Data Management (A.A. 2023/24) – exam of 05/06/2024
Solutions

Problem 1 A transaction T is called *eager* if all its requests for an exclusive lock appear just after the "begin" action of T (and, therefore, before every other action of T). A lock-based scheduler with both shared and exclusive locks is called *disciplined* if it behaves like a 2PL scheduler with the only difference that it aborts all the transactions that are not eager.

1.1 Show a schedule that is accepted by the 2PL scheduler (with both shared and exclusive locks) and is not accepted by a disciplined scheduler.

1.2 Prove or disprove that a disciplined 2PL sheduler can avoid to deal with deadlock management.

Solution 1

1.1 We adopt the classical notion of "schedule accepted by a scheduler (in this case, a disciplined scheduler)": S is accepted by a disciplined scheduler D if, given S in input to D, the output produced by D is exactly S.

We could even show a schedule with just one transaction where exclusive locks are asked after some actions and therefore is not eager, but we prefer to show a more interesting schedule, one without lock operations. The following schedule

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

is obviously accepted by the 2PL scheduler (with both shared and exclusive locks), but is not accepted by a disciplined scheduler, because either transaction 1 is not eager and is therefore aborted by the disciplined scheduler, or transaction 1 is eager and acquires the exclusive lock both on x and y, thus forcing transaction 2 to be put in a waiting stage (and resuming when transaction 1 ends) when trying to execute $w_2(y)$.

1.2 We disprove that a disciplined 2PL sheduler can avoid to deal with deadlock management by considering the following schedule S:

$$w_2(y)\, w_1(x)\, r_1(y)\, r_2(x).$$

It is easy to see that we can add lock commands to S to make all transactions eager and to make S following the 2PL protocol. At the same time it is also easy to see that a deadlock occurs when a disciplined scheduler processes S, because $r_1(y)$ must wait for transation 2 to release the exclusive lock on y and at the same time $r_2(x)$ must wait for transation 1 to release the exclusive lock on x. This implies that a disciplined scheduler cannot avoid dealing with deadlock management.

Problem 2 Consider the relations `R(A,`<u>`B`</u>`)` and `S(`<u>`B`</u>`,`<u>`C`</u>`)`, where the attributes forming the key are underlined. Each of them is stored in a heap with 1.000 pages. We have to compute the equi join of `R` and `S` on the condition `R.B` = `S.B`, knowing that we have M free frames available in the buffer. For the two cases of $M = 3$ and $M = 32$, answer the following questions. (2.1) Is the block-nested loop applicable? If the answer is negative, provide a motivation; if the answer is positive, illustrate the algorithm and tell which is its cost in terms of number of page accesses. (2.2) Illustrate what you think is the most efficient algorithm, and tell which is its cost in terms of number of page accesses.

Solution 2

α) Let us consider the case where $M = 3$.

2.1 The block-nested loop algorithm (which, in this case, coincides with the page-based nested loop) for the equi-join is always applicable and the cost in this case is $B(R) + \lceil B(R)/(M-2) \rceil \times B(S) = 1.000 + \lceil 1.000/1 \rceil \times 1.000 = 1.001.000$.

2.2 In this case, the simple sort join and the multi-pass merge-sort join behave in the same way. Either one of them is the most efficient algorithm: we sort the two relations using 3 frames, and then we compute the join by a "merging step". The cost is $2 \times 1.000 \times log_2 1000 + 2 \times 1.000 \times log_2 1000 + 2.000 = 42.000$.

β) Let us consider the case where $M = 32$.

2.1 The block-nested loop algorithm for the equi-join is always applicable and the cost in this case is $B(R) + \lceil B(R)/(M-2) \rceil \times B(S) = 1.000 + \lceil 1.000/30 \rceil \times 1.000 = 35.000$.

2.2 Since the attribute `B` is a key for `R`, we do not have the problem of too large joining fragments for the two relations. It follows that the most efficient algorithm is the multi-pass merge-sort join algorithm: we (iteratively) produce a number of sorted sublists less than or equal to $M - 2$ using K passes, and then we compute the join by a "merging step". Since $32 \times 31 < 2.000$ and $32 \times 31 \times 31 > 2.000$, we need $K = 3$ passes for producing the sorted sublists. Therefore, the cost is $4 \times 1.000 + 4 \times 1.000 + 2.000 = 10.000$. Note that the cost of the simple sort join is $6 \times 1.000 + 6 \times 1.000 + 2.000 = 14.000$.

