H` in `S`. Finally, we know that there are 10 processors $P_1, \dots, P_{10}$ that we can use for processing queries, each with 80 free frames available. Consider the query that computes the equi-join between `R` and `S` on the condition `C = H`, and compare the following two cases:

3.1 The query is executed at processor P_0.

3.2 The query is executed in parallel after sending the tuples of the two relations to processors $P_1, \dots, P_{10}$.

For each of the above cases, illustrate the algorithm you would use and discuss its cost in terms of number of page accesses (case 1.1) and elapsed time (case 1.2).

Solution 3

3.1 The query is executed at processor P_0. We cannot use a one-pass algorithm for computing the join, but, since $300 \times 299 > 52.000 + 20.000$, we can use a two-pass algorithm based on sorting. We know that we have two types of such algorithms, the simple sort-based join algorithm, and the sort-merge join algorithm and we know that the latter is more efficient, as far as the problem of "fragments with too many joining tuples" does not occur.

Let us compute the maximum number of tuples joining between the two relations, by computing the number of tuples in `R` having the same value in the joining attribute `C` as $52.000 \times 100/500 = 10.400$, and the number of tuples in `S` having the same value in the joining attribute `H` as $20.000 \times 100/500 = 4.000$. This means that we need at least $4.000/100 = 40$ pages in the buffer to hold all the tuples of the maximum fragments of `S`, and apply the one-pass join for such fragments. Now, in the case of sort-merge join algorithm, we have $52.000/300 = 174$ sublists for `R` and $20.000/300 = 67$ sublists for `S`, and therefore, since $300 - (174 + 67) = 59 > 40$ we do have sufficient room in the buffer for holding the fragments of `S` with tuples with the same value of `H` during the merging phase of the algorithm. We conclude that we choose the sort-merge join algorithm, with a cost of $3 \times (52.000 + 20.000) = 216.000$.

3.2 The query is executed in parallel after sending the tuples of the two relations to processors $P_1, \dots, P_{10}$.

We assume that we have a good hash function on the values in the attributes `C` and `H` so that the hash-based partitioning distributes the tuples of the two relations uniformly. Using such hash function, we read the two relations `R` and `S` at processor P_0 and we send to each of the 10 processors $P_1, \dots, P_{10}$ 5.200 pages of `R` and 2.000 pages of `S`. Let us now consider the situation at any of the 10 processor, say processor P_i. We immediately notice that we cannot use a one-pass or a two-pass algorithm (e.g.. based on sorting) for computing the join, because $5.200 + 2.000 > 80 \times 79$, but we can use a three-pass algorithm, because $5.200 + 2.000 < 80 \times 79$. Also, notice the number of tuples in `R` having the same value in the joining attribute `C` as $520.000/50 = 10.400$, and the number of tuples in `S` having the same value in the joining attribute `H` as $200.000/50 = 4000$. This means that, again, we need at least 40 pages in the buffer to hold all the tuples of the maximum fragments of `S`, and apply the one-pass join for such fragments. In the case of three-pass sort-merge join algorithm, we have $5.200/80 = 65$ sublists for `R` and $2.000/80 = 67$ sublists for `S` generated in the first pass, and only one sublist for each of the two relations generated at the second pass. So, we have plenty of buffer frames available for holding all the tuples of the fragments of `S` with the same value for `S`, and apply the one-pass join for such fragments during the third pass (the "merging phase"). We conclude that we again choose the sort-merge join algorithm, and, if we first store the two relations in the secondary storage when they arrive at processor P_i, and then we execute the sort-merge join algorithm, the elapsed time will be $52.000 + 20.000$ (for reading the two relations at processor P_0) + $5.200 + 2.000$ (for storing the relation in the processor P_i) $+5 \times (5.200 + 2.000) = 115.200$ page accesses as elapsed time. We can actually do better by avoiding to store the two relations when they arrive at the processor, simply producing the corresponding sorted sublists, thus obtaining $52.000 + 20.000 + 4 \times (5.200 + 2.000) = 100.800$ page accesses as elapsed time.

Problem 4
Consider the following schedule S:

$$B_1\ w_1(X)\ B_2\ w_2(Y)\ B_3\ w_3(Z)\ c_3\ B_4\ r_4(Z)\ r_4(Y)\ w_4(W)\ B_5\ r_5(W)\ c_4\ w_5(W)\ c_5\ w_1(Y)\ c_1\ w_2(X)\ c_2$$

where the action B_i means "begin transaction T_i", the initial value of every item X, Y, Z, W is 200 and every write action increases the value of the element on which it operates by 200. 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 4
We first deal with the isolation level "read committed". We remind the reader that such isolation level does not prevent the unrepeatable read anomaly (that, however, cannot occur in our case) nor the lost update anomaly (that actually occurs).

