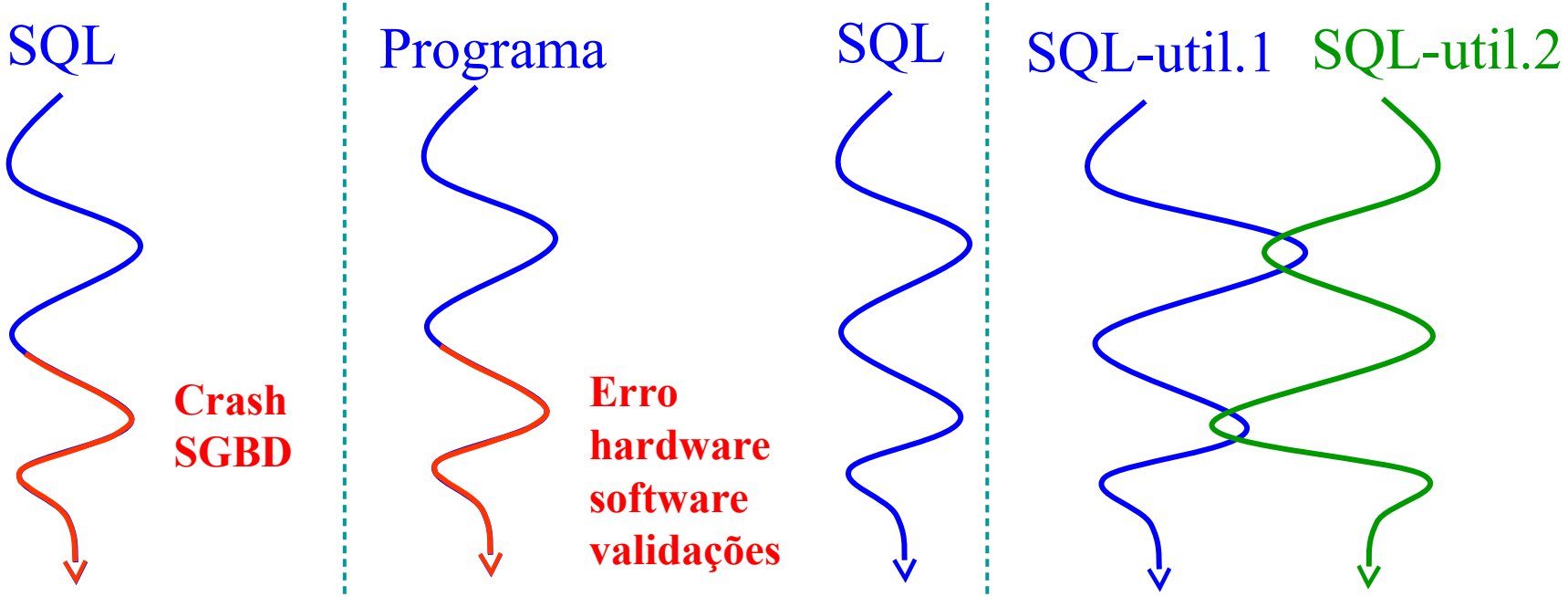
Processamento Transaccional

(1ª parte)

Transacções

Transacções porquê?



Caso 1
Recuperação

Caso 2
anular processamento anterior

Caso 3
Controlar a interferência

Transacções

Acção atómica: aquela que, quando executada num determinado nível de abstracção, ou é executada completamente com sucesso, (produzindo todos os seus efeitos), ou, então, não produz quaisquer efeitos directos ou laterais.

Uma acção pode ser atómica num nível de abstracção mais elevado, mas não num nível de abstracção mais baixo.

Por exemplo, uma instrução SQL é atómica ao nível do SQL, mas quando executada por instruções de um CPU é realizada por um conjunto de instruções cuja execução parcial pode, a este nível, produzir efeitos.

Terá, então, de haver mecanismos que permitam garantir a atomicidade no nível de abstracção adequado.

As transacções são uma forma de os programadores poderem definir, com base em conjuntos de acções (com determinadas características) num nível de abstracção, acções atómicas num nível de abstracção superior.

Transacções – tipos de acções

Não protegidas: aquelas cujo efeito não necessita de ser anulado. Por exemplo, uma operação sobre um ficheiro temporário.

Protegidas: aquelas cujo efeito pode e tem de ser anulado ou reposto se a transacção falhar ou se o valor de um grânulo tiver de ser reposto. Por exemplo, a escrita de um valor num registo.

Reais: aquelas acções, tipicamente sobre objectos físicos, cujo efeito não pode, em geral, ser anulado. Por exemplo, o lançamento de um míssil.

Transacções – objectivos e propriedades

Principais objectivos:

- **Fornecer mecanismos de recuperação em caso de falhas do sistema**
- **Facilitar o tratamento de erros ao nível das aplicações (programação)**
- **Fornecer mecanismos que permitam controlar de forma eficiente as interferências entre aplicações que concorrem no acesso aos mesmos recursos**

Propriedades ACID:

- **Atomicidade (Atomicity)**

  Uma transacção é indivisível no seu processamento

- **Consistência (Consistency preservation)**

  Uma transacção conduz a BD de um estado consistente para outro estado consistente

- **Isolamento (Isolation)**

  As transacções concorrentes não devem interferir umas com as outras durante a sua execução

- **Durabilidade (Durability)**

  O resultado de uma transacção válida deve ser tornado persistente (mesmo na presença de falhas, após commit)

Transacções – escalonamentos

Um escalonamento (história) de um conjunto de transacções {T1, ...,Tn} é uma ordenação S das operações em cada um dos Ti tal que todas as acções de cada Ti aparecem em S pela mesma ordem em que ocorrem em Ti.

Exemplos:

Sejam:

T1 = <r(x1),w(x2),r(x3)>
T2 = <w(x1),r(x2),w(x4)>

São escalonamentos:

S1 = <r(t1,x1),w(t1,x2),r(t1,x3),w(t2,x1),r(t2,x2),w(t2,x4)>
S1 = <w(t2,x1),r(t2,x2),w(t2,x4),r(t1,x1),w(t1,x2),r(t1,x3) >
S3 = <r(t1,x1),w(t2,x1),w(t1,x2),r(t2,x2),r(t1,x3),w(t2,x4)>

Não é um escalonamento:

<w(t1,x2),r(t1,x1) ,w(t2,x4),r(t1,x3),w(t2,x1),r(t2,x2)>

Duas operações num escalonamento S conflituam se se verificarem, simultâneamente, as seguintes condições:

1. **As operações pertencem a transacções diferentes**
2. **Ambas as operações acedem ao mesmo item de dados**
3. **Pelo menos uma das operações é uma operação de escrita**

Transacções – escalonamentos

Um escalonamento S diz-se "cascadeless" (não exibe o efeito cascading abort ou cascading rolback) se nenhuma das suas transacções ler um item escrito por outra transacção ainda não terminada.

Exemplos:

Não é "cascadeless":

S1 = <r(t1,x1),w(t1,x1),r(t2,x1),r(t1,x2),w(t2,x1),w(t1,x2),a(t1),a(t2)>

Quando t1 aborta, t2 tem de abortar também (efeito cascata)

É cascadeless:

S2 = <r(t1,x1),w(t1,x1),r(t1,x2),w(t2,x1),w(t1,x2),a(t1),r(t2,x1),a(t2)>

Transacções – escalonamentos

Um escalonamento S é recuperável se não existir nenhuma transacção que faça commit tendo lido um item depois de ele ter sido escrito por outra transacção ainda não terminada com commit. (nestes escalonamentos, nenhuma transacção terminada com sucesso necessita de ser anulada quando outras transacções falham – em nenhuma circunstância, por exemplo, em caso de crash).

Não ser recuperável => não ser "cascadeless"

Exemplos:

Não é recuperável:

Sn = <r(t1,x1),w(t1,x1),r(t2,x1),r(t1,x2),w(t2,x1),c(t2),w(t1,x3),c(t1)>

São recuperáveis:

Sr1 = <r(t1,x1),r(t2,x1),w(t1,x1),r(t1,x2),w(t2,x1),c(t2),w(t1,x2),c(t1)>

Sr2= <r(t1,x1),w(t1,x1),r(t2,x1),r(t1,x2),w(t2,x1),w(t1,x2),a(t1),a(t2)>

(a últimas não é "cascadeless"– ver slide anterior)

Transacções – escalonamentos

Um escalonamento S diz-se estrito se nenhuma das suas transacções ler nem escrever um item escrito por outra transacção ainda não terminada.

Exemplos:

Não é estrito:

S1 = <r(t1,x1),w(t1,x1),r(t2,x1),r(t1,x2),w(t2,x1),c(t1),c(t2)>

É estrito:

S2 = <r(t1,x1),w(t1,x1),r(t1,x2),c(t1),r(t2,x1),w(t2,x1),c(t2)>

Transacções – escalonamentos

Um escalonamento S diz-se "série" se para toda a sua transacção T as operações de T são executadas consecutivamente, sem interposição de operações de outras transacções. Limitam a concorrência, fornecem os resultados possíveis sem concorrência. Para N transacções, existem N! escalonamentos série possíveis.

Exemplos:

Não é série:

S1 = <r(t1,x1),w(t1,x1),r(t2,x1),r(t1,x2),w(t2,x1),c(t1),c(t2)>

São série:

S2 = <r(t1,x1),w(t1,x1), r(t1,x2), c(t1), r(t2,x1), w(t2,x1), c(t2)>

S2 = < r(t2,x1), w(t2,x1), c(t2), r(t1,x1),w(t1,x1), r(t1,x2), c(t1)>

Do pronto de vista de processamento transacional são considerados correctos os resultados de qualquer dos escalonamentos série, dado que eles correspondem a execuções sem interferência, dependendo a ordem porque as transações são executadas do carácter assíncrono na sua recepção do SGBD

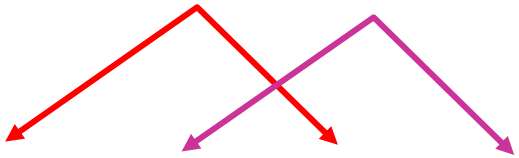
Transacções – escalonamentos

Dois escalonamentos são equivalentes do ponto de vista de conflito se a ordem de quaisquer duas operações conflituosas for a mesma nos dois escalonamentos.

Exemplos

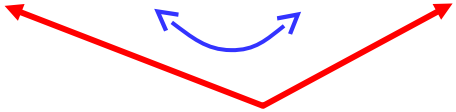
Sejam

S1 = <r(t1,x1),r(t2,x2),w(t2,x1),w(t1,x2),c(t1),c(t2)>



S2 = <r(t1,x1),r(t2,x2),w(t1,x2),w(t2,x1),c(t1),c(t2)>

S3 = <r(t1,x1), w(t1,x2), r(t2,x2), w(t2,x1),c(t1),c(t2)>



S1 e S2 são equivalentes do ponto de vista de conflito, mas nenhum deles é equivalente a S3.

Transacções – escalonamentos

Um escalonamento S diz-se "serializável (do ponto de vista de conflito)" se for equivalente do ponto de conflito a um dos escalonamentos "série" possíveis com as transacções de S.

Quando os escalonamentos serializáveis terminam com sucesso de toda as suas transações, produzem os mesmos resultados que os escalonamentos séria a que são equivelentes do ponto de vista de conflito

Exemplos:

Sejam t1 = <r(x1),w(x1), r(x2), c> e t2 = <r(x1), w(x1), c>

Os dois escalonamentos série possíveis são:

S1 = <r(t1,x1),w(t1,x1),r(t1,x2), c(t1), r(t2,x1), w(t2,x1), c(t2)>
S2 = < r(t2,x1), w(t2,x1), c(t2), r(t1,x1),w(t1,x1), r(t1,x2), c(t1)>

São serializáveis:

S3 = <r(t1,x1),w(t1,x1),r(t2,x1), r(t1,x2),w(t2,x1), c(t1),c(t2)>
S4 = <r(t2,x1), w(t2,x1),r(t1,x1),w(t1,x1),c(t2),r(t1,x2),c(t1)>

Não é serializável:

S5 = < r(t2,x1), r(t1,x1),w(t2,x1),w(t1,x1),c(t2),r(t1,x2),c(t1)>

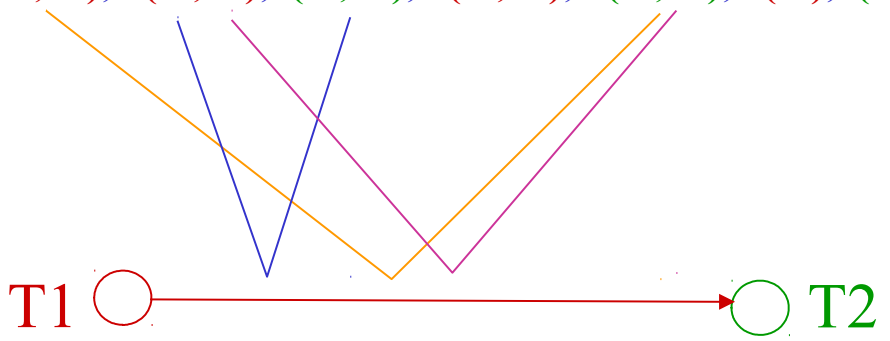
Transacções – grafos de precedências e escalonamentos serializáveis

Desenhar um vértice por cada transacção do escalonamneto.

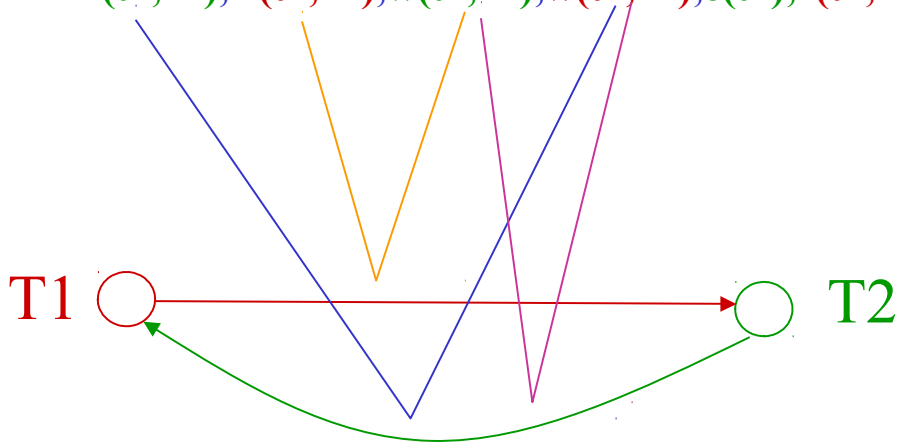
Por cada par conflituoso a1(Ti,x), a2(Tj,x) tal que a1 precede a2 desenhar um arco de Ti para Tj.

Se existirem ciclos, o escalonamento não é serializável

S3 = <r(t1,x1),w(t1,x1),r(t2,x1), r(t1,x2),w(t2,x1), c(t1),c(t2)>



S5 = < r(t2,x1), r(t1,x1),w(t2,x1),w(t1,x1),c(t2),r(t1,x2),c(t1)>

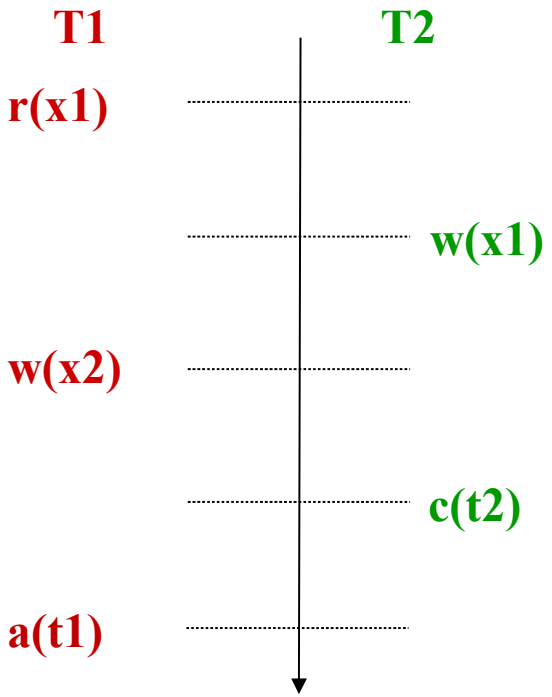


Transacções – escalonamentos

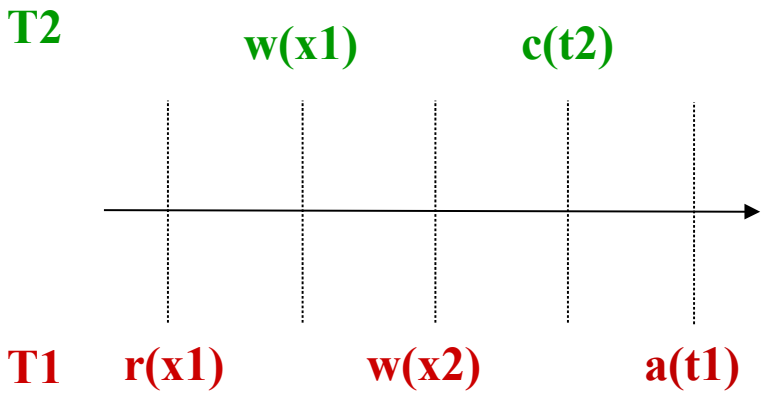
Os escalonamentos podem ser mostrados como linhas de tempo, nas quais as ações são colocadas nos tempos correspondente às posições que ocupam no escalonamento.

Exemplo:

S = <r(t1,x1), w(t2,x1), w(t1,x2), c(t2), a(t1)>



ou



Transacções – escalonamentos (exercício 1)

Considere as seguintes transações:

T1 = r(x1), w(x2), c
T2 = w(x1), w(x2), c
T3 = r(x1), r(x3), c
T4 = w(x2),w(x1),c
T5 = r(x4),w(x4),c

1. **Indique um escalonamento que envolva T1 e outra transação e seja não recuperável.**
2. **Indique um escalonamento que envolva T1 e outra transação e seja recuperável, mas não "cascadeless"**
3. **Indique um escalonamento que envolva T1 e e T3 e que seja não recuperável.**
4. **Indique quais são os escalonamentos serializáveis que envolvam T2 e T3.**
5. **Indique quais são os escalonamentos serializáveis que envolvam T4 e T5.**
6. **Indique um escalonamento que seja "cascadeless", mas não estrito.**

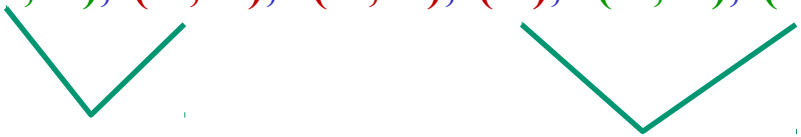
Nota: se não existirem escalonamentos com as características indicadas, justifique porquê.

Transacções – escalonamentos (exercício 1)

T1 = r(x1), w(x2), c
T2 = w(x1), w(x2), c
T3 = r(x1), r(x3), c
T4 = w(x2),w(x1),c
T5 = r(x4),w(x4),c

1. **Indique um escalonamento que envolva T1 e outra transação e seja não recuperável.**

S = w(t2,x1),r(t1,x1),w(t1,x2),c(t1),w(t2,x2),c(t2)



Transacções – escalonamentos (exercício 1)

T1 = r(x1), w(x2), c
T2 = w(x1), w(x2), c

2. **Indique um escalonamento que envolva T1 e outra transação e seja recuperável, mas não "cascadeless"**

S = w(t2,x1),r(t1,x1),w(t1,x2),w(t2,x2),c(t2),c(t1)

Transacções – escalonamentos (exercício 1)

T1 = r(x1), w(x2), c
T3 = r(x1), r(x3), c

3. **Indique um escalonamento que envolva T1 e e T3 e que seja não recuperável.**

   Não existe tal escalonamento porque T1 e T2 não possuem ações conflituantes, todos os escalonamentos que apenas envolvem estas duas transações são serializáveis

Transacções – escalonamentos (exercício 1)

T2 = w(x1), w(x2), c
T3 = r(x1), r(x3), c

4. **Indique quais são os escalonamentos serializáveis que envolvam T2 e T3.**
   Dado que apenas conflituam w(t1,x1) com r(t1,x1) teremos os seguintes escalonamentos serializáveis:

   Todos os que começam por w(t2,x1):
   S1 = w(t2,x1), w(t2,x2),c(t2), r(t3,x1),r(t3,x3), c(t3)
   S2 = w(t2,x1), r(t3,x1), w(t2,x2),c(t2), r(t3,x3), c(t3)
   …

   Todos os que começam por r(t3,x1):
   S3 = r(t3,x1),r(t3,x3),c(t3), w(t2,x1), w(t2,x2),c(t2)
   S4 = r(t3,x1),w(t2,x1),r(t3,x3),c(t3), w(t2,x2),c(t2)
   …

Transacções – escalonamentos (exercício 1)

T4 = w(x2),w(x1),c
T5 = r(x4),w(x4),c

5. Indique quais são os escalonamentos serializáveis que envolvam T4 e T5.

Como entre T4 e T5 não existem ações que conflituem, todos os escalonamentos são serializáveis.

Em geral, para N ações de T4 e K ações de T5 que não conflituem, teremos

$$C_K^{N+K} = C_N^{N+K}$$ **escalonamentos serializáveis diferentes**

Exemplo: N =2, K = 2
X1X2X3X4 (4 posições). Arranjando as formas de dispor N (ou K) das ações nestas posições, as restantes K (ou N) ficam determinadas.
Neste caso, existem 4!/(2!x2!) = 6 escalonamentos diferentes.

