Data Bases II
Exercises

Christian Rossi

Academic Year 2023-2024

Abstract

The course aims to prepare software designers on the effective development of database applications.

First, the course presents the fundamental features of current database architectures, with a specific emphasis on the concept of transaction and its realization in centralized and distributed systems.

Then, the course illustrates the main directions in the evolution of database systems, presenting approaches that go beyond the relational model, like active databases, object systems and XML data management solutions.

Contents

CHAPTER 1

Exercise session I

1.1 Anomalies classification

Can the following schedules produce anomalies? c_i and a_i indicate the transactional decision commit and abort.

1. $r_1(x)w_1(x)r_2(x)w_2(y)\ a_1\ c_2$
2. $r_1(x)w_1(x)r_2(y)w_2(y)\ a_1\ c_2$
3. $r_1(x)r_2(x)r_2(y)w_2(y)r_1(z)\ a_1\ c_2$
4. $r_1(x)r_2(x)w_2(x)w_1(x)\ c_1\ c_2$
5. $r_1(x)r_2(x)w_2(x)r_1(y)\ c_1\ c_2$
6. $r_1(x)w_1(x)r_2(x)w_2(x)\ c_1\ c_2$

Solution

1. We have a serial execution, but with the abort of the first transaction. Since the second transaction reads the modified value of x before the abort, we have a dirty read.
2. We have a serial execution and the two transactions require different resources, so there are no anomalies.
3. There are no anomalies because the last operation of the first transaction works on a different resource.
4. Both transactions first reads in sequence the resource x and then updates it without considering the updated value, so we have a lost update.
5. There are no anomalies because the last operation of the first transaction works on a different resource.
6. We have a serial execution, so the schedule is correct.

1.2 Anomalies classification

The following schedule may produce 2 anomalies: a lost update and a phantom update. Identify them.

$$r_1(x)r_2(x)r_3(x)w_1(x)r_4(y)w_2(x)r_4(x)w_4(y)r_3(y)w_4(x)r_5(y)w_6(y)w_5(y)w_7(y)$$

Solution We can write the schedule in the following way:

$r_1(x)$			$w_1(x)$										
	$r_2(x)$				$w_2(x)$								
		$r_3(x)$						$r_3(y)$					
				$r_4(y)$		$r_4(x)$	$w_4(y)$		$w_4(x)$				
										$r_5(y)$		$w_5(y)$	
											$w_6(y)$		
													$w_7(y)$

And, considering both definitions, we can see that there is a lost update with transactions T_1 and T_2 and a phantom update with T_3 and T_4.

1.3 Schedule classification

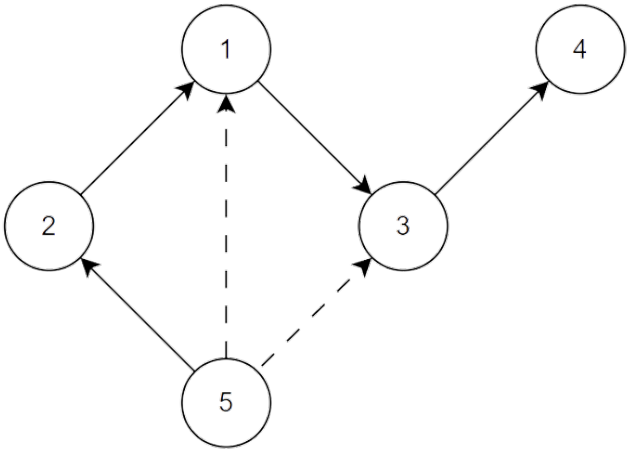
Classify the following schedule with respect to CSR and VSR classes:

$$r_1(x)r_2(y)w_3(y)r_5(x)w_5(u)w_3(s)w_2(u)w_3(x)w_1(u)r_4(y)w_5(z)r_5(z)$$

Solution Since CSR contains VSR we check with the conflict graph. To do so we first divide the schedule based on the resources:

- $x : r_1\ r_5\ w_3$
- $y : r_2\ w_3\ r_4$
- $z : w_5\ r_5$
- $s : w_3$
- $u : w_5\ w_2\ w_1$

The nodes are $\{1, 2, 3, 4, 5\}$ and the arcs are found with the write-write or write-read relations found in the previous groups. So we have the following graph:



Some arcs can be omitted if the nodes are connected in another way (in this case we can remove arcs $\{\{5,1\},\{5,3\}\}$).

There are no cycles, so the schedule is CSR (and also VSR).

1.4 Schedule classification

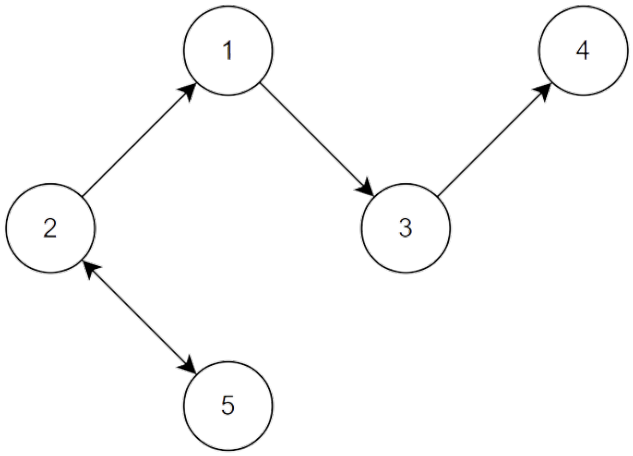
Classify the following schedule with respect to CSR and VSR classes:

$$r_2(u)w_2(s)r_1(x)r_2(y)w_3(y)r_5(x)w_5(u)w_3(s)w_2(u)w_3(x)w_1(u)r_4(y)w_5(z)r_5(z)$$

Solution Since CSR contains VSR we check with the conflict graph. To do so we first divide the schedule based on the resources:

- $x : r_1\ r_5\ w_3$
- $y : r_2\ w_3\ r_4$
- $z : w_5\ r_5$
- $s : w_2\ w_3$
- $u : r_2\ w_5\ w_2\ w_1$

The nodes are $\{1,2,3,4,5\}$ and the arcs are found with the write-write or write-read relations found in the previous groups. So we have the following graph:



It is possible to see that there is a cycle between two and five. The definition of VSR states that we need to have the same reads-from relations and final writes. So, we try to find a view-equivalent schedule that is also CSR. One possible solution is simply to swap the two writes on the resource u and that is sufficient to eliminate the cycle. So, the schedule:

$$r_2(u)w_2(s)r_1(x)r_2(y)w_3(y)r_5(x)w_5(u)w_2(u)w_3(s)w_3(x)w_1(u)r_4(y)w_5(z)r_5(z)$$

is CSR and also VSR.

1.5 Schedule classification

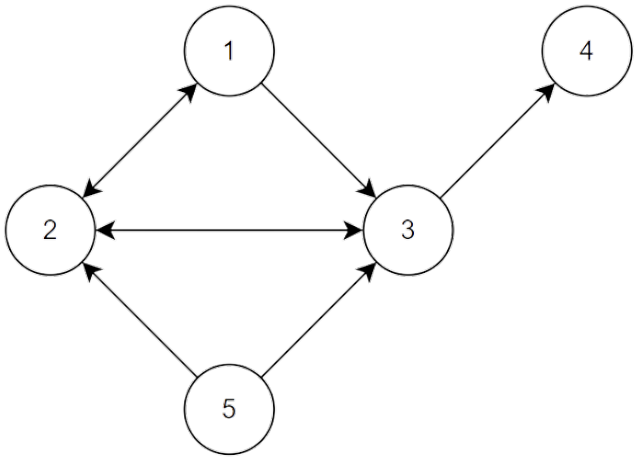
Classify the following schedule with respect to CSR and VSR classes:

$$r_1(x)r_2(y)w_3(y)r_5(x)w_5(u)w_3(s)w_2(u)w_3(x)w_1(u)r_4(y)w_5(z)r_5(z)r_2(u)w_2(s)$$

Solution Since CSR contains VSR we check with the conflict graph. To do so we first divide the schedule based on the resources:

- $x : r_1\ r_5\ w_3$
- $y : r_2\ w_3\ r_4$
- $z : w_5\ r_5$
- $s : w_3\ w_2$
- $u : w_5\ w_2\ w_1\ r_2$

The nodes are $\{1, 2, 3, 4, 5\}$ and the arcs are found with the write-write or write-read relations found in the previous groups. So we have the following graph:



In this case it is not possible to find a VSR schedule because it is impossible to do so without changing the final write on s.

1.6 Schedule classification

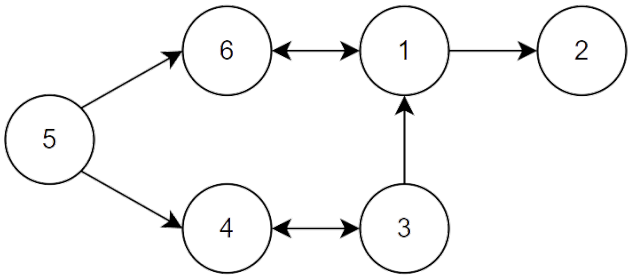
Classify the following schedule with respect to CSR and VSR classes:

$$r_5(x)r_3(y)w_3(y)r_6(t)r_5(t)w_5(z)w_4(x)r_3(z)w_1(y)r_6(y)w_6(t)w_4(z)w_1(t)w_3(x)w_1(x)r_1(z)w_2(t)w_2(z)$$

Solution Since CSR contains VSR we check with the conflict graph. To do so we first divide the schedule based on the resources:

- $t:\ r_6\ r_5\ w_6\ w_1\ w_2$
- $x:\ r_5\ w_4\ w_3\ w_1$
- $y:\ r_3\ w_3\ w_1\ r_6$
- $z:\ w_5\ r_3\ w_4\ r_1\ w_2$

The nodes are $\{1, 2, 3, 4, 5, 6\}$ and the arcs are found with the write-write or write-read relations found in the previous groups. So we have the following graph:



We have two cycles. It is impossible to find a VSR schedule because only the conflict between four and three can be eliminated (the other one changes a read-write relation).