- $w_1(X)$: T_1 writes 400 on X in the local store.
- $w_2(Y)$: T_2 writes 400 on Y in the local store.
- $w_3(Z)$: T_1 writes 400 on Z in the local store.
- c_3: T_3 commits and the value 400 for Z is written in the database.
- $r_4(Z)$: T_4 reads 400.
- $r_4(Y)$: T_4 reads 200.
- $w_4(W)$: T_4 writes 400 on W in the local store.
- $r_5(W)$: T_5 reads 200.
- c_4: T_4 commits and the value 400 for W is written in the database.
- $w_5(W)$: T_5 writes 600 on W in the local store (**a kind of lost update anomaly**).
- c_5: T_5 commits and the value 600 for W is written in the database.

- $w_1(Y)$: not executed, because T_2 holds the write lock on Y; so T_1 is suspended because it must wait for the end of T_2.
- $w_2(X)$: not executed because T_1 holds the write lock on X; a deadlock is recognized, T_2 is aborted and T_1 is resumed, writing the value 400 on Y in the local store
- c_1: T_1 commits and the value 400 both for X and for Y is written in the database.
- c_2: T_2 was aborted and now it rollbacks.

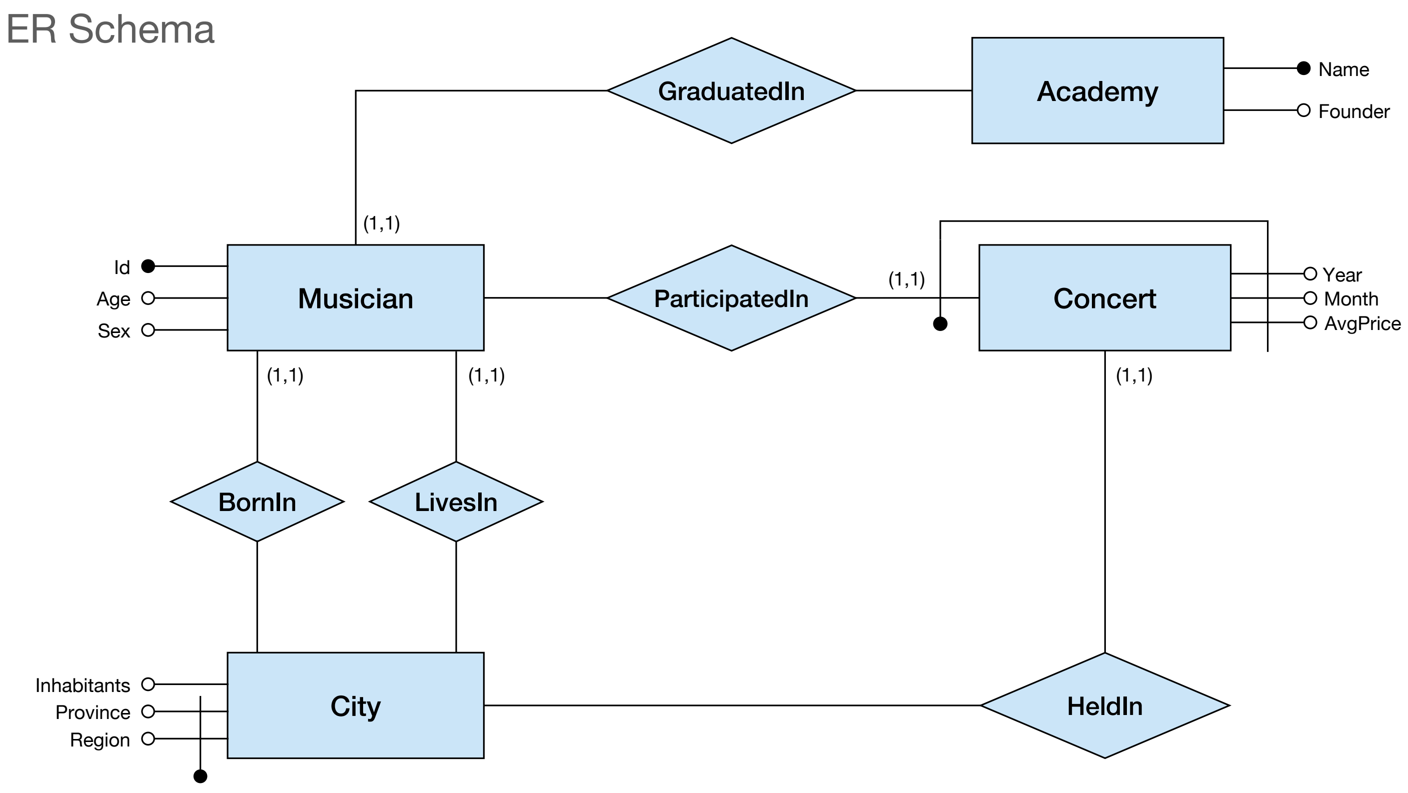
We now deal with the isolation level "repeatable read" (in bold the difference with respect to the previous case). We remind the reader that such isolation level prevents both the unrepeatable read anomaly (that, however, cannot occur in our case) and the lost update anomaly (that can occur).

