Solutions of Test-2 and Test-3 of 2022

Companies: 100 x 50 = 5.000 bytes -> 2 blocks
Routes: 2000 x 50 = 100.000 bytes -> 25 blocks
Flights: 200.000 x 25 = 5.000.000 bytes -> 1221 blocks
Travels: 400.000 x 50 = 20.000.000 bytes -> 4883 blocks
VaccinationCert: 800.000 x 100 = 80.000.000 bytes -> 19532 blocks
Persons: 1.000.000 x 200 = 200.000.000 bytes -> 48829 blocks
travelLegs: 20.000.000 x 50 = 1.000.000.000 bytes -> 244.141 blocks

2nd test

1a) THIS PART OF THE SYLLABUS WAS ALREADY EVALUATE IN TEST 1

1b) THE FORMULAE ARE IN THE BOOK

(use smaller relation as outer relation if both do not fit into memory)
$b_r * b_s + b_r$ transfers and $2 * b_r$ seeks

(use smaller relation as inner relation if it fits in memory)

or $b_r + b_s$ transfers and 2 seeks

1b1) travels has 4883 blocks while travelLegs has 244.141 blocks therefore none fits in memory, consequently it is better to use travels as outer relation and travelLegs as inner relation

1b2) the best ordering is ((routes,companies),travelLegs)
The first join can be performed in memory, resulting in at most 2*25 = 50 blocks that afterwards can be used as inner relation to join efficiently with the travelLegs

Let's justify the answer by estimating the number of tuples according to the formulae given (see slides 16.46 onwards). It is clear that the ordering companies $companies \bowtie travelLegs$ is terrible since it corresponds to a cartesian product returning 100 x 20 x 10^6 = 2 x 10^9 tuples . So, the remaining interesting join orderings are $routes \bowtie companies$ or $routes \bowtie travelLegs$.

Regarding $routes \bowtie companies$, the number of estimated tuples for the join are at most the number of tuples in routes since the only common attribute is idC which is the primary key of companies, therefore each tuple of routes will join at most with one tuple of companies (we do not have information about the existence of foreign keys…). Therefore, the resulting number of tuples is at most 2000.

Regarding $routes \bowtie travelLegs$, a similar analysis would result in the number of estimated tuples being at most 20 x 10^6, since in this case the common attribute is numR the key of Routes.

1c) THIS PART OF THE SYLLABUS WAS ALREADY EVALUATED IN TEST 1

1d) We need to create a foreign key in attribute IdC in Routes referencing Routes(IdC). In this case, the join is superfluous.

```
SELECT Routes.*
FROM Companies NATURAL INNER JOIN Routes
WHERE Origin = 'Lisboa'
```

1e) By the same reason, it is superfluous the join with persons in the auxiliary view flightP1111, so it becomes

with *flightP1111* **as**
(select *NumR, Date* **from** *travelLegs* **natural inner join** *travels* ***where*** *IdP = 1111***)**

The date condition on travel legs should be put outside in the exists in order to reduce the number of tuples being tested in the correlated subquery

select *nameP*
from *travelLegs* **natural inner join** *travels* **natural inner join** *persons*
where *person.Age* > 70 **and** *travelLegs.Date = 10/05/2022* ***and***
exists (select * from *flightP1111* **where** *flightP1111.Date = 10/05/2022* **and** *travelLegs.NumR = flightP1111.NumR)*

We can now substitute the auxiliary view:

select *nameP*
from *travelLegs* **natural inner join** *travels* **natural inner join** *persons*
where *person.Age* > 70 **and** *travelLegs.Date = 10/05/2022* ***and***
exists (select *NumR, Date* **from** *travelLegs tl* **natural inner join** *travels tr*
where *IdP = 1111* **and** *tr.Date = 10/05/2022* **and** *travelLegs.NumR = tl.NumR)*

Now one can use the semi-join operator to evaluate the query (see the slide 16.59 for details).

2a) THIS PART OF THE SYLLABUS WAS ALREADY EVALUATED IN TEST 1

2b) Consider the join of tables with foreign keys that result in one tuple after each join. By doing a “blind” join minimization it might result in a lot of time trying to optimize the join order since the minimization operation is exponential even using dynamic programming.

2c) WE DID NOT COVER PARALLELISM IN THIS YEAR’S EDITION OF THE COURSE

3rd test

1a) In rigorous two-phase locking a transaction must hold all locks till commit/abort.

Step	Transaction T_1:	Transaction T_2:

1	SELECT *
	 FROM Routes
	 WHERE numberR = 1	

2		SELECT *
		 FROM Routes
		 WHERE numberR = 2

3	UPDATE Routes
	 SET locOrigin = 7
	 WHERE numberR = 2;	

4		UPDATE Routes
		 SET locOrigin = 7
		 WHERE numberR = 1;

The first and second steps lock the tuples in shared mode, transaction T1 in step 3 tries to lock in exclusive mode and has to wait for T2 to release the lock, and a similar situation occurs in step 4 originating a deadlock.

1b) In serializable mode there are two possible executions, T1 followed by T2, or T2 followed by T1. In the first execution, after T1 committing the value of age is 6, and therefore the execution finishes with age being 7. In the second execution (T2,T1), the complete execution sets age to the value 12.

There are several alternative executions all providing different values of the age attribute for idP=12345678, besides the ones presented before that correspond to schedule 1,2,3,4,A,B,C, and A,B,C,1,2,3,4, respectively.

First suppose, that none of the transaction see the commit of the other, then age can be either 6 or 2, depending on the one that commits last. E.g., 1,2,3,A,B,C,4 (age :=6), or 1,2,3,A,B,4,C (age := 2).

