TÉCNICO LISBOA

Bases de Dados 2015/2016
Controlo de Concorrência

Helena Galhardas

Sumário

- Serialização e Protocolos de *Locking*
  - 2PL – *Two Phase Locking*
- Tratamento de *Deadlocks*
- Níveis de Isolamento em SQL
- Aquisição Automática de Locks

TÉCNICO LISBOA

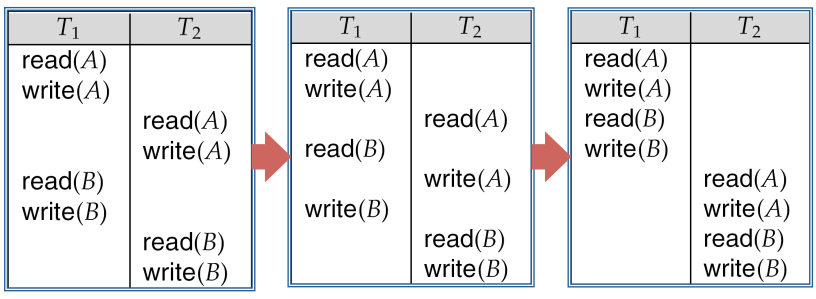
Referências

- Raghu Ramakrishnan, Database Management Systems, 3ª edição: Cap. 17

Conflitos (recap)

Existe **conflito** se as instruções de T_1 e T_2 operarem sobre o mesmo objecto e, pelo menos, uma delas for uma escrita

– se não houver conflito, instruções podem ser trocadas.

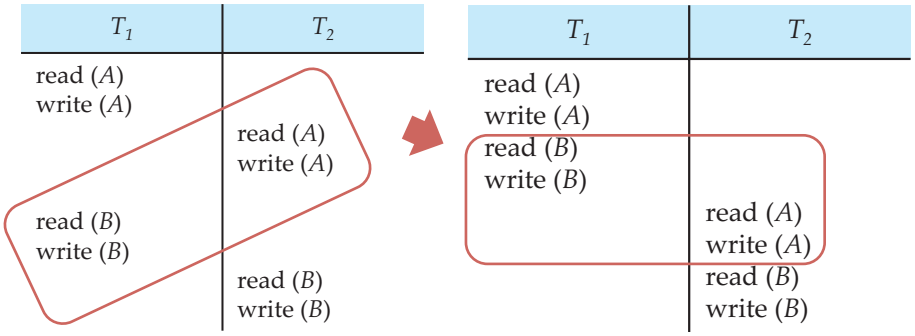


Serialização em Conflitos

- Dois escalonamentos são **equivalentes em conflitos (*conflict-equivalent*)** se
  - envolverem o mesmo conjunto de acções das mesmas transacções
  - ordenarem cada par de acções em conflito das duas transacções da mesma maneira
- Se um escalonamento S pode ser transformado num escalonamento S' mediante uma série de trocas de ordem de execução de instruções, dizemos que S e S' são **equivalentes em conflitos**
- Um escalonamento S é **serializável em conflitos (*conflict-serializable*)** se é equivalente em conflitos a um escalonamento série
- Qualquer escalonamento serializável em conflitos também é serializável

TÉCNICO LISBOA

Ex. Escalonamento Serializável em Conflitos



TÉCNICO LISBOA

Ex. Escalonamento não Serializável em Conflitos

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

Não podemos trocar instruções de forma a obter um dos escalonamentos série < T3, T4 > ou < T4, T3 >

TÉCNICO LISBOA

Serializável e serializável em conflitos

Segundo os critérios, este escalonamento não é serializável em conflitos, mas é serializável

– mantém A+B

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

TÉCNICO LISBOA

Execução Concorrente: Anomalias (recap)

- Três tipos fundamentais
  - *Dirty Read*
  - *Unrepeatable Read*
  - *Lost Update*

TÉCNICO LISBOA

Como Evitar Anomalias?