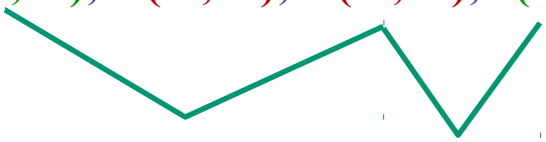
Transacções – escalonamentos (exercício 1)

T1 = r(x1), w(x2), c
T2 = w(x1), w(x2), c
T3 = r(x1), r(x3), c
T4 = w(x2),w(x1),c
T5 = r(x4),w(x4),c

6. Indique um escalonamento que seja "cascadeless", mas não estrito.

Por exemplo:

S1 = r(t1,x1), w(t1,x2), w(t2,x1), w(t2,x2), c(t2),c(t3)

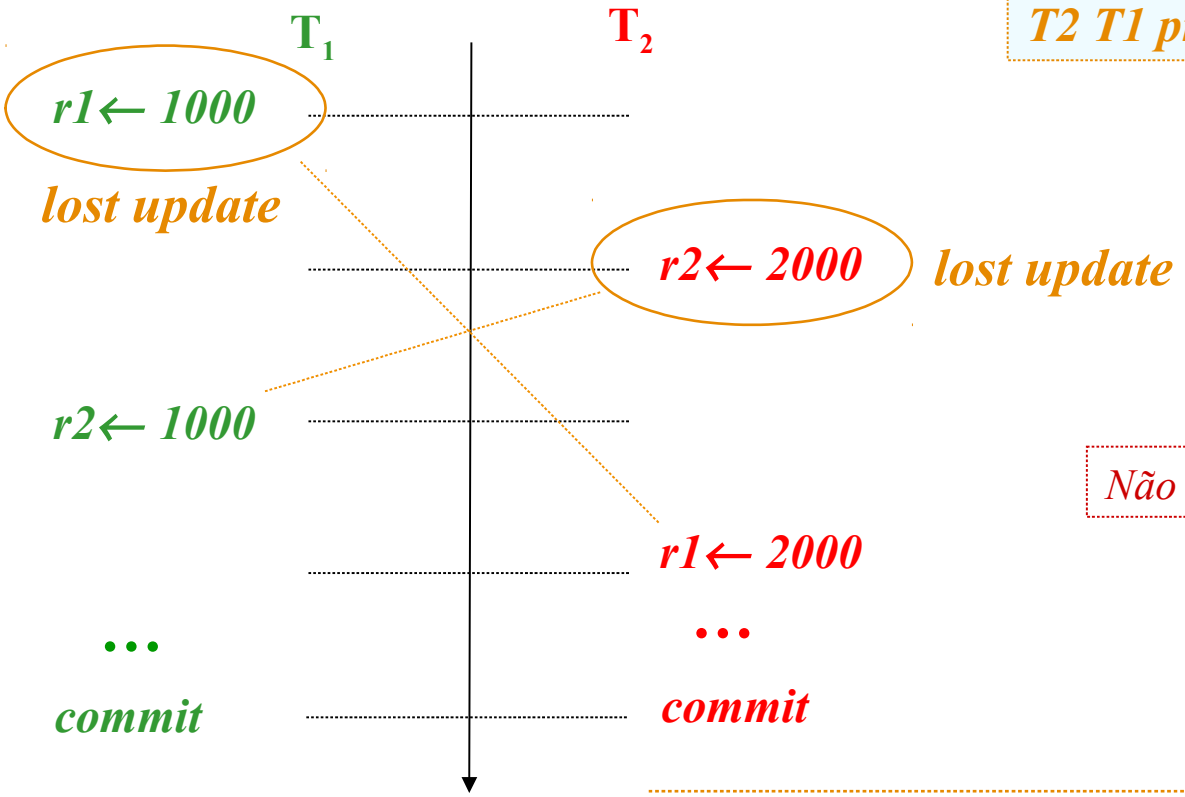


S2 = w(t4,x2), w(t2,x1), w(t4,x1), w(t2,x2), c(t4),c(t2)

Transacções e concorrência - Anomalias

"overwriting uncommited data" *(conflito W/W)*

T1 T2 produz r1 = 2000, r2 = 2000
T2 T1 produz r1 = 1000, r2 = 1000

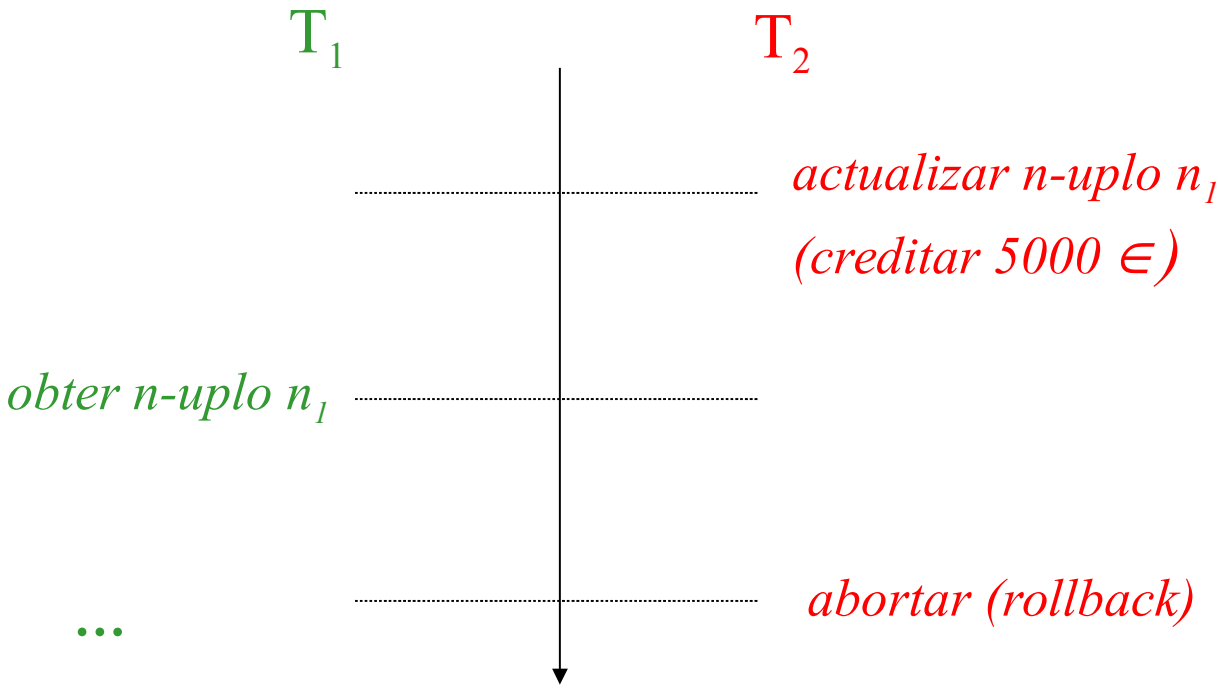


Não ocorre com a norma ISO SQL

Escalonamentos não estritos

Transacções e concorrência - Anomalias

"uncommited dependency" *(conflito W/R)*

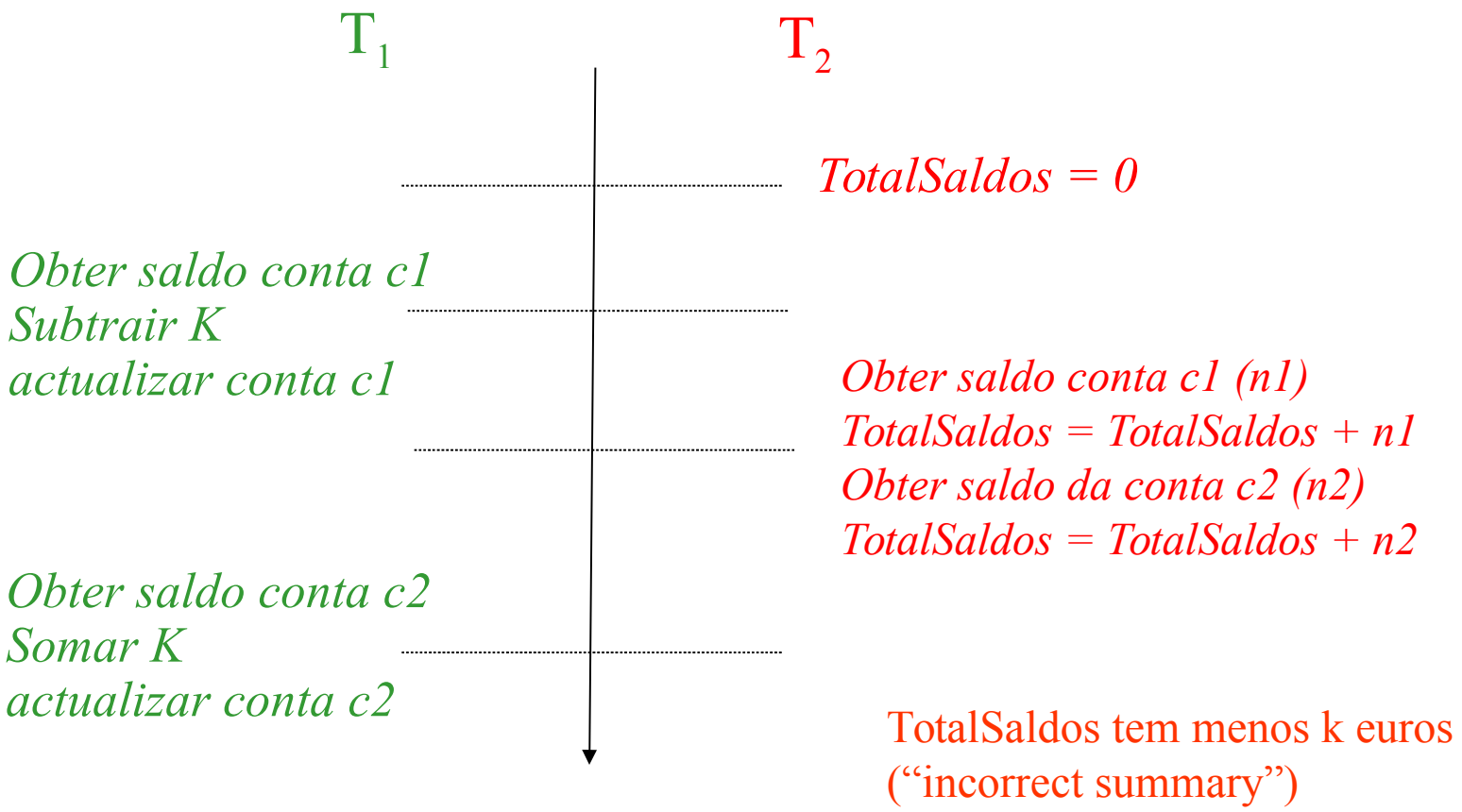


"dirty read" ou "temporary update"

Escalonamentos que exibem cascading rollback

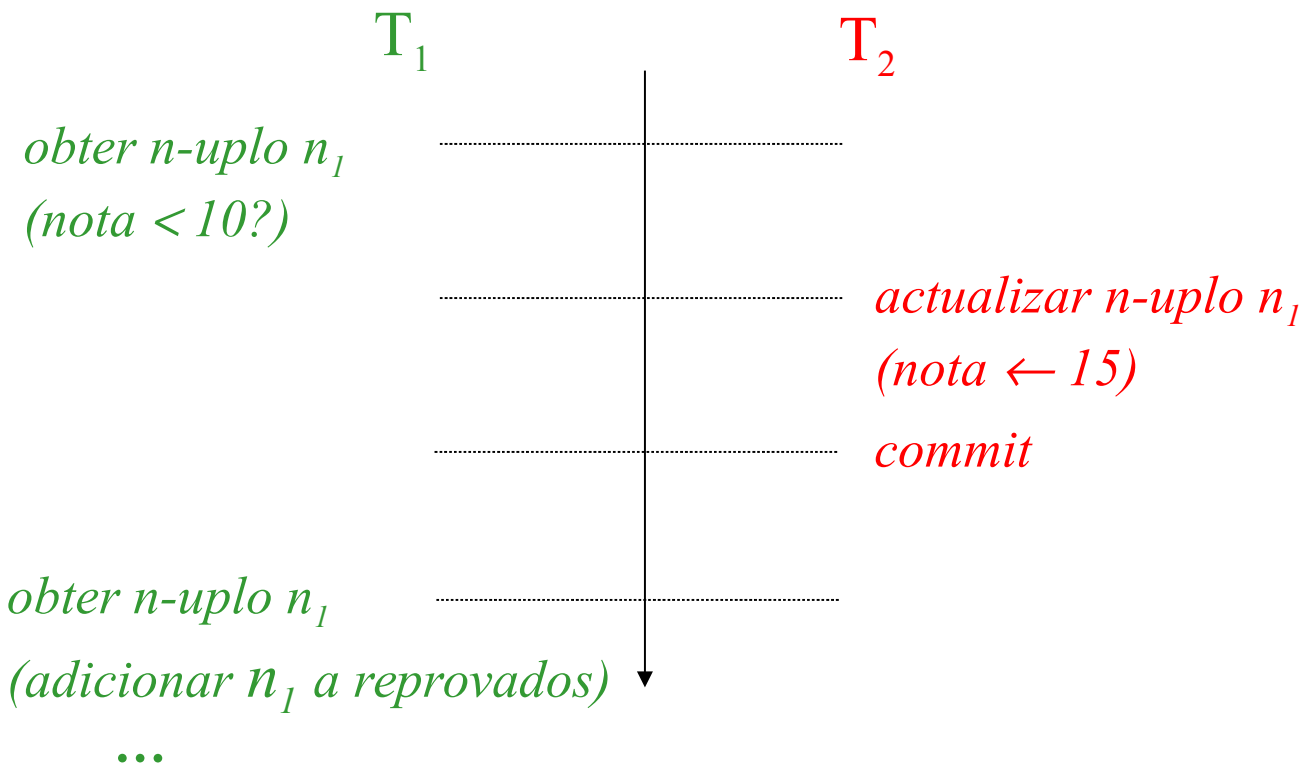
Transacções e concorrência - Anomalias

"uncommited dependency"



Transacções e concorrência - Anomalias

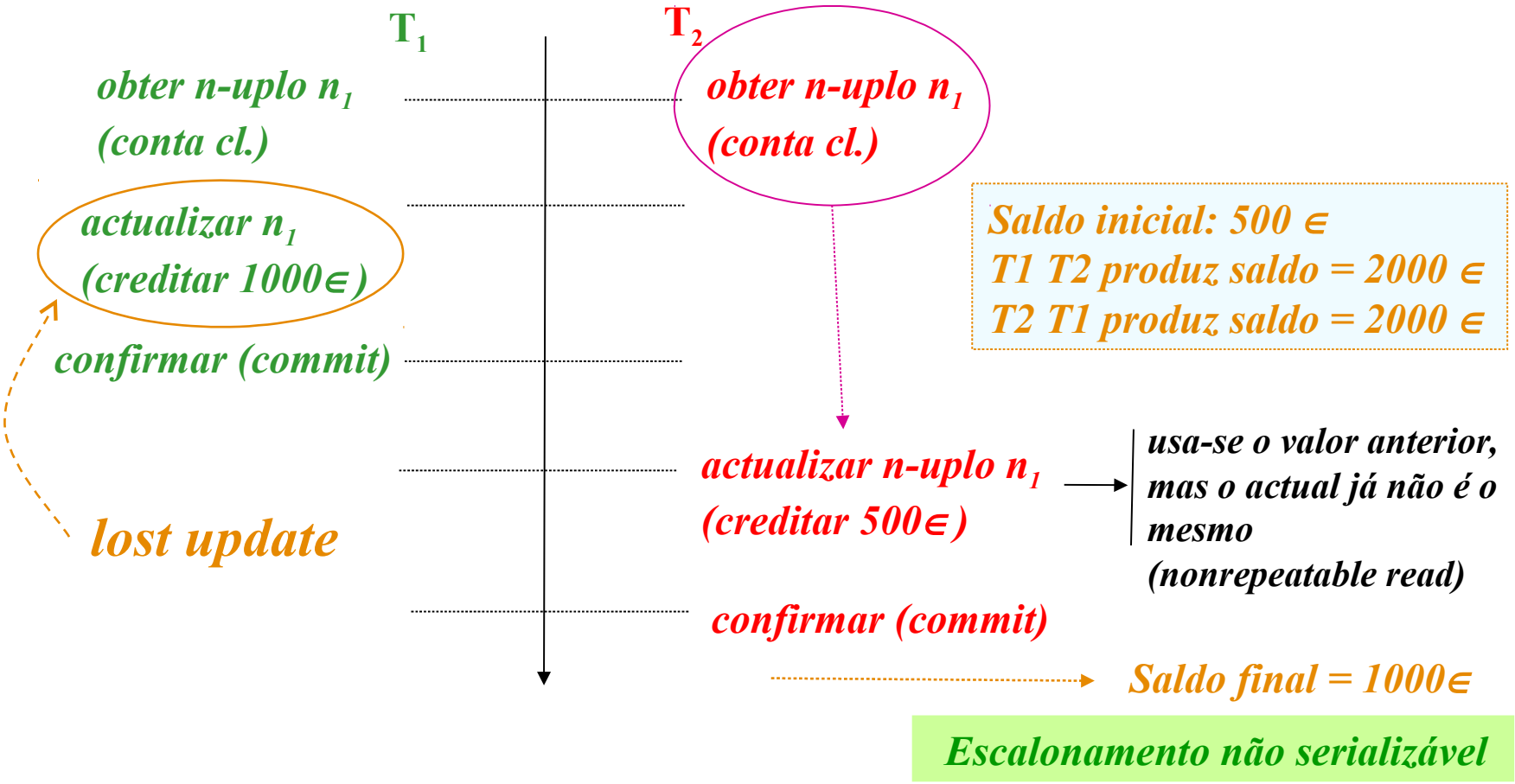
"nonrepeatable read", ou "inconsistent analysis" *(c. R/W)*



Escalonamento não serializável

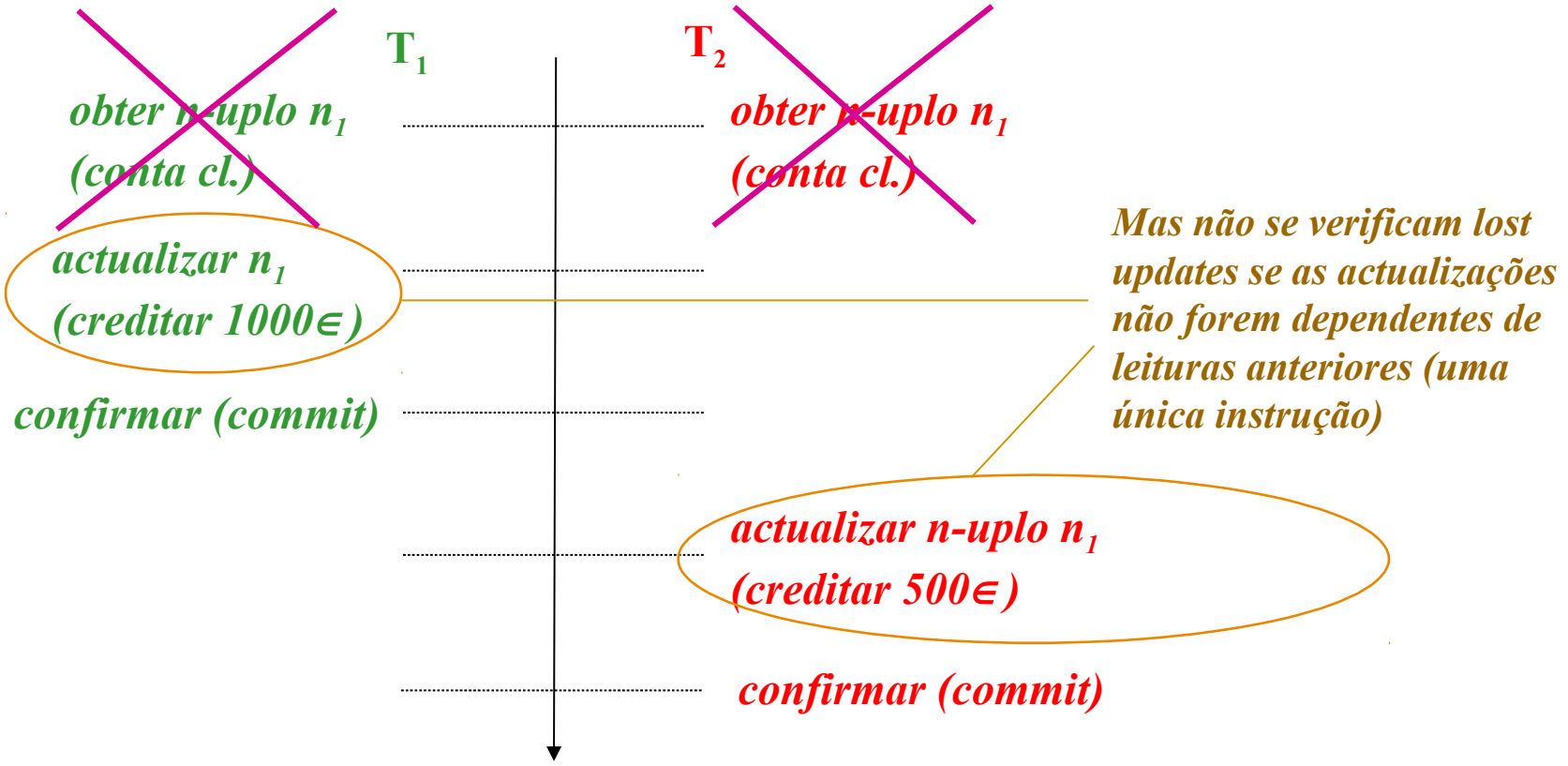
Transacções e concorrência - Anomalias

"nonrepeatable read" pode causar "lost updates"



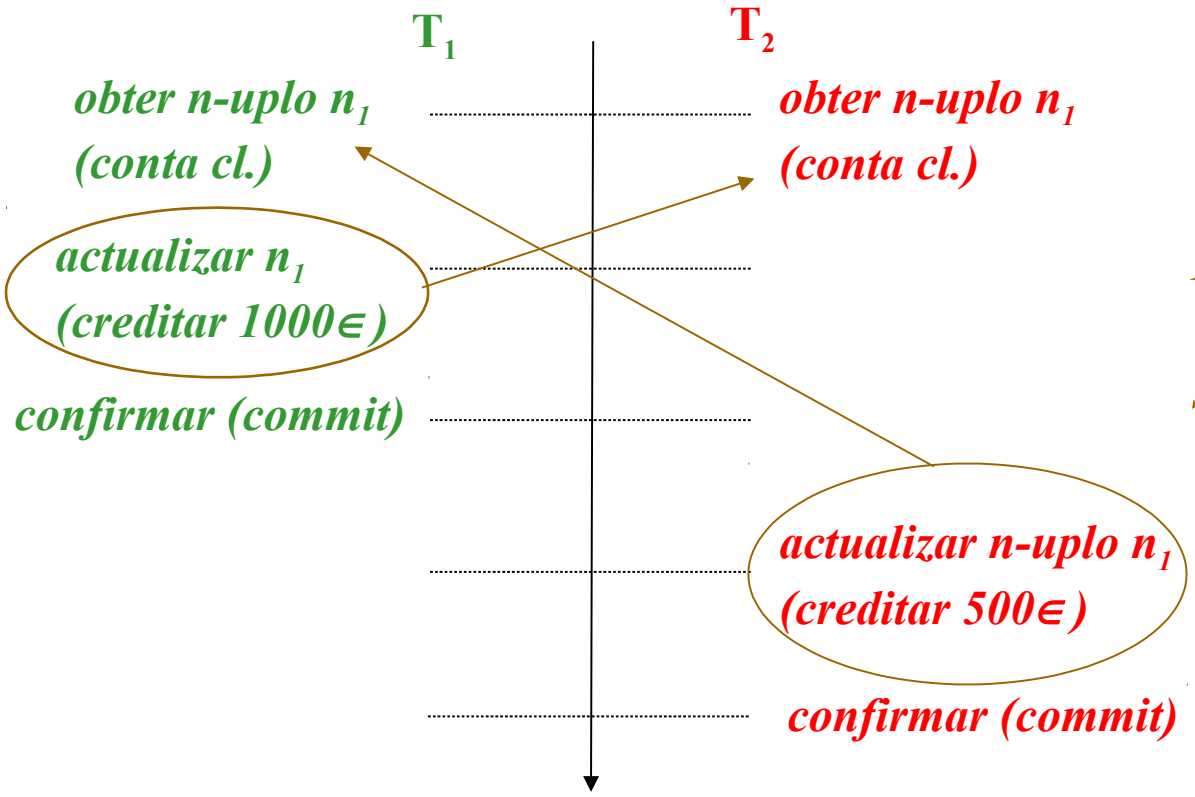
Transacções e concorrência - Anomalias

"nonrepeatable read" pode causar "lost updates"