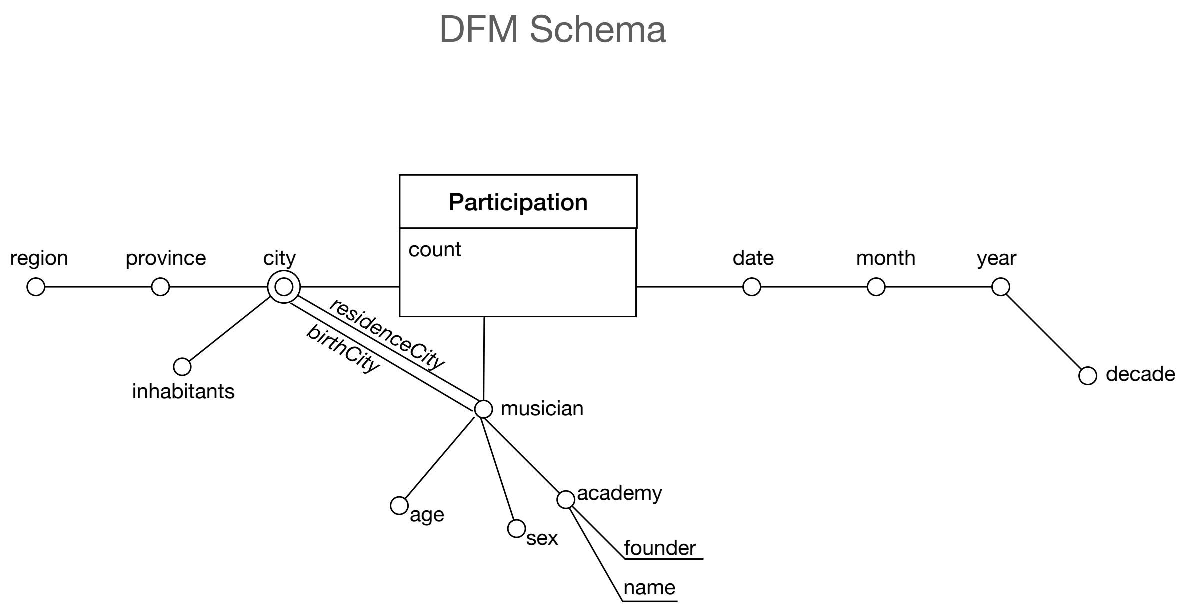
- $w_1(X)$: T_1 writes 400 on X in the local store.
- $w_2(Y)$: T_2 writes 400 on Y in the local store.
- $w_3(Z)$: T_1 writes 400 on Z in the local store.
- c_3: T_3 commits and the value 400 for Z is written in the database.
- $r_4(Z)$: T_1 reads 400.
- $r_4(Y)$: T_1 reads 200.
- $w_4(W)$: T_4 writes 400 on W in the local store.
- $r_5(W)$: T_1 reads 200.
- c_4: T_4 commits and the value 400 for W is written in the database.
- $w_5(W)$: T_5 **aborted with message: "ERROR: could not serialize access due to concurrent update"**.
- c_5: T_5 rollbacks.
- $w_1(Y)$: not executed, because T_2 holds the write lock on Y; so T_1 is suspended because it must wait for the end of T_2.
- $w_2(X)$: not executed because T_1 holds the write lock on X; a deadlock is recognized, T_2 is aborted and T_1 is resumed, writing the value 400 on Y in the local store
- c_1: T_1 commits and the value 400 both for X and for Y is written in the database.
- c_2: T_2 was aborted and now it rollbacks.