- Intercalar operações de ***Lock/Unlock*** nas acções que compõem a transacção
  - Transacção: sequência de operações de 6 tipos básicos
    - *Read, Write, Commit, Rollback*
    - ***Lock, Unlock***
  - O que quase todos os sistemas reais usam

- Outros mecanismos de controlo de concorrência:
  - Etiquetagem (*timestamps*)
  - Baseados em Validações (Optimistas)

TÉCNICO LISBOA

Locks

- Existem 2 modos de bloquear o acesso a um objecto
  - **modo partilhado** (**S - *Shared***)
    - o objecto pode ser lido, mas não escrito
    - várias transacções podem possuir este *lock* num dado momento
  - **modo exclusivo** (**X - *eXclusive***)
    - o objecto pode ser lido e escrito
    - apenas uma transacção pode possuir este *lock* num dado momento

TÉCNICO LISBOA

Transações Bem-Formadas

- Quando as transações são **bem-formadas**
  - Cada uma das ações de *read, write, unlock* **é precedida** por uma acção *lock*
  - Todas as ações de *lock* **seguidas** de um *unlock*

TÉCNICO LISBOA

Locks na Resolução de Anomalias

T1: Lock-X Saldo	
T1: Read Saldo	**T2: Lock-X Saldo**
T1: Saldo := Saldo+100	**espera**
T1: Write Saldo	**espera**
T1: Unlock Saldo	**espera**
	T2: Read Saldo
	T2: Saldo := Saldo+200
	T2: Write Saldo
	Unlock Saldo

Compatibilidade entre *locks*

- Uma transacção que pede o *lock* de um objecto
  - Só continua quando o *lock* é finalmente concedido
  - Só recebe o *lock* quando não houver ou forem libertados todos os *locks* incompatíveis
- Matriz de compatibilidade entre *locks*

	S	X
S	true	false
X	false	false

S = partilhado (*shared*)
X = exclusivo

Exemplo com Transacções Concorrentes

T_1: **read**(B)
$B := B - 50$
write(B)
read(A)
$A := A + 50$
write(A)

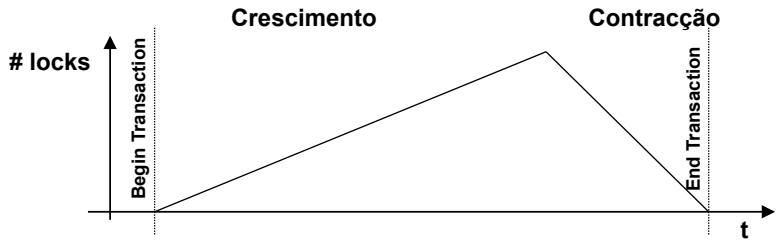
T_2: **read**(A)
read(B)
display(A+B)

- T_2 pode dar um resultado incoerente quando executada em paralelo com T_1
- *Locking* **pode não ser suficiente** para garantir serialização - se A e B são actualizados por T1 entre a leitura de A e B por T2, então a soma mostrada está errada!

TÉCNICO LISBOA

Transacções Bi-faseadas

- **Protocolo de lock em 2 fases (2PL)**
  - **Todas as acções de *lock* antecedem todas as acções de *unlock***
  - **Fase de Crescimento/aquisição** (*growing phase)*
    - Apenas se adquirem *locks*
  - **Fase de Contracção/libertação** (*shrinking phase*)
    - Apenas se libertam *locks*



TÉCNICO LISBOA

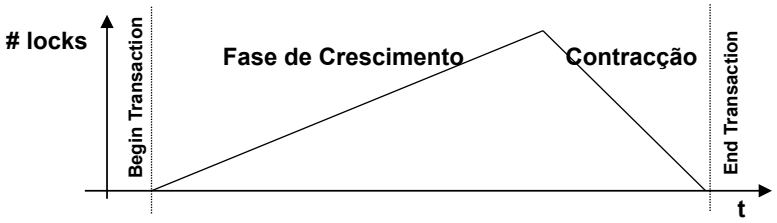
Protocolo de *lock* em duas fases (2PL)