Transacções e concorrência - Anomalias

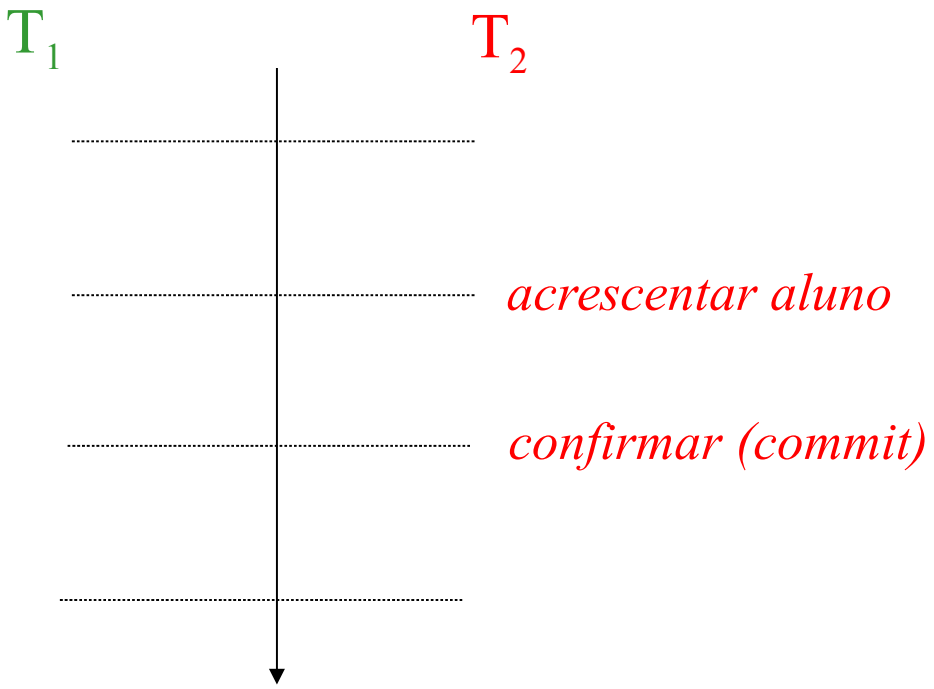
Com repeatable read teriamos um deadlock



Deadlock:
Uma das transacções seria abortada

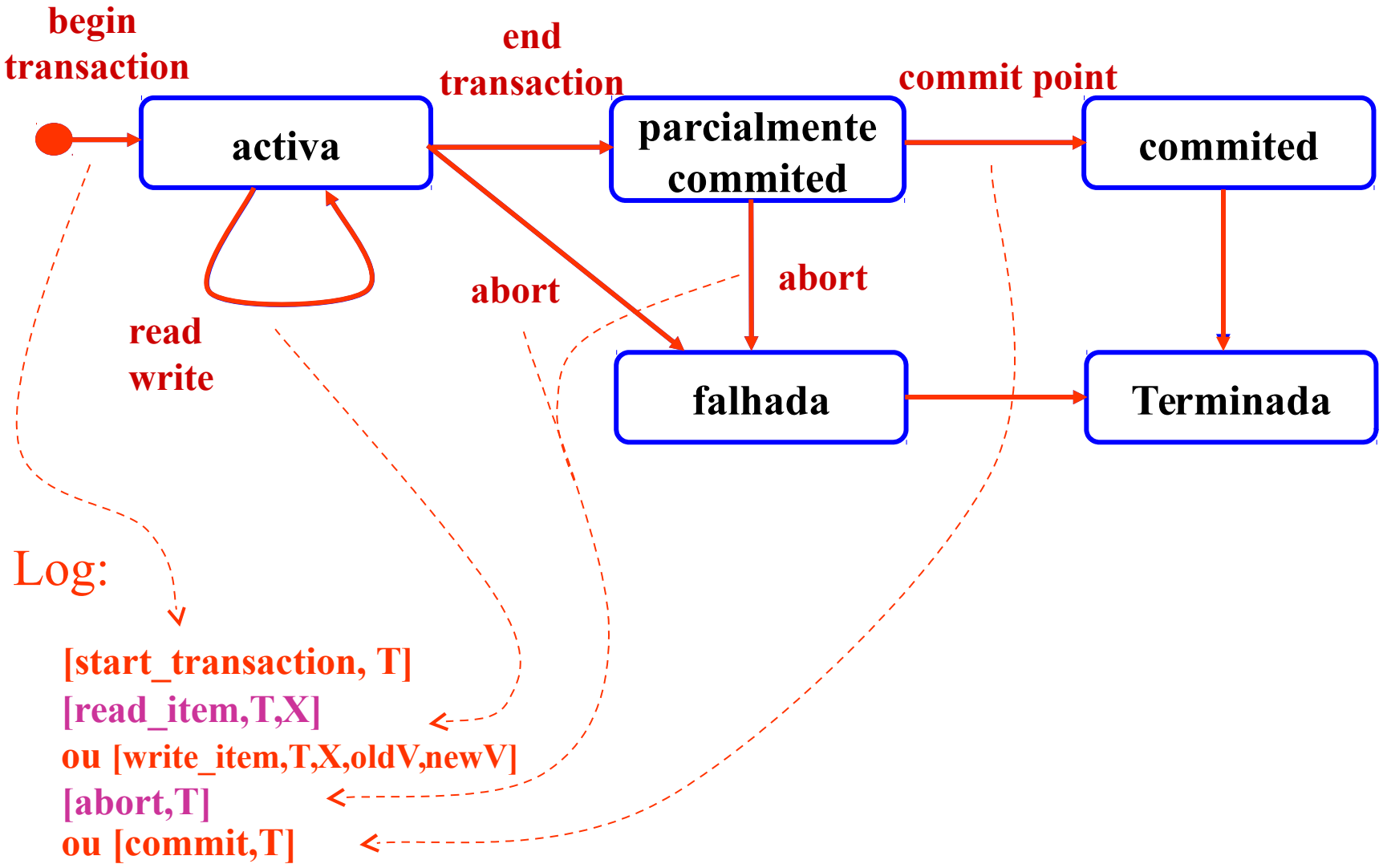
Transacções e concorrência - Anomalias

"phanton tuples" *(c. R/W)*



Não é uma anomalia que tenha a ver com a definição de conflito vista anteriormente.
Controlo mais difícil: predicate locking

Transacções – estados



Transacções – estados

Activa: é o estado após início da transação e mantém-se enquanto se forem realizando operações de leitura e escrita sobre os dados.

Parcialmente commited: quando se indica que a transacção deve terminar com sucesso, entra-se neste estado. Nele é garantido que todos os dados são transferidos para disco (force-writing) e só se isso acontecer é que a transacção atinge o commit point

Commited: a transaccão entra neste estado quando atinge o commit point (escreve [commit, T] no log)

Falhada: a transacção vem para este estado se for abortada no seu estado activa ou se os testes realizados no estado parcialmente commited falharem (escreve [abort, T] no log)

Terminada: a transação deixa de existir no sistema

Transacções – nível de isolamento em SQL2011

<set transaction statement> ::=
SET [LOCAL] TRANSACTION <transaction characteristics>

<start transaction statement> ::=
START TRANSACTION [<transaction characteristics>]

<transaction characteristics> ::=
[<transaction mode> [{ <comma> <transaction mode> }...]]

<transaction mode> ::= <isolation level> | <transaction access mode>
| <diagnostics size>

<transaction access mode> ::= READ ONLY | READ WRITE

<isolation level> ::= ISOLATION LEVEL <level of isolation>

<level of isolation> ::= READ UNCOMMITTED | READ COMMITTED
| REPEATABLE READ | SERIALIZABLE

Transacções – nível de isolamento em SQL SERVER 2012

Em SQL Server 2012

```
SET TRANSACTION ISOLATION LEVEL
  {   READ COMMITTED     (por omissão)
    | READ UNCOMMITTED
    | REPEATABLE READ
    | SERIALIZABLE
    | SNAPSHOT
  }
```

Pode alterar-se o nível dentro da transacção (com algumas limitações nos modos SNAPSHOT)

Se se alterar o nível de isolamento dentro dum procedimento armazenado ou gatilho, o nível de isolamento existente antes do SP ou gatilho é reposto na sua desactivação

Notar que nos SGBDs comerciais, não é habitual haver a realização explícita de escalonamentros (sobretudo para utilizações interectivas isso seria complicado de realizar). A opção mais comum consiste em deixar que a interferência entre transações ocorra sendo os escalonamentos obtidos em função dos níveis de isolamento usados que poderão conduzir a atrasos em algumas transações ou rooback de outras.

Transacções em SQL2011 – níveis de isolamento

Nível de isol.	Anomalia		
	dirty read	nonrep. read	phanton
read uncomm.	**sim**	**sim**	**sim**
read comm.	**não**	**sim**	**sim**
repeat. read	**não**	**não**	**sim**
serializable	**não**	**não**	**não**

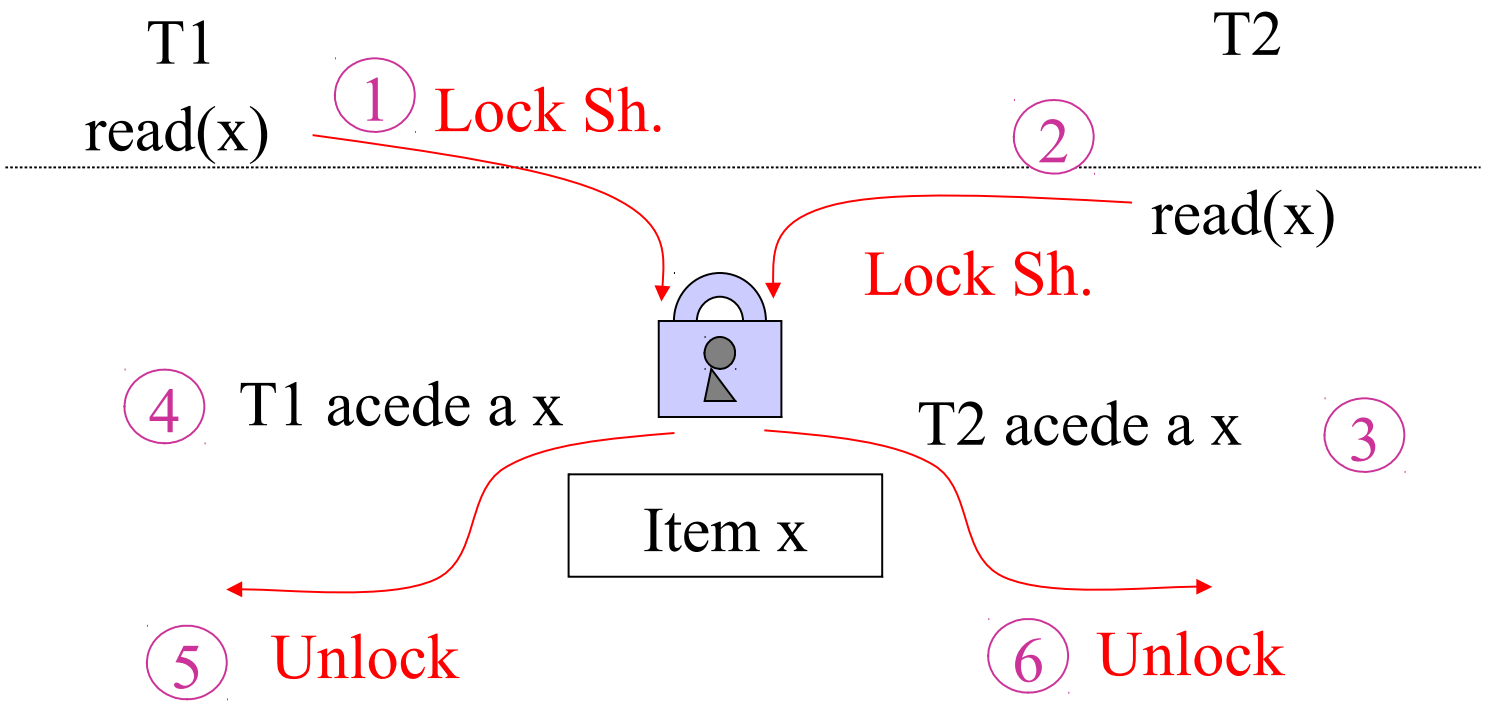
- **Uma transacção é sempre bem comportada relativamente às outras (write é bem formada e de duas fases), ou seja, não existe a anomalia *overwriting uncommited data*. O nível *read uncommited* só é possível com modo read only.**
- **Cada transacção protege-se das outras tanto quanto necessário, escolhendo o nível de isolamento adequado.**
- **Para se evitarem *lost updates*, todas as transacções têm de ter o nível de isolamento *repeatable read* ou superior, ou, então, as actualizações não podem depender de valores resultantes de leituras anteriores (actualizações de uma única acção *update*).**

Transacções – o protocolo *two phase lock* (2pL)

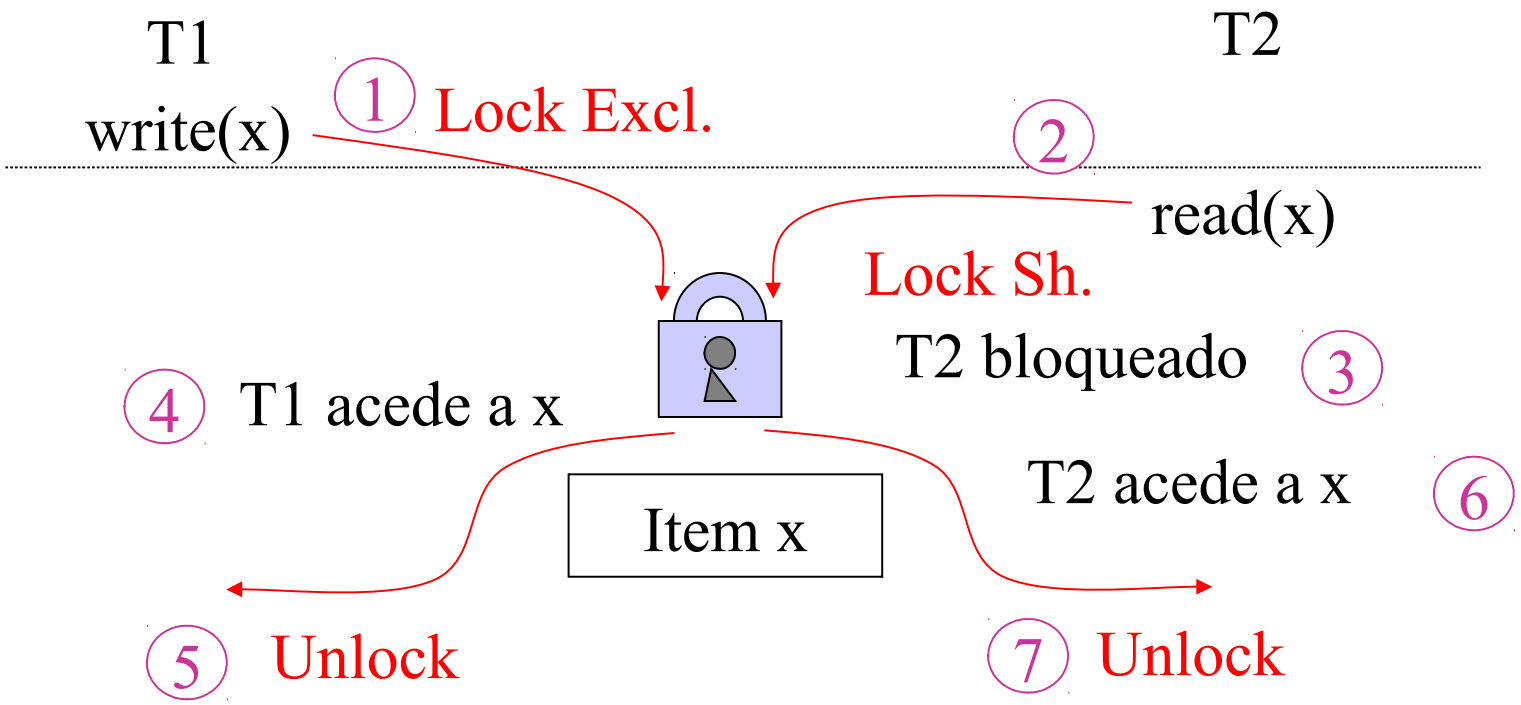
Matriz de compatibilidade

	Transacção 2			
	Modo	Unlock	Shared	Exclusive
Transacção 1	Unlock	**Sim**	**Sim**	**Sim**
	Shared	**Sim**	**Sim**	**Não**
	Exclusive	**Sim**	**Não**	**Não**

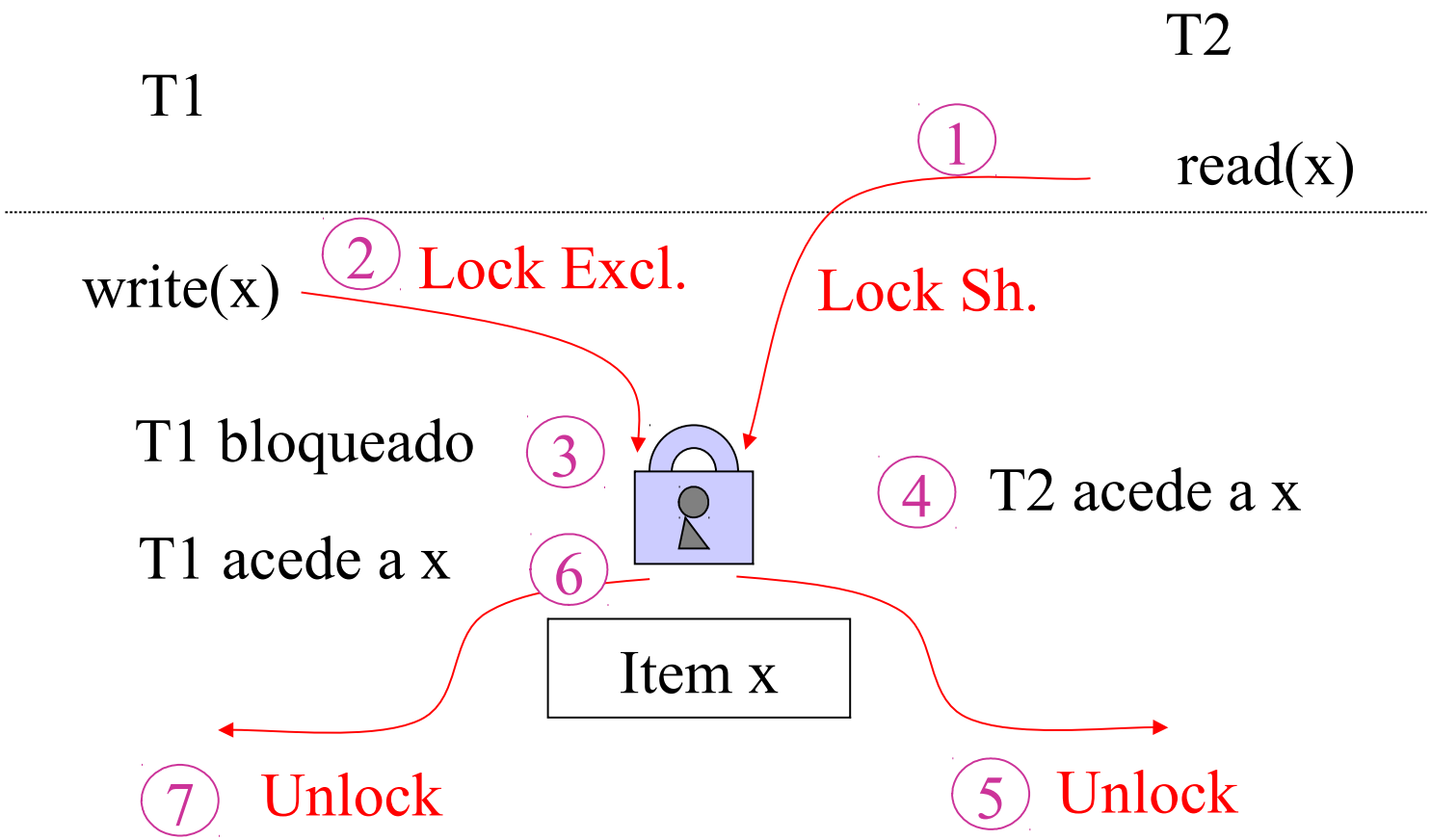
Transacções – o protocolo *two phase lock* (2pL)



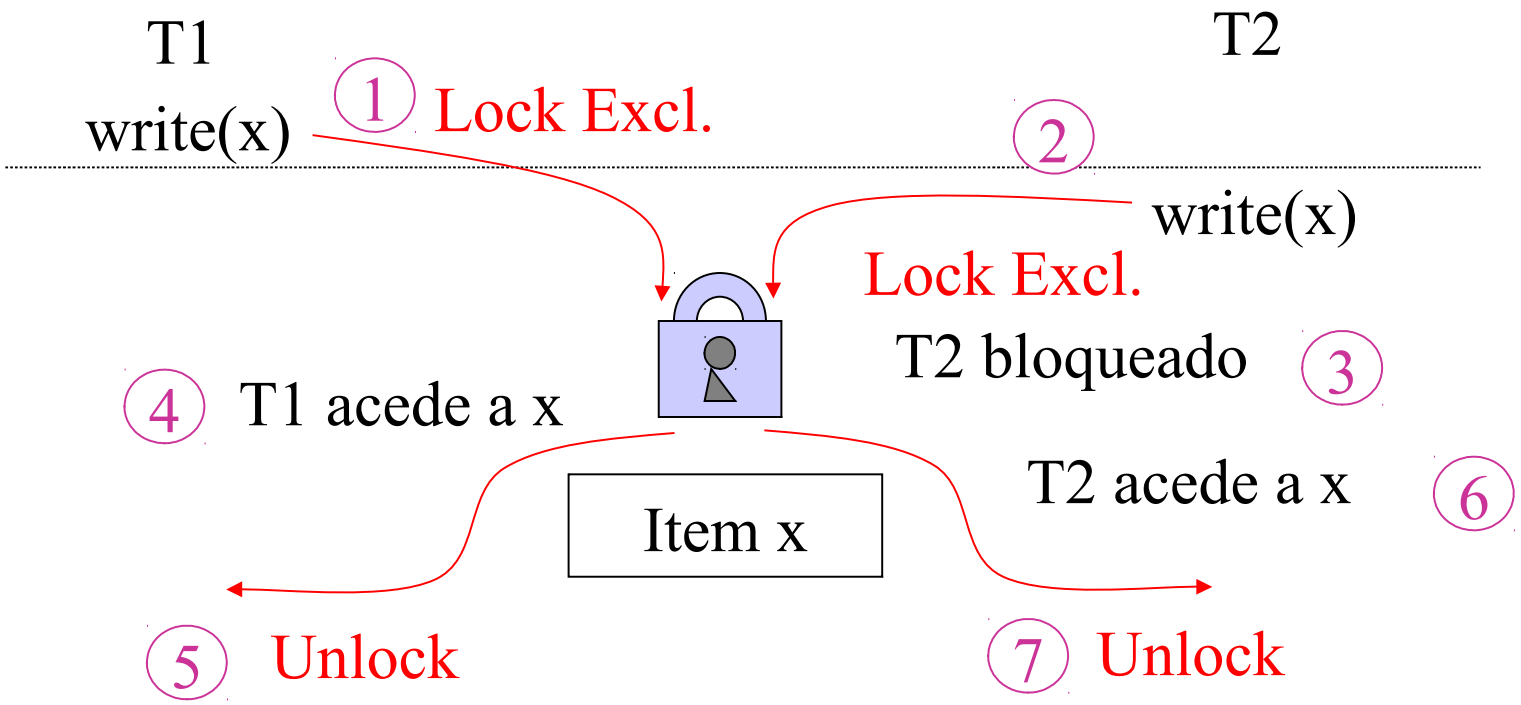
Transacções – o protocolo *two phase lock* (2pL)



