Resolução 2º teste SBD (2016/17)

1a) A instrução SELECT em SQL efetuam operações de leitura sobre items e UPDATE e INSERT de escrita.

Passo	T1	T2
1	read(TVI)	
2	write(RTP1)	
3		write(TVI)
4		read(RTP1) read(RTP2) read(TVI)
5		write(SIC)
6	read(RTP1) read(RTP2)	

Existe um conflito entre 1:read(TVI) e 3:write(TVI) e 2:write(RTP1) e 4:read(RTP1). Como não existe conflito entre as leituras no passo 6 e as escritas em 3, 4 e 5 é possível trocar as instruções do passo 6 com as do 3, 4 e 5, dando origem ao escalonamento, justificando que o escalonamento dado é seriali´zavel por conflito e equivalente a T1 seguido de T2.

Passo	T1	T2
1	read(TVI)	
2	write(RTP1)	
6	read(RTP1) read(RTP2)	
3		write(TVI)
4		read(RTP1) read(RTP2) read(TVI)
5		write(SIC)

Como resposta alternativa pode-se ver que o grafo de precedências tem apenas dois nós com T1 e T2 e com um arco de T1 para T2 pois T1 e T2 são conflituantes e T1 acedeu a dados primeiro que T2. Como não existe ciclo no grafo, logo o escalonamento é serializável por conflito.

1b) Um escalonamento que ilustra bem o protocolo de concorrência baseado em timestamps é o seguinte:

Passo	Transação 1 (ts=1)	Transação 2 (ts=2)	Item TVI(R,W)=TVI(0,0)
	write(TVI)		TVI(0,1)
		read(TVI)	TVI(2,1)
	write(TVI)		conflito pois 1 < 2 e T1 é desfeita

O protocolo de locking adequado para controlar a concorrência é o two phase-locking evitando interferências indevidas (em baixo temos o estrito pois libertamos locks exclusivos após o commit). Assim, obteríamos:

Passo	Transação 1	Transação 2	Observação
1	lock-x(TVI)		
2	write(TVI)		
		lock-s(TVI)	T2 fica à espera
3	write(TVI)		
4	commit		
5	unlock(TVI)		
6		read(TVI)	

Como se vê, a transação 2 ficará à espera até a transação T1 libertar o lock exclusivo e lerá um valor consistente (o produzido pela transação T1 no final).

COMENTÁRIO: repare-se também que é essencial utilizar o 2PL pois se libertássemos o lock antes poderíamos ter interferências indesejadas entre as transações (o que muitos fizeram no teste...):

Passo	Transação 1	Transação 2	Observação
1	lock-x(TVI)		
2	write(TVI)		
3		lock-s(TVI)	T2 espera um pouco...
4	unlock(TVI)		
5		read(TVI)	Prossegue e lê um valor que não deveria...
6	lock-x(TVI)		
7	write(TVI)		
8	unlock(TVI)		
9	commit		

1d) Em modo SNAPSHOT ISOLATION uma transação apenas "vê" o que estava confirmado antes do início da sua execução e o que foi alterado por ela própria.

Assim, a tabela N irá conter no final os seguintes três tuplos:

N	Justificação
1	T1 só vê o tuplo inserido por ela
3	T2 só vê os tuplos inseridos por ela
2	T3 vê o tuplo inserido por T1 e o tuplo inserido por ela

1e) O cenário mais simples e o da cópia primária em que para obter um lock de um determinado item de dados todas as transações enviam um pedido ao gestor de locks desse item. No cenário abaixo T1 obtém o lock primeiro, liberta e seguidamente T2 obtém esse lock e executa, e finalmente é executada T3. Não ocorre deadlock

Passo	Transação 1	Transação 2	Transação 3
	lock-x(SIC)		
		lock-x(SIC)	
			lock-x(SIC)
	write(SIC)		
	commit		
	unlock(SIC)		
		write(SIC)	
		commit	
		unlock(SIC)	
			write(SIC)
			commit
			unlock(SIC)

No segundo cenário temos 3 cópias e cada transação tem de pedir a pelo menos 2 réplicas que lhe seja atribuído o lock. Como cada transação efetua o pedido a pelo menos dois locais podemos ter a situação em que T1 obtém um lock numa réplica, T2 noutra e T3 na outra. Assim, nenhuma tem uma maioria de locks criando um deadlock (este tipo de deadlock não pode ocorrer num sistema centralizado).

2a) O protocolo de 2PL serve para controlo de concorrência garantindo isolamento das transações pois os escalonamentos obtidos são serializáveis por conflito. O protocolo tem duas fases:

- Fase de crescimento (growing): só se podem obter locks, não podendo libertar qualquer lock (pode-se contudo fazer upgrade dos locks, isto é promover locks partilhados a exclusivos)
- Fase de decrescimento (shrinking): libertam-se locks não podendo voltar a efetuar-se locks

As transações podem ser serializadas pela ordem de obtenção de todos os seus locks (ou seja, quando termina a primeira fase). Contudo, podem ocorrer deadlocks no 2PL e são possíveis rollbacks em cascata. Existem duas variantes:

- estrito: a transação mantém os seus locks exclusivos até terminar (commit ou abort), assim evitam-se rollbacks em cascata
- rigoroso: a transação mantém todo os seus locks até terminar (commit ou abort), assim evitam-se rollbacks em cascata e a serialização das transações é por ordem de confirmação.

2b) Para se escrever um bloco de memória para disco num SGBD com recuperação baseada em logs e buffering de logs e disco temos de realizar as seguintes operações exatamente por esta ordem:

- Impedir escritas no bloco a escrever (por exemplo com lock/latch do bloco)
- Escrever o log por ordem para armazenamento estável
- Escrever o bloco de memória para disco
- Permitir escritas no bloco alterado

2c) Podemos utilizar um gestor de locks único e centralizado para todo o sistema, o que é de fácil implementação mas é um ponto único de falha e de possível contenção do sistema.

Alternativamente, podemos ter um sistema distribuído em que cada local mantém o seu grafo de espera (wait-for) e o envia para o detetor de deadlocks. A implementação é um pouco mais complexa pois exige enviar o grafo local para o detetor de deadlocks, mas o maior problema é a possibilidade de existirem falsos positivos (i.e. o coordenador erradamente indica que existe um deadlock quando não existe).

COMENTÁRIO: alguns de vós indicaram que poderia ser utilizado um sistema baseado em timeouts, o que foi considera correto e tem o mesmo problema de falsos positivos (possivelmente ainda mais...) pois há sistemas que implementam desta forma a detecção de deadlocks distribuídos dada a sua simplicidade. Outra solução brevemente mencionada na aula foi uma versão totalmente distribuída recorrendo a um algoritmo de obtenção de um estado global do sistema (*distributed snapshot*) mas que pode exigir um elevado número de troca de mensagens.