- Cada transacção tem que adquirir um *lock* S (partilhado) sobre um objecto antes de o ler, e um *lock* X (exclusivo) antes de o escrever
- Se uma transacção detém um *lock* X sobre um objecto, nenhuma outra transacção pode obter um *lock* (S ou X) sobre esse objecto
- **Uma transacção não pode pedir *locks* adicionais depois de ter libertado algum dos seus *locks***
- Existem 4 variantes: 2PL, S2PL, R2PL, C2PL

TÉCNICO LISBOA

Protocolo de Lock em 2 fases (**2PL**)

- Todas as ações de LOCK antecedem todas as ações de UNLOCK
- Após o primeiro unlock, cada transacção não pode pedir mais *locks*



- O 2PL garante a serialização, e corresponde à ordem das transações vista pelo momento em que fizerem o último lock
- O 2PL não garante escalonamentos recuperáveis

TÉCNICO LISBOA 17

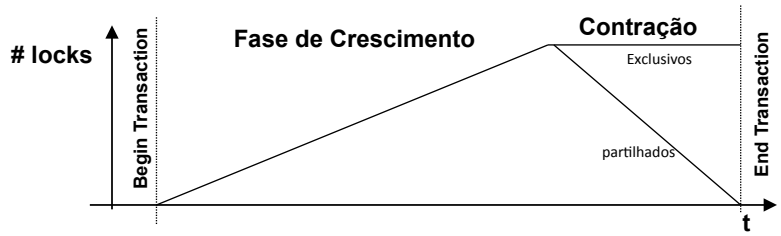
Exemplo com *2PL*

T_1:	T_2:
lock-X(*B*)	**lock-S**(*A*)
read(*B*)	**read**(*A*)
B := *B* – *50*	**lock-S**(*B*)
write(*B*)	**read**(*B*)
lock-X(*A*)	**unlock**(*B*)
read(*A*)	**unlock**(*A*)
A := *A* + *50*	**display**(*A*+*B*)
unlock(*B*)	
write(*A*)	
unlock(*A*)	

TÉCNICO LISBOA

Protocolo Strict 2PL (**S2PL**)

- Todas as ações de LOCK antecedem todas as ações de UNLOCK
- Todos os *unlocks* exclusivos só podem acontecer após o abort or commit

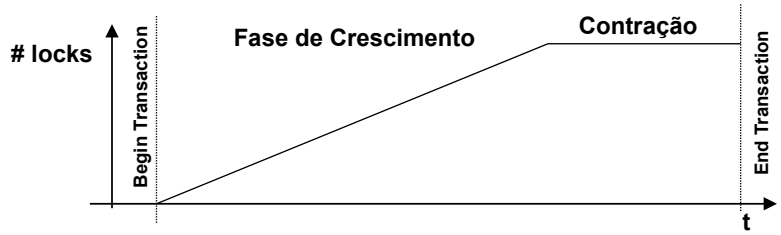


- O S2PL garante a serialização, e corresponde à ordem com que fizeram o último lock
- O S2PL garante a recuperação e sem *rollbacks* em cascata

TÉCNICO LISBOA 19

Protocolo Rigorous 2PL **(R2PL)**

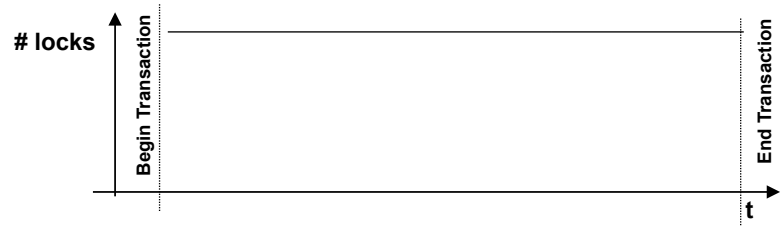
- Todas as ações de LOCK antecedem todas as ações de UNLOCK
- Todos os *unlocks* só podem acontecer após o *rollback* or *commit*.



- O R2PL garante a serialização, e corresponde à ordem das transações pela ordem que fazem o commit.
- O R2PL garante a recuperação e sem *rollbacks* em cascata

Protocolo
Conservative 2PL (C2PL)

- Todos os *locks* são pedidos antes do *Begin Transaction*
- Todos os unlocks só podem acontecer após o rollback ou *commit*.