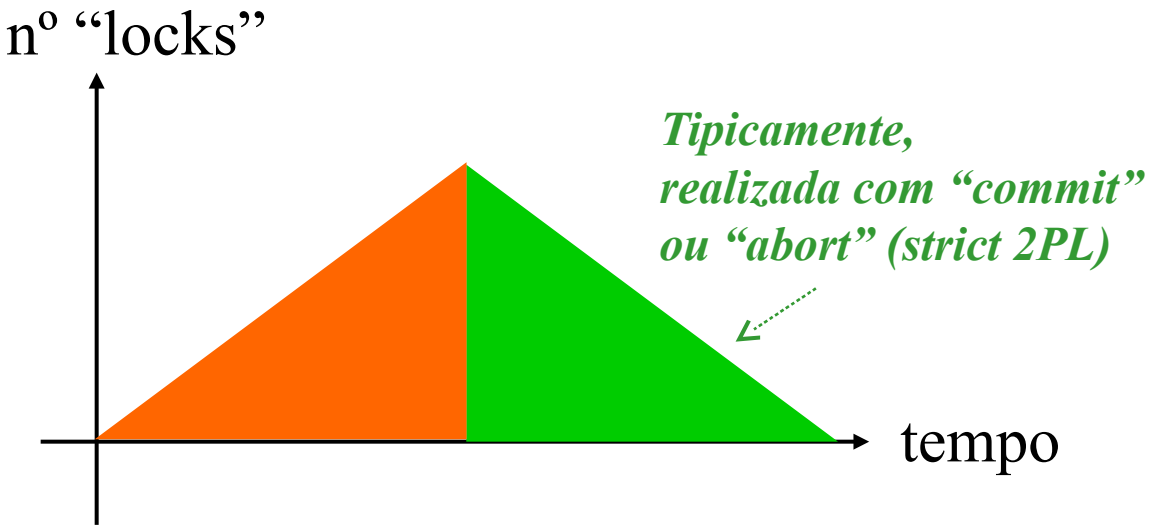
Transacções – o protocolo *two phase lock* (2pL)



Transacções – o protocolo *two phase lock* (2pL)



Transacções – o protocolo *two phase lock* (2pL)



Acção bem formada: protegida por um par *lock/unlock*

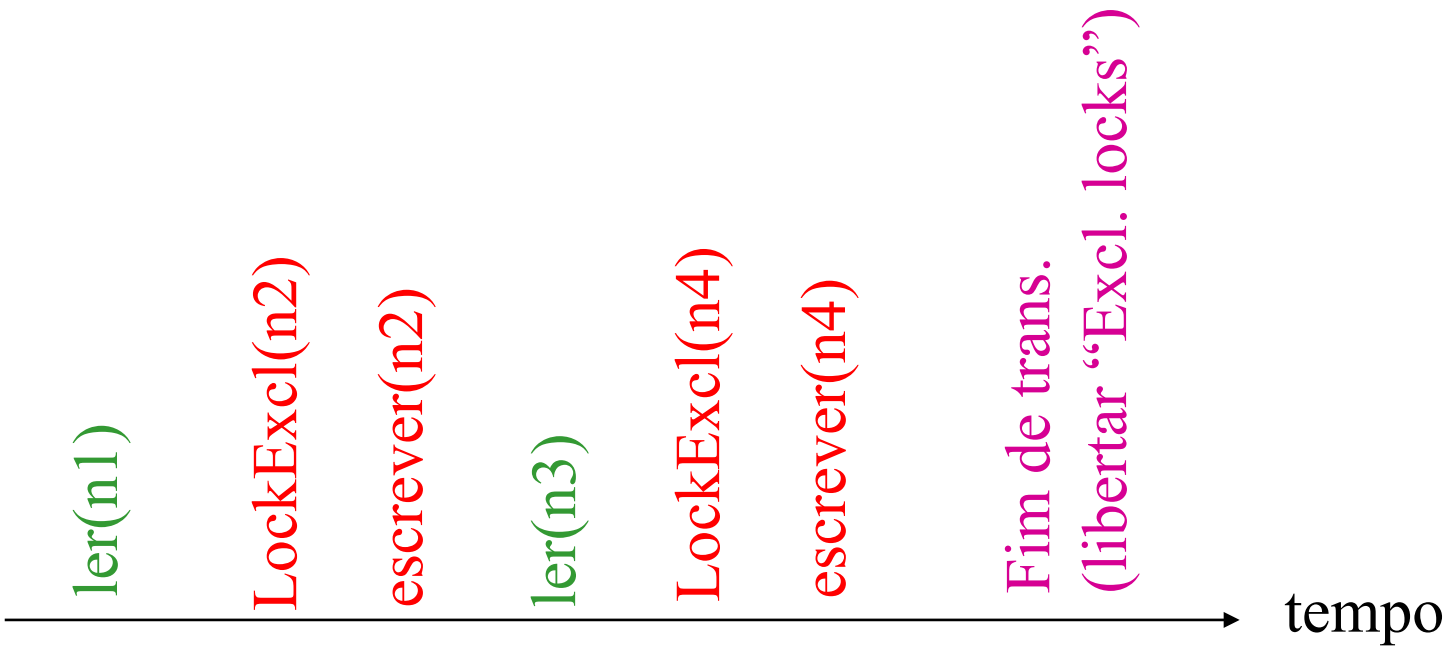
Acção de duas fases: não executa *unlock* antes de *locks* de outras acções da mesma transacção

Transacção bem formada: todas as suas acções são bem formadas

Transacção de duas fases: todas as suas acções são de duas fases

Transacções com 2PL e níveis de isolamento SQL 2011

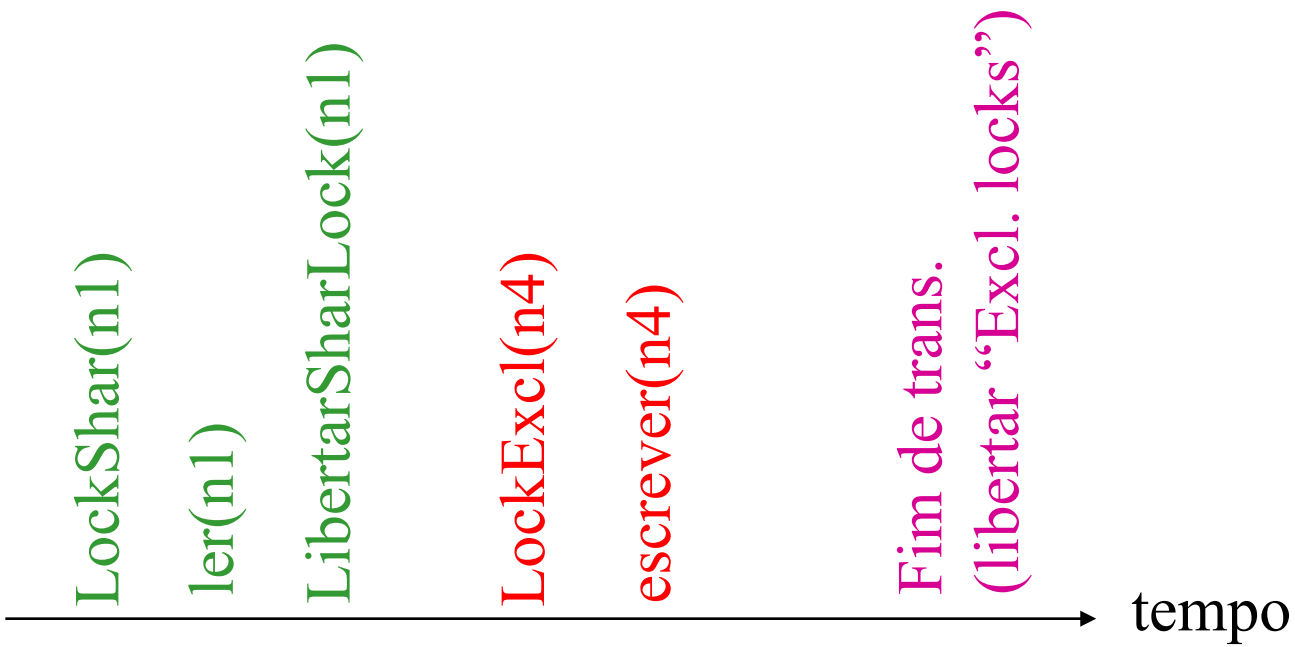
read uncommited



Ler – não é bem formada
Escrever – bem formada e de 2 fases

Transacções com 2PL e níveis de isolamento SQL 2011

read commited

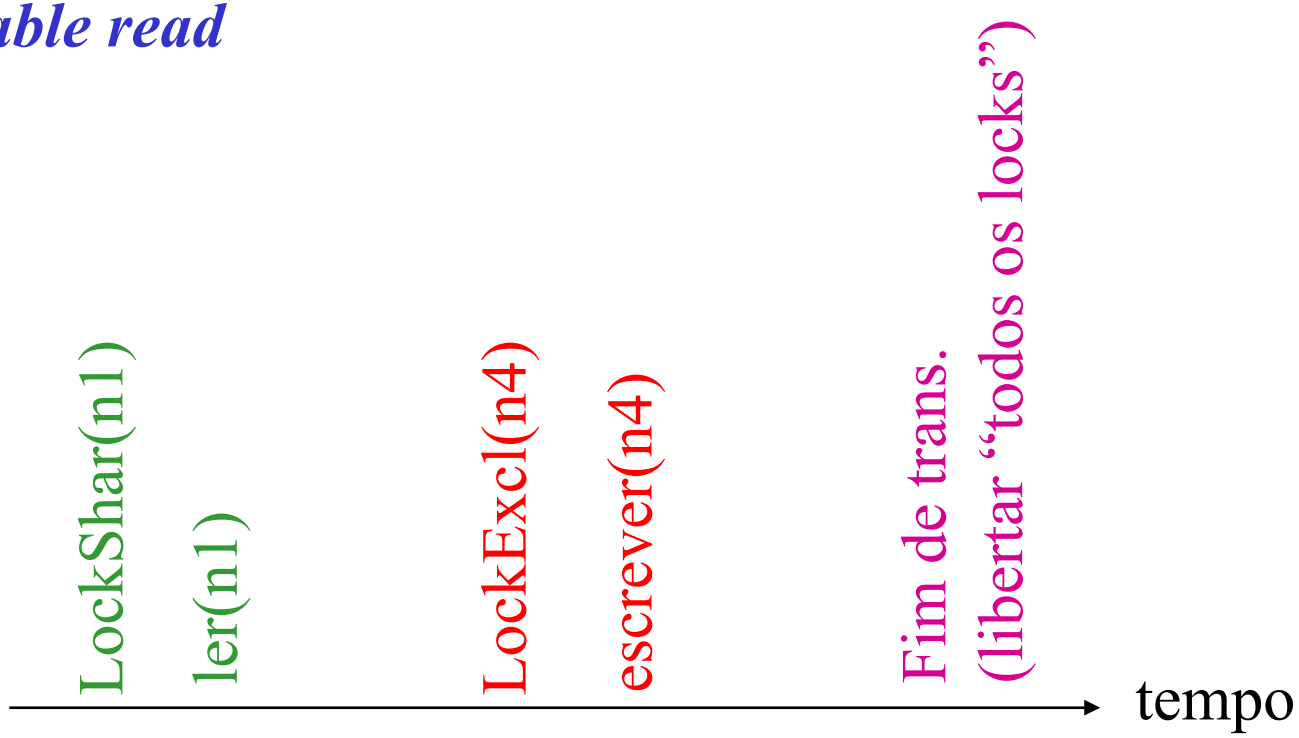


Ler – bem formada

Escrever – bem formada e de 2 fases

Transacções com 2PL e níveis de isolamento SQL 2011

repeatable read

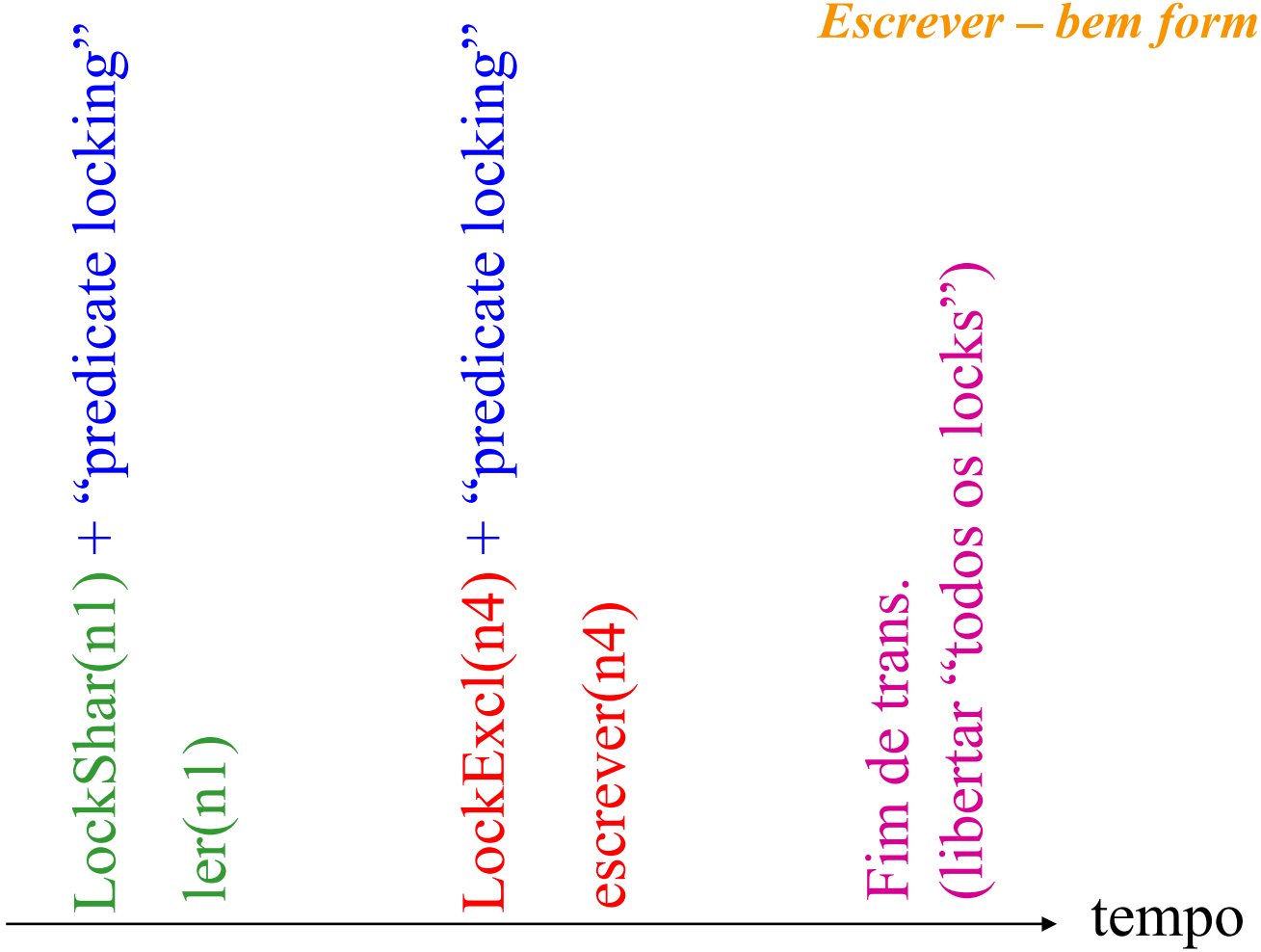


Ler – bem formada e 2 fases
Escrever – bem formada e de 2 fases

Transacções com 2PL e níveis de isolamento SQL 2011

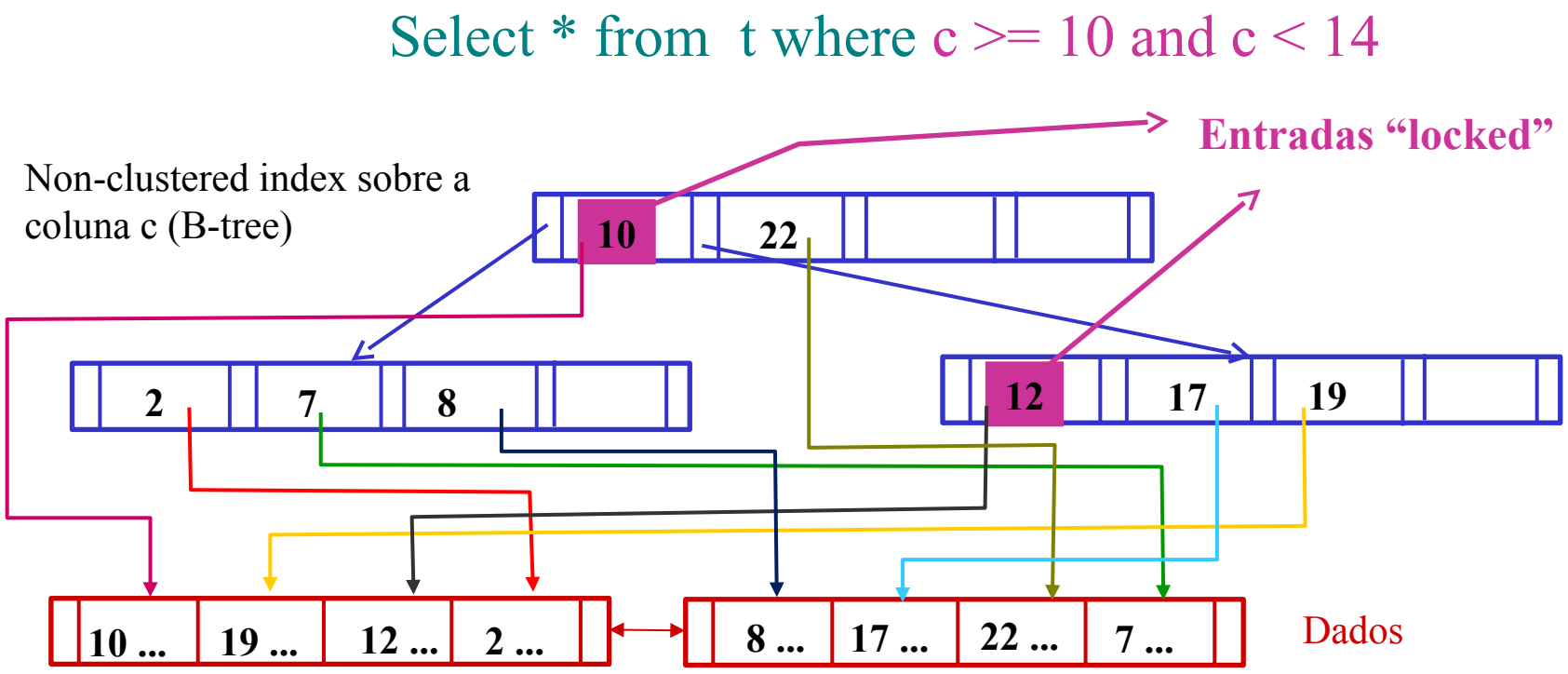
serializable

Ler – bem formada e de 2 fases
Escrever – bem formada e de 2 fases



Transacções com 2PL e níveis de isolamento SQL 2011

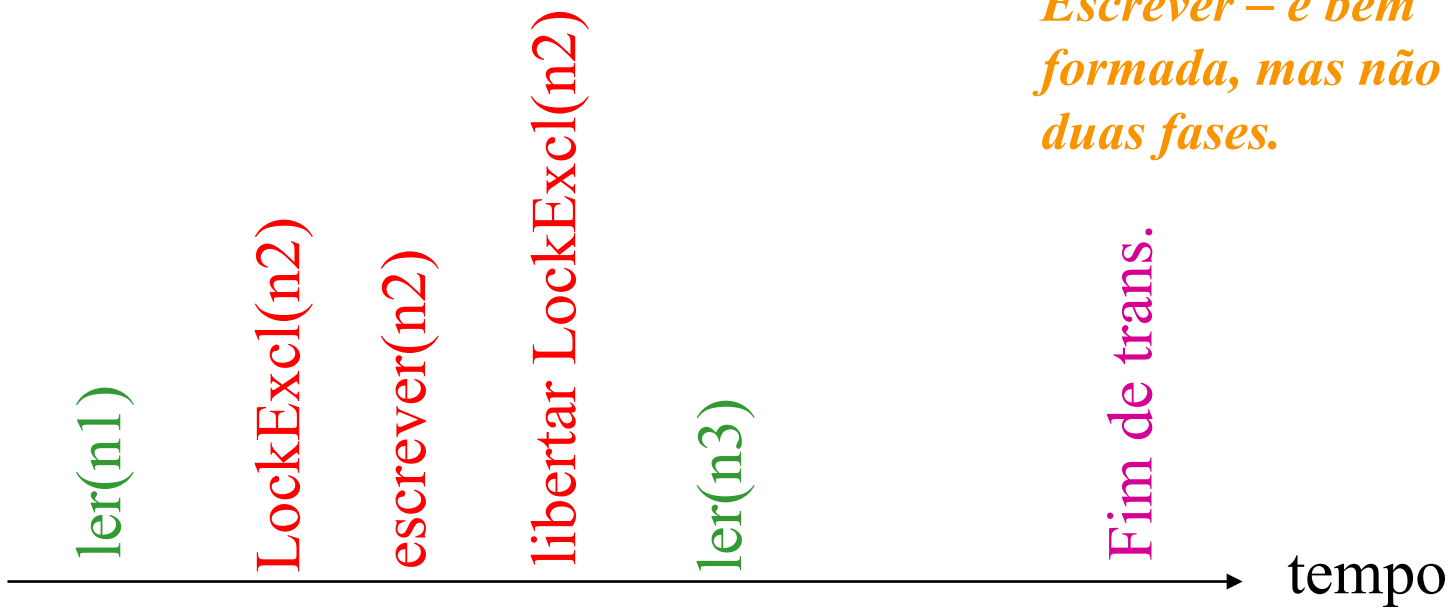
Na prática, o *predicate locking* costuma ser substituído por um lock a toda a tabela, ou, existindo um índice usável sobre as colunas da condição WHERE, podem ser realizados locks a parte do índice



Transacções com 2PL

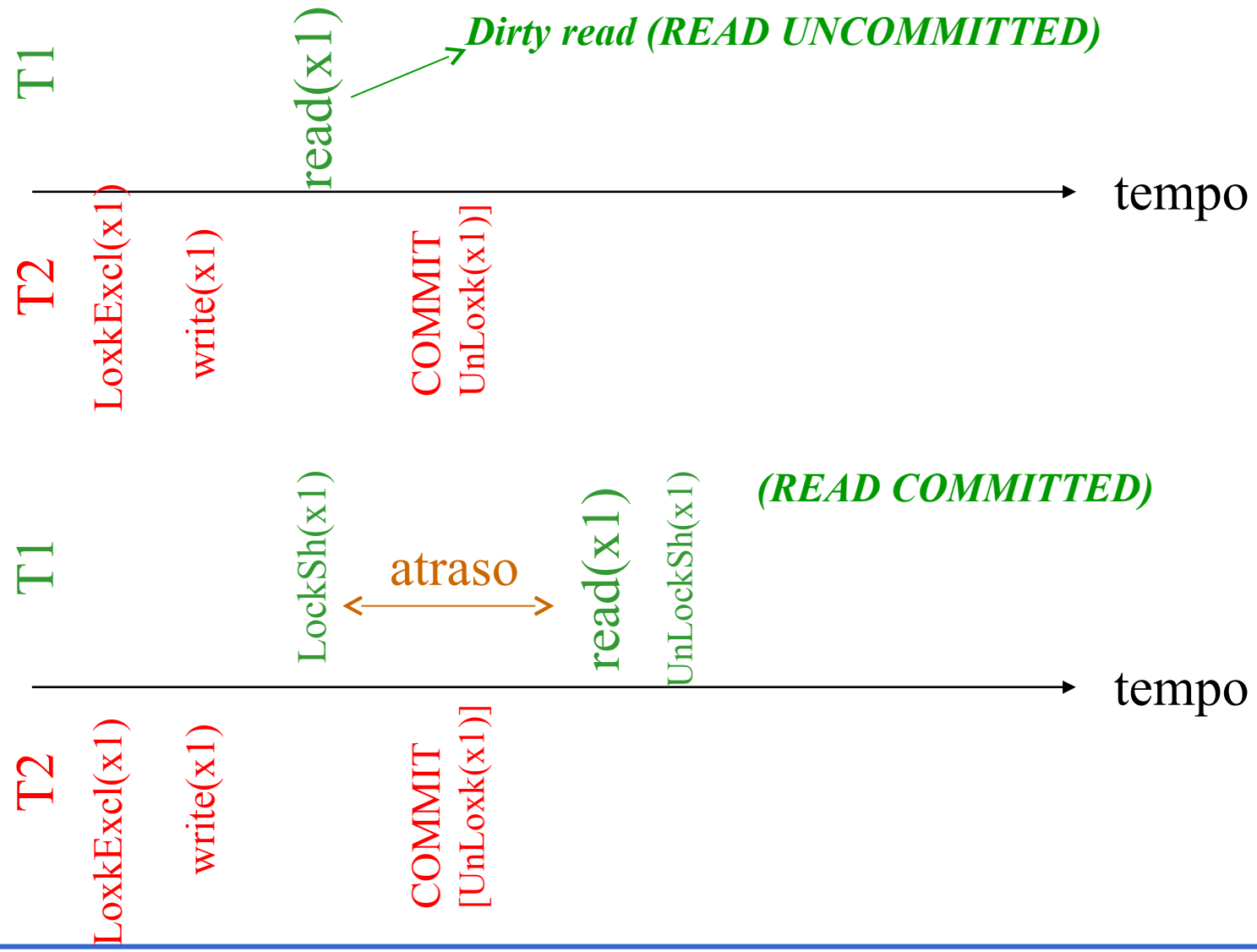
***chaos** (não compatível com a norma ISO SQL 2011, mas possível em alguns sistemas)*

Ler – não é bem formada
Escrever – é bem formada, mas não é de duas fases.