CHAPTER 2

Exercise session II

2.1 Schedule classification

Classify the following schedule with respect to 2PL and strict 2PL classes:

$$r_1(x)r_2(y)w_3(y)r_5(x)w_5(u)w_3(s)w_2(u)w_3(x)w_1(u)r_4(y)w_5(z)r_5(z)$$

Solution For strict 2PL we assume that all transactions commit and release all locks immediately after their last operation, and check if releases can be executed at commit time.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
S						w_3						
U					w_5		$\searrow_5$ w_2 $\downarrow_2$		w_1 $\downarrow_1$			
X	r_1			r_5				$\searrow_1$ w_3				
Y		r_2 $\searrow_2$	w_3							r_4		
Z											w_5	r_5 $\downarrow_5$

S clearly cannot be in strict 2PL. The contradictions are:

- T_1 must release X before 8.
- T_2 must release Y before 7.
- T_5 must release U before 12.

For 2PL we have:

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
S						$\nearrow_3$ w_3		$\searrow_3$				
U					$\nearrow_5$ w_5 $\searrow_5$		$\nearrow_2$ w_2 $\searrow_2\nearrow_1$		w_1 $\searrow_1$			
X	$\nearrow_1$ r_1			$\nearrow_5$ r_5		$\searrow_5$		$\searrow_1\nearrow_3$ w_3 $\searrow_3$				
Y		$\nearrow_2$ r_2 $\searrow_2$	$\nearrow_3$ w_3						$\searrow_3$	$\nearrow_4$ r_4 $\searrow_4$		
Z				$\nearrow_5$							w_5	r_5 $\searrow_5$

It is also not in 2PL: an assignment is not possible for T_2 (which must release Y before locking U).

2.2 Schedule classification

Classify the following schedule with respect to 2PL and strict 2PL classes:

$$r_4(x)r_2(x)w_4(x)w_2(y)w_4(y)r_3(y)w_3(x)w_4(z)r_3(z)r_6(z)r_8(z)w_6(z)w_9(z)r_5(z)r_10(z)$$

Solution For strict 2PL we assume that all transactions commit and release all locks immediately after their last operation, and check if releases can be executed at commit time.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
X	r_4	r_2 $\searrow_2$	w_4				w_3								
Y				w_2 $\downarrow_2$	w_4	r_3									
Z								w_4	r_3	r_6	r_8	w_6	w_9	r_5	r_{10}

It is therefore clear that the schedule cannot be in 2PL-strict, due to T_2 and T_4: T_2 ends after 4, but T_4 wants to write X at 3, and T_2 would thus be required to release X earlier, which is impossible if T_2 has to keep all locks until after 4.

For 2PL we have:

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
X	$\nearrow_4$ r_4	$\nearrow_2$ r_2 $\searrow_2$	$\nearrow_4$ w_4		$\searrow_4$		$\nearrow_3$ w_3			$\searrow_3$					
Y		$\nearrow_2$		w_2 $\searrow_2$	$\nearrow_4$ w_4 $\searrow_4$	$\nearrow_3$ r_3				$\searrow_3$					
Z			$\nearrow_4$					w_4 $\searrow_4$	$\nearrow_3$ r_3	r_6	r_8 $\searrow_3$	w_6	w_9	r_5	r_{10}

We need to look at those acquisitions that must be anticipated and to those releases that must be delayed to not violate the 2PL rules. T_4 can only get the XL on X only after 2 and on Y after 4 and has to release Y before 6 and X before 7. Thus, the lock on Z must be acquired before 6. T_2 can get all the locks at the beginning and release them immediately after each use. T_3 can acquire X, Y and Z just before using them and release them all before 12. All other transactions (T_6, T_9, T_5, T_{10}) clearly pose no problems.

2.3 Schedule classification

Classify the following schedule with respect to 2PL and strict 2PL classes:

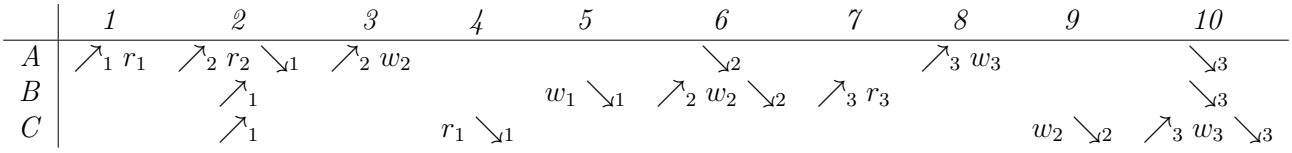
$$r_1(A)r_2(A)w_2(A)r_1(B)w_1(C)w_2(C)r_3(C)w_3(A)w_2(B)w_3(B)$$

Solution For strict 2PL we assume that all transactions commit and release all locks immediately after their last operation, and check if releases can be executed at commit time.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
A	r_1	r_2 $\searrow_1$	w_2					w_3		
B					w_1 $\downarrow_1$	w_2	r_3			
C				r_1					w_2	w_3

The schedule is not strict 2PL.

For 2PL we have:



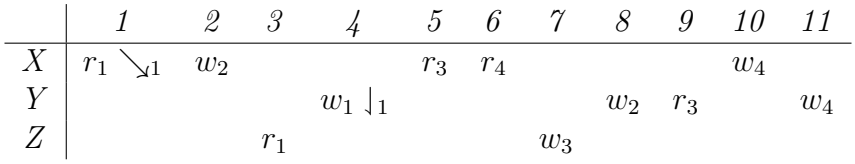
The schedule is 2PL.

2.4 Schedule classification

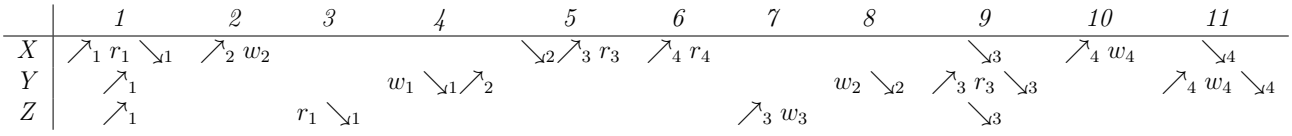
Classify the following schedule with respect to 2PL and strict 2PL classes:

$$r_1(x)w_2(x)r_1(z)w_1(y)r_3(x)r_4(x)w_3(z)w_2(y)r_3(y)w_4(x)w_4(y)$$

Solution For strict 2PL we assume that all transactions commit and release all locks immediately after their last operation, and check if releases can be executed at commit time.



The schedule is not strict 2PL.
For 2PL we have:



The schedule is 2PL.

2.5 Update locks

Given the schedule:

$$r1(x)r2(x)r3(y)w3(y)w1(x)w2(y)$$

show the sequence of lock and unlock requests produced by the transactions in a 2PL execution, in a system with update lock (available locks: SL, UL, XL).

Solution The locking phases with update locks are the following:

X	Y
$\mathrm{UL}_1(x)$	
$r_1(x)$	
$\mathrm{SL}_2(x)$	
$r_2(x)$	
	$\mathrm{UL}_3(y)$
	$r_3(y)$
	$\mathrm{XL}_3(y)$[upgrade]
	$w_3(y)$
	$\mathrm{rel}(\mathrm{XL}_3(y))$
	$\mathrm{XL}_2(y)$
$\mathrm{rel}(\mathrm{SL}_2(x))$	
$\mathrm{XL}_1(x)$[upgrade]	
$w_1(x)$	
$\mathrm{rel}(\mathrm{XL}_1(x))$	
	$w_2(y)$
	$\mathrm{rel}(\mathrm{XL}_2(y))$

2.6 Update locks

Update lock was introduced to contrast deadlocks. Can we state that deadlocks are impossible in the presence of update locks?

1. If so, concisely explain why.
2. If not, provide a counter-example.

Solution

1. Clearly deadlocks are possible in the presence of UL. Indeed, UL only makes deadlock less likely, by preventing one type of deadlock, due to update patterns, when two transactions compete for the same resource ($r_1(x)r_2(x)w_1(x)w_2(x)$).
2. Consider two distinct resources X and Y, and two transactions that want to access them in this order: $r_1(X)r_2(Y)w_1(Y)w_2(X)$. It is likely that they end up in deadlock, especially if the system on which they run applies 2PL. UL is totally irrelevant here, because there is no update pattern.

CHAPTER 3

Exercise session III

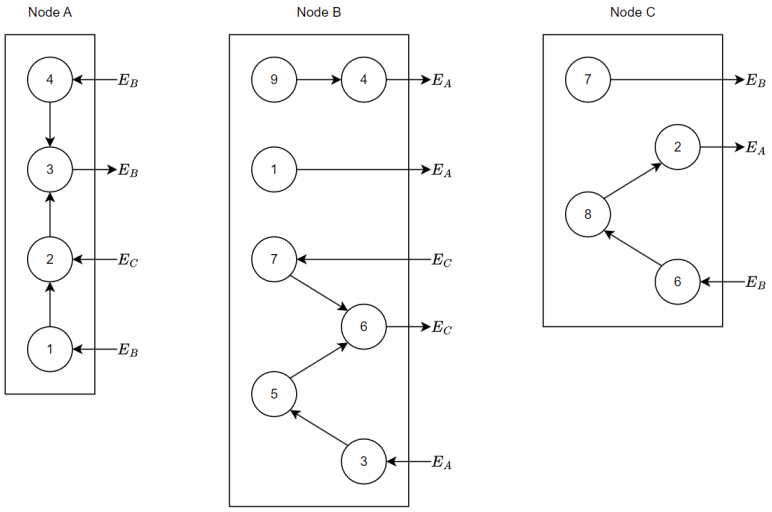
3.1 Obermarck's algorithm

Consider the following waiting conditions:

- Node A: $E_B \to t_1, t_1 \to t_2, E_C \to t_2, t_2 \to t_3, t_3 \to E_B, E_B \to t_4, t_4 \to t_3$
- Node B: $E_A \to t_3, t_3 \to t_5, t_5 \to t_6, t_6 \to E_C, E_C \to t_7, t_7 \to t_6, t_9 \to t_4, t_4 \to E_A, t_1 \to E_A$
- Node C: $E_B \to t_6, t_6 \to t_8, t_8 \to t_2, t_2 \to E_A, t_7 \to E_B$

Simulate the Obermarck algorithm and indicate whether there is a distributed deadlock.