- O C2PL garante a serialização, e corresponde à ordem das transações pela ordem que fazem o *commit*.
- O C2PL garante a recuperação e sem *rollbacks* em cascata
- O C2PL garante que não existem *deadlocks* (mas pode existir *starvation*)

Discussão

- Porquê transacções bem-formadas?
  - Evitam anomalias
- Porquê transacções bi-faseadas?
  - Garantem isolamento (pode ser demonstrado)

2PL vs Strict 2PL

- 2PL assegura que os escalonamentos são serializáveis em conflitos
  - Uma ordem em série equivalente é dada pela ordem pela qual as transacções entram na fase de decrescimento
- Em adição, Strict 2PL assegura que os escalonamentos são recuperáveis, e não existem *rollbacks* em cadeia

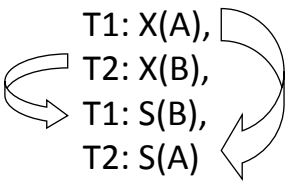
Sumário

- Serializabilidade e Protocolos de *Locking*
  - 2PL – *Two Phase Locking*
- ➢ **Tratamento de *Deadlocks***
- Níveis de Isolamento em SQL
- Aquisição Automática de Locks

TÉCNICO LISBOA

Escalonamento e *Deadlocks*

- Ao escalonar a execução, há que prever o tratamento de ***deadlocks***
  - um grupo de transacções fica bloqueado por cada uma se encontrar à espera da libertação de *locks* detidos por outras transacções do grupo.



TÉCNICO LISBOA

Situação de *deadlock* – exemplo

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

T_3 bloqueia porque T_4 tem um *lock* incompatível

T_4 bloqueia porque T_3 tem um *lock* incompatível

- T_3 ou T_4 terá que sofrer *rollback* para resolver este impasse (***deadlock***)

Técnicas para anular *Deadlocks*

- **Prevenção**
  - Ordenar transacções por antiguidade (*Timestamping*)
- **Detecção**
  - Estabelecer *timeouts*
    - Abortar transacções que não avançam há algum tempo
    - Problemático estabelecer o valor do *timeout*
  - Criar Grafo de Espera (*wait-for*)
    - Quebrar o ciclo, usualmente por *rollback* da transacção mais recente

Prevenção de *Deadlock*

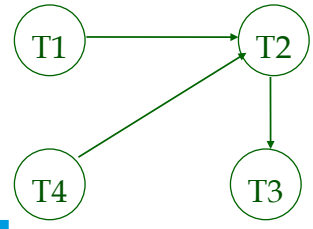
- Atribuir prioridades às transacções baseadas em ***timestamps*** (atribuídos às transacções quando se iniciam)
  - Maior prioridade implica transacção mais velha
- Se Ti pretende um *lock* que entra em conflito com um *lock que* Tj possui, duas políticas possíveis:
  - ***Wait-Die***: Se Ti tem prioridade mais alta, Ti espera por Tj; senão aborta
    - Transacção mais velha pode ter que esperar que mais nova liberte *lock*; transacções mais novas nunca esperam por mais velhas – são abortadas
    - Transacção pode ser abortada várias vezes antes de aquirir o *lock*
  - ***Wound-wait***: Se Ti tem prioridade mais alta, Tj aborta; senão Ti espera
    - Transacção mais velha força o *rollback* da mais nova em vez de esperar por ela; transacções mais novas podem esperar por mais velhas
    - Pode envolver menos *rollbacks que wait-die*
- Se uma transacção recomeça, ter a certeza que fica com o seu *timestamp* original

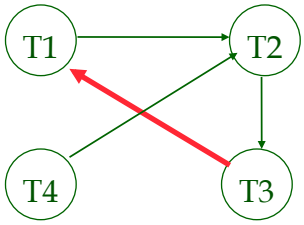
Detecção de *Deadlock*

- Criar um grafo ***waits-for***:
  - Nós são transacções activas
  - Existe um arco de Ti para Tj se Ti espera que Tj liberte um *lock*
- Periodicamente verificar se existem ciclos no grafo *waits-for*
  - Se existirem ciclos, então abortar uma das transacções envolvidas no ciclo e libertar os seus *locks*