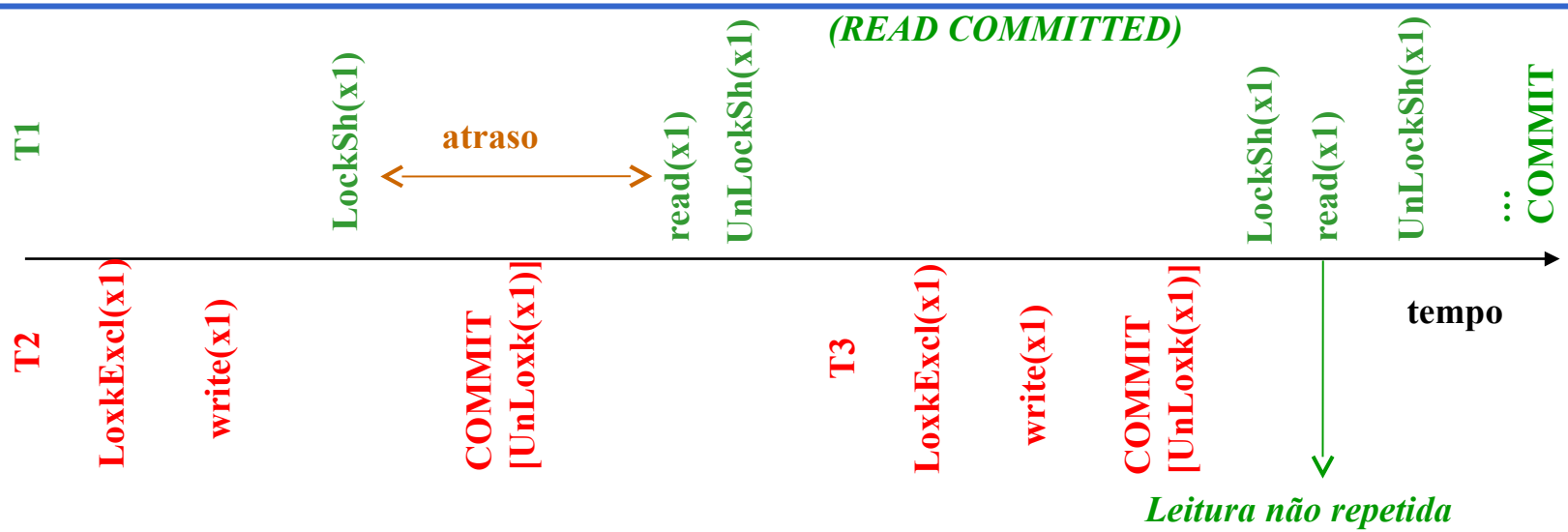


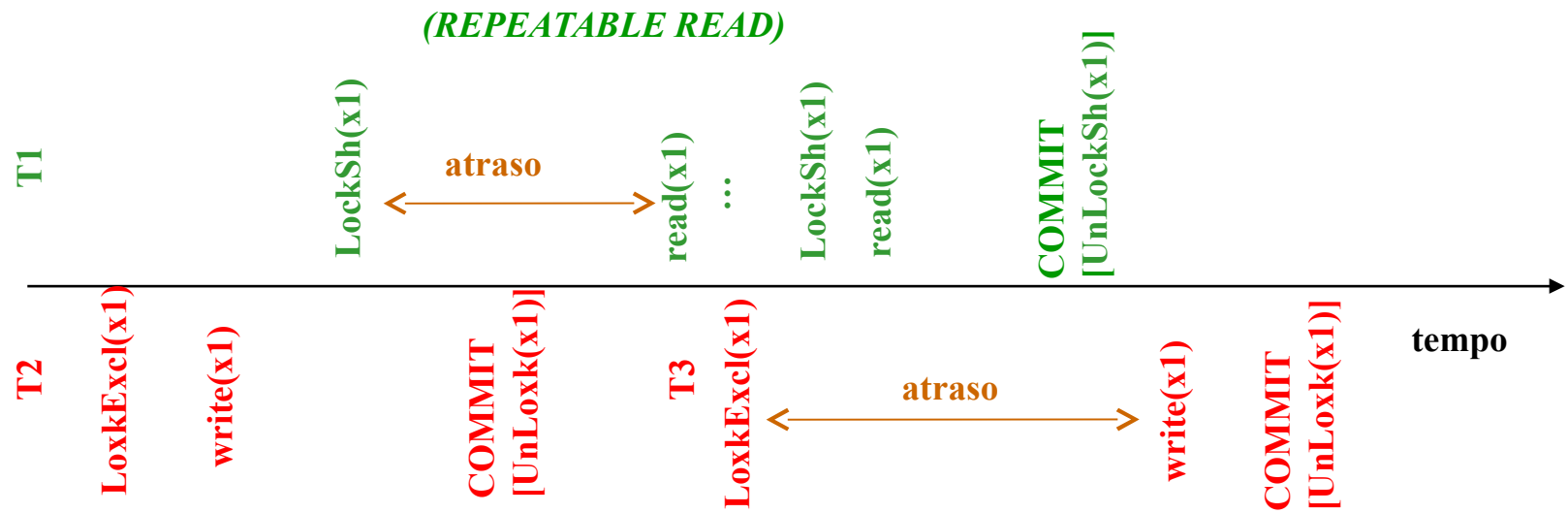
Apresenta a anomalia "overwriting uncommited data"

Transacções com 2PL e níveis de isolamento SQL 2011 (exemplos)

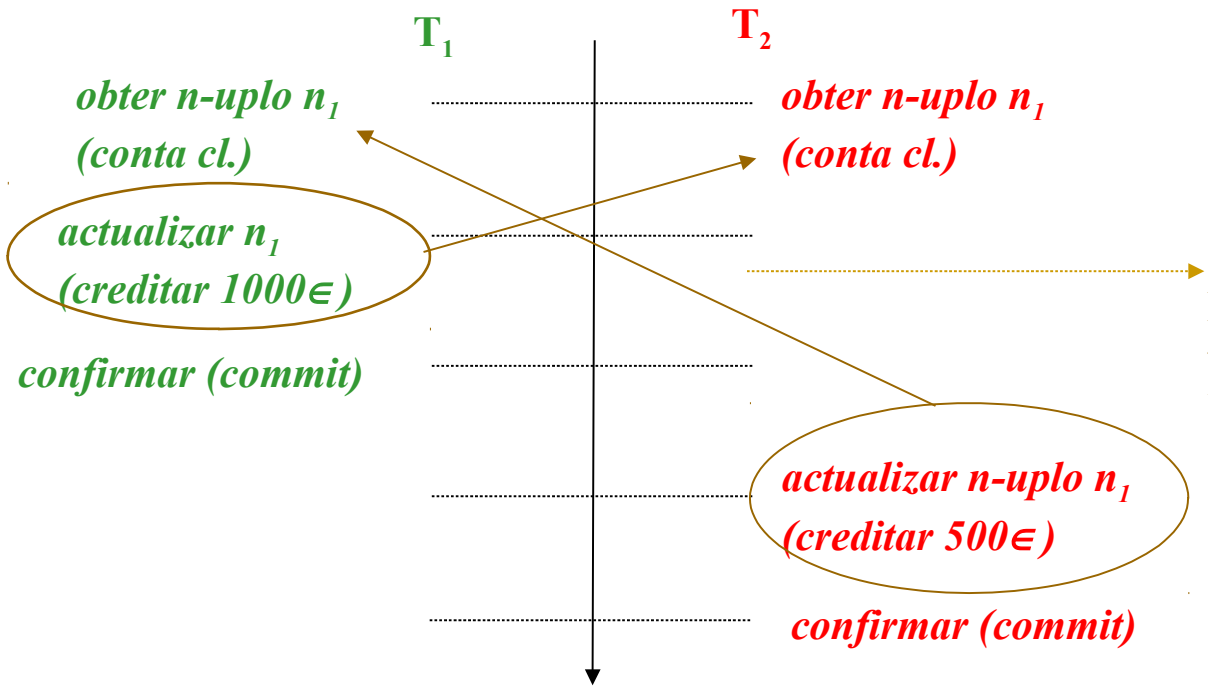


Transacções com 2PL e níveis de isolamento SQL 2011 (exemplos)





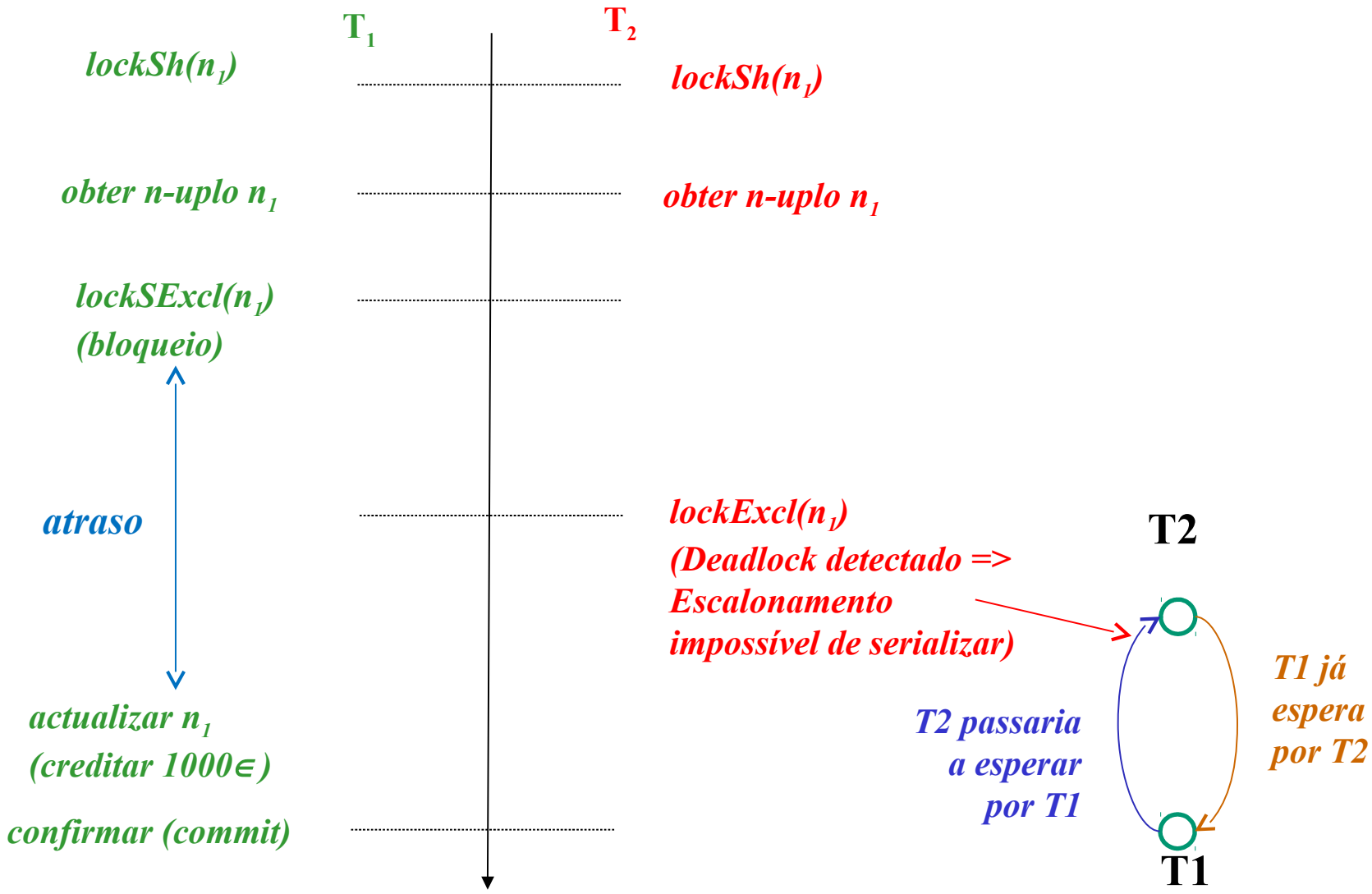
Transacções com 2PL – deadlocks



Se ambas as transações se executarem com nvel de isolamento igual ou superior a REPEATABLE READ, verifica-se um deadlock Uma das transações deve ser abortada (detecção de deadlocks ou timeouts)

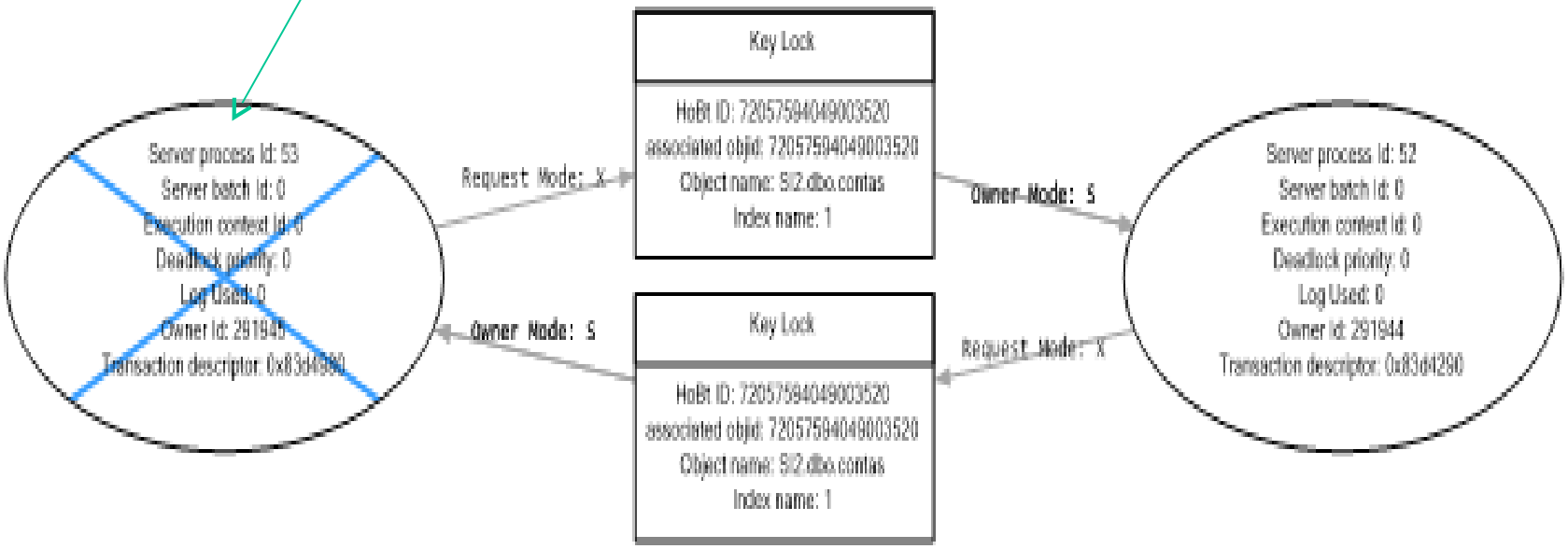
Para alguns tipos de processamento (lotes) as transacções abortadas podem ser reiniciadas automaticamente, mas para processamantos interactivos e quando se usa uma linguagem de programação a controlar a transacção isso não é viável. Logo, tais aplicações devem estar preparadas para lidarem com a possibilidade de as transacções falharem por vários motivos, entre os quais o serem abortadas pelo SGBD.

Transacções com 2PL – deadlocks



Transacções com 2PL – deadlocks

Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 53) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.



Transacções com 2PL –starvation

Se o esquema de selecção de qual das transacções bloqueadas terá acesso ao item for injusto, uma transacção pode ficar idefenidamente à espera (*starvation*).

Pode ser resolvido de várias formas. Por exemplo, adoptando uma disciplina FIFO no acesso aos itens.

No Sql Server não é relevante, dado que as transações eleitas como vítimas não saõ re-executadas de forma automática. Como são abortadas, deixam de existitir no sistema. As aplicações é que têm a responsabilidade de lidar com estas situações.

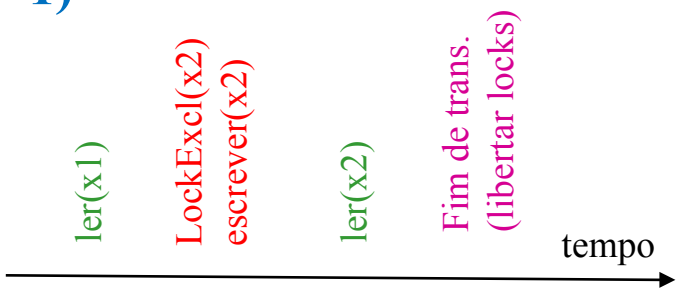
Transacções – Graus de isolamento (J. Gray, 1993)

- Grau 0 – *uma transacção 0º não altera dados alterados por outras transacções de 1º ou superior (bem formada em write) (chaos)*
- Grau 1 – *uma transacção 1º não exibe as anomalias "overwriting uncommited data" (read uncommited)*
- Grau 2 – *uma transacção 2º não exibe as anomalias "overwriting uncommited data" e "dirty reads" (read commited)*
- Grau 3 – *uma transacção 3º não exibe as anomalias "overwriting uncommited data" e "non repeatable reads" (consequentemente, não exibe o efeito "dirty reads") (repeatable read)*

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 1)

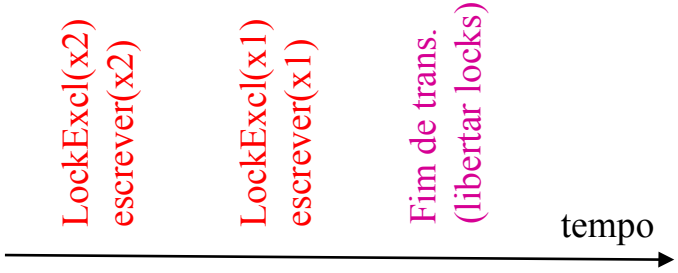
Sabendo que o SGBD é compatível com a norma ISSO 2011, diga, para cada uma das seguintes transações, qual o nível de isolamento sob o qual ela se executa:

1)

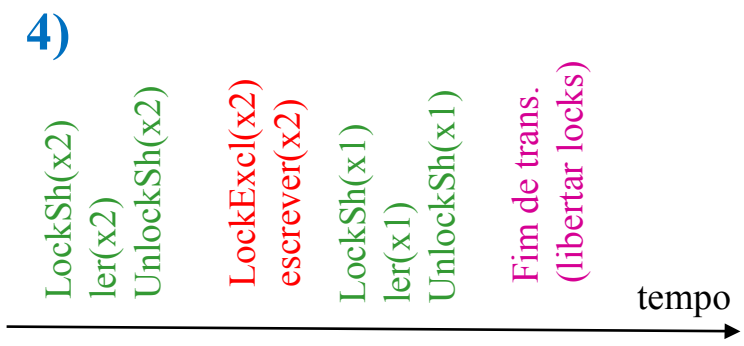


2)

3)

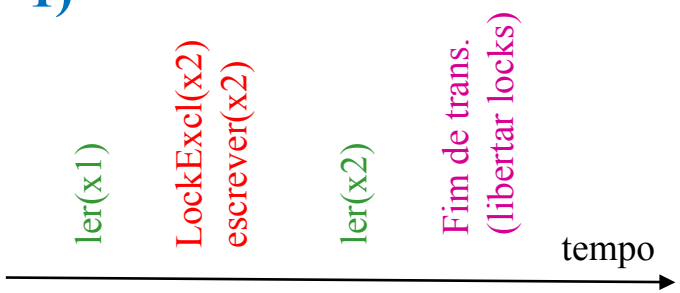


4)



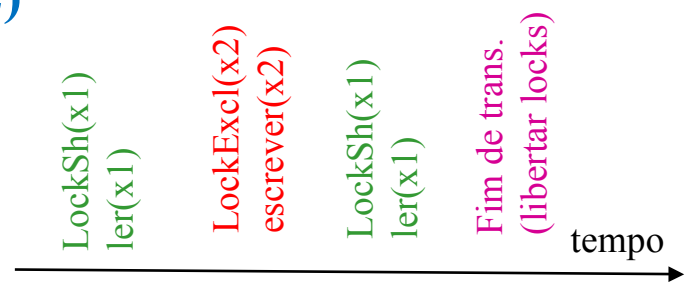
Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 1)

1)



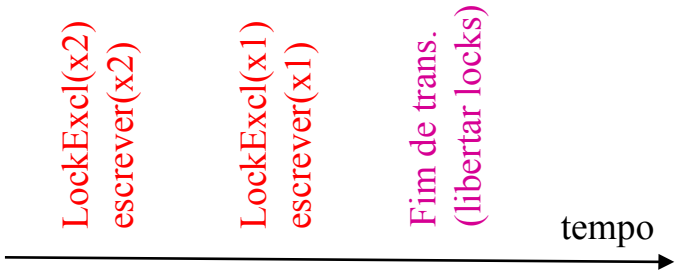
Read Uncommitted

2)



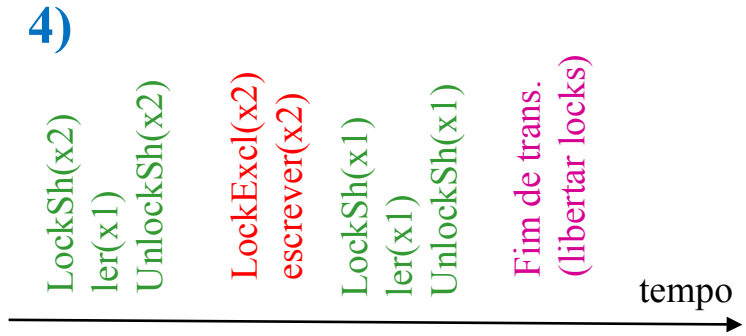
Repeatable Read

3)



Desconhecido
(e indiferente)

4)



Read Committed

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 2)

Considere as seguintes transações:

T1 = read(x1),write(x2),read(x1),c
T2 = write(x1),read(x2),c
T3 = write(x1),c
T4 = write(x2), write(x1),c
T5 = read(x3), write(x4),c
T6 = write(x1), write(x2),c

Indique níeveis de isolamento e escalonamentos (incluindo a indicação de potenciais locks e unlocks) que tenham as seguintes características:

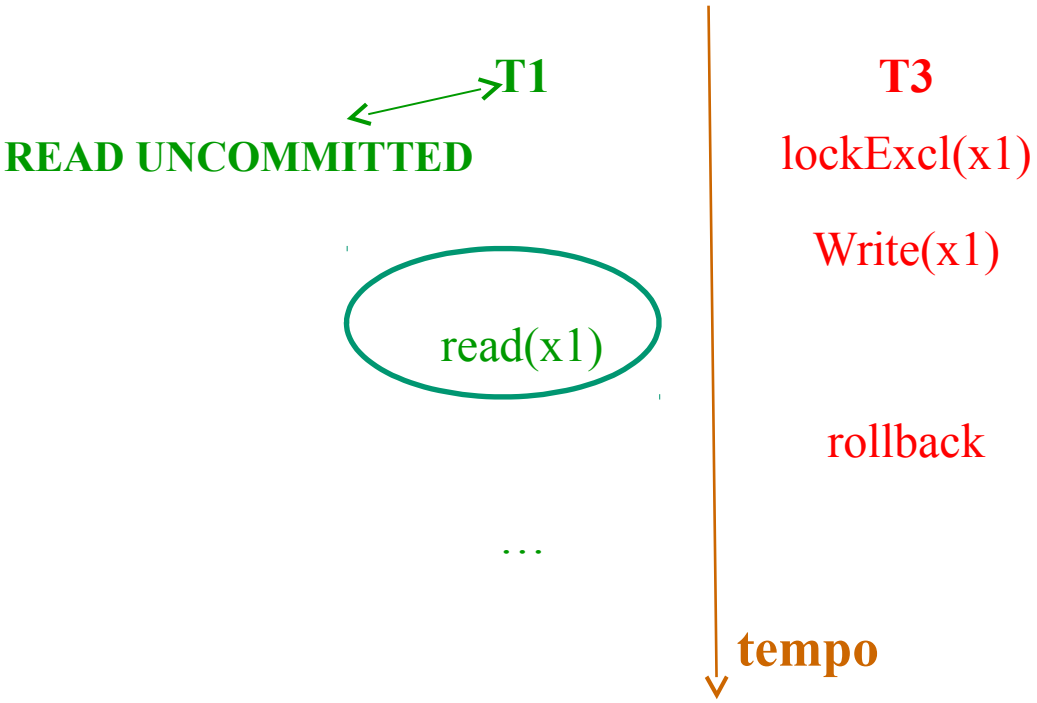
1. **Envolva apenas T1 e T3 e possua um dirty read.**
2. **Envolva apenas T1 e T3 e possua um non-repeatable read, mas nenhum dirty read.**
3. **Envolva T1 e apenas uma outra das transações e origine uma situação de deadlock.**
4. **Envolva apenas T2 e T4 e origine uma situação de deadlock.**
5. **Envolva T5 e apenas uma outra das transações e origine uma situação de dirty read.**
6. **Envolva T6 e T4 e origine uma situação de dirty read.**
7. **Envolva T6 e T4 e origine uma situação de deadlock.**

Nota: se não existirem escalonamentos com as característcas indicadas, justifique a razão para isso.