Problem 5 (only for students enrolled in an A.Y. before 2021/22 who do **not** do the project)
Consider a database B about musicians and concerts, where: (*i*) for each musician we are interested in the id, the age, the sex, the city of birth, the city (s)he is living in, the music academy where (s)he graduated and the concerts to which (s)he participated; (*ii*) for each concert we are interested in the average price of the ticket, in the month and the year when it was held, with the decade of the year, and the city where the concert was held; (*iii*) for each city we are interested in the province, the region and the number of inhabitants; (*iv*) for each music academy we are interested in the name and the founder. We want to build a data warehouse on all the above data for various analyses on participation of musicians to concerts in

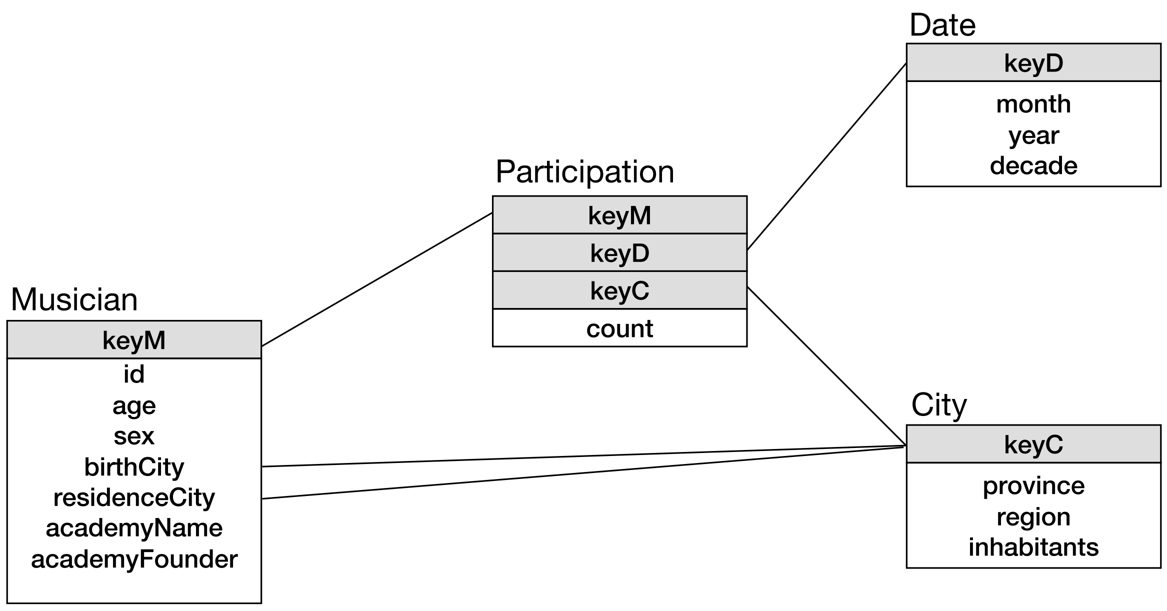
the last 50 years. You are asked to show (1) the ER schema of the database, (2) the DFM schema of the data warehouse, (3) the corresponding star schema (optionally, with the queries used to populate the star schema tables), and (4) based on such tables, the SQL query that computes the number of musicians who participated in concerts, for each region of the city of birth of the musicians, and for each music academy where the musicians graduated.

Solution 5


ER Schema
GraduatedIn
Academy
Name
Founder
(1,1)
Id
Age
Sex
Musician
ParticipatedIn
(1,1)
Concert
Year
Month
AvgPrice
(1,1)
(1,1)
(1,1)
BornIn
LivesIn
Inhabitants
Province
Region
City
HeldIn


DFM Schema
Participation
count
region
province
city
inhabitants
residenceCity
birthCity
musician
date
month
year
decade
age
sex
academy
founder
name

Star Schema



SQL query

Number of musicians who participated in concerts, for each region of the city of birth of the musician, and for each music academy where the musician graduated

```
SELECT c.region, m.academyName, count(*) as noOfMusicians
FROM Participation p JOIN Musician m ON m.id =  p.keyM
          JOIN City c ON m.birthCity = c.keyC
GROUP BY c.region, m.academyName
```