Exemplo

T1: S(A), R(A), S(B)
T2: X(B),W(B) X(C)
T3: S(C), R(C) X(A)
T4: X(B)





TÉCNICO LISBOA

Após Detecção: Escolha da Vítima

Várias Alternativas:

- A Xact que está a bloquear
- A Xact mais nova
- A Xact que usou menos recursos
- A Xact que usou menos *locks*
- A Xact que foi re-inicializada menos vezes
- A Xact que elimina mais ciclos (se houver + que 1)

TÉCNICO LISBOA

Prevenção vs. Detecção

Prevenção

- Pode levar ao cancelamento de muitas transações

Detecção

- *Deadlocks* podem impedir acesso aos dados por algum tempo
- Aumentar a frequência da detecção consome tempo

Detecção é o método preferido porque *deadlocks* são raros e envolvem poucas transacções, tipicamente

TÉCNICO LISBOA

Sumário

- Serialização e Protocolos de *Locking*
  - 2PL – *Two Phase Locking*
- Tratamento de *Deadlocks*
- ➢ **Níveis de isolamento em SQL**
- Aquisição Automática de Locks

TÉCNICO LISBOA

Concorrência vs. Isolamento

- SQL Standard
  - assume isolamento completo entre transacções
- SGBD comerciais
  - permitem relaxamento dos mecanismos de prevenção de anomalias (de forma a aumentar a concorrência)

Níveis fracos de coerência

- Algumas aplicações podem viver com níveis mais fracos de coerência de dados, permitindo escalonamentos que não são serializáveis
  - Ex: transacção só de leitura que pretende obter um saldo total aproximado de todas as contas
  - Ex: estatísticas de base de dados calculadas para optimização de interrogações podem ser aproximadas
  - Estas transacções não necessitam ser serializáveis com respeito a outras transacções
- **Compromisso coerência vs desempenho**

Níveis de isolamento em SQL (1)

Nível de isolamento Anomalias de dados	*dirty reads*	*non-repeatable reads*	*phantom reads*
SERIALIZABLE	não	não	não
REPEATABLE READ	não	não	possível
READ COMMITTED	não	possível	possível
READ UNCOMMITTED	possível	possível	possível

- ***phantom read****:* fazendo a mesma consulta duas vezes, o número de registos pode ser diferente, se entretanto outra transacção inseriu um tuplo e fez *commit*
- ***Non-repeatable read****:* fazendo a mesma consulta duas vezes, cada registo pode conter dados diferentes, se entretanto outra transacção modifica o registo e faz *commit*
- ***dirty read****:* é possível ler dados alterados por outras transacções activas que ainda nem sequer fizeram *commit*.

TÉCNICO LISBOA

Níveis de isolamento em SQL (2)

- ***Serializable***: garante que:
  - transacção T lê apenas as modificações efectuadas por transacções *committed*
  - nenhum valor lido ou escrito por T é modificado por outra transacção até T se ter completado
  - se T lê valores baseados numa condição de procura, este conjunto não é modificado por outras transacções até T estar completo
  - Implementa *Strict 2PL*
- ***Repeatable read****:* assegura que:
  - T lê apenas modificações efectuadas por transacções *committed*
  - nenhum valor lido ou escrito por T é mudado por outra transacção antes de T terminar.
  - *Strict 2PL* mas só faz *lock* de objectos individuais e não conjuntos de objectos (não faz *index locking*) – ver mais tarde

TÉCNICO LISBOA

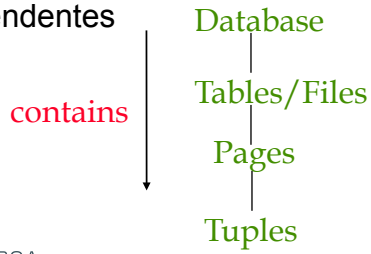
Níveis de isolamento em SQL (3)