Problem 3 Consider the relations `R(A,B,C,D,E,F)` and `S(A,B,G,H)`, where `R` contains 260.000 tuples (20 tuple per page), `R` contains 1.000 distinct values uniformly distributed in the integer attribute `A`, there is a hash-based index for `S` on the integer attribute `A`, there is a B$^+$-tree index for `R` on attribute `A`, and the buffer has 205 free frames available. Consider the query type Q (where `x` and `y` are integer constants):

```
select A,B from R where A ≥ x and A < x + 20
union
select A,B from S where A = y
```

for which we remind the reader that the SQL `union` operator performs the union of two sets (or bags) by eliminating duplicates in the result. Illustrate the algorithm you would use to answer queries of type Q, and tell which is the cost of the algorithm in terms of number of page accesses.

Solution 3

We present the solution assuming the "worst case" scenario for the B$^+$-tree index for `R`: unclustering, dense and using alternative 2. Note, however, that other assumptions are also perfectly acceptable. Let us first count the number of leaves of the tree and its fan-out. Since 20 tuples of `R` fit in one page, we assume that 120 values fit in one page, which means that 120/2 = 60 data entries (and, analogously, 60 index entries) fit in one page. Taking into account the 67% rule, every leaf contains 40 data entries; also, since (30 + 60)/2 = 45, we assume 45 to be the fan-out of the tree. Since the total number of data entries to store in the leaves is 260.000, the number of leaves is 260.000 / 40 = 6.500. How many leaves we should access? Since `R` contains 1.000 distinct values uniformly distributed in the integer attribute `A`, we conclude that for every value of `A` there are 260.000/1.000 = 260 tuples of `R` with that value in `A`, and since we have to consider 20 values in the range appearing in the condition of the query, we have $260 \times 20 = 5.200$ data entries to consider, corresponding to 5.200/40 = 130 relevant leaves to access and, in the worst case, 5.200 pages (one for each data entry) of `R` to access.

The algorithm is as follows. We first use the hash-based index for loading in one frame of the buffer the tuple t (if any) of `S` with `A = y`. We then use the B$^+$-tree index for `R` to get to the first leaf satisfying the condition `A = x` and we scan the 130 relevant leaves of the index for accessing all data entries satisfying the condition `A < x + 20`, and for each such data entry we access the page of `R` with the right tuple and store the pair ⟨`A,B`⟩ of such tuple in the buffer, if it is different from all the pairs appearing in the buffer. Note that in the worst case we have to store 5.200 pairs in the buffer; since 120 values fit in one page, 60 pairs fit in one frame of the buffer, and therefore in the worst case we need $5.200/60 = 87$ frames in the buffer, which is much less than the 205 frames available. At the end, we copy to the output all the tuples stored in buffer.

As for the cost, we count 1 access for the use of the hash-based index, $log_{45}\,6.500 = 3$ accesses for reaching the first leaf, 130 pages for accessing the relevant leaves and finally 5.200 pages for retrieving the tuples of `R`. The cost is therefore 5.334 page accesses.

As we said before, other assumptions for the B^{+}-tree index for `R` are acceptable, in particular, assuming the index clustering and sparse. With this assumption, which implies the arbitrary assumption that `R` is sorted on `A`, the algorithm is similar, but requires a smaller number of page accesses.

Problem 4 Consider the relations `R(A,B,C)` and `S(A,B,C)`, where (*i*) `R` has ⟨`A,B,C`⟩ as the key and contains 100.000 tuples stored in a heap with 1.000 pages; (*ii*) `S` contains 12.000 tuples stored as a heap of 120 pages. Our buffer has 100 free frames available, and we have to compute the difference between `R` and `S` (i.e., `R - S`), with the goal of *maximizing efficiency*, obviously in terms of number of page accesses. Illustrate the algorihm you would choose and tell which is its cost in terms of number of page accesses.

Solution 4

Since we cannot apply the one-pass algorithm, the most obvious algorithm to think of is the two-pass algorithm based on sorting: we produce 2 sorted sublists for `S` and 10 sorted sublists for `R` and then we execute the "merging pass" to compute the bag difference. This algorithm costs $3 \times (1.000 + 120) = 3.360$.

