DI-FCT/UNL **11 de junho de 2019**

Sistemas de Bases de Dados

2.º teste (com consulta limitada: 1 folha identificada) - Duração: 2 horas

N.º:		**Nome:**	

Grupo 1 (10 valores: 1 + 1,2 + 2,4 + 3,2 + 2,2)

1 a) Considere a tabela ***alunos****(id, nota)* já com os tuplos (1,18) e (2,13), previamente criada com:

```
create table alunos(id int primary key,
                    nota int check(nota between 0 and 20));
```

Apresente um escalonamento de transações com operações apenas sobre esta tabela, que seja serializável de vista, mas não de conflito:

T1	T2	T3
Select * from alunos;		
	Update alunos Set nota =20 where id=1;	
Update alunos Set nota =16 where id=1;		
		Update alunos Set nota =19 where id=1;

1 b) O protocolo de *locking* de 2 fases garante escalonamentos serializáveis de conflito? Se sim, diga como seria dada a ordem de serialização. Se não, dê um exemplo.

Yes; transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).

1 c) Para cada um dos 4 escalonamentos abaixo (onde *sel*, *upd* e *ins* representam, respetivamente, *select*, *update*, *insert*, sobre tabelas com colunas *a* e *b*), indique (com 'X') se é ou não serializável de conflito. Em caso afirmativo, indique também uma serialização que lhe seja equivalente (de conflito).

Nota: Nesta alínea cada opção certa vale 0,6 valores, e cada opção errada vale –0,6 valores (negativo, portanto). Uma resposta afirmativa correta mas com serialização incorreta será cotada a 50% (i.e. 0,3 valores positivos).

	Não serializável	**Serializável**	**Serialização**
S1:	X		
S2:		X	T3, T1, T2
S3:	X		
S4:		X	T2,T1

S1

T1	**T2**
upd s set a=0	
	upd t set a=0
sel * from s	
sel * from t	
	sel * from s
	sel * from t

S2

T1	**T2**	**T3**
sel * from t		
sel * from s		
		sel * from t
		upd v set a=0
	upd v set b=0	
upd t set a=0		
	sel * from v	
upd s set a=0		
	upd s set a=0	

S3

T1	**T2**	**T3**	**T4**
	sel * from t		
		sel * from s	
			upd t set a=0
sel * from s			
sel * from t			
		upd s set a=0	
	upd s set b=0		
upd t set a=1			

S4

T1	**T2**
	sel * from s;
sel * from t where a=1;	
ins into s(a,b) values (1,1);	
	upd t set b=1 where a=0;
	sel * from t;
sel count(*) from s;	

N.º:	

1 d) Voltando à tabela ***alunos****(id, nota)* da alínea *a* (com tuplos <1,18> e <2,13>), considere as seguintes transações executando em *Snapshot Isolation* :

T1	T2
Insert into alunos values (3,12);	
Update alunos set nota=nota-1 where mod(id, 2) = 1;	
	Select * from alunos;
	Update alunos set nota=nota+1 where mod(nota, 2) = 1;
Commit;	
	Commit;

I. Alguma transação abortará ou bloqueará? Se sim, qual e onde/quando?

Não

II. O que verá o utilizador como resultado da instrução '*select * from alunos*' dentro de T2?

{(1,18), (2,13)}

III. Qual o conteúdo da tabela *alunos* após a conclusão de todo o escalonamento?

{(1,17), (2,14), (3,11)}

1 e) Como sabe, o uso de *locks* pode levar a *deadlocks*.

I. Indique uma possível estratégia de controlo de concorrência para evitar *deadlocks* (sem abortar transações).

Require that each transaction locks all its data items before it begins execution (predeclaration).
(or Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order.)

II. Indique agora uma estratégia baseada nos *timestamps* das transações para quebrar *deadlocks* (abortando transações).

wait-die scheme — older transaction may wait for younger one to release data item. (older means smaller timestamp). Younger transactions never wait for older ones; they are rolled back instead.
(or wound-wait scheme — older transaction wounds (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.)

III. Num esquema baseado em *timeout*, há um limite de tempo para a obtenção dum *lock* — caso seja ultrapassado, a transação aborta e recomeça. Quais as vantagens e inconvenientes deste esquema?

simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Grupo 2 (10 valores: 2 + 2 + 0,9 + 1,3 + 1,4 + 1 + 1,4)

2 a) Considere o modelo simplificado de *log* dado nas aulas, e o seguinte escalonamento executado em modo *read committed*, onde inicialmente X, Y, e Z têm o valor 1.