Solution We need to construct the graph with the given constraints, that is:

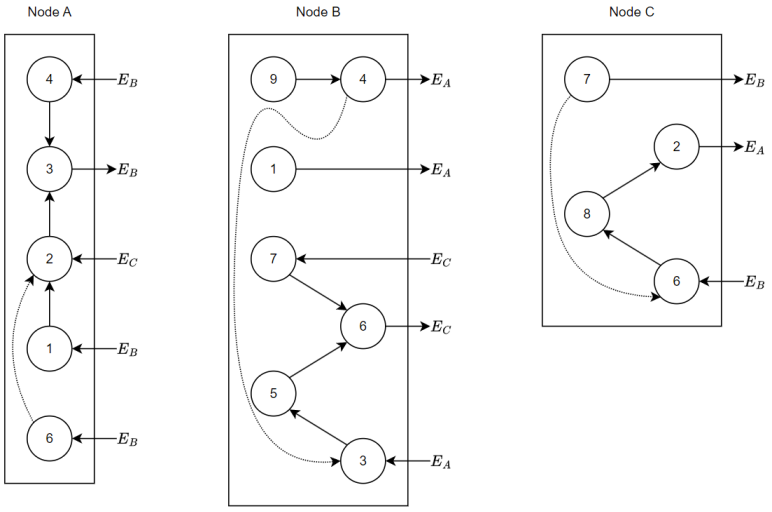


We have to check all the nodes where the sender has a lower value than the receiver. In this case we have that the update is sent to the other distributed node. The interesting cases are highlighted in the image. So, we now have to add the nodes:

- $4 \to 3$ in E_B.
- $7 \to 6$ in E_C.

- $6 \to 2$ in E_A.

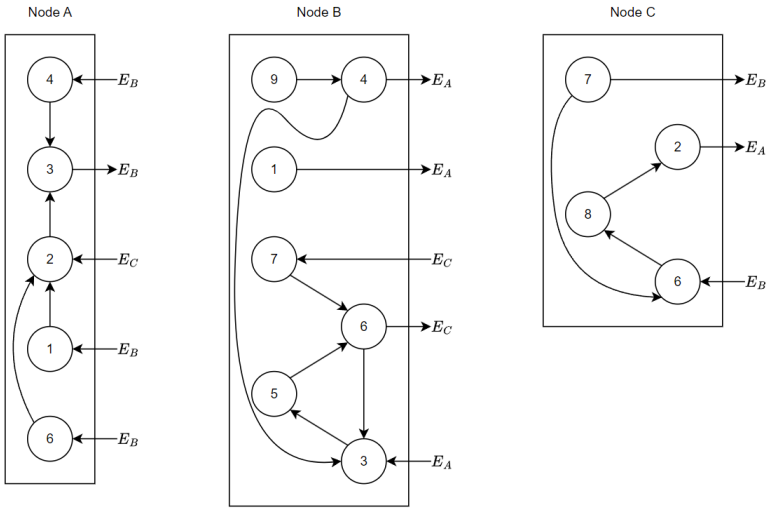
If the numbered node is not present we can add it to the graph of the distributed node. We obtain the following graphs:



We have to check if other messages are sent. We have:

- $6 \to 3$ in E_B.

So the updated graph is:



We have found a cycle, so there is a deadlock.

3.2 Obermarck's algorithm

The nodes A, B, and C of a distributed transactional system are aware of the following remote and local waiting conditions:

- Node $A : E_B \to t_3, E_C \to t_2, t_1 \to E_C, t_3 \to t_5, t_5 \to t_1$
- Node $B : E_C \to t_2, t_3 \to E_A, t_2 \to t_3$

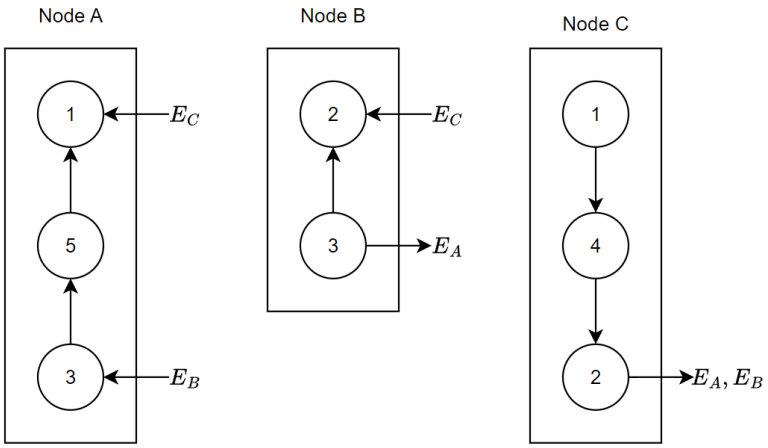
- Node $C : t_2 \to E_A, t_2 \to E_B, t_1 \to t_4, t_4 \to t_2$

Execute the Obermarck's algorithm twice, with different conventions:

1. Sending messages of the form $E_X \to t_i \to t_j \to E_Y$ forward (toward node Y) and only if and only if $i > j$.
2. With the opposite conventions, so if and only if $i > j$

Discuss the outcome, and explain it, taking into account the properties of the algorithm and the initial conditions.

Solution The graph is the following:



1. We have to add the nodes (connections are distributed):
   - $3 \to 1$ in E_C.
   - $3 \to 2$ in E_A.
   - $3 \to 2$ in E_B.

   By adding those nodes we found that the third one creates a deadlock.
2. We have to add the node (connections are distributed):
   - $2 \to 3$ in E_A.

   No cycles are found, so no deadlocks found.

The algorithm is independent of the conventions but the initial conventions, but the initial conditions must be consistent and complete. On a faulty dataset even the best algorithm returns untrustworthy results. In this case we have a link missing between node A and C.

3.3 Schedule classification

Classify the following schedule with respect to timestamps:

$$r_4(x)r_2(x)w_4(x)w_2(y)w_4(y)r_3(y)w_3(x)w_4(z)r_3(z)r_6(z)r_8(z)w_6(z)w_9(z)r_5(z)r_{10}(z)$$

Solution We can identify pairs of operations that cause killings:

- $X : r_4 r_2 w_4 w_3$
- $Y : w_2 w_4 r_3$
- $Z : w_4 r_3 r_6 r_8 w_6 w_9 r_5 r_{10}$

On X we have that w_3 is too late with respect to r_4 and w_4. On Y we have that r_3 is late with respect to w_4. On Z we have that r_3 is late with respect to w_4, w_6 with respect to r_8, r_5 with respect to both w_6 and w_9. So, the schedule is not in TS-mono.

The schedule is also outside TS-multi, because $w_3(X)$ comes too late ($r_4(X)$ was already given the initial version instead) and also because $w_6(Z)$ is late with respect to $r_8(Z)$. The other five reasons were due to reads (that are always accepted in TS-multi).

3.4 Schedule classification

Classify the following schedule with respect to timestamps:

$$r_1(x) r_2(y) w_3(y) r_5(x) w_5(u) w_3(s) w_2(u) w_3(x) w_1(u) r_4(y) w_5(z) r_5(z)$$

Solution We can identify pairs of operations that cause killings:

- $S : w_3$
- $U : w_5 w_2 w_1$
- $X : r_1 r_5 w_3$
- $Y : r_2 w_3 r_4$
- $Z : w_5 r_5$

S, Y and Z are ok, U is ok only if the Thomas rule is applied, and X is not ok for both TS-mono and TS-multi.

3.5 Schedule classification

Classify the following schedule:

$$r_1(X) w_1(Y) w_2(Y) w_3(Z) r_1(Z) w_4(X) r_4(Y) w_3(X) r_5(Y) w_5(X)$$

Solution First, we check if it is CSR:

- $X : r_1 w_4 w_3 w_5$
- $Y : w_1 w_2 r_4 r_5$
- $Z : w_3 r_1$

We found a cycle between the nodes three and one, so it is not CSR. It is also not VSR. We now check for TS using the same list: we find that $w_4 w_3$ are not in the correct order, so it is not in TS-mono, neither in TS-multi (it is a write that causes the problem).

3.6 Schedule classification

Classify the following schedule:

$$r_1(x)r_2(y)w_3(x)r_5(z)w_6(z)w_2(x)w_3(y)r_7(z)w_4(x)$$

Solution First, we check if it is CSR:

- $X : r_1 w_3 w_2 w_4$
- $Y : r_2 w_3$
- $Z : r_5 w_6 r_7$

There is a cycle between two and three, so it is not CSR, but by swapping $w_3 w_2$ we can obtain a VSR schedule without changing the schedule.

The schedule is not TS-mono because we have $w_3 w_2$ and so also non TS-multi.

3.7 Schedule classification

Given the resources above and the following transactions:

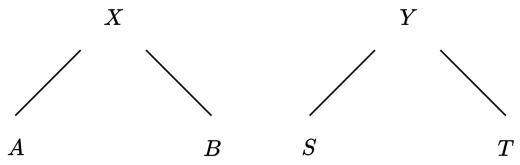
$$T_1 : r_1(C)w_1(B)w_1(C)$$

$$T_2 : w_2(A)r_2(C)$$

Consider that T_1 and T_2 can only be scheduled in 10 different ways (two serial, eight interleaved). What can be stated about the 2PL-strict compatibility of these schedules?