However, it is worth asking whether this is really the most efficient method. Obviously the two-pass algorithm based on hashing would be an option, but it has the same cost. One could think that there is no other method that could be considered. However, notice that the number of pages of `S` is small and therefore it is worth checking if the block-nested loop can be applied and which is its cost. We remind the reader that the block-nested loop cannot be used for computing the difference between two bags. However, in our case, the first operand (`R`) has a key, and therefore, is a set, not a bag. This implies that no tuple of `R` can appear in two different blocks, and therefore the block-nested loop can be used. The cost of the block-nested loop algorithm in this case is $1.000 + \lceil 1.000/98 \rceil \times 120 = 2.320$, which is lower than the one of the two-pass algorithm based on sorting. So, the most efficient algorithm is the block-nested loop.

Problem 5 (only for students who do **not** do the project)
Let B be a relational database with relations `Agency(id,nation)`, `Travel(tcode,agencyid,duration)`, `Visit(tcode,citycode)`, `Participate(tcode,person)`, `Likes(person,citycode)`, where (*i*) each travel agency is located in a nation, (*ii*) each travel is offered by an agency, has a duration, includes a set of at most 10 cities to visit (relation `Visit`) and accepts at most 50 participants (relation `Participate`); (*iii*) each person likes a set of at most 20 cities (relation `Likes`).

5.1 Describe how would you organize a property graph database G in order to represent the relational database B. In particular, (*i*) specify how nodes, edges, labels, etc. of G are used in order to capture the information stored in the tables of B and (*ii*) choose a few tuples for the relations in B, and show the specific property graph database G obtained by applying the chosen representation method.

5.2 Consider the query Q *on the property graph database* that, given the `tcode` of a specific travel T, returns every person who participates to T and likes at least one city visited by T. Choose *one* of the following questions and answer it. (*i*) Assuming the graph database organized as decided for point (5.1), formulate the query on such graph database using a Cypher-like syntax. (*ii*) Assuming that every page of our system contains 1.000 values and there are 10 free buffer frames, illustrate a possible algorithm that the graph database system can use to answer the query and tell which is its cost in terms of number of page accesses.

