Data Management – AA 2015/16 – exam of 08/1/2016

Problem 1
Consider the following schedule $S = w_4(x)\, r_3(z)\, r_2(x)\, w_3(y)\, r_2(z)\, w_2(y)\, r_4(z)\, w_1(y)$, and answer the following questions:

- Tell whether there exists a serial schedule on the same transactions of S that is view equivalent to S but is not conflict equivalent to S.
- Tell whether S is conflict serializable, and, if so, exhibit one serial schedule which is conflict-equivalent to S.
- Tell whether S is in 2PL with shared and exclusive locks, motivating the answer.
- Tell whether S follows the strict 2PL protocol (with shared and exclusive locks), motivating the answer.

Solution 1

- The serial schedule T_4, T_2, T_3, T_1 has the same read-from relation and the same final-write-set as S, whereas it has the conflicting action pair $(w_2(y), w_3(y))$ that appears in different order with respect to S. Therefore, such serial schedule is view equivalent but not conflict equivalent to S.
- The serial schedule T_4, T_3, T_2, T_1 is conflict equivalent to S, and therefore S is conflict serializable.
- S is in 2PL with shared and exclusive locks. To show that, it is sufficient to note that (i) T_4 can anticipate the shared lock on z, needed for executing $r_4(z)$ before unlocking x (unlocking x is necessary to let T_2 read x), and (ii) all other transactions can unlock their elements without violating the 2PL protocol.
- In order to follow the strict 2PL protocol, every transaction must keep the exclusive locks until the commit. However, T_4 must release the lock on x before reading z, and therefore before the commit. Therefore, S does not follow the strict 2PL protocol (with shared and exclusive locks).

Problem 2
Prove or disprove the following claim: if S is a conflict serializable schedule on two transactions, then S is in 2PL with both shared and exclusive locks (or, in other words, S is accepted by the 2PL scheduler with both shared and exclusive locks).

Solution 2
The claim can be disproved by noticing that the following schedule:

$$w_1(x)\, r_2(x)\, r_1(x)$$

is conflict-serializable, but is not in 2PL with both shared and exclusive locks.

Observe that, however, in the above schedule, transaction T_1 writes an element, and then reads the same element. Since such a read is useless, one might ask whether, without such situations, the claim holds. Indeed, we will prove that, if no transaction reads an element that it has already written, the claim holds. We will do that by showing that if S is a schedule on two transaction that is not in 2PL with both shared and exclusive locks, and where no transaction reads an element that it has already written, then it is not conflict serializable.

Suppose that S is not in 2PL with both shared and exclusive locks. This means that

1. there is an action a of one of the two transactions, say transaction T_1, that comes before another action b of the other transaction (i.e., T_2), and such that one of the actions, say b, is a write on an element X and a is a read or a write on the same element X (i.e., S looks like $\ldots, r_1(X), \ldots, w_2(X), \ldots$);
2. there is at least another action a' of T_1 after the action b whose corresponding lock operation (shared lock if a' is a read, or exclusive lock if a' is a write) cannot be executed before the unlock of T_1 of x because of the presence of another action b' fo T_2 before a' (i.e., S looks like $\ldots, r_1(X), \ldots, w_2(X), \ldots, b', \ldots, a'$).

The fact that a' of T_1 cannot be executed before the unlock of T_1 of x because of the presence of b' of T_2 implies that b' is in conflict with a' (i.e., a' and b' operate on the same element, and b' is a write if a' is a read, and is a read or write if a' is a write). It is easy to see that the precedence graph associated to S contains an edge from T_1 to T_2 (because of the actions a and b), and an edge from T_2 to T_1 (because the actions b' and a'), and this implies that S is not conflict serializable.

Problem 3
We have a relation `R`, stored in a heap file, with 810.000 tuples and with the primary key constituted by the attributes `A` and `B`. Suppose that each attribute and each pointer occupies 10 bytes, and the size of each page in our system is 300 bytes. Consider the following query, where `a` and `b` are constants:

```
select * from R where A=a and B=b
```

and tell how many page accesses do we need to answer the query in these two cases: (*i*) if we use a sorted index on `R` with search key <`A`,`B`>; (*ii*) if we use a B$^+$-tree index on `R` with the same search key.

Solution 3
Since `R` is a heap file and the primary key of `R` is the search key of both indexes, both indexes are unclustering, and therefore dense. Also, since each index is unclustering, it cannot use alternative 1, and since the search key is a primary key of the relation, it cannot use alternative 3. We conclude that both indexes use alternative 2. Every data entry occupies 30 bytes, and therefore every page has space for 300/30 = 10 data entries.