Solution We note that the only potential conflict is between $w_1(C)$ and $r_2(C)$. But these are also the last operations of their respective transactions: to check for compatibility, we can always assume that the commit occurs right after the last operation, and that all locks are released in favor of the other one. But this argument is independent of the order of the operations, and is therefore valid for all the eight interleaved schedules. So, all the schedules are strict 2PL.

3.8 Hierarchical lock



Given the resource hierarchy above, is the following schedule compatible with a 2PL-strict scheduler that applies hierarchical locking?

$$r_1(A)w_1(S)w_2(T)r_2(A)w_1(A)$$

Solution The lock manager works as follows:

	X	A	B	Y	S	T	
$r_1(A)$	ISL_1	-	-	-	-	-	
	ISL_1	SL_1	-	-	-	-	
$w_1(S)$	ISL_1	SL_1	-	IXL_1	-	-	
	ISL_1	SL_1	-	IXL_1	XL_1	-	
$w_2(T)$	ISL_1	SL_1	-	$\text{IXL}_{1,2}$	XL_1	-	
	ISL_1	SL_1	-	$\text{IXL}_{1,2}$	XL_1	XL_2	
$r_2(A)$	$\text{ISL}_{1,2}$	SL_1	-	$\text{IXL}_{1,2}$	XL_1	XL_2	
	$\text{ISL}_{1,2}$	$\text{SL}_{1,2}$	-	$\text{IXL}_{1,2}$	XL_1	XL_2	
commit T_2							End of T_2
	$\mathbf{ISL}_1$	SL_1	-	IXL_1	XL_1	-	
$w_1(A)$	$\mathbf{IXL}_1$	$\mathbf{SL}_1$	-	IXL_1	XL_1	-	
	IXL_1	$\mathbf{XL}_1$	-	IXL_1	XL_1	-	
commit T_1							End of T_1

So, it is compatible with a strict 2PL scheduler.

3.9 Hierarchical lock

Consider the following short schedule occurring on a system with hierarchical lock over a hierarchy where $PagA$ contains tuples t_1 and t_2:

$$r_1(PagA), w_2(t1), w_1(t2)$$

Show a possible sequence of locks, unlocks, lock upgrades, and lock downgrades performed by transactions T_1 and T_2 such that the schedule is in 2PL.

Solution We have that:

	PagA	**t2**	**t1**
$\text{SIXL}_1(PagA)$	SIXL_1	-	-
$\text{XL}_1(t2)$	SIXL_1	XL_1	-
$r_1(PagA)$			
$\text{U-SL}_1(PagA)$	$\text{IXL}_1, \text{IXL}_2$	XL_1	-
$\text{IXL}_2(PagA)$	$\text{IXL}_1, \text{IXL}_2$	XL_1	-
$\text{XL}_2(t1)$	$\text{IXL}_1, \text{IXL}_2$	XL_1	XL_2
$w_2(t1)$			
$\text{U-XL}_2(t1)$	$\text{IXL}_1, \text{IXL}_2$	XL_1	-
$\text{U-IXL}_2(PagA)$	IXL_1	XL_1	-
Commit T_2			
$w_1(t2)$			
$\text{U-XL}_1(t2)$	IXL_1	-	-
$\text{U-IXL}_1(PagA)$	-	-	-
Commit T_1			

So the schedule is 2PL.

CHAPTER 4

Exercise session IV

4.1 Ranking and skyline queries

Consider a distributed setting with three data sources, ranking basketball players according to their offensive rating (off), defensive rating (def) and rebounds (reb). An associated score in $[0, 1]$ is indicated (the higher, the better). Sorted access is available.

R_0 (off)	Player	R_1 (def)	Player	R_2 (reb)	Player
0.8	9	0.9	4	0.6	1
0.8	8	0.8	5	0.5	3
0.8	2	0.7	7	0.3	4
0.7	7	0.6	9	0.3	2
0.6	3	0.5	0	0.3	0
0.6	1	0.4	6	0.2	8
0.5	6	0.2	3	0.2	6
0.5	5	0.2	2	0.1	5
0.5	0	0.0	8	0.0	9
0.2	4	0.0	1	0.0	7

1. Determine the top-3 players according to their median rank using MedRank.

2. Remove the first data source and consider the scoring function

$$\text{MAX}(o) = \max\{\text{def}(o), \text{reb}(o)\}$$

   Determine the top-2 players according to MAX using the algorithms B_0 and NRA.

3. Assume now that random access is also available. Determine the top-2 players according to MAX with the algorithms TA and FA.

4. Consider now the scoring function

$$\text{SUM}(o) = \text{def}(o) + \text{reb}(o)$$

   equally weighing all partial scores. What are the top-2 players according to SUM?

5. Assume now that all the data regarding the players are centralized in a single data source, in which the players are available sortedly according to SUM. Use SFS to determine the skyline of the players.

6. Identify the players in the 2-skyband, and in the 3-skyband.

Solution

1. A player is considered when, while doing sorted access it appears in at least two tables. If we check the first row we find the following players: $\{9, 4, 1\}$ and the tables are three, so no player appears in at least two tables. For the second iteration we have found $\{9, 4, 1, 8, 5, 3\}$, so again no duplicate player. With the third row we have $\{9, 4, 1, 8, 5, 3, 2, 7, 4\}$. We have found the first player that appears in at least two rankings, but we need two more players, so we can check the fourth row: $\{9, 4, 1, 8, 5, 3, 2, 7, 4, 7, 9, 2\}$. With this iteration we have found other three players, and the algorithm stops. The content of the buffer at this step is the following:

Player	First row	Second row	Third row
4	1	3	?
2	3	4	?
7	3	4	?
9	1	4	?
1	1	?	?
8	2	?	?
5	2	?	?
3	2	?	?

With this algorithm we have found that the best players are: nine, four, two, and seven. In particular, the player four is the best due to the median rank and the other three are equally good.

2. Since we have $k = 2$, for the B_0 algorithm we need to make two sorted access to the first two rows, and add the objects to the buffer:

Player	R_1 (def)	R_2 (reb)	MAX
4	0.9	?	0.9
5	0.8	?	0.8
1	?	0.6	0.6
3	?	0.6	0.6

For each object in the buffer we have to make some random accesses to find the missing values. The buffer will be updated like follows.

Player	R_1 (def)	R_2 (reb)	MAX
4	0.9	0.3	0.9
5	0.8	0.1	0.8
1	0.0	0.6	0.6
3	0.2	0.6	0.6

The best two players in the buffer are four and five.

For the NRA algorithm we have to make a sorted access to each row until the upper bound of the first element out of the top-k is less or equal than the lower bound of the worst top-k. With the sorted access to the first row we have the following buffer.

Player	$\boldsymbol{R_1}$ **(def)**	$\boldsymbol{R_2}$ **(reb)**	**Lower bound**	**Upper bound**
4	0.9	?	0.9	0.9
1	?	0.6	0.6	0.9

And the threshold point has the following coordinates $(0.9, 0.6)$, and since the scoring function is MAX we have that its score is 0.9. After the second sorted access we have:

Player	$\boldsymbol{R_1}$ **(def)**	$\boldsymbol{R_2}$ **(reb)**	**Lower bound**	**Upper bound**
4	0.9	?	0.9	0.9
5	0.9	?	0.8	0.8
1	?	0.6	0.6	0.8
3	?	0.6	0.5	0.8

And the threshold point has the following coordinates $(0.8, 0.5)$, and since the scoring function is MAX we have that its score is 0.8. Since the lower bound of three is equal to the upper bound of one the algorithm stops.

3. With FA we have to make sorted accesses until we find k elements that appears in each column. In this case (excluding R_0) we need to make five sorted accesses to find two elements in both tables (four and zero). The elements added to the buffer are:

Player	$\boldsymbol{R_1}$ **(def)**	$\boldsymbol{R_2}$ **(reb)**	**Score**
4	0.9	0.3	0.9
5	0.8	?	0.8
7	0.7	?	0.7
1	?	0.6	0.6
9	0.6	?	0.6
3	?	0.5	0.5
0	0.5	0.3	0.5
2	?	0.3	0.3

We can now complete the buffer with random accesses to the score, and we obtain the final buffer.

Player	$\boldsymbol{R_1}$ **(def)**	$\boldsymbol{R_2}$ **(reb)**	**Score**
4	0.9	0.3	0.9
5	0.8	0.1	0.8
7	0.7	0.0	0.7
1	0.0	0.6	0.6
9	0.6	0.0	0.6
3	0.2	0.5	0.5
0	0.5	0.3	0.5
2	0.2	0.3	0.3

The best players are: four and five.

For the TA we have to make sorted access row by row and complete the value with sorted accesses in the other column. We have also to compute the threshold as the scoring function return value of the considered row. For the first row we have the following buffer.

Player	R_1 (def)	R_2 (reb)	Score
4	0.9	0.3	0.9
1	0.0	0.6	0.6

The threshold in this case is the maximum of the values of the first row, that is 0.9. So, we need to do another iteration, that creates the following buffer.

Player	R_1 (def)	R_2 (reb)	Score
4	0.9	0.3	0.9
5	0.8	0.1	0.8

The threshold in this case is the maximum of the values of the second row, that is 0.8. Since it is less or equal than the worst object's score in the buffer, the algorithm halts. We have found that also with TA the best players are four and five.

4. We decide to use TA algorithm with the given scoring function and $k = 2$. The steps are the same as the previous point, except for the scoring function. After accessing the first row we have the following buffer:

Player	R_1 (def)	R_2 (reb)	Score
4	0.9	0.3	1.2
1	0.0	0.6	0.6

The threshold in this case is the sum of the values of the first row, that is 1.5. Since it is greater than 0.6 we have to do another iteration. If we read the second row we obtain the following buffer.

Player	R_1 (def)	R_2 (reb)	Score
4	0.9	0.3	1.2
5	0.8	0.1	0.9

The threshold in this case is the sum of the values of the first row, that is 1.3. Since it is greater than 0.9 we have to do another iteration. If we read the third row we obtain the following buffer.