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 2)

T1 = read(x1),write(x2),read(x1),c
T3 = write(x1),c

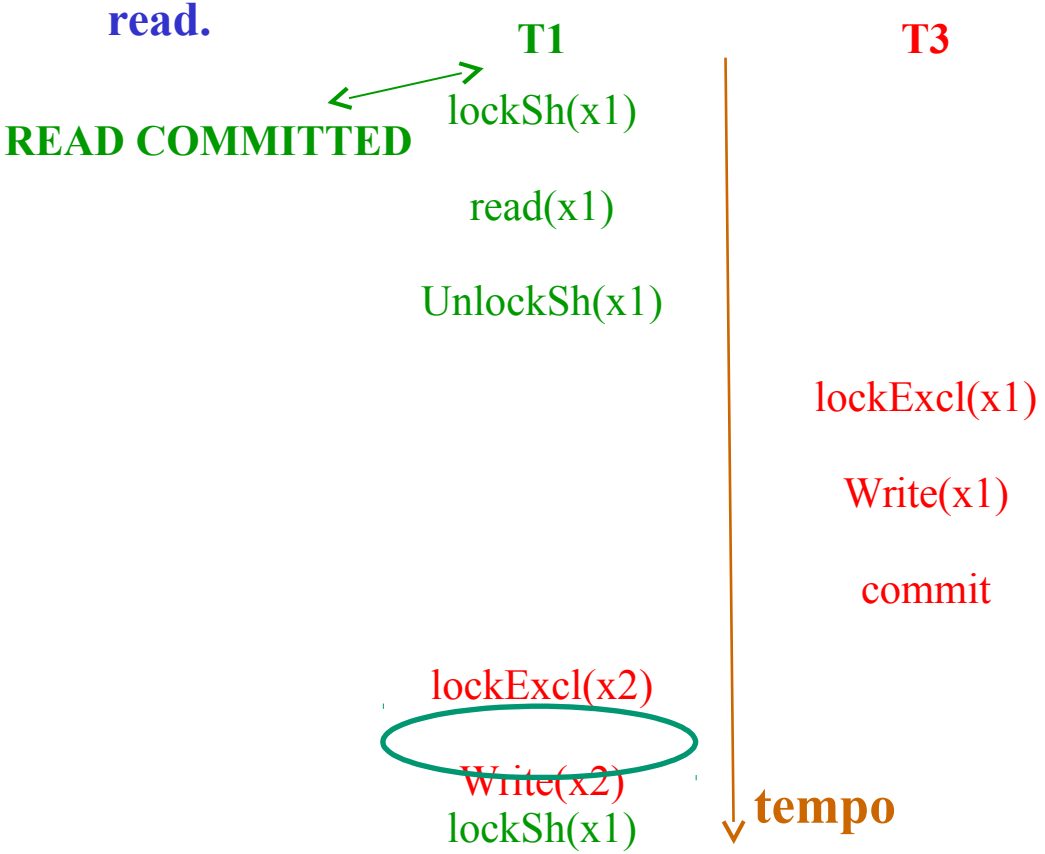
1. **Envolva apenas T1 e T3 e possua um dirty read.**



Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 2)

T1 = read(x1),write(x2),read(x1),c
T3 = write(x1),c

2. **Envolva apenas T1 e T3 e possua um non-repeatable read, mas nenhum dirty read.**



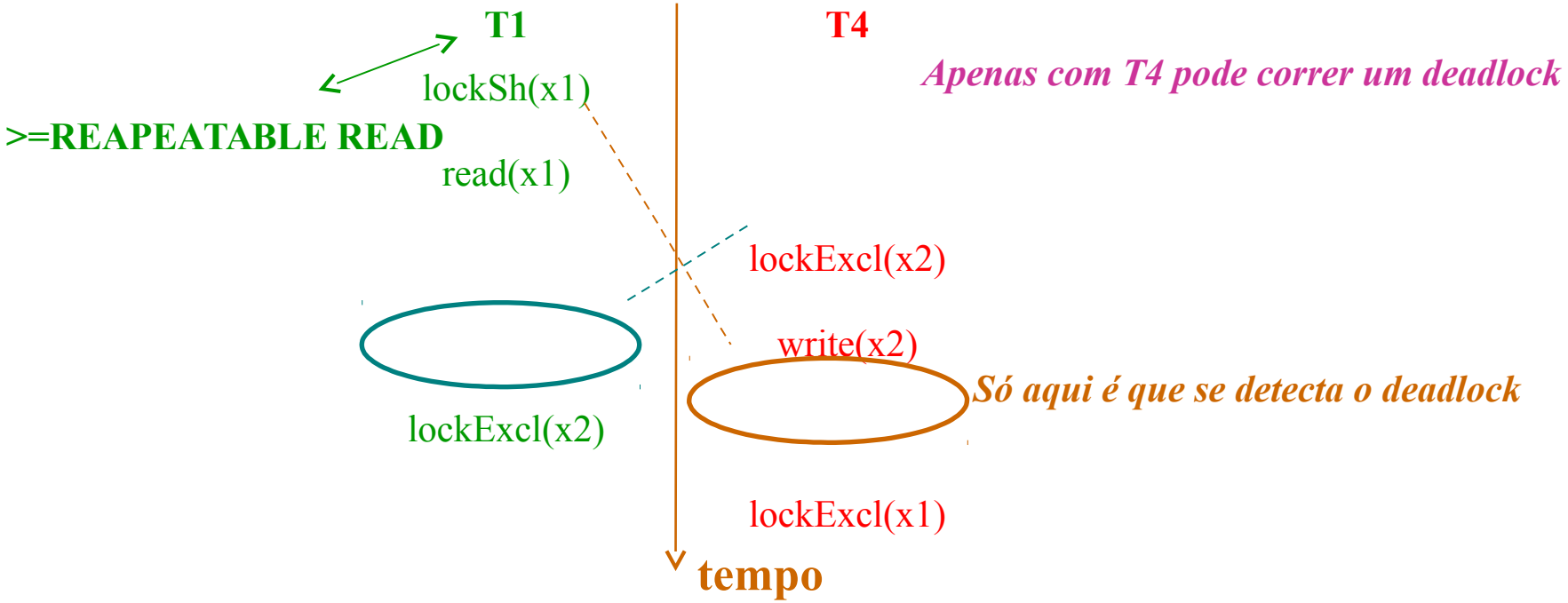
Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 2)

T1 = read(x1),write(x2),read(x1),c
T2 = write(x1),read(x2),c
T3 = write(x1),c
T4 = write(x2), write(x1),c
T5 = read(x3), write(x4),c
T6 = write(x1), write(x2),c

Para haver deadlock, tem de verificar-se o seguinte:

1. ***Cada uma das transações tem de ter duas ações de leitura ou escrita***
2. ***Tem de haver ações conflituantes cruzadas, ou seja: as primeiras dessas ações de ambas as transações não conflituam entre si (logo ambas podem deixar o item locked) mas cada uma delas conflitua com as segunda ação da outra transação (logo haverá um bloqueio mútuo).***

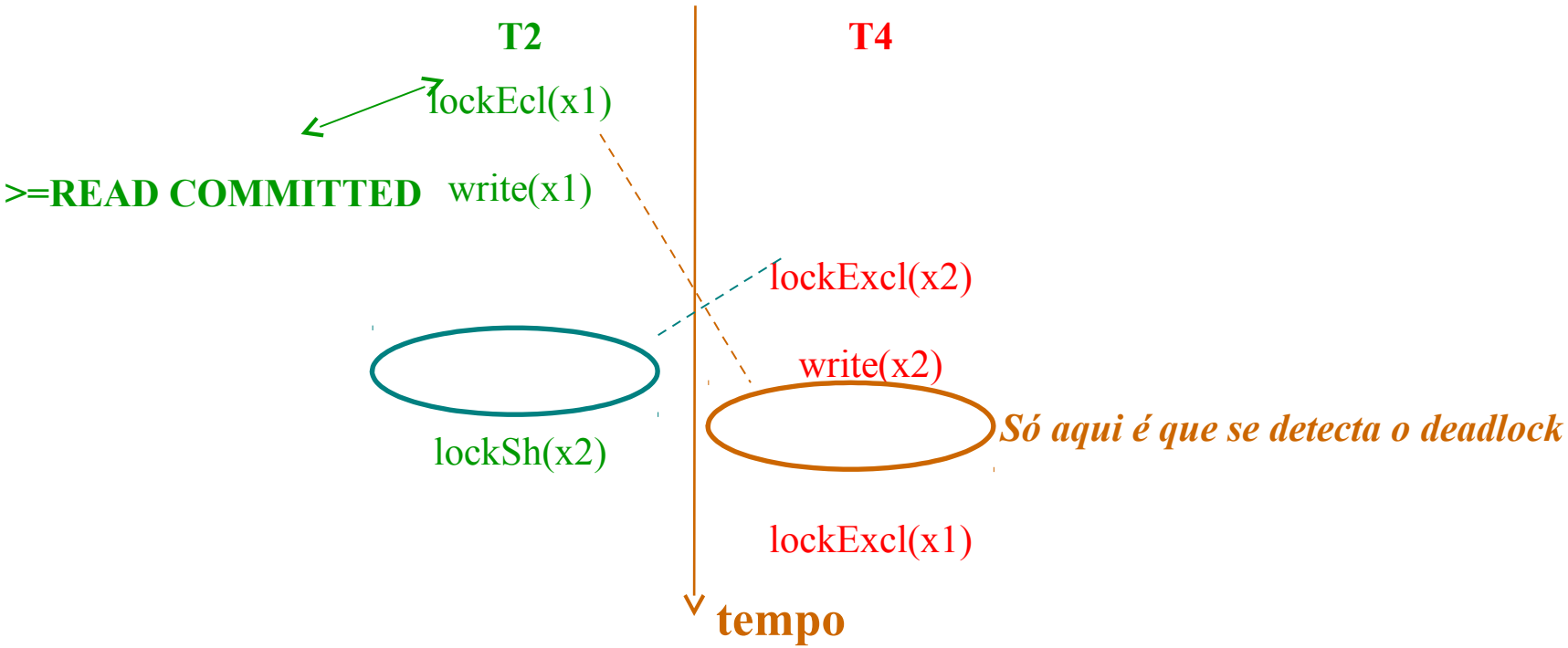
3. Envolva T1 e apenas uma outra das transações e origine uma situação de deadlock.



Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 2)

T2 = write(x1),read(x2),c
T4 = write(x2), write(x1),c

4. **Envolva apenas T2 e T4 e origine uma situação de deadlock.**



Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 2)

T1 = read(x1),write(x2),read(x1),c
T2 = write(x1),read(x2),c
T3 = write(x1),c
T4 = write(x2), write(x1),c
T5 = read(x3), write(x4),c
T6 = write(x1), write(x2),c

5. **Envolva T5 e apenas uma outra das transações e origine uma situação de dirty read.**

Não existe, pois T5 não partilha dados com nenhuma das outras transações
Neste caso, todos os escalonamentos são "cascadeless".

Em geral, para N ações da transação T5 e K ações de outra transação, teremos:

$$C_K^{N+K} = C_N^{N+K}$$ **escalonamentos diferentes sem situações de dirty reads.**

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 2)

T4 = write(x2), write(x1),c
T6 = write(x1), write(x2),c

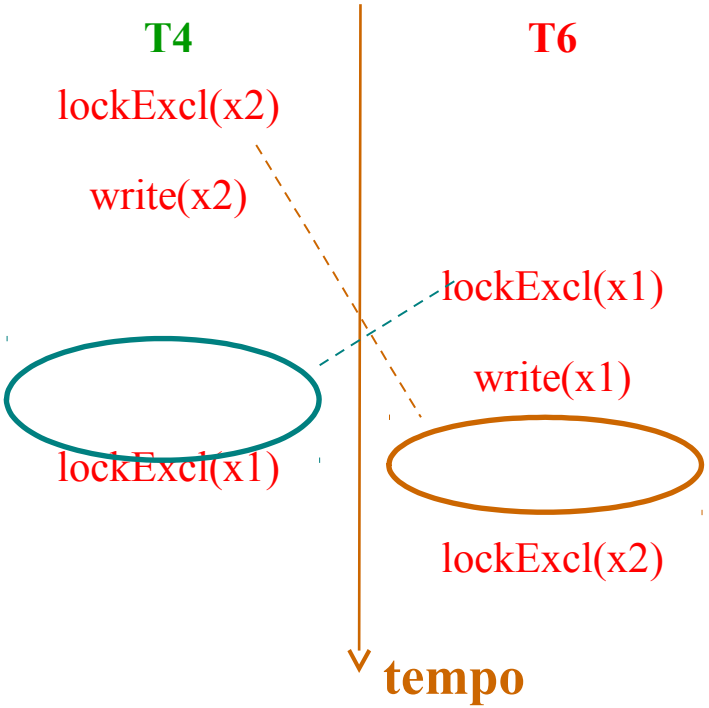
6. **Envolva T6 e T4 e origine uma situação de dirty read.**

Não existe, pois ambas as transações apenas realizam escritas

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 2)

T4 = write(x2), write(x1),c
T6 = write(x1), write(x2),c

7. **Envolva T6 e T4 e origine uma situação de deadlock.**



Dado que apenas existem escritas, verifica-se com qualquer nível de isolamento

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 3)

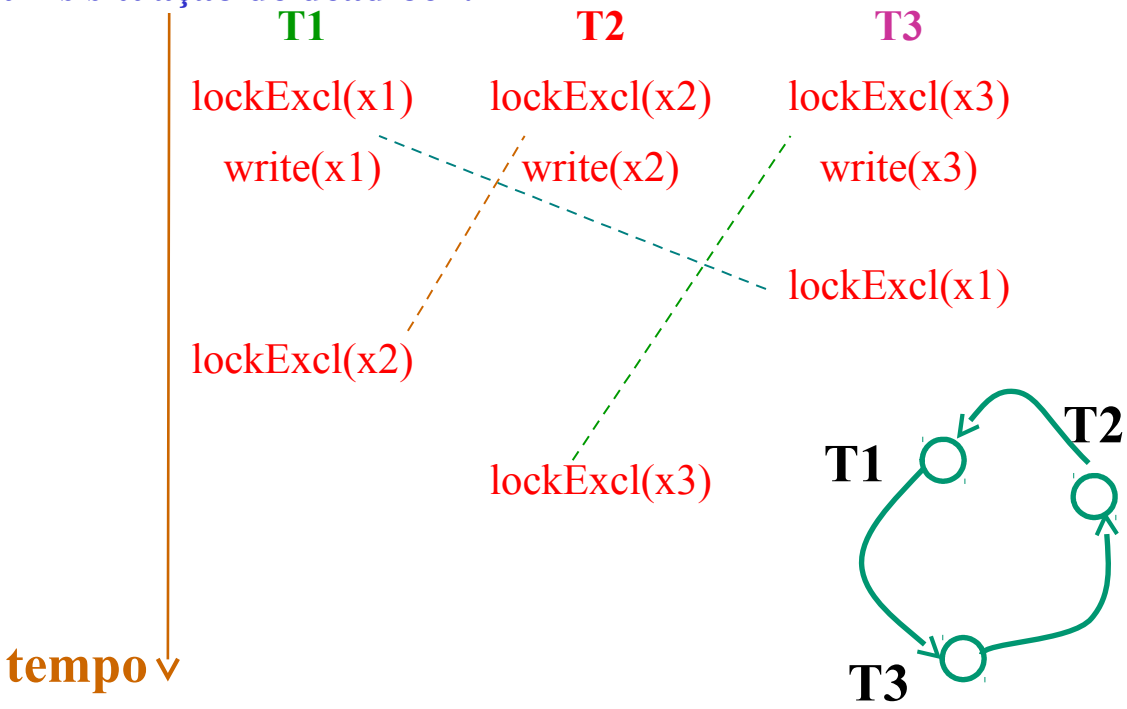
Considere as seguintes transações:

T1 = write(x1),write(x2),c
T2 = write(x2),write(x3),c
T3 = write(x3),write(x1),c

Diga se existe algum escalonamento que envolve ações de todas ou algumas destas transações que conduza a ums situação de deadlock.

É fácil verificar que qualquer dos escalonamentos que apenas envolvam ações de duas das transações não provocam deadlock porque não existem ações conflituantes cruzadas.

Mas o escalonamento da figura provoca uma situação de deadlock



Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 4)

Teoricamente, sem *predicate locking* ou algo equivalente, o fenómeno *phanton tuples* pode manifestar-se das seguintes duas formas:

1. **Aparecimento de registos novos entre dois percursos de uma tabela com a mesma condição WHERE, devido à realização de inserções ou actualizações de registos.**

2. **Desaparecimento de registos entre dois percursos de uma tabela com a mesma condição WHERE , devido à realização de remoções ou actualizações de registos.**

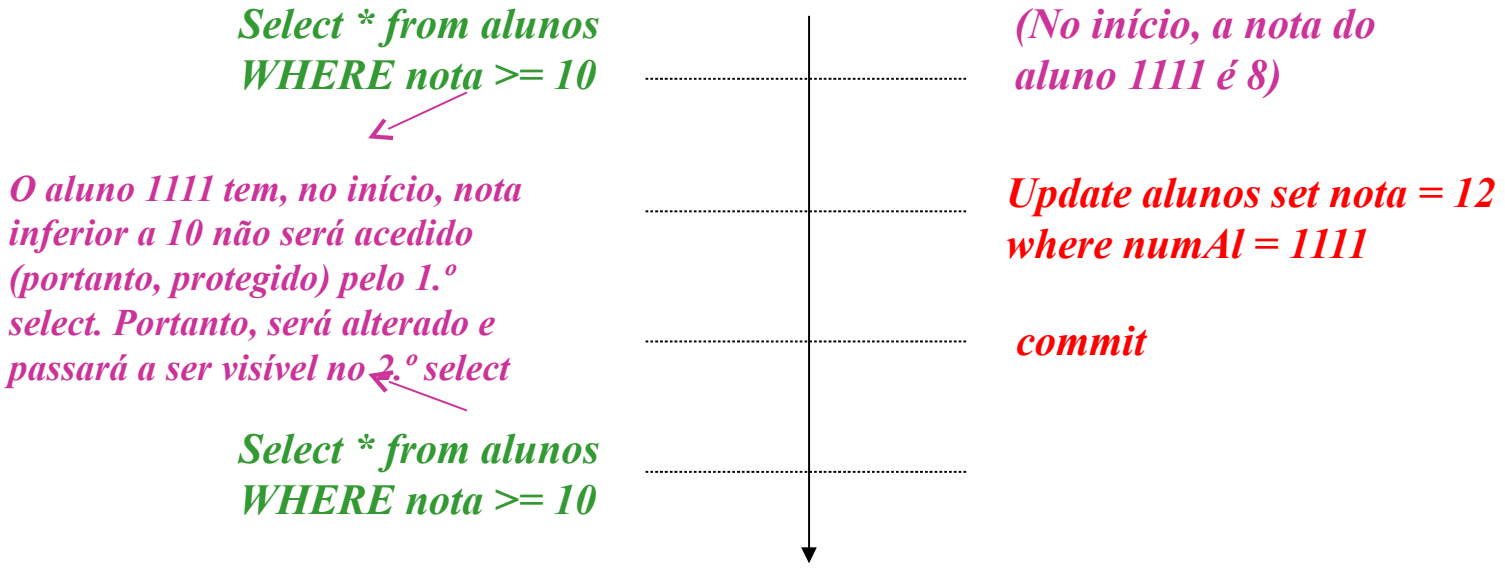
Para cada uma das formas, explique, detalhadamente, como ela se manifesta na norma ISO, para as várias operações que a podem originar.