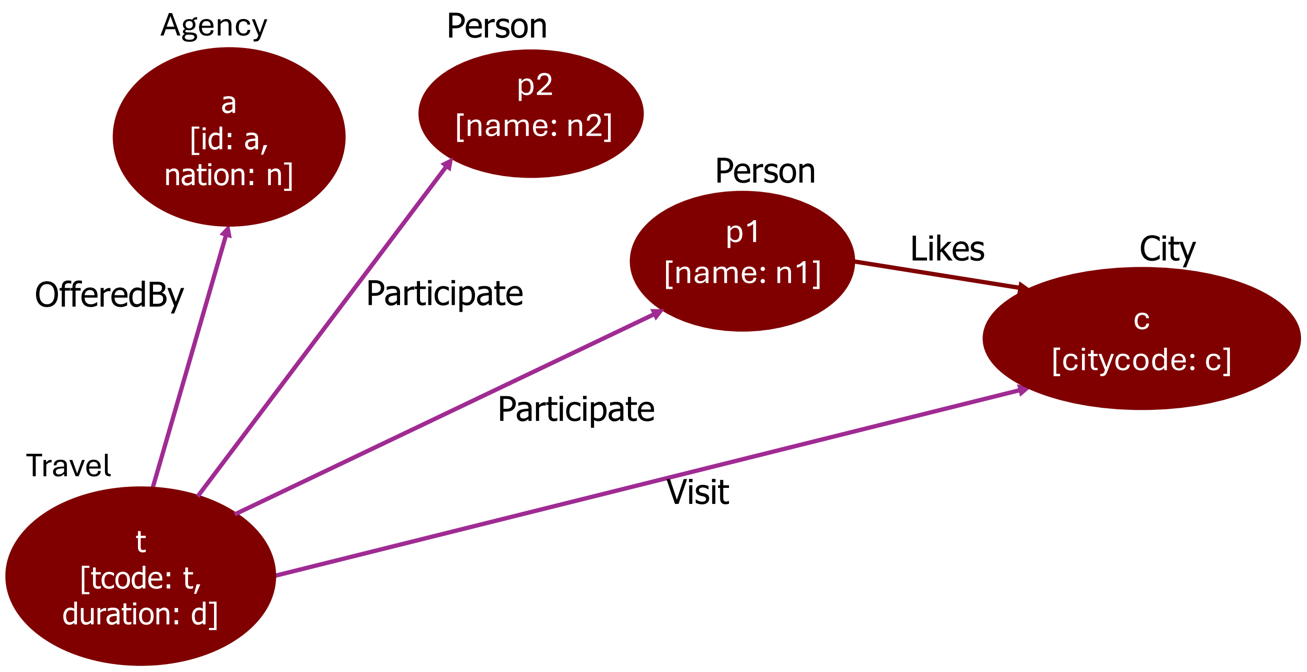
Solution 5

5.1 The following is a possible organization: every tuple of `Agency` is represented by a node labeled `Agency`, with `id` and `nation` as properties. Every tuple of `Travel` is represented by a node labeled `Travel` with properties `tcode` and `duration`. We also have nodes labeled `Person` (with property `name`), and nodes labeled `City` (with property `citycode`). For every tuple of `Travel` with `tcode` t and `agencyid` a, we also have an edge labeled `OfferedBy` from the node corresponding to t to the node corresponding to a. For every tuple of `Visit` with `tcode` t and `citycode` c, we have an edge labeled `Visit` from the node corresponding to t to the node corresponding to c. For every tuple of `Participate` with `tcode` t and `person` p, we have an edge labeled `Participate` from the node corresponding to t to the node corresponding to c. Finally, for every tuple of `Likes` with `person` p and `citycode` c, we have an edge labeled `Participate` from the node corresponding to t to the node corresponding to c.

As an example, the following relational database

```
Agency(a,n)

Travel(t,a,d)

Visit(t,c)

Participate(t,p1)

Participate(t,p2)

Likes(p1,c)
```

is represented by the following property graph database (where we indicate inside the node both the identifier of the node and its properties):



5.2.*i* Assuming that the code of the travel T we are interested in is 123, the query can be expressed by the following Cypher expression:

```
MATCH (x:Travel) -[Participate]-> (y:Person) -[Likes]-> (z:City)
WHERE x.tcode=123 AND (x) -[Visit]-> (z)
RETURN x
```

5.2.*ii* To answer the question we have to decide about the representation method in secondary storage for the graph. We choose to follow the method based of adiacency lists, stored in secondary storage. Since a node in a graph plays the role a tuple in a relational database, we make the usual assumption that to every node n an identifying pair ⟨`pageid,slotaddress`⟩ is associated, specifying the page and the slot address within the page where the data about n reside. In other words, such pair is used as an identifier of the node. The fields of the slot are the identifier of the node n, its properties, and the information about the outgoing edges. In particular, every edge outgoing from n is represented as a pointer to the target node together with the possibile properties of the edge, where the pointer is again an identifying pair ⟨`pageid,slotaddress`⟩. The collection of such pointers (each one together with the properties of the corresponding edge, form the adiacency list associated to n.

A possible algorithm for answering query Q is as follows.

1. Knowing the node n_T corresponding to the travel T, we know the page id of the page P where the node is stored. We access P and we load into one frame F_1 all the node id of n_T and the pointers corresponding to all its outgoing edges). Note that the pointers corresponding to the outgoing edges labeled `Visit` are at most 10 and the pointers corresponding to the outgoing edges labeled `Participate` are at most 50. Therefore, we have to store the id of n_T and 60 pointers, that fit perfectly in F_1.
2. We use a frame F_2 to load, one by one, the pages whose id are stored in F_1 in the pointers corresponding to the edges labeled `Participate` (at most 50). Consider one such page P_s, say corresponding to edge e whose target is person s and note that P_s contains at most 20 pointers to node labeled `City` corresponding to the cities liked by s. We compare all such pointers to the pointers corresponding to the edges labeled `Visit` outgoing fro n_T stored in frame F_1: if at least one of them appears in F_1, we put s in the output, otherwise we ignore it.

Obviously, 10 frames are sufficient for the above algorithm. As for the cost of the algorithm, we count the page accesses as follows: 1 page access for step 1 and 50 page accesses for step 2. The total number of accesses is therefore 60.