The remaining possible different value for age can be obtained by the schedule, 1,2,A,B,C,3,4 returning also age := 6.

1c) A typical example would be:

Step	Transaction T_1:	Transaction T_2:

1	SELECT MAX(NumR) INTO n
	 FROM Routes;	

2		SELECT MAX(NumR) INTO n
		 FROM Routes;

3	INSERT INTO Routes
	 VALUES(:n+1, …, …)	

4		INSERT INTO Routes
		 VALUES(n+2, …, …)

Suppose the maximum number present in Routes is 2000, then if execute the above schedule it will be inserted two tuples, one with id=2001 by T1, and another with id =

2002 by T2. However, if we execute the transactions serially, it is not possible to obtain such a result. In fact, T1 followed by T2 inserts two tuples, one with NumR = 2001, and the other with NumR = 2003. If T2 is followed by T1 then, the primary keys of the tuples would be NumR=2002 and NumR=2003. Therefore, no serial execution coincides with the execution in snapshot isolation mode.

1d)

Step	Transaction T_1:	Transaction T_2:

1	SELECT *
	FROM Routes
	WHERE numberR = 1	

2		UPDATE Routes
		SET locOrigin = 2
		WHERE numberR = 2;

3		UPDATE Routes
		SET locOrigin = 2
		WHERE numberR = 1;

4	UPDATE Routes
	SET locOrigin = 1
	WHERE numberR = 2;	

If we use Thomas' Write rule the previous schedule would execute without blocking (the last write is an obsolete write, therefore is ignored), and would correspond to the execution of T1 followed by T2. Of course, the scheduled would enter a deadlock using for instance rigorous two phase locking.

1e) WE DID NOT COVER PARALLEL/DISTRIBUTED QUERIES IN THIS YEAR'S EDITION OF THE COURSE

2a) See the husbands and wives problem in the lab session about transactions and deferrable constraints. In summary, it might be necessary to impose constraints that depend one of the another, and consequently only after all the data is inserted in the table it is possible to verify them.

2b) See slide 19.13. However, the optimization was not explained in the lecture but in a nutshell, it is no longer necessary to keep the old value of an update since the disk blocks are only changed at commit and therefore it is not necessary to undo any work (however, the recovery algorithm must be changed).

2c) See slide 23.6, and the 2PC protocol is to guarantee that a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites (i.e. to guarantee atomicity in distributed transactions).

3rd test 2020/21

I just add the solutions of the two exercises we solved in the lab sessions, namely 1a) and 1b)

1a)

The schedule is not conflict serializable because it is not possible to swap any two operations since they conflict: 1st with 2nd is a read-write conflict, 2nd with the 3rd is a write-write conflict, and 3rd with the 4th is also a write-write conflict. Since the original schedule is not a serial one, therefore the schedule is not conflict serializable.

The schedule is in fact serializable, and it is equivalent to the serial execution T2, T1, T3. The operation T3 is a blind write, and "cleans" the interference between T2 and T1. As a final remark, the schedule is view serializable, as it would be expected because of the blind write used to make it serializable.

1b)

Consider the schedule 1, 2, 3, A, B, C, 4, 5, D, E. Assume the database is empy.
We will overview the effects in the database in the modes read uncommitted, read committed, and snapshot isolation:

	Read Uncommitted	Read Committed	Snapshot Isolation
1			
2	{}	{}	{}
3	Inserts tuple (1,…)	Inserts tuple (1,…)	Inserts tuple (1,…)
A			
B	{(1,…)}	{}	{}
C	Inserts tuple (2,…)	Inserts tuple (2,…)	Inserts tuple (2,…)
4	{(1,…),(2,…)}	{(1,…)}	{(1,…)}
5			
D	{(1,…),(2,…)}	{(1,…),(2,…)}	{(2,…)}
E			

Notice that T1, followed by T2 returns in the select operations 2, 4, B, D the results 2={}, 4={{(1,…)}, B={{(1,…)}, D={(1,…),(2,…)}, respectively.

In execution T2 followed by T1, we get a symmetrical situation with B, D, 2, 4 returning respectively B={}, D={{(2,…)}, 2={{(2,…)}, 4={(1,…),(2,…)}

It is immediate to see that none of the above the executions correspond to a serial one.

Extra question about recovery system

Consider the following log record of a database management system whose normal operation was interrupted by a failure. Indicate the final state of the log and the database after crash recovery (assume that the only records in the database are A, B and C).

<T1 **start**>
<T2 **start**>
<T1,A,1,2>
<**checkpoint** {T1,T2}>
<T2, B, 1, 3>
<T3 **start**>
<T1,A,1>
<T3,C,1,4>
<T3 **commit**>
<T1 **abort**>
Crash ...

The log will contain additionally the records:

<T2,B,1>
<T2 abort>

After the recovery the values A, B and C will be 1, 1 and 4 respectively.

Log	Redo	Undo
<T1 **start**>		
<T2 **start**>		Writes <T2 abort> in the log undolist = {}
<T1,A,1,2>		
<**checkpoint** {T1,T2}>	Redo phase starts here undolist = {T1,T2}	
<T2, B, 1, 3>	B := 3	B := 1 Writes <T2,B,1> in the log
<T3 **start**>	undolist = {T1,T2,T3}	
<T1,A,1>	A := 1	
<T3,C,1,4>	C := 4	
<T3 **commit**>	undolist = {T1,T2}	
<T1 **abort**>	undolist = {T2}	

The 1st phase is the redo, starting from the last checkpoint till the end of the low, while the undo phase scans the log backwards till the undolist is empty.