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 4)

1. **Podem ocorrer com ambas as operações:**

   a) **Com inserções ocorrem pelo facto de os registos novos não poderem ser protegidos (por não existirem);**

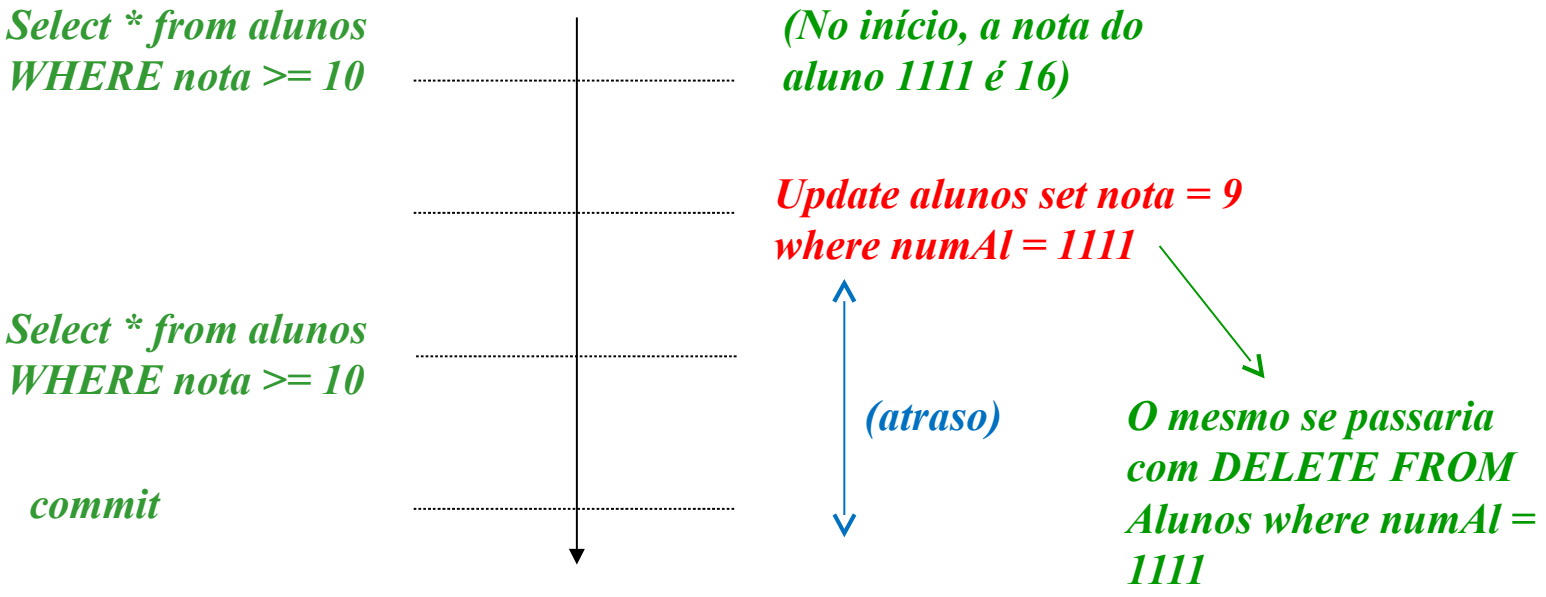
   b) **Com actualizações ocorrem <u>APENAS</u> se um registo não acedido numa leitura, por não satisfazer a condição WHERE, for alterado entretanto e passar a satisfazer a condição WHERE.**



Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 4)

2. **Desaparecimento de registos entre dois percursos de uma tabela com a mesma condição WHERE , devido à realização de remoções ou actualizações de registos.**

Esta situação não ocorre na norma se o nível de isolamento for REPEATABLE READ, dado que cada aluno acedido é protegido por um lock shared que apenas é libertado no fim da transação que o acede.



Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 5)

Sabendo que a tabela alunos apenas tem os alunos 1111 e 2222 e estes alunos têm, respectivamente, 6 e 3 inscrições, diga quantos shared locks irão potencialmente existir com a seguinte instrução:

```
Select * from alunos as a
INNER JOIN
(select numAl, count(*) as nInscr from Inscr
                                    group by numAl having count(*) > 4) as I
on a.numAl = i.numAl
```

Admitindo que não existirá escalamento de locks, o número de locks será:

- 9 para acesso à tabela Inscr, dado que ela tem de ser percorrida para cada aluno
- 1 para acesso ao aluno de número 1111 na tabela alunos que é o que tem mais do que 4 inscrições (o aluno 2222 não é acedido dado que a query marcada a verde não produz resultados)

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 6)

Sabendo que a tabela alunos apenas tem os alunos 1111 e 2222 e estes alunos têm, respectivamente, 6 e 3 inscrições, diga quantos shared locks irão potencialmente existir com a seguinte instrução:

```
Select * from alunos
where
(select count(*) from Inscr where Inscr.numAl = alunos.numAl) > 4
```

Admitindo que não existirá escalamento de locks, o número de locks será:

- 9 para acesso à tabela Inscr, dado que ela tem de ser percorrida para cada aluno
- 2 para acesso ao aluno 1111 e 2222, dado que, o uso da subquery correlacionada, obriga ao percurso de todos os alunos para se poder passar o numero de cada um para a subquery

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 7)

Sabe-se que a tabela alunos apenas tem os alunos 1111 e 2222 e que a tabela Inscr foi preenchida da seghuinte forma:

```
insert into inscr values(1111,'si1')
insert into inscr values(1111,'si2')
insert into inscr values(1111,'lc ')
insert into inscr values(1111,'aed')
insert into inscr values(1111,'se1')
insert into inscr values(1111,'se2')
insert into inscr values(2222,'si1')
insert into inscr values(2222,'si2')
insert into inscr values(2222,'aed')
```

Diga quantos shared locks e excusive locks irão potencialmente existir com a instrução:

```
update alunos set nomeAl = 'xx' where numAl = 1111
    and (select count(*) from Inscr where numAl = 1111) > 4
```

Admitindo que não existirá escalamento de locks, o número de locks será:

- 1 exclusive lock sobre o aluno 1111
- 6 shared locks para acesso à tabela Inscr, dado que ela tem 6 registos relativos ao aluno 1111

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 8)

Sabe-se que a tabela alunos apenas tem os alunos 1111 e 2222 e que a tabela Inscr foi preenchida da seghuinte forma:

```
insert into inscr values(1111,'si1')
insert into inscr values(1111,'si2')
insert into inscr values(1111,'lc ')
insert into inscr values(1111,'aed')
insert into inscr values(1111,'se1')
insert into inscr values(1111,'se2')
insert into inscr values(2222,'si1')
insert into inscr values(2222,'si2')
insert into inscr values(2222,'aed')
```

Diga quantos shared locks e excusive locks irão potencialmente existir com a instrução:

```
update alunos set nomeAl = 'xx' where numAl = 1111
    and exists (select * from Inscr where Inscr.numAl = alunos.numAL)
```

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 8)

```
insert into inscr values(1111,'si1')
insert into inscr values(1111,'si2')
insert into inscr values(1111,'lc ')
insert into inscr values(1111,'aed')
insert into inscr values(1111,'se1')
insert into inscr values(1111,'se2')
insert into inscr values(2222,'si1')
insert into inscr values(2222,'si2')
insert into inscr values(2222,'aed')
```

```
update alunos set nomeAl = 'xx' where numAl = 1111
    and exists (select * from Inscr where Inscr.numAl = alunos.numAL)
```

Admitindo que não existirá escalamento de locks, o número de locks será:

- 1 exclusive lock sobre o aluno 1111
- Para os shared locks a situação é mais complexa, dependendo da forma como o exists for realizado. Como basta apenas testar a existência de um registo, bastará realizar um shared lock sobre esse, mas poderão ser mais (todos os correspondentes ao aluno 1111, no pior caso). Portanto, MUITO CUIDADO!!!!

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 9)

Sabe-se que a tabela alunos apenas tem os alunos 1111 e 2222 e que a tabela Inscr foi preenchida da seghuinte forma:

```
insert into inscr values(1111,'si1')
insert into inscr values(1111,'si2')
insert into inscr values(1111,'lc ')
insert into inscr values(1111,'aed')
insert into inscr values(1111,'se1')
insert into inscr values(1111,'se2')
insert into inscr values(2222,'si1')
insert into inscr values(2222,'si2')
insert into inscr values(2222,'aed')
```

Diga quantos shared locks e excusive locks irão potencialmente existir com a instrução:

```
update alunos set nomeAl = 'xx' where
   not exists (select * from Inscr where Inscr.numAl = alunos.numAL)
```

Tal como no exercício anterior, para o not exists basta apenas testar a existência de um registo, pelo que bastará realizar um shared lock sobre esse, mas poderão ser mais (todos os correspondentes a ambos os alunos, no pior caso).
Existirão zero exclusive locks.

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 10)

Sabendo que a base de dados BD_CP é read only, defina o nível de isolamento adequado para a seguinte transação (que tem como objectivo criar numa outra base de dados – a corrente – tabelas com os códigos postais relativos aos distritos de Lisboa e Porto):

```
SET TRANSACTION  ISOLATION LEVEL ?
Begin transaction
 insert into CP_lisboa select * from BD_CP.dbo.CodPostal WHERE Distrito = 'LISBOA'
 insert into CP_Porto select * from BD_CP.dbo.CodPostal WHERE Distrito = 'PORTO'
COMMIT
```

Dado que a base de dados BD_CP é read only, os seus dados podem ser considerados sempre estáveis (as alterações terão de ocorrer de forma especial).

Assim sendo, o nível de isolamento adequado é READ UNCOMMITTED

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 11)

Indique qual o nível de isolamento mais adequado para as transações que executem o seguinte código:

```
SET TRANSACTION ISOLATION LEVEL ?
BEGIN TRANSACTION
If Exists(select * from Alunos Where NumAl = 1111)
Begin
    insert into AlunosAEISEL values(1111,'caloiro')
End
Commit
```

Se o nível de isolamento for read uncommitted podem surgir dois problemas;

1. **Se outra transação concorrente apagar o aluno 1111 antes do EXISTS mas depois abortar estas pensam (erradamente) que o aluno não existe.**
2. **Se outra transação concorrente introduzir o aluno 1111 antes do EXISTS mas depois abortar, estas pensam (erradamente) que o aluno existe**

Assim sendo, o nível de isolamento adequado é READ COMMITTED. Admitindo que existe uma chave estrangeira de AlunosAEISEL para Alunos, se outra transação apagar o aluno 1111, a inserção dará um erro, pelo que não é necessário o nível de isolamento REPEATABLE READ. Também se poderia pensar que seria necessário o nível de isolamento SERIALIZABLE para protecção contra inserções na tabela Alunos com o número 1111, depois de se avaliar o exists com false e antes de commit, mas o resultado (não inserção e AlunosISEL) seria igual ao resultado do escalonamento série T1T2, onde T1 executa este código e T2 realiza a inserção do aluno 1111.

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 12)

Diga qual o nível de isolamento adequado para o seguinte código:

```
create proc getY @x int, @y float output
as
  select @y = y from t where x = @x

create proc putY @x int, @y float output
as
  update t set y = @y where x = @x


set transaction isolation level ?
begin tran
declare @v float
exec getY 1, @v output
set @v = @v+500
exec putY 1, @v
commit
```

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 12)

Aparentemente, bastaria o nível de isolamento READ COMMITTED, mas com esse nível de isolamento, poderiam ocorrer lost updates. Se, por exemplo, duas transações concorrentes executassem o mesmo código, o resultado poderia ser diferente do resultado de qualquer dos escalonamentos série (se o valor inicial de y fosse 1000, o valor final nessa coluna deveria ser 2000)

Para evitar essa situação, o nível de isolamento adeqaudo é REPEATABLE READ.

É a única situação em que uma transação não depende de si própria para se proteger. Neste caso se uma transação concorrente (por exemplo, com as mesmas ações) correr com nível de isolamento inferior a REPEATABLE READ pode observar-se o efeito "lost update".

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 13)

Diga qual o nível de isolamento adequado para o seguinte código:

```
CREATE PROC P @tab varchar(2), @c int
As
begin

  if exists(select * from t1 where c1 = @c)
      if @tab = 't2'
         insert into t2 values(@c)
      else
         if @tab = 't3'
              insert into t3 values(@c)
end

SET TRANSACTION ISOLATION LEVEL ?
Begin transaction
 exec p 't2', 1111
 exec p 't3', 1111
commit
```

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 13)

Depreende-se do código que se pretende (usando o SP P) realizar algo equivalente a:

```
SET TRANSACTION ISOLATION LEVEL ?
Begin transaction
If exists (select * from t1 where c1 = 1111)
Begin
    insert into t2 values(1111)
    insert into t3 values(1111)
End
Commit
```

Se for este o caso, com a imposição do uso do SP P, então o nível de isolamento deveria ser, pelo menos, REPEATABLE READ, pois, de outro modo, entre a primeira e segunda execuções do SP alguém pode apagar o registo existente em t1.
No entanto, mesmo com repeatable read, também se pode verificar um resultado indesejável se a tabela t1 não contiver o registo com c1 = 1111 e entre as duas execuções do SP, outra transção o inserir.

Assim, o nível de isolamento adequado é SERIALIZABLE

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 13)

Interessante notar que:

1. **Com o nível de isolamento READ COMMITED o resultado produzido seria o mesmo do código:**

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
Begin transaction
If exists (select * from t1 where c1 = 1111)
      insert into t2 values(1111)
If exists (select * from t1 where c1 = 1111)
      insert into t3 values(1111)
Commit
```

2. **Com o nível de isolamento REPEATABLE READ, o resultado seria o mesmo do código:**

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
Begin transaction
Declare @i int = 0
If exists (select * from t1 where c1 = 1111)
Begin
    insert into t2 values(1111)
    insert into t3 values(1111)
    set @i = 1
End
If exists (select * from t1 where c1 = 1111) and @i = 0
     insert into t3 values(1111)
Commit
```

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 14)

O SP seguinte realiza a inserção de um novo produto (na tabela cuja criação também se indica) apenas se se o seu preço não for inferior a 60% da média dos preços dos produtos já existentes:

```
CREATE TABLE Produto (
      cod varchar(10) PRIMARY KEY,
      preco float NOT  NULL,
      NumOrdem int identity
)

CREATE PROC  InsProduto(@cod varchar(10), @preco float)
AS
begin
  declare @m float
  set transaction isolation level ?
  begin tran
  select @m = avg(preco) from produto
  if (@m is null OR @preco >= 0.6*@m)
      insert into Produto values(@cod,@preco)
  commit
End
```

Indique qual o nível de isolamento que deve ser utilizado no interior do SP

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 14)

Suponhamos que a tabela tem inicialmente os seguintes valores:

Cod	Preço	NumOrdem
P0	1	1

Suponhamos as transações t1 e t2 a executarem, respectivamente exec InsProd 'P1', 1 e exec InsProd 'P2', 3.

Os escalonamentos série produzem os seguintes resultados:

T1T2

Cod	Preço	NumOrdem
P0	1	1
P1	1	2
P2	3	3

0.6*média = 0.6

3 >= 0.6

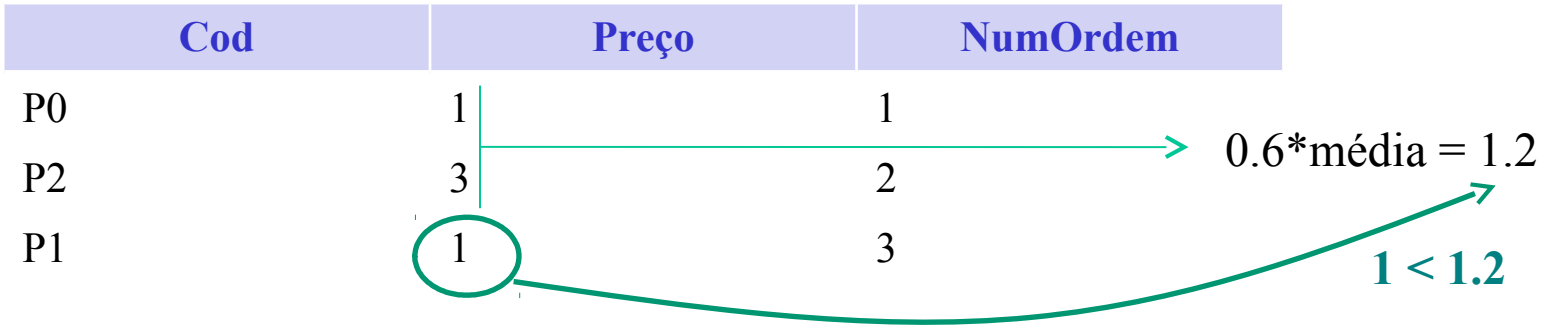
T2T1

Cod	Preço	NumOrdem
P0	1	1
P2	3	2

0.6*média = 1.2

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 14)

Mas se for possível que ambas as transações calculem a média apenas quando existe P0, ambas irão inserir os seus produtos, por uma ordem não determinista, sendo possível obter-se:



Que não corresponde aos resultados produzidos por nenhum dos escalonamentos série.

Se o nível de isolamento for SERIALIZABLE, ambas as transações ficarão bloqueadas (isto é, existirá um deadlock sobre o "predicate lock") e uma delas será abortada.
Se for a t1, obtemos o resultado da série T2T1, mesmo que posteriormente se volte a executar T1, ela não fará a inserção.
Se for a t2 e a voltarmos a executar, o resultado obtido será o da série T1T2.

Notar, no entanto, que se não existisse a coluna identity, com nível REPEATABLE READ obteríamos o resultedo de uma das séries, mas a solução ficaria muito dependente da estrutura da tebela.

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 15)

Considere as duas transações seguintes:

T1:

```
set transaction isolation level
                        REPEATABLE READ
begin tran
declare @v float
select @v = y from t where x = 1
set @v = @v+500
update t set y = @v where x = 1
commit
```

T2:

```
set transaction isolation level
                        REPEATABLE READ
begin tran
declare @v float
select @v = y from t where x = 1
set @v = @v+1000
update t set y = @v where x = 1
commit
```

Discuta a serializabilidade dos escalonamentos possíveis entre estas transações.

Transacções com 2PL e níveis de isolamento SQL 2011 (exercício 15)

Considere as duas transações seguintes:

T1:

```
set transaction isolation level
                    REPEATABLE READ
begin tran
declare @v float
select @v = y from t where x = 1
set @v = @v+500
update t set y = @v where x = 1
commit
```

T2:

```
set transaction isolation level
                    REPEATABLE READ
begin tran
declare @v float
select @v = y from t where x = 1
set @v = @v+1000
update t set y = @v where x = 1
commit
```

Os dois escalonamentos série T1T2 e T2T1 produzem o mesmo resultado.
Os pares de ações mutuamente conflituantes estão marcadas com cor verde e vermelho.
Qualquer escalonamento onde uma das transações execute o *update* antes da outra realizar o *select* é serializável.
Se ambas transações realizarem o *update* depois da outra ter realizado *select* e antes desta realizar commit existe uma situação de deadlock (que indica que o escalomento conjunto das transações não pode ser seriálizável). Mas, como uma das transações é abortada, obtém-se um escalonamento serializável, por apenas envolver uma transação. A repetição da outra transação produzirá um escalonamento série.
Se todas as transações concorrentes usarem um nível de isolamento igual ou superior a REPEATABLE READ, os escalonamentos resultantes são serializáveis, no sentido antes exposto.

Transacções – início e fim

Início:

- **Implícitas (ISO SQL 1992)**

  No SQL-Server 2012:

  SET IMPLICIT_TRANSACTIONS {ON|OFF}

- **Explícitas**

  Em ISO SQL (1999 – 2011):

  START TRANSACTION [<transaction characteristics>]

  No SQL-Server 2012 **: BEGIN TRAN[SACTION] [*nome*]**

Terminação:

- **Com AUTO-COMMIT**
- **ROLLBACK [WORK]**

  (também ROLLBACK TRAN[SACTION [nome]) no SQL Server 2012)
- **COMMIT [WORK]**

  (também COMMIT TRAN[SACTION [nome]) em SQL Server 2012)

Transacções – início e fim

No Sql Server com SET IMPLICIT TRANSACTIONS ON, as seguintes instruções sobre a sessão iniciam uma transação, se não existir uma em curso:

CREATE, DROP, ALTER TABLE,

SELECT, INSERT, DELETE, UPDATE,

OPEN, FETCH,

BEGIN TRANSACTION,

GRANT , REVOKE,

TRUNCATE TABLE

Transacções –controlo de concorrência baseado em timestamps

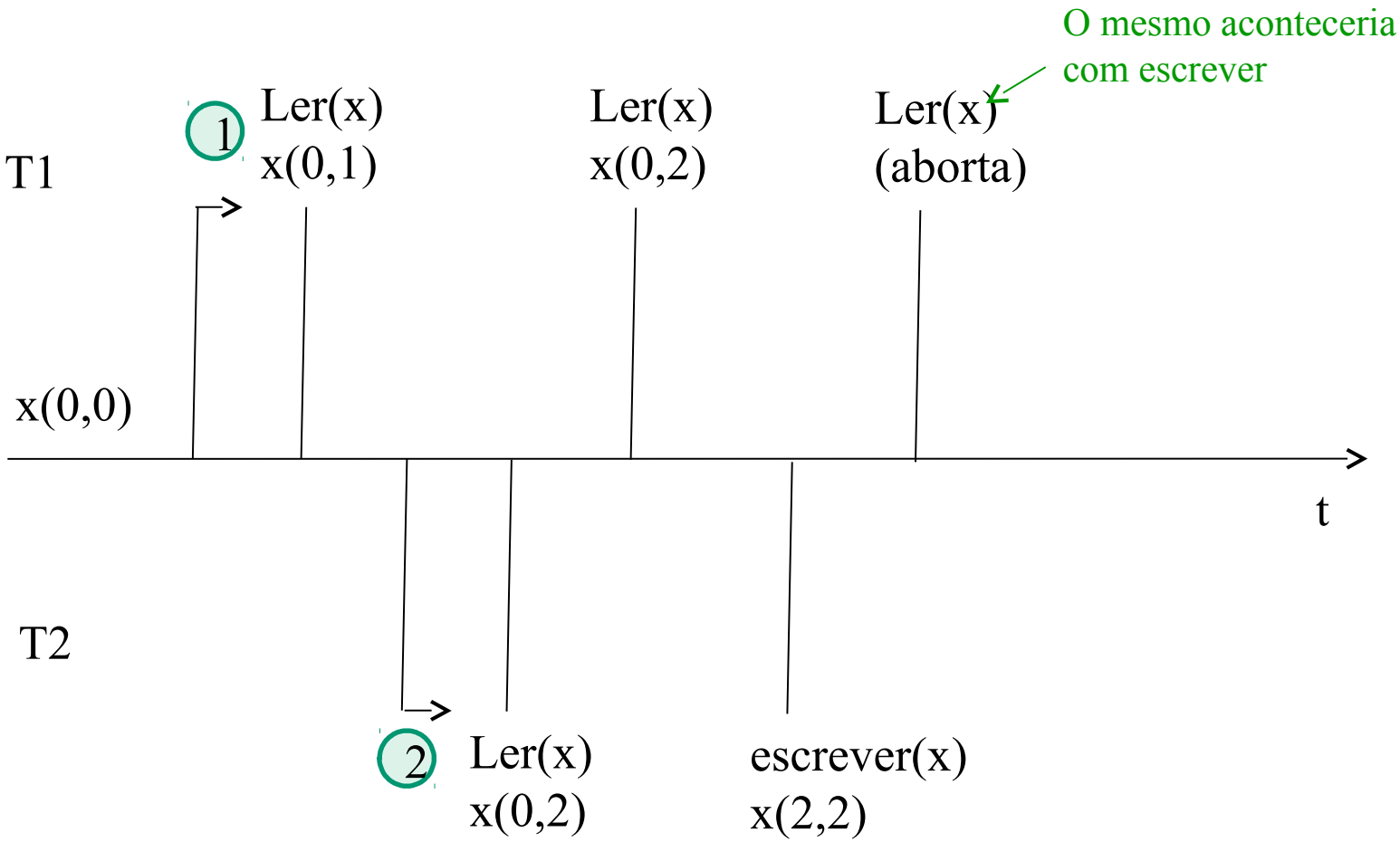
A cada transacção (T) é associado um timestamp dependente do tempo em que foi criada (ts(T)).
Cada item (X) tem associados os timestamps das últimas transacções que o acederam para leitura e escrita (tr(X) e tw(X)), os quais são usados para ordenar os acessos aos itens.

```
READ (T,X)
Se tw(X) <= ts(T) então
     realizar leitura;
    tr(X) ← MAX(tr(X), ts(T))
Senão
    abortar T
Fim READ
```

```
WRITE(T,X)
Se (tw(X) <= ts(T)) e (tr(X) <= ts(T)) então
   realizar escrita;
   tw(X) ← ts(T)
Senão
   abortar T
Fim WRITE
```

Naturalmente, os escalonamentos gerados são sempre serializáveis.
Mas geram escalonamentos não *cascadeless* e não recuperáveis (devido a leituras de dados não validados). Uma variante (*strict t.s. ordering*) consiste em atrasar as transacções que acedam a itens escritos por transacções anteriores não terminadas, produzindo escalonamentos estritos e serializáveis do ponto de vista de conflito.

