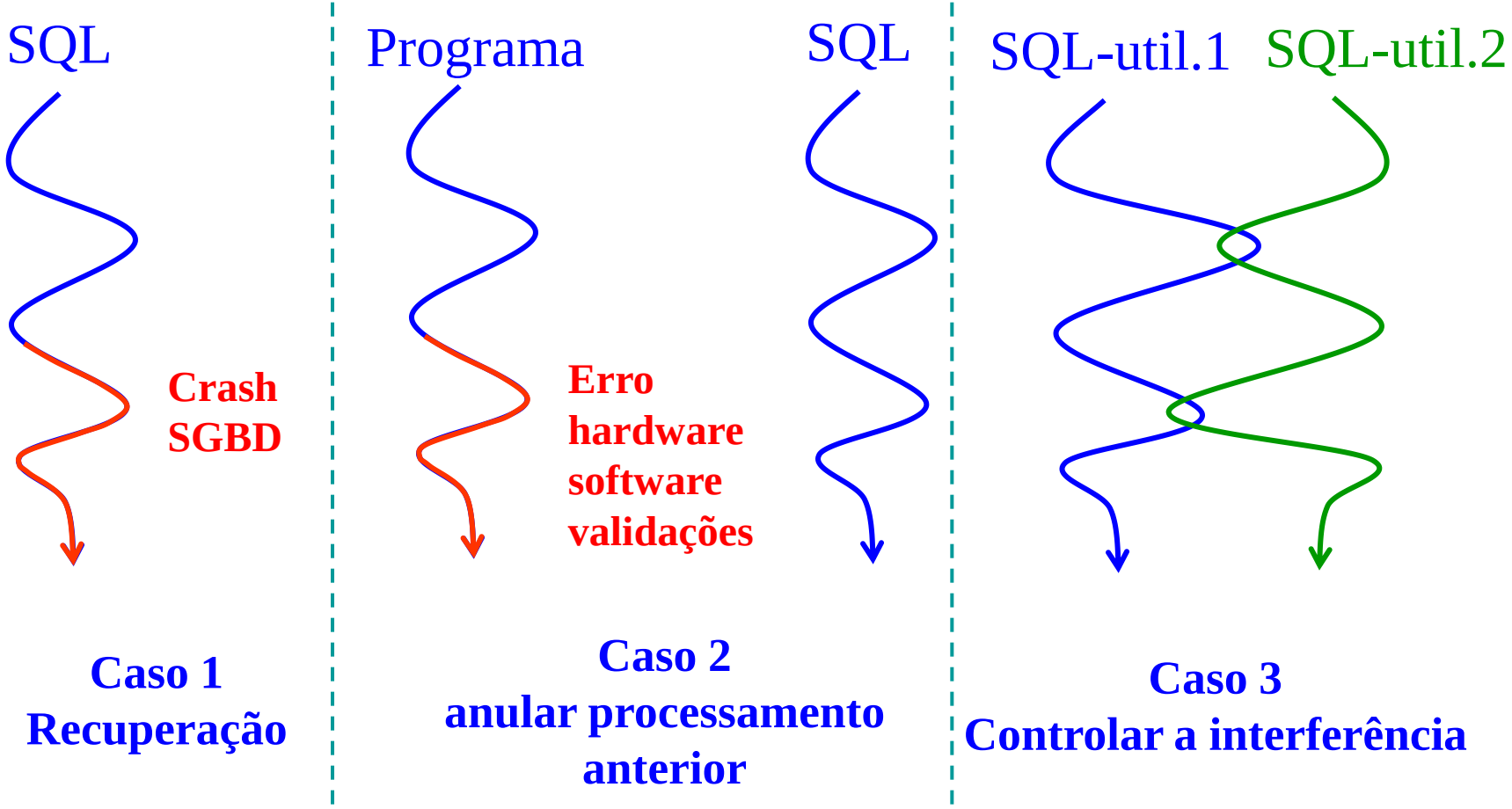
Processamento Transaccional

(1ª parte)

Transacções

Transacções porquê?



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)>
S2 = <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)>

Transacções – escalonamentos

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**

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, nehuma transacção terminada necessita de ser anulada quando outras transacções falham (em nenhuma circunstância, por exemplo, em caso de crash).

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 = <r1(t1,x1),w1(t1,x1),r2(t2,x1),r1(t1,x2),w2(t2,x1),a(t1),a(t2)>

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)>

mas é recuperável

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 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).

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)>

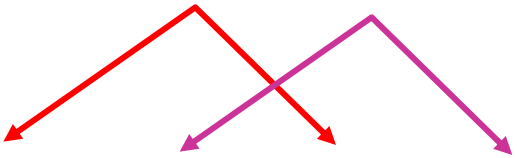
S2 = < r(t2,x1), w(t2,x1), c(t2), r(t1,x1),w(t1,x1), r(t1,x2), c(t1)>

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

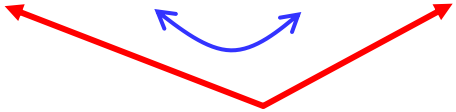
Sejam



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 nehum 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 "serie" possíveis com as transacções de S.

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)>

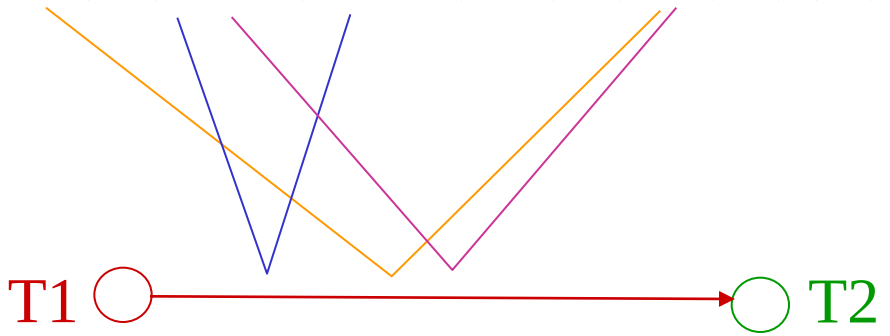
Transacções – grafos de precedências

Desenhar um vértice por cada transacção do escalonamento.

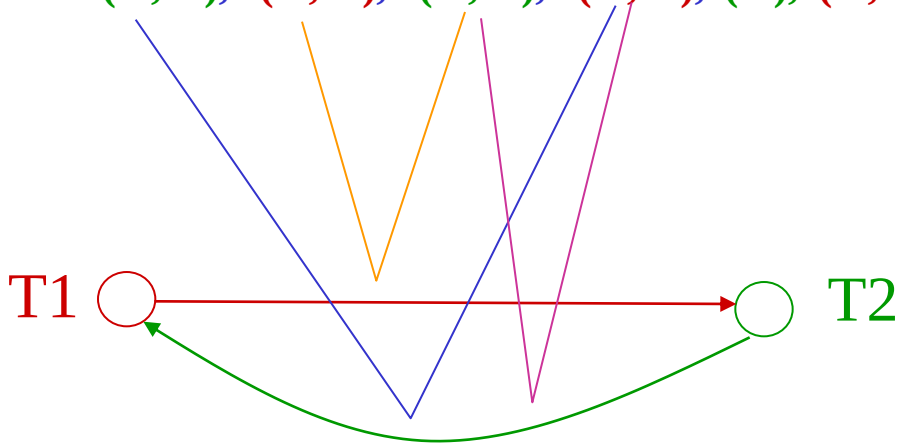
Por cada par conflituoso a1(Ti,x), a2(Tj,x) tal que a1 precede a2 desenhar um arco de Ti para Tj.

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)>