T1	X ← 3			Z ← 2*Y	X ← Z+X
T2		Y ← X+2	Commit		

I. Escreva a sequência de registos de *log* referentes a este escalonamento.

<T1 start>
<T1, X, 1,3>
<T2 start>
<T2, Y, 1,3>
<T2 commit>
<T1, Z, 1,6>
<T1, X, 3,9>

II. Apresente agora a sequência adicional de registos fruto do abortar da transação T1.

<T1, X, 3>
<T1, Z, 1>
<T1, X, 1>
<T1 abort>

2 b) Complete (à direita) o registo de *log* abaixo, após uma falha do sistema. I.e. acrescente os registos associados à sua recuperação.

<T4, X, 3,4>
<checkpoint {T3,T4}>
<T4, Y, 0,1>
<T5 start>
<T4 commit>
<T5, Z, 3,4>
<checkpoint {T3,T5}>
<T3, D, 2,1>
<T6 start>
<T6, F, 9,4>
<T3, D,2>
<T1 start>
<T1, B, 3,2>
<T3 abort>
<T5, D, 2,5>
<T2 start>
<T2, O1, operation-begin>
<T2, B, 2,5>
<T5, C, 2,4>
<T2, O1, operation-end, (B,-3)>
<T2, A, 3,4>
<T1 commit>
<T2, O4, operation-begin>
<T2, B, 5,4>
<T2, O4, operation-end, (B,+1)>
<T2, B, 5>
<T2, O4, operation-abort>
<T5, O3, operation-begin>
<T5, B, 5,0>

<T5, B, 5>
<T2, A, 3>
<T2, B, 5,2>
<T2, O1, operation-abort>
<T5, C, 2>
<T2 abort>
<T5, D, 2>
<T6, F, 9>
<T6 abort>
<T5, Z, 3>
<T5 abort>

N.º:	

2 c) Qual o mecanismo base para permitir *fuzzy checkpointing*?
(Não é necessário descrever o processo)

Store a pointer to the last checkpoint record in a fixed position, last_checkpoint, on disk

2 d) Para uma tabela ***clientes****(id, nome, cidade, saldo)*, dê um exemplo (baseando-se nos seus atributos) de fragmentação vertical e escreva uma consulta SQL que, nesse caso, poderia beneficiar de processamento paralelo.

Clientes1(id, nome, cidade), Clientes2(id, saldo)

Select * from clientes where cidade='Lisboa' and saldo > 1000000

2 e) No protocolo de '*commit* em 2 fases' em sistemas distribuídos, o que tem de fazer o coordenador *C* duma transação *T*, e o que tem de acontecer para se realizar o *commit*?

In phase 1, C asks all participants to prepare to commit transaction T: it adds the record <prepare T> to the log, forces log to stable storage, and sends 'prepare T' messages to all sites at which T executed.
C must receive a 'ready T' message from all such sites (phase 2), in order to commit T, in which case it records the decision in the log (in stable storage) and informs the participants.

2 f) Qual o interesse da estratégia de *semijoin*? I.e. para que serve?

Para calcular a junção de 2 tabelas r1 num site S1 e r2 num site S2, basta primeiramente enviar para S2 a projeção de r1 nas colunas de junção (ao invés de todo o r1), e aí calcular o semijoin de r2 com r1 (que dá os tuplos de r2 que contribuem para a junção). Depois é só enviar essa parte (ao invés de todo o r2) para S1 e lá calcular a junção final, para aí apresentar o resultado. Assim, envia-se um menor volume de dados pela rede, possibilitando melhor desempenho.

2 g) Considere as *Queries LDAP* e como são especificadas.

I. De que formas se pode especificar uma query LDAP (i.e. como é que pode ser passada ao sistema)?
Por URL ou com uma API.

II. O que deve e o que pode constar numa tal *query*?
A base (um nó no diretório de informação onde começar a pesquisa), composta por servidor e *Distinguished Name*; uma condição de pesquisa, e o seu escopo. Pode-se ainda indicar os atributos a devolver (e, adicionalmente, também indicar limites nos resultados e recursos, e especificar se os *aliases* devem ser automaticamente des-referenciados).

III. Que operações de álgebra relacional permitem tais *queries*?
Apenas seleção e projeção.