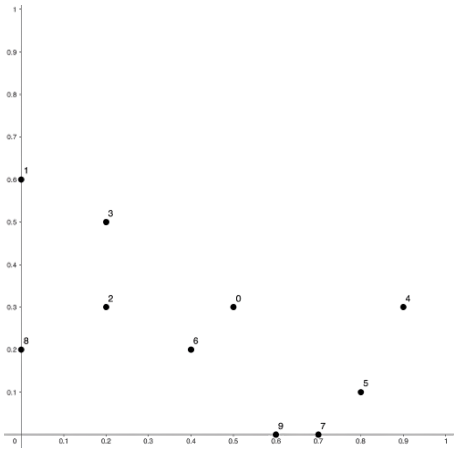
Player	R_1 (def)	R_2 (reb)	Score
4	0.9	0.3	1.2
5	0.8	0.1	0.9

The threshold in this case is the sum of the values of the first row, that is 1.0. Since it is greater than 0.9 we have to do another iteration. If we read the fourth row we obtain the following buffer.

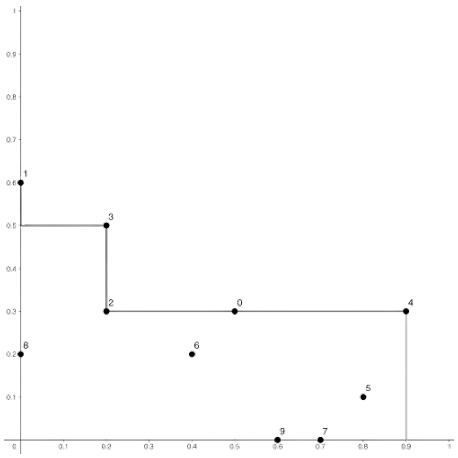
Player	R_1 (def)	R_2 (reb)	Score
4	0.9	0.3	1.2
5	0.8	0.1	0.9

The threshold in this case is the sum of the values of the first row, that is 0.9. Since it is equal to 0.9 the algorithm halts. We found that the best player for this scoring function are four and five.

5. It is possible to draw the points with coordinates (R_1, R_2).



We access the first row and since the window has no elements we add it to it. We check for every tuple if it dominates the inserted one. Only the tuples three and one are not dominated by the tuple four, so the final window contains the points four, three and one. Graphically we have that the skyline of the given dataset is the following.



6. Players zero and five are dominated only by player four, so they are part of the 2-skyband. Players one, three and four are part of the skyline, so they are obviously also part of the 2-skyband. The 3-skyband is made of the players one, three, four, six and seven according to the definition.

4.2 Ranking and skyline queries

Consider the following three ranked lists reporting statistics about European soccer teams:

Goals	Team	Possession	Team	Passes	Team
75	PSG	63.2	BMU	89.9	PSG
73	BMU	62.1	BAR	88.9	MCY
70	ATA	61.8	PSG	88.4	BAR
68	MCY	61.6	MCY	87.5	BMU
68	BDO	59.0	LIV	86.6	BDO
66	LIV	58.5	BDO	84.4	LAZ
63	BAR	55.7	ATA	83.9	LIV
60	LAZ	50.1	LAZ	83.4	ATA

1. Apply an algorithm that does not use random access to determine the best team according to the scoring function:

$$S(\text{team}) = \max\{\text{team.Goals}, \text{team.Possession}, \text{team.Passes}\}$$

2. On this dataset, could you do better than the previous algorithm, in terms of access cost, with an algorithm that also uses random access?

3. Apply now the TA to determine the best team according to the same scoring function as in one.

4. Apply now the TA to determine the best team according to the following scoring function:

$$S(\text{team}) = \text{team.Goals} + \text{team.Possession} + \text{team.Passes}$$

5. Determine the skyline of the dataset.

Solution

1. Since the scoring function is MAX we can use the B_0 algorithm. Find the best team means that we have $k = 1$, so we need to make only one sorted access. By accessing the first row we obtain the following buffer.

Player	Goals	Possession	Passes	Score
PSG	75	?	89.9	89.9
BMU	?	63.2	?	0.8

   The maximum score is given to PSG, that is the best team according to the MAX scoring function.

2. No algorithm could do better than reading at least the top scores on each list.

3. We have again $k = 1$. We make a sorted access to the first row to compute the threshold value and the objects to insert in the buffer.

Player	Goals	Possession	Passes	Score
PSG	75	?	89.9	89.9
BMU	?	63.2	?	0.8

The threshold point has a value that is the MAX of the scores in the first row, that is 89.9. We now make random accesses to complete the data in the buffer (remember that the algorithm searches the missing data when it inserts a new object in the buffer).

Player	Goals	Possession	Passes	Score
PSG	75	61.8	89.9	89.9

Note that the passes for PSG are accessed before via random access, and later via sorted access. So at each iteration (except for the third column) the algorithm makes a sorted access and two random accesses. Since the threshold has the same value as the score of PSG, the algorithm halts and returns PSG as the best team.

4. We have to apply again TA with $k = 1$, but with a different scoring function, so the result may change. The check for the first row is the same as the previous point, except for the score.

Player	Goals	Possession	Passes	Score
PSG	75	61.8	89.9	226.7

The threshold is the sum of the values in the first row 228.1. The threshold is greater than PSG score, so we need to make another iteration. In the next row we find: BMU (223.7), BAR (213.5), and MCY (218.5). All these scores are lower than PSG one, so the buffer remains the same. The threshold point has a value of 224, that is less than PSG score, so the algorithm halts. The best team with SUM is again PSG.

5. It is possible to represent the dataset in a graph with coordinates (Goals, Possession, Passes). By inspecting the ordered table we add PSG to the window and all the other teams that are not dominated by it. The only teams not dominated by PSG are BMU and BAR. We have found that the skyline of the dataset is composed by three teams: PSG, BMU, and BAR.

4.3 Ranking and skyline queries

Consider the following two ranked lists of hotels:

Rating	Hotel	Stars	Hotels
0.8	C	0.8	F
0.4	G	0.6	E
0.4	F	0.5	G
0.3	D	0.4	A
0.3	A	0.2	D
0.2	E	0.2	C
0.1	B	0.2	B

1. Apply FA and TA to determine the top hotel according to the scoring function:

$$\text{MAX}(o) = \max\{\text{Rating}(o), \text{Stars}(o)\}$$

2. Indicate the number of sorted accesses and random accesses executed on each source.
3. Could FA make fewer sorted accesses than TA?
4. Could FA make fewer random accesses than TA?
5. TA is instance optimal: can any algorithm cost overall less than TA?

Solution

1. For the FA we have to make sorted access until we found k objects in all tables. In this case we need to find only one element that is in both columns. To do so we need three sorted accesses, and we obtain the following buffer.

Hotel	Rating	Stars	Score
F	0.4	0.8	1.2
C	0.8	?	0.8
G	0.4	0.5	0.9
E	?	0.6	0.6

We need to find the missing values with two random accesses, and we have

Hotel	Rating	Stars	Score
F	0.4	0.8	1.2
C	0.8	0.2	1.0
G	0.4	0.5	0.9
E	0.2	0.6	0.8

The best hotel according to the given scoring function is F.

The TA checks rows until the value of the threshold is greater than the worst score in the top-k. After accessing the first row we have:

Hotel	Rating	Stars	Score
F	0.4	0.8	1.2

With a threshold of 1.6. After accessing the second row we have the same buffer, and a threshold value of 1.0, that is less than F's score, so the algorithm halts. The best hotel found is again F.

2. With FA we have made six sorted accesses (three for Rating, and three for Stars) and two random accesses (one for Rating, and one for Stars), so the total is eight. With TA we have made four sorted accesses (two for Rating, and two for Stars) and four random accesses (two for Rating, and two for Stars), so the total is eight.

3. No, because FA stops its sorted access phase when all potential top-k objects (according to any possible scoring function) have been seen, so at least k objects are no worse than the threshold according to any scoring function.

4. Yes, in this exercise we have an example.

5. Yes, it is possible.

4.4 Ranking and skyline queries

Consider the following lists reporting statistics about soccer players.

Goals	**Player**	**Assists**	**Player**
25	LEW	17	DEB
21	RON	15	SAN
19	MES	12	MES
18	MBA	6	NEY
15	ILI	5	MBA
14	SAN	5	ILI
13	NEY	3	RON
8	DEB	3	LEW

1. Apply the TA to determine the top-2 players according to the scoring function:

$$S(\text{player}) = \text{player.Goals} + \text{player.Assists}$$

2. Reuse as much as possible the answer of the previous question to indicate the reached depth and the number of sorted and random accesses to determine, with TA, the top-1 player according to the same scoring function as the previous point.

3. Discuss whether any algorithm could attain a lower execution cost than the cost incurred by TA to determine the top-1 player.

4. Apply now the TA to determine the top-2 players according to the scoring function:

$$S(\text{player}) = |\text{player.Goals} - \text{player.Assists}|$$

5. Determine the skyline of the dataset.

Solution

1. The execution of the TA algorithm needs four rounds to stop on the given dataset. At the fourth round we have the following buffer.

Player	**Goals**	**Assists**	**Score**
MES	19	12	31
SAN	14	15	29

With a threshold value of 24. We found that the best two players for the given scoring function are MES and SAN.

2. The procedure is the same as the previous point, but it stops at the third round since the threshold condition is already verified at this step. The buffer is the following.

Player	**Goals**	**Assists**	**Score**
MES	19	12	31

   With a threshold of 31. We have found that the best player is MES.

3. FA would stop at depth three as well, but only needs four random accesses to find the top player.

4. The given function is not monotone, so the TA can return wrong results. In this case the algorithm will fail, returning DEB instead of RON.

5. After sorting the dataset we insert MES in the window. The only players that are not dominated by him are: SAN, LEW, and DEB. So, we have that the skyline is composed by these three players.