- ***Read committed****:* assegura que:
  - T lê apenas as modificações efectuadas por transacções *committed*
  - Nenhum valor escrito por T é mudado por outra transacção até T ter terminado, mas pode ser lido
  - Obtém *locks* exclusivos antes de escrever objectos e mantém esses locks até ao fim
  - Obtém *locks* partilhados antes de ler mas liberta-os imediatamente
- ***Read uncommitted***:
  - Pode ler modificações efectuadas a um objecto por uma transacção activa
  - Não obtém locks partilhados antes de ler um objecto

TÉCNICO LISBOA

Locks com múltiplas granularidades

- Decidir a que granularidade fazer o *lock* (tuplos vs. páginas vs. tabelas).
- Hierarquia de granularidades de objectos (árvore de objectos)
  - *Lock* sobre um nó tranca esse nó e, implicitamente, todos os seus descendentes



TÉCNICO LISBOA

Fantasmas (*Phantom problem*)

```
T1: SELECT MIN(S.age)
    FROM Sailors
    WHERE S.rating = 8
T2: UPDATE Sailors
    SET age = 20
    WHERE S.name = 'Joe' AND S.rating = 8
```

- T1 adquire *lock* partilhado sobre tuplos com rating = 8
- T2 adquire *lock* exclusivo sobre o tuplo a alterar

Locks com **diferentes granularidades** aumentam o grau de concorrência permitido

TÉCNICO LISBOA

Fantasmas (*Phantom problem*)

```
T3: INSERT INTO Sailors(name, age,
rating) VALUES ('Mary', 19, 8)
```

- T1 adquiriu *lock* partilhado sobre linhas com rating = 8
- T3 pode inserir o novo tuplo e adquirir um *lock* exclusivo sobre esse tuplo.
  - Se este tuplo tiver um valor de idade menor do que todos os outros, T1 retorna um resultado que depende se se executa antes ou depois de T3 – **Problema do fantasma!**

TÉCNICO LISBOA

Instrução SQL

- `Set transaction isolation level serializable read only`
- Por omissão, é *serializable* e *read write*

Sumário

- Serializabilidade e Protocolos de *Locking*
  - 2PL – *Two Phase Locking*
- Gestão de *locks*
- Tratamento de *Deadlocks*
- ➢ **Aquisição Automática de Locks**

Aquisição Automática de Locks

- Cada T_i produz os pedidos de *read/write* sem invocação explícita das operações de *lock/unlock*
  - O código para os processar é gerado automaticamente
- No final, após *commit* ou *abort*, todos os *locks* são libertados

TÉCNICO LISBOA

Aquisição Automática de *Locks* (cont.)

read(*D*) processada como:

```
if Ti has a lock on D
then
    read(D)
else begin
    if necessary wait until no other
     transaction has a lock-X on D

    grant Ti a lock-S on D;

    read(D)
end
```

write(*D*) processada como:

```
if Ti has a lock-X on D
then
    write(D)
else begin
   if necessary wait until no other trans.
    has any lock on D,
   if Ti has a lock-S on D
   then upgrade lock on D to lock-X
   else
        grant Ti a lock-X on D
        write(D)
   end
```