Transacções –controlo de concorrência baseado em timestamps



Transacções –controlo de concorrência baseado em versões e t.s.

São mantidas várias versões dos itens à medida que estes vão sendo alterados. Por exemplo, pode manter-se para cada item X um conjuntgo de versões X1, ..., Xn, cada uma associada a timestamps (tr(Xi) e tw(Xi)) da última transacção que sobre ele realizou uma leitura e da transacção que lhe deu origem (escrita).

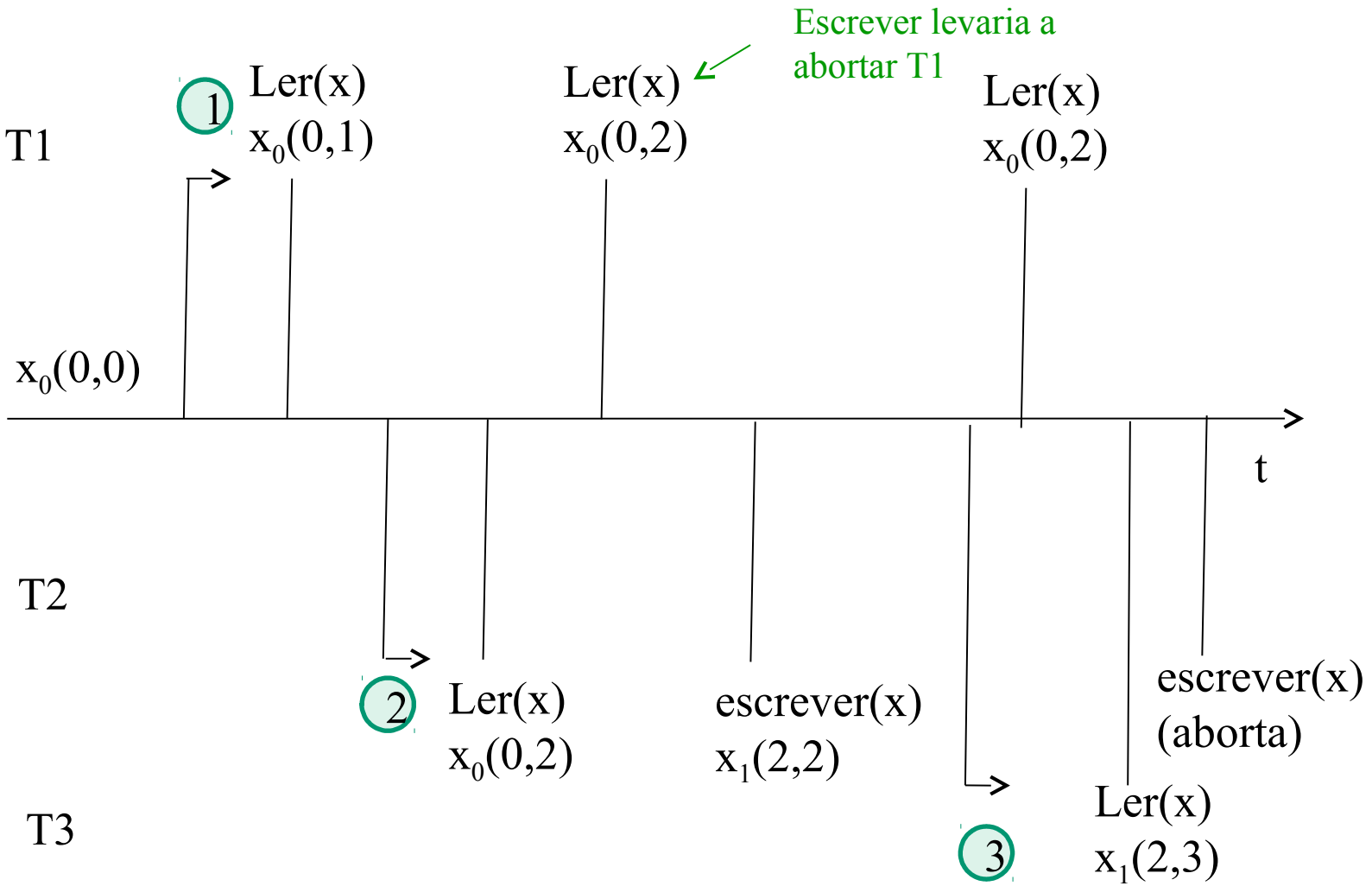
```
WRITE(T, X)
    i ← índice da última versão de X
    Enquanto tw(Xi) > ts(T) fazer
        i ← índice da versão anterior de i;
    Se tr(Xi) >  ts(T) então
        Abortar;
    Senão
        Executar escrita, inserindo nova versão (k),
            após a versão i;
        tw(Xk) ← tr(Xk) ← ts(T)
Fim WRITE
```

```
READ (T, X)
        i ← índice da última versão de g
        Enquanto tw(Xi) > ts(T) fazer
            i ← índice da versão anterior de i;
        Executar leitura sobre versão i;
        tr (Xi) ← MAX(tr(Xi), ts(T))
        Fim READ
```

Problemas:

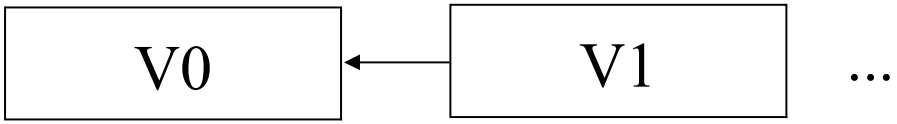
- Espaço gasto com as versões
- Escalonamentos não recuperáveis e não cascadeless (leituras de dados não validados). Uma possível solução é atrasar o commit de tr. dependentes de versões não validadas.

Transacções –controlo de concorrência baseado em timestamps



Transacções – multi-versões em Sql Server

O Sql Server (a partir da versão 2005) introduziu o nível de isolamento SNAPSHOT, o qual constitui uma alternativa aos níveis de isolamento READ COMMITTED, REPEATABLE READ e SERIALIZABLE, mas sem o uso de "shared locks". Para isso, usa-se uma variante dos sistemas multiversão só para leitura.
Cada item pode ter várias versões (validadas) alojadas na tempdb. A cada versão é associado o timestamp da sua criação.



Quando a alteração a uma linha é validada (commited) é criada uma nova vesão através da qual se tem acesso às anteriores. A cada versão associa-se o tempo de criação (tw(Vi)). Mas o valor actual é sempre guardado definitivamente na BD.
Se uma transacção T (iniciada no tempo ts(T)) faz um acesso para leitura à linha, percorrem-se todas as versões em tempdb à procura da primeira cujo valor tw(Vi) é menor ou igual a ts(T). É desta versão que T lê os dados.
Se uma transacção que já fez acesso a uma das versões tentar actualizar o item quando já existem versões mais recentes, é abortada.

Não é necessário manter-se um valor tr(Vi) por cada versão porque não se alteram as versões (são só para leitura).
Escalonamentos sempre recuperáveis (versões estão validadas)

Transacções – multi-versões em Sql Server

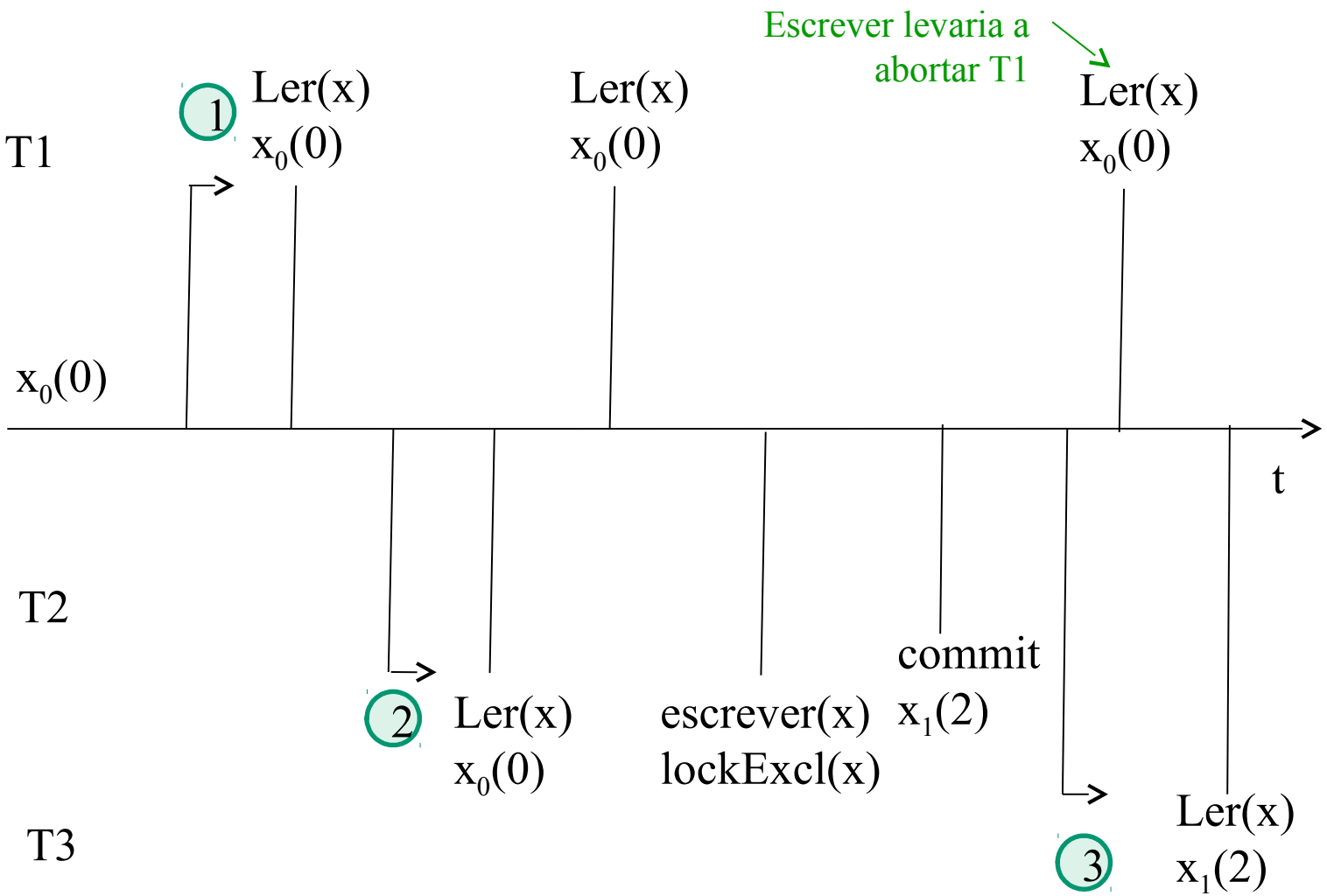
snapshot isolation level
(visão simplificada)

```
READ (T, X)
    i ← índice da última versão de g
    Enquanto tw(Xi) > ts(T) fazer
        i ← índice da versão anterior de i;
    Executar leitura sobre versão i;
    Fim READ
```

```
WRITE(T, X)
    i ← índice da última versão de X
    Se tw(Xi) > ts(T) então
        Abortar;
    Senão
        Alterar X (com lock excl.)
Fim WRITE
```

```
COMMIT(T,X):
...
Para cada registo do log na forma [write_item,T,X,oldV,newV]
    Criar nova versão (Xk) de X e fazer tw(Xk) ← ts(T)
```

Transacções –controlo de concorrência baseado em timestamps



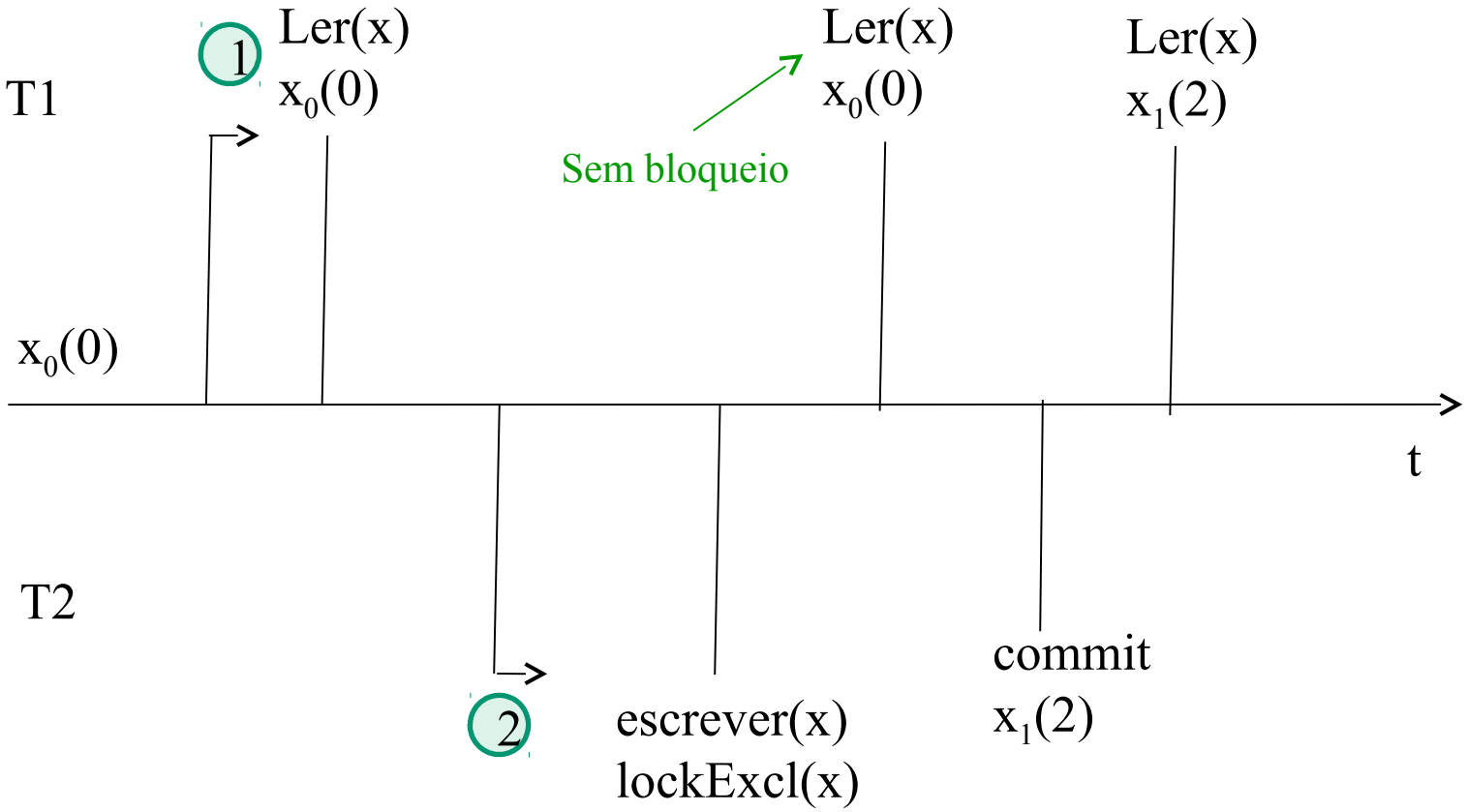
Transacções – multi-versões em Sql Server 2005

Existe também a variante SNAPSHOT do nível READ COMMITED no qual é cada operação de leitura que vê sempre a versão mais recente dos dados já validados, mas podem ocorrer non-repeatable reads (READ COMMITED). O SGBD associa à transacção um timestamp que vai variando e que é igual ao timestamp mantido pelo SDBD na altura em que se inicia a operação.

Em snapshot o timestamp da transacção era constante e igual ao timestamp mantido pelo SGBD na altura do lançamento da operação (ou da execução da sua primeira operação).

Snapshot permite níveis de consistência dos dados equivalentes a repeatable read sem bloqueio.

Transacções –controlo de concorrência baseado em timestamps



Diferimento de teste de restrições

<domain definition> ::=
CREATE DOMAIN <domain name> [AS] <predefined type>
[<default clause>] [<domain constraint>...] [<collate clause>]

<domain constraint> ::=
[<constraint name definition>] <check constraint definition>
[*<constraint characteristics>*]

<column constraint definition> ::=
[<constraint name definition>] <column constraint>
[*<constraint characteristics>*]

<table constraint definition> ::=
[<constraint name definition>] <table constraint>
[*<constraint characteristics>*]

Diferimento de teste de restrições

<constraint name definition> ::=
CONSTRAINT <constraint name>

<check constraint definition> ::=
CHECK <left paren> <search condition> <right paren>

<constraint characteristics> ::=
<constraint check time> [[NOT] DEFERRABLE] [<constraint enforcement>]
| [NOT] DEFERRABLE [<constraint check time>] [<constraint enforcement>]

Diferimento de teste de restrições

<constraint check time> ::=
INITIALLY DEFERRED | INITIALLY IMMEDIATE (por omissão)
(define o valor no início de cada transacção)

IMMEDIATE – testada no fim de cada instrução
DEFERRED – testada quando voltar a ser IMMEDIATE (implicita ou explicitamente)
INITIALLY DEFERRED não pode coexistir com NOT DEFERRABLE

Diferimento de teste de restrições

<set constraints mode statement> ::=
SET CONSTRAINTS <constraint name list> { DEFERRED | IMMEDIATE }

<constraint name list> ::=
ALL
| <constraint name> [{ <comma> <constraint name> }...]

válida apenas para a transacção corrente, se existir, ou, caso não exista, para a próxima transacção

A instrução commit implica SET CONSTRAINTS ALL IMMEDIATE implícito para a transacção corrente

Diferimento de teste de restrições

```
begin transaction
create table tr1 (i int primary key, j int)
create table tr2 (i int primary key, j int constraint c1 references tr1(i))
alter table tr1 add constraint c foreign key(j) references tr2(i)
commit

start transaction
set constraint c deferred
insert into tr1 values(1,2)
insert into tr2 values(2,1)
set constraint c immediate
insert into tr1 values(3,2)
commit

start transaction
set constraint c deferred
insert into tr1 values(5,6)
insert into tr2 values(6,5)
-- Set constraint c immediate – não necessário
commit
```

Diferimento de teste de restrições em SQL Server 2005

```
ALTER TABLE [ database_name . [ schema_name ] . | schema_name . ]
                                                        table_name
{
   <alteração de coluna>
   | <adição de coluna>
   | <remoção de coluna ou restrição>
   | [ WITH { CHECK | NOCHECK } ]
        { CHECK | NOCHECK } CONSTRAINT  { ALL | constraint_name [,...n ] }
   | ...
}
```

MUITO CUIDADO:

- **Não é o mesmo que SET CONSTRAINTS, pois a restrição nunca é testada, se usarmos with nocheck.**
- **Se não voltarmos a activar a restrição, ela nunca mais será testada.**
- **Com alter table ... with check check ... é feita a validação para os todos os dados já existentes**
- **Só funciona para restrições CHECK e FOREIGN KEY**

Diferimento de teste de restrições em SQL Server 2005

```
begin transaction
create table tr1 (i int primary key, j int)
create table tr2 (i int primary key, j int constraint c1 references tr1(i))
alter table tr1 add constraint c foreign key(j) references tr2(i)
commit

begin transaction
alter table tr1 nocheck constraint c
insert into tr1 values(1,2)
insert into tr2 values(2,1)
alter table tr1 with check check constraint c
insert into tr1 values(3,2)
commit

begin transaction
alter table tr1 nocheck constraint c
insert into tr1 values(5,6)
insert into tr2 values(6,5)
alter table tr1 with check check constraint c
commit
```

reparar nisto

é sempre necessário

SQL – XACT_ABORT em SQL Server

```
SET XACT_ABORT OFF;
delete from conta
BEGIN TRANSACTION;
INSERT INTO conta VALUES (7777,0); -- é inserido
INSERT INTO conta VALUES (7777,0); -- erro
INSERT INTO conta VALUES (8888,0); -- é inserido
COMMIT TRANSACTION;
```

```
SET XACT_ABORT ON;
delete from conta
BEGIN TRANSACTION;
INSERT INTO conta VALUES (7777,0); -- não é inserido
INSERT INTO conta VALUES (7777,0); -- Erro
INSERT INTO conta VALUES (8888,0); -- não é inserido
COMMIT TRANSACTION;
```

```
SET XACT_ABORT OFF;
delete from conta
INSERT INTO conta VALUES (7777,0);
BEGIN TRANSACTION;
INSERT INTO conta VALUES (9999,0); -- é inserido
UPDATE conta Set numero = 7777 WHERE numero=9999 -- erro
INSERT INTO conta VALUES (8888,0); -- é inserido
COMMIT TRANSACTION;
```

```
SET XACT_ABORT ON;
delete from conta
INSERT INTO conta VALUES (7777,0);
BEGIN TRANSACTION;
INSERT INTO conta VALUES (9999,0); -- não é inserido
UPDATE conta Set numero = 7777 WHERE numero=9999 -- erro
INSERT INTO conta VALUES (8888,0); -- não é inserido
COMMIT TRANSACTION;
```

SQL – XACT_ABORT em SQL Server

```
SET XACT_ABORT OFF;
delete from conta
INSERT INTO conta VALUES (7777,0);
BEGIN TRANSACTION;
INSERT INTO conta VALUES (9999,0); -- é inserido
UPDATE conta Set numero = 7777 WHERE numero=9999 -- erro
INSERT INTO conta VALUES (8888,0); -- é inserido
COMMIT TRANSACTION;
```

O modo XACT_ABORT OFF é mais versátil, dado que, se se executa o código dentro de um bloco begin try, um erro ("severity "igual ou superior a 11) transfere a execução para o bloco begin catch respectivo, onde podemos fazer o tratamento mais adequado (ROLLBACK, se for o caso).

```
SET XACT_ABORT OFF
delete from conta
INSERT INTO conta VALUES (7777,0);
BEGIN TRY
BEGIN TRANSACTION
INSERT INTO conta VALUES (9999,0); -- não é inserido
UPDATE conta Set numero = 7777 WHERE numero=9999 -- erro
INSERT INTO conta VALUES (8888,0); -- não é inserido porque não é executado
COMMIT TRANSACTION; END TRY
BEGIN CATCH
    ROLLBACK;
     throw
END CATCH
```

Bibliografia

Ramez Elmasri and Shamkant B. Navathe, Fundamentals of Database Systems, Addison Wesley

Jim Gray, Andreas Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufman, 1993

Philip A. Bernstein, Eric Newcomer, Principles of Transaction Processing for the Systems Professional, Morgan Kaufman, 1997

Microsoft SQL Server 2012 Books Online (http://technet.microsoft.com/en-us/library/ms130214.aspx)

ANSI/ISO/IEC International Standard (IS) Database Language SQL—Part 2: Foundation (SQL/Foundation) (ISO/IEC 9075-2:2011)