CHAPTER 5

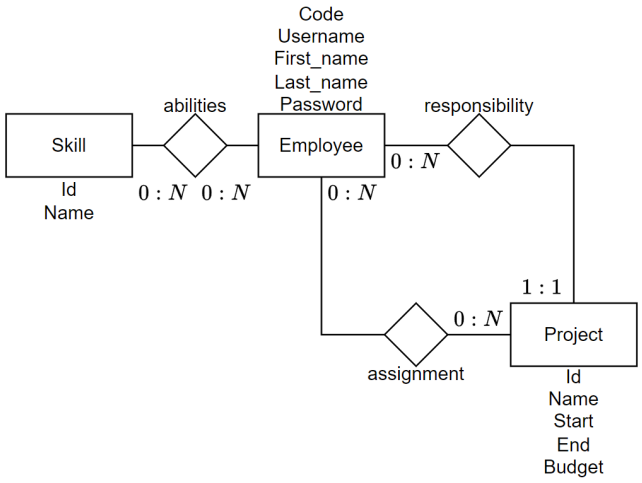
Exercise session V

5.1 Java Persistence API

An application lets an employee edit the projects he is responsible for. A project has a name, a start and end date and a budget. The employee can create projects and assign other employees to them. Employees have a name, a code. Employees also have one or more skills. After logging in, the employee accesses a home page that displays:

- The list of his projects, ordered by end date descending and with the list of employees allocated to each one.
- The list of other employees, with the last name and the list of skills possessed.
- A form for creating a project.
- A form for choosing a project and assigning it to another employee. After creating a project or an assignment, the home page is displayed again, with the information updated.

The entity relationship model is the following:



The relational model is:

- Project(<u>id</u>, name, start, end, budget, responsible)

- Emp_prj(empid, prjid)
- Employee(code, firstname, lastname, username, pwd)
- Skill_emp(skillid, empid)
- Skill(id, label)

Given the specifications write the entity classes of the ORM mapping, including annotations for the attributes and for the relationships, fetch type of attributes and of relationships, and operation cascading policies for relationships (when not by default).

Solution We start by checking all the relationships in the E-R diagram:

- Responsibility: from employee to project we have to use the annotations:

  ```
  @OneToMany
  @OrderBy("end DESC")
  ```

  From project to employee we have to use these annotations:

  ```
  // This annotation can be omitted since it is implicit
  @ManyToOne
  ```

  The owner of the relation is entity project.

- Assignment: from project to employee we have to use the annotations:

  ```
  @ManyToMany
  // To let the client access the employees working in a project via
      ↪ relationship navigation
  FetchType.EAGER
  ```

  From employee to project we have to use these annotations:

  ```
  // This annotation can be omitted since it is implicit
  @ManyToMany
  ```

  The owner of the relation can be either project or employee.

- Abilities: from employee to skill we have to use the annotations:

  ```
  @ManyToMany
  ```

  From skill to employee we have to use these annotations:

  ```
  // This annotation can be omitted since it is implicit
  @ManyToMany
  ```

  The owner of the relation can be either skill or employee.

We can now define the three entity mappings. The entity employee is defined as:

```
@Entity
@NamedQueries({
    @NamedQuery(name = "Employee.findAllButOne", query ="SELECT e FROM Employee e
        ↪ WHERE e.code <> :empid"),...})
public class Employee implements Serializable {
    ...
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int code;

    private String firstname;
    private String lastname;
    private String password;
    private String username;

    @ManyToMany(mappedBy = "employees", fetch = FetchType.EAGER)
    private List<Project> assignedProjects;

    @OneToMany(mappedBy = "manager", fetch = FeatchType.EAGER)
    @OrderBy("end DESC")
    private List<Project> managedProjects;

    @ManyToMany(mappedBy = "employees", fetch = FetchType.EAGER)
    private List<Skill> skills;
    ...
}
```

The entity project is defined as:

```
@Entity
@NamedQuery(name="Project.findAll",query="SELECT p FROM Project p")
public class Project implements Serializable {
    ...
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;

    private String name;
    private int budget;

    @Temporal(TemporalType.DATE)
    private Date start;

    @Temporal(TemporalType.DATE)
    private Date end;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name="emp_prj",
              joinColumns={@JoinColumn(name="projid")},
              inverseJoinColumns={@JoinColumn(name="empid")})
    private List<Employee> employees;
```

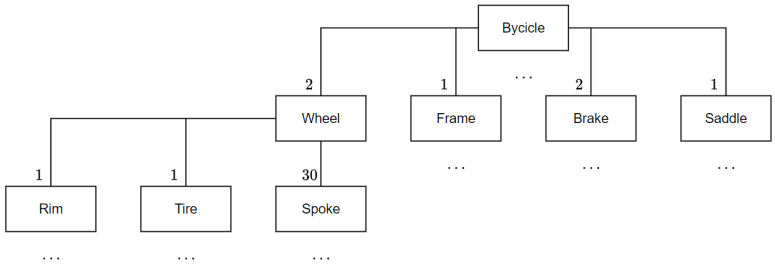
```
    @ManyToOne
    @JoinColumn(name="responsible")
    private Employee manager;
    ...
}
```

The entity skill is defined as:

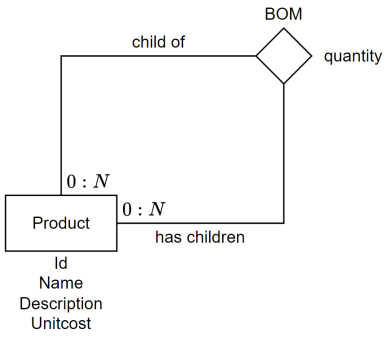
```
@Entity
public class Skill implements Serializable {
    ...
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String label;
    @ManyToMany
    @JoinTable(name="skill_emp",
               joinColumns={@JoinColumn(name="skillid")},
               inverseJoinColumns={@JoinColumn(name="empid")}
               )
    private List<Employee> employees;
    ...
}
```

5.2 Java Persistence API

An application permits the management of the bill of materials (BoM) of products. The BoM is the hierarchical description of a product in terms of the sub-products that comprise it. At each level but the last one, a product is associated with the components that make it, each with a quantity. The application allows the user to create BoMs. A BoM is progressively assembled by attaching to a product its sub-products and specifying the number of units of sub-products that make one unit of the parent product. The editor accesses the HOME PAGE with the list of current BoMs and where s/he can create a new top level product and view the existing BOMs. The editor can add product-sub-products links to a product, modify the quantity of a product-sub-products link, delete a product or product-sub-products link. Products have an identifier, a name a description and a unit cost. An example of BoM is the following:



The entity relationship model is the following:



The relational schema DDL of the given database is:

```
CREATE TABLE 'product' (
'id'           INT            NOT NULL AUTO_INCREMENT,
'unitcost'     INT            NOT NULL,
'name'         VARCHAR(45)    NOT NULL,
'description'  VARCHAR(45)    DEFAULT NULL,
PRIMARY KEY ('id')
)
CREATE TABLE 'subparts' (
'father'       INT            NOT NULL,
'child'        INT            NOT NULL,
'quantity'     INT            NOT NULL,
PRIMARY KEY ('father','child'),
KEY 'childtoproduct_idx' ('child'),
CONSTRAINT 'childtoproduct' FOREIGN KEY ('child') REFERENCES 'product' ('id'),
CONSTRAINT 'fathertoproduct' FOREIGN KEY ('father') REFERENCES 'product' ('id')
)
```

The relational model is:

- Product(id, unitcost, name, description)
- Subparts(father, child, quantity)

Given the specifications, write the entity classes of the ORM mapping, including annotations for the attributes and for the relationships, fetch type of attributes and of relationships, and operation cascading policies for relationships (when not by default).

Solution Given the specifications write the entity classes of the ORM mapping, including annotations for the attributes and for the relationships, fetch type of attributes and of relationships, and operation cascading policies for relationships (when not by default).

- BoM: from father product to children product we have to use the annotations:

  ```
  @ManyToMany
  ```

  From children product to father product we have to use these annotations:

  ```
  @ManyToMany
  ```

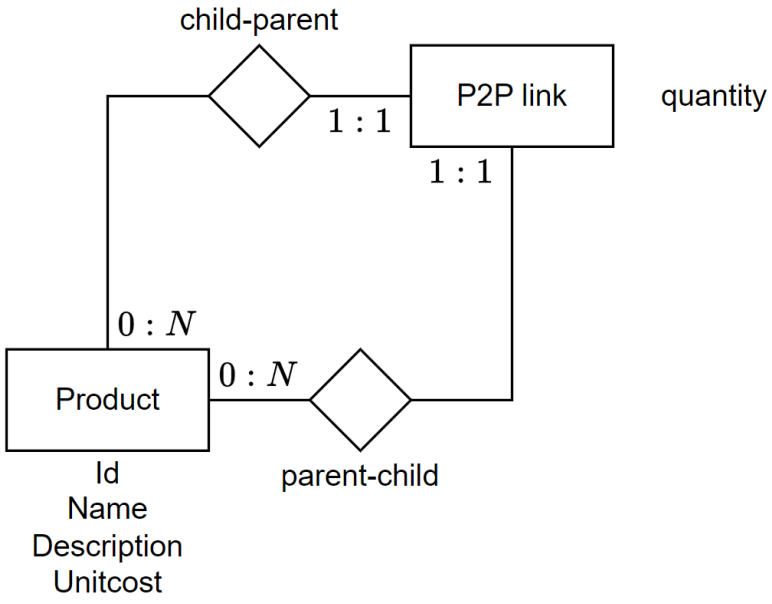
  The owner of the relation is entity project.