Sumário

- Controlo de concorrência
- Próxima aula: Gestão de recuperação

Bases de dados dinâmicas

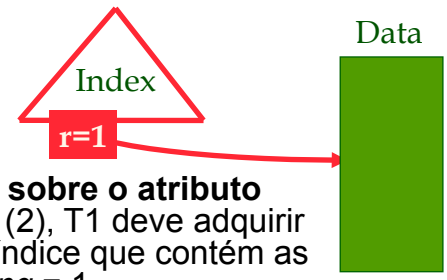
- Se relaxarmos a assumpção que a BD é um conjunto fixo de objectos, **mesmo o protocolo Strict 2PL não assegura serializabilidade**
- Exemplo: T1 percorre a tabela Sailors para encontrar o marinheiro mais velho para cada um dos ratings, = 1 e = 2
  - T1 adquire lock sobre todas as páginas contendo registos de marinheiros *rating* = 1, e encontra o marinheiro **mais velho** (e.g., *age* = 71)
    - Assume-se que se usam *locks* ao nível da página
  - Depois, T2 insere um **novo** marinheiro: *rating* = 1, *age* = 96
    - Pode ser inserido numa página que não contém outros marinheiros com rating = 1 => lock X sobre esta página não entra em conflito com nenhum dos locks de T1
  - T2 também **apaga** o marinheiro mais velho com *rating* = *2* (que tem, por exemplo, *age* = 80), e faz *commit* (liberta os seus locks)
    - T2 adquire um lock sobre a página que contém o marinheiro mais velho com rating =2
  - T1 agora adquire lock sobre todas as páginas contendo marinheiros com *rating* = 2, e encontra o **mais velho** e.g., *age* = 63).
- Resultado da execução concorrente é 71 e 63 , mas se T1 se tivesse executado antes de T2, o resultado era 71 e 80; e se T2 antes de T1 era: 96 e 63

O problema

- A execução concorrente de T1 e T2 **não é equivalente a nenhuma em série** das duas transacções, porque:
  - T1 implicitamente assume que adquiriu um *lock* sobre o conjunto de todos os registos de marinheiros com *rating* = 1
    - Mas esta assumpção só é válida se nenhum registo de marinheiro fôr adicionado enquanto T1 se executa; senão, existe um registo **fantasma** adicionado por T2!
    - T1 não adquiriu um *lock* sobre todos os registos necessários
    - É necessário um mecanismo para garantir que isso aconteça: ***index and predicate locking***
- Exemplo em que serializabilidade em conflitos não garante serializabilidade, porque o conjunto de objectos da BD não é fixo

TÉCNICO LISBOA

Index Locking



- Se existir um **índice denso sobre o atributo *rating*** usando a Alternativa (2), T1 deve adquirir um *lock* sobre a página do índice que contém as entradas de dados com *rating* = 1.
  - Se não existirem registos nenhuns com *rating* = 1, T1 tem que adquirir um *lock* sobre a página do índice onde essa entrada de dados estaria se existisse
- Se **não existir este índice**, então T1 tem que adquirir um *lock* sobre todas as páginas e adquirir um *lock* sobre o ficheiro ou tabela que as contém prevenindo assim que novas páginas sejam adicionadas e consequentemente que novos registos com *rating* = 1 sejam adicionados

TÉCNICO LISBOA

Predicate Locking

- Conceder um ***lock* sobre todos os registos que satisfaçam um predicado lógico**, por exemplo rating = 1
  - *Index locking* é um caso especial de *predicate locking* no qual um índice suporta uma implementação eficiente de the predicate lock.
- Em geral, *predicate locking* implica um elevado custo de implementação e não é suportado
  - testar um predicado antes de adquirir cada lock representaria a execução de um número muito elevado de instruções
- Solução: ***Locks* com múltiplas granularidades**!

TÉCNICO LISBOA

Solução: Novos modos de *lock*

- Permite que as transacções façam *lock* em cada nível, mas com um protocolo especial usando novos ***locks* de intenção S e X**
- Antes de fazer *lock* a um objecto, transacção tem que ter *locks* de intenção sobre todos os seus antecessores
- Para *unlock*, ir do específico para o geral.
- **Modo SIX**: Como S & IX ao mesmo tempo

	--	IS	IX	S	X
--	√	√	√	√	√
IS	√	√	√	√	
IX	√	√	√		
S	√	√		√	
X	√				

TÉCNICO LISBOA

Protocolo de *locking* com múltiplas granularidades

1. Cada transacção começa pela raiz da hierarquia
2. Para obter um *lock* S ou IS sobre um nó, tem que possuir *lock* IS ou IX sobre o nó pai
3. Para obter *lock* X, IX, ou SIX sobre um nó, tem que possuir um *lock* IX ou SIX sobre um nó pai
4. Libertar nós de baixo para cima