(*i*) In the case of sorted index, we need to store 810.000 data entries, and therefore we need 810.000/10 = 81.000 pages, which means that we need $\log_2 81.000 + 1 = 17 + 1 = 18$ page accesses to answer the query.

(*ii*) In the case of B$^+$-tree index, every index entry page has room for 10 index entries, which means that we can assume that the fan out is (10 + 10/2) / 2 = 7. Taking into account the 67% occupation rule, we need 810.000 / 6 = 135.000 leaves to store the data entries, which means that we need $\log_7 135.000 + 1 = 7 + 1 = 8$ page accesses to answer the query.

Problem 4
Let `R` be a relation stored in a heap file with 12.250 pages, consider the operation O of duplicate elimination on `R`, and answer the following questions about executing O in the two cases described below (as usual, the cost must be expressed in terms of number of page accesses).

- *Case* (1)*: the buffer has* M = 13.000 *free frames.* Can we use the block nested loop algorithm? If yes, which is the cost? Can we use the one pass algorithm? If yes, which is the cost?
- *Case* (2)*: the buffer has* M = 115 *free frames.* Can we use the block nested loop algorithm? If yes, which is the cost? Can we use the two pass algorithm? If yes, which is the cost?

Solution 4

- *Case* (1)*: the buffer has* M = 13.000 *free frames.* In this case, the whole relation fits in the buffer, and both algorithms (block nested loop, and one pass) load the relation in the buffer, and then compute the result working in main memory. So, ignoring as usual the cost of writing the output, the cost in both cases is 12.250.
- *Case* (2)*: the buffer has* M = 115 *free frames.* In this case, we can surely use the block nested loop, that uses 115 -2 buffer frames, and whose cost is $12.250 \times (12.250 - 113)/(2 \times 113) = 657.869$. As for the two pass algorithm, we have that $12.250 \leq (115 - 2)^2$, and therefore we can use the two pass algorithm based on sorting, whose cost is $3 \times 12.250 = 36.750$.

Problem 5
Consider the relation `Club(code,city)`, whose key is `code`, with 90.000 tuples, and the relation `Member(clubCode,number,year,country,job)`, whose key is `(clubCode,number)`, with 2.000.000 tuples. We know that the various clubs have approximately the same number of members, that every attribute and every pointer occupies 10 bytes, and that the size of each page is 1.800 bytes. The most important queries on such relations are the following:

Query 1	**Query 2**	**Query 3**

`select *`
`from Club`
`order by code`
	`select clubCode, number`
	`from Member`
	`where clubCode >=1 and clubCode <= 10`
		`select number, clubCode, city`
		`from Club, Member`
		`where code = clubCode`

Answer the following questions:

1. Which is the method you would choose for representing the two relations so as to make the execution of the above queries efficient?
2. Taking into account the chosen method, for each of the above queries, which is the algorithm you would choose for answering the query, and which is the cost of the algorithm in terms of number of page accesses?

Solution 5

1. The relation `Club` can be stored in a file sorted on `code`, so as to support Query 1. Also, we define a B$^+$-tree primary index on search key `(clubCode,number)`, so as Query 2 can access only the index in its range-based search, and Query 3 can be based on computing the join using the index.

2. In order to estimate the cost of the queries, we observe the following:

   - Every tuple of `Club` has size 20 bytes, which means that we have $1.800/20 = 90$ tuples in each page. It follows that `Club` is stored in $90.000/90 = 1.000$ pages.
   - Every data entry occupies 30 bytes, and therefore $1.800/30 = 60$ data entries fit in one page. Taking into account the 67% occupance rule, we conclude that every leaf contain 40 data entries, which means that we have 2.000.000 / 40 = 50.000 pages in the leaves of the index.
   - Eevry index entry occupies 30 bytes, and therefore $1.800/30 = 60$ index entries fit in one page. We can therefore assume that the fan-out of the tree is $(60 + 30)/2 = 45$.
   - Every club has an average of 2.000.000 / 90.000 = 22,22 members.

   We are now ready to evaluate the cost of the three queries.

   - Query 1 costs 1.000 page accesses.
   - In query 2, we need $\log_{45} 50.000 = 3$ page accesses to reach the right leaf, and then we need to access to further $(10 \times 22,22)/40 = 6$ pages. In total, we have 9 page accesses.
   - In query 3, we need to scan the relation `Club`, and for each value of `code`, we need to access the index using such value (this is possible, because the index conforms the condition). On the average, for each value of `code`, we have to consider 22,22 members, and therefore, we can assume that the access to the index requires 3 pages plus one more page, i.e., 4 page accesses. We conclude that Query 3 requires $1.000 + 90.000 \times 4 = 367.000$ page accesses.