-

The entity product is defined as:

```
@Entity
@NamedQueries({
@NamedQuery(name = "BomProduct.findAll", query = "SELECT p FROM BomProduct p"),
@NamedQuery(name = "BomProduct.findAllTop", query = "SELECT p FROM BomProduct p
    ↪ WHERE p.fathers IS EMPTY") })

public class BomProduct implements Serializable {
...
private static final long serialVersionUID = 1L;
@Id @Column(name="id") @GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
private String description;
private String name;
private int unitcost;
...
@ElementCollection(fetch = FetchType.EAGER)
@CollectionTable(name = "subparts",joinColumns = @JoinColumn(name = "father"))
@MapKeyJoinColumn(name = "child")
@Column(name = "QUANTITY")
private Map<BomProduct, Integer> subparts;

@ManyToMany
@JoinTable(name = "subparts",
          joinColumns = @JoinColumn(name = "child"),
          inverseJoinColumns = @JoinColumn(name ="father"))
private List<BomProduct> fathers;
...
}
```

The better way is to make the many-to-many relationships with attributes a weak entity like in the following image.



The entity P2PLinkID is defined as:

```
@Embeddable
public class P2PLinkID implements Serializable {
private static final long serialVersionUID = 1L;
private int father;
private int child;

public P2PLinkID() { }

public P2PLinkID(int father, int child) {
    super();
    this.father = father;
    this.child = child;
}
...
}
```

The entity P2PLink is defined as:

```
@Entity
public class P2PLink implements Serializable {
private static final long serialVersionUID = 1L;

@EmbeddedId
private P2PLinkID id;

@ManyToOne
@MapsId("father") // reference to the foreign key attribute
@JoinColumn(name = "father")
private BomProduct father;

@ManyToOne
@MapsId("child") // reference to the foreign key attribute
@JoinColumn(name = "child")
private BomProduct child;

private int quantity;
...
}
```

The entity product is defined as:

```
@Entity
public class BomProduct implements Serializable {
private static final long serialVersionUID = 1L;

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
private String description;
private String name;
private int unitcost;

// getters setters and constructors
```

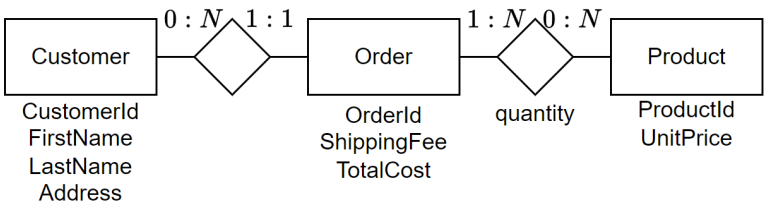
```
@OneToMany(mappedby="father")
private List<P2PLink> children;

@OneToMany(mappedby="child")
private List<P2PLink> fathers;
...
}
```

CHAPTER 6

Exercise session VI

6.1 Java Persistence API

An application manages the shipment of orders to customers from an inventory. The data are organized according to the following conceptual model.



The application permits the operator to select a customer, open his/her list of orders, select an order and the list of its products with the ordered quantity associated with each product in the order. Customers are in the order of millions, orders per customer are in the order of thousands and products per order are in the order of tens. Show the JPA entities (with all their attributes) that map the domain objects of the conceptual model, taking into account the above-mentioned access paths of the application and data cardinalities. When designing the annotations for the relationships, specify the owner side of the relationship, the mapped-by attribute, the fetch policy and the cascading policies you consider more appropriate to support the access required by the web application.

Solution We start by checking all the relationships in the E-R diagram:

- Orders/orderedBy: from customer to order we have to use the annotations:

  ```
  @OneToMany
  FetchType.LAZY
  ```

  From order to customer we have to use these annotations:

  ```
  // This annotation can be omitted since it is implicit
  @ManyToOne
  ```

  The owner of the relation is order.

- Contains/containedIn: from order to product we have to use the annotations:

  `FetchType.EAGER`

  From product to order we have to use these annotations:

  ```
  // This annotation can be omitted since it is implicit
  @ManyToMany
  ```

  The owner of the relation is product.

We can now define the three entity mappings. The entity customer is defined as:

```
@Entity
public class Customer implements Serializable {

    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private int customerId;
    private String firstName;
    private String lastName;
    private String address;

    @OneToMany(mappedBy="customer", fetch = FetchType.LAZY)
    private List<Order> orders;
    . . .
}
```

The entity product is defined as:

```
@Entity
public class Product implements Serializable {

    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private int productId;
    private int unitPrice;

    @ManyToMany
    @JoinTable(name = "product_order",
               joinColumns = @JoinColumn(name = "productId"),
               inverseJoinColumns = @JoinColumn(name ="orderId"))
    private List<Order> orders;
    . . .
}
```

The entity order is defined as:

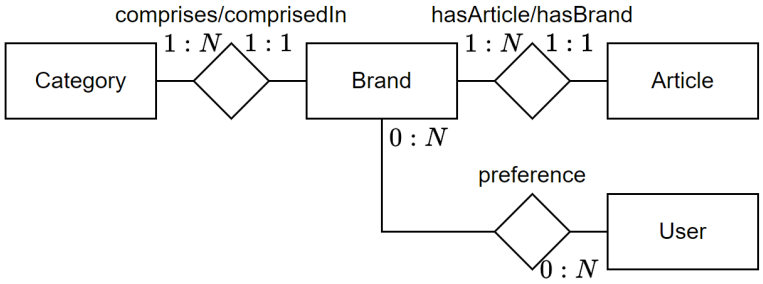
```
@Entity
public class Order implements Serializable {

    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private int orderId;
    private int shippingFee;
    private int totalCost;

    @ManyToOne
```

```
    @JoinColumn(name="customer")
    private Customer customer;

    @ElementCollection(fetch = FetchType.EAGER)
    @CollectionTable(name = "product_order",
    joinColumns = @JoinColumn(name = "orderId"))

    @MapKeyJoinColumn(name = "productId")
    @Column(name = "quantity")
    private Map<Product, Integer> products;
    . . .
}
```

6.2 Java Persistence API

An e-commerce web application manages data about catalogs of articles belonging to different brands and categories. After logging in, the user can access a HOME page where he can start to drill down the catalog. In the HOME page, he can select one category (e.g., "sports apparel") from a list of categories to see its brands, which are shown in a brands page. For the list of brands in the brands page, he can select one brand (e.g., "Adidas") of the chosen category to see its articles, which are listed in an articles page. By choosing one article from the list in the articles page he can see the details of the chosen article in the article page. The details of an article include: code, name, price, picture and the brand it belongs to. Clicking on the brand attribute of an article leads back to the articles page showing all the articles of that brand. When a user displays the details of an article, the application creates a relationship between the user and the article's brand, to record that the user may have a preference for such a brand. The categories are in the order of tens, the brands in the order of hundreds, and the articles in the order of hundreds of thousands.

Write the entity classes of the ORM mapping, including annotations for the attributes and for the relationships, fetch type of attributes and of relationships, and operation cascading policies for relationships (when not by default). Motivate the design choices. Specify the named queries used by the methods of the business objects.



Solution We start by checking all the relationships in the E-R diagram:

- Comprises/comprisedIn: from category to brand we have to use the annotations:

  ```
  @OneToMany
  FetchType.EAGER
  ```

Note that remove can be cascaded. From brand to category we have to use these annotations:

```
// This annotation can be omitted since it is implicit
@ManyToOne
```

The owner of the relation is brand.

- Articles/hasBrand: from brand to article we have to use the annotations:

```
@OneToMany
FetchType.LAZY
```

Note that remove can be cascaded. From article to brand we have to use these annotations:

```
@ManyToOne
```

The owner of the relation is article.

- Preference: from user to brand we have to use the annotations:

```
@ManyToMany
FetchType.EAGER
```

From brand to user we have to use these annotations:

```
// This annotation can be omitted
@ManyToOne
FetchType.LAZY
```

The owner of the relation can be either user or brand.

We can now define the four entity mappings. The entity category is defined as:

```
@Entity
public class Category{

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int categoryId;
    private String categoryName;

    @OneToMany(mappedBy="category", fetch = FetchType.EAGER,
    cascade = CascadeType.REMOVE)
    private List<Brand> brands;
    ...
}
```

The entity brand is defined as:

```
@Entity
public class Brand {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int brandId;
    private String brandName;
```

```
    @ManyToOne
    @JoinColumn(name="category")
    private Category category;

    @OneToMany(mappedBy="brand", fetch = FetchType.LAZY,
    cascade=CascadeType.REMOVE)
    private List<Article> articles;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable( name="usr_brand", joinColumns={
              @JoinColumn(name="brandid")},
              inverseJoinColumns={@JoinColumn(name="userid")})
    private List<User> interestedUsers;
    ...
}
```

The entity article is defined as:

```
@Entity
public class Article{

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int articleId;
    private String articleCode;
    private String articleName;
    private double price;

    @Basic(fetch = FetchType.LAZY) @Lob
    private byte[] picture;

    @ManyToOne
    @JoinColumn(name="brand")
    private Brand brand;
    ...
}
```

The entity user is defined as:

```
@Entity
public class User implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private String password;
    private String username;

    @ManyToMany(mappedBy = "interestedUsers", fetch = FetchType.EAGER)
    private List<Brand> preferredBrands;
    ...
}
```

6.3 Triggers

Consider the following relational schema:

- EXAMINATION (StudentID, Course, EDate, Grade)
- PRECEDENCERULE (PrecedingCourse, FollowingCourse)

With the following integrity constraint: it is forbidden to take an exam if there is a preceding course that has not been passed yet.

1. Find the operations that can violate the constraint.
2. Based on the common sense experience, choose the operation that will actually violate the constraint.
3. Write a trigger for that operation, that rolls back the effects when the constraint is violated.

Solution

1. The operations that can violate the integrity constraint are:
   - `INSERT` of a new EXAMINATION, when there are preceding courses that are not sustained.
   - `UPDATE` of the Date or Course on EXAMINATION.
   - `DELETE` of an EXAMINATION.
   - `UPDATE` of an existing PRECEDENCERULE.
   - `INSERT` of a new PRECEDENCERULE.
2. The operation that most typically will violate the rule is the first one. Updates and deletions on examinations should occur exceptionally, and only to handle specific errors. Changes to precedence rules, instead, may affect older exams. One solution to this issue is to extend the schema with a date from which the precedence rule is activated and a date in which the rule is to be considered invalidated.
3. The requested trigger is:

```
CREATE TRIGGER MissingPrecedence
BEFORE INSERT ON EXAMINATION
FOR EACH ROW
WHEN EXISTS (
    SELECT *
    FROM PRECEDENCERULE
    WHERE FollowingCourse = NEW.Course
        AND PrecedingCourse NOT IN (
            SELECT Course
            FROM EXAMINATION
            WHERE StudentID = NEW.StudentID
            AND EDate < NEW.EDate)
    )
ROLLBACK;
```

6.4 Triggers

Consider the following schema describing a system for hiring rehearsal rooms to musical groups. Start and end time of reservations contain only hours (minutes are always "00"). Use of rooms can only happen if there is a corresponding reservation, but can start later or end earlier. All rooms open at 7:00 and close at 24:00.

- USER (SSN, Name, Email, Type)
- RESERVATION (UserSSN, RoomCode, Date, StartTime, EndTime)
- ROOM (RoomId, Type, CostPerHour)

1. Write a trigger that prevents the reservation of already booked rooms.
2. Suppose that usage data are inserted into a table Usage only after the room has been actually used. Enrich the schema to track the number of hours that have been reserved but not used by each user, and write a (set of) trigger(s) that track such a number and set the "type" of a user to "unreliable" when he totalizes 50 hours of unused reservations.

Solution

1. The requested trigger is:

```
CREATE TRIGGER ThouShallNotBook
BEFORE INSERT INTO Reservation
FOR EACH ROW
WHEN EXISTS (
    SELECT *
    FROM RESERVATION
    WHERE RoomCode = NEW.RoomCode
    AND Date = NEW.Date AND
    StartTime < NEW.EndTime AND EndTime > NEW.StartTime
    )
ROLLBACK;
```

2. We add and modify the tables as follows:

   - USAGE (UserSSN, RoomCode, Date, StartTime, EndTime)
   - USER (SSN, Name, Email, Type, WastedHours)

   The requested triggers are:

```
CREATE TRIGGER UpdateWastedHours
AFTER INSERT ON USAGE
FOR EACH ROW
UPDATE USER
SET WastedHours = WastedHours +
    ( SELECT EndTime - StartTime - ( NEW.EndTime - NEW.StartTime )
      FROM Reservation
      WHERE RoomCode = NEW.RoomCode AND Date = NEW.Date
      AND StartTime <= NEW.StartTime AND EndTime >= NEW.EndTime )
```

```
WHERE SSN = NEW.UserSSN

CREATE TRIGGER UpdateType
AFTER UPDATE OF WastedHours ON USER
FOR EACH ROW
WHEN old.WastedHours < 50 AND NEW.WastedHours >= 50
UPDATE USER
SET Type = "Unreliable"
WHERE SSN = OLD.SSN;
```

CHAPTER 7

Exercise session VII

7.1 Triggers

Consider the following relational schema:

- STUDENT (ID, Name, Address, Phone, Faculty, Year1, Campus, EarnedCredits)
- ENROLLMENT (StudID, CourseCode, Year1, Date)
- EDITION (CourseCode, Year1, Teacher, Semester, #Students, #ExternalStuds)
- COURSE (CourseCode, Name, Credits, Campus)

1. Write a trigger that rolls back the creation of a new ENROLLMENT if the referenced STUDENT and/or EDITION does not exist in the corresponding table.
2. The #ExternalStuds attribute in EDITION represents the number of students enrolled to the course, that are associated to a different Campus with respect to the one where the COURSE is held. Write a trigger that updates (if needed) the value of EDITION.#ExternalStuds when a Student moves from a campus to another.

Solution

1. The requested trigger is:

```
CREATE TRIGGER CheckEnrollment
AFTER INSERT ON ENROLLMENT
WHEN NOT EXISTS (
    SELECT *
    FROM STUDENT
    WHERE ID = NEW.StudID
    )
OR NOT EXISTS (
    SELECT *
    FROM EDITION
    WHERE CourseCode = NEW.CourseCode AND Year1 = NEW.Year1
    )
ROLLBACK;
```

2. Modifying the Campus of a student has the following effects. The student becomes an "External" student for all the courses he is enrolled into that are located in the campus he is leaving. The student becomes an "Internal" student for all his courses located in the campus where he is moving. There are two counters to update. The requested triggers are:

```
CREATE TRIGGER CheckCampus1
AFTER UPDATE OF Campus ON STUDENT
FOR EACH ROW
BEGIN
    UPDATE EDITION
    SET #ExternalStuds = #ExternalStuds + 1
    WHERE (CourseCode, Year1) IN (
        SELECT CourseCode, Year1
        FROM ENROLLMENT
        WHERE StudID = OLD.ID
        )
    AND CourseCode IN (
        SELECT CourseCode
        FROM COURSE
        WHERE Campus = OLD.Campus
        );
END

CREATE TRIGGER CheckCampus2
AFTER UPDATE OF Campus ON STUDENT
FOR EACH ROW
BEGIN
    UPDATE EDITION
    SET #ExternalStuds = #ExternalStuds - 1
    WHERE (CourseCode,Year1) IN (
        SELECT CourseCode, Year1
        FROM ENROLLMENT
        WHERE StudID = OLD.ID
        )
    AND CourseCode IN (
        SELECT CourseCode
        FROM COURSE
        WHERE Campus = NEW.Campus
        );
END
```

The use of old/new is fundamental to distinguishing the values of the Campus attribute and to identify the campus that the students are leaving/joining. Instead, using old or new is equivalent for the ID attribute (that value is not changed by the update event). If (by any chance) the update re-assigns to the Campus attribute the same value (i.e., if old.Campus = new.Campus), the counters remain unaltered.

7.2 Triggers

A logistics company manages shipments via trucks. Each driver owns one or more trucks and has a number of colleagues who can drive his/her trucks. A truck makes zero or more trips. A trip specifies that a given truck must travel from an origin to a destination on a certain date. When the driver completes a trip, a report is created with the number of hours travelled. The following logical schema is given:

- TRUCK(plate, status, ownerId)
- DRIVER(id, hoursTravelled, status)
- TRIP(TripId, date, origin, destination, truckPlate, driverId)
- TRIPREPORT(tripId, hoursTravelled)
- ALTERNATE(driverId, alternativeDriverId)

Suppose that when a new trip is created, the initial values for the truckPlate and driverId are set to `NULL`. Design a set of triggers that implement the following behaviors:

1. When a truck is assigned to a trip, if the status of the truck is maintenance the operation is prevented from proceeding, and the trigger raises an exception with text Truck under maintenance. After the assignment of an available truck to a trip, a trigger tries to automatically assign a suitable driver, giving priority to the truck's owner. He/she can be assigned only if his/her status is authorized and he/she is available for that date. Otherwise, an alternate driver is searched: the one with the least cumulative travel hours among those with authorized status available for that date is chosen. If a suitable driver is not found, the trigger raises an exception with text No driver available.

2. When a trip report is created, the total number of hours travelled by the driver must be updated. If the hours s/he has travelled exceeds 1000, his/her status is set to non-authorized and his/her assignments to the trips posterior to the reported one are set to `NULL`.

Solution

1. The requested triggers are:

```
CREATE TRIGGER CheckMaintenance
BEFORE UPDATE OF truckPlate ON TRIP
FOR EACH ROW
BEGIN
    DECLARE truck_status VARCHAR(20);
    SELECT status INTO truck_status
    FROM TRUCK
    WHERE PLATE = NEW.truckPlate;
    IF truck_status = 'maintenance' THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Truck under maintenance';
    END;
END;

CREATE TRIGGER AssignDriver
```

```
AFTER UPDATE OF truckPlate ON TRIP
FOR EACH ROW
BEGIN
    DECLARE owner_id INT;
    DECLARE suitable_driver_id INT;
    SELECT ownerId INTO owner_id
    FROM TRUCK
    WHERE PLATE = NEW.truckPlate;
    SELECT ID INTO suitable_driver_id
    FROM DRIVER
    WHERE ID = owner_id AND status = 'authorized'
    AND NOT EXISTS (
        SELECT *
        FROM TRIP T
        WHERE T.driverId = owner_id
        AND T.date = NEW.date
    )
    IF suitable_driver_id IS NULL THEN
        SELECT A.ALTERNATEDRIVERID INTO suitable_driver_id
        FROM ALTERNATE A
        JOIN DRIVER D ON A.ALTERNATEDRIVERID = D.ID
        WHERE A.DRIVERID = owner_id
        AND D.status = 'authorized'
        AND NOT EXISTS (
            SELECT *
            FROM TRIP T
            WHERE T.driverId = A.ALTERNATEDRIVERID
            AND T.date = NEW.date
            )
        ORDER BY D.hoursTravelled ASC
        LIMIT 1;
    END IF;
    IF suitable_driver_id IS NOT NULL THEN
        UPDATE TRIP
        SET driverId = suitable_driver_id
        WHERE NEW.TRIPID = TRIPID;
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'No driver available';
    END IF;
    END IF;
END;

CREATE TRIGGER UpdateHours
AFTER INSERT ON TRIPREPORT
FOR EACH ROW
BEGIN
    UPDATE DRIVER
    SET hoursTravelled += NEW.hoursTravelled
    WHERE ID = (SELECT driverId FROM TRIP WHERE TRIPID = NEW.TRIPID);
END;
```

```
CREATE TRIGGER UpdateDriverStatus
AFTER UPDATE OF hoursTravelled ON DRIVER
FOR EACH ROW
WHEN NEW.hoursTravelled > 1000
    BEGIN
        UPDATE DRIVER
        SET status = 'non-authorized'
        WHERE ID = NEW.ID;
        UPDATE TRIP
        SET driverId = NULL
        WHERE driverId = (SELECT driverId FROM TRIP WHERE TRIPID =
            ↪ NEW.TRIPID)
        AND date > NOW();
    END;
```

CHAPTER 8

Exercise session VIII

CHAPTER 9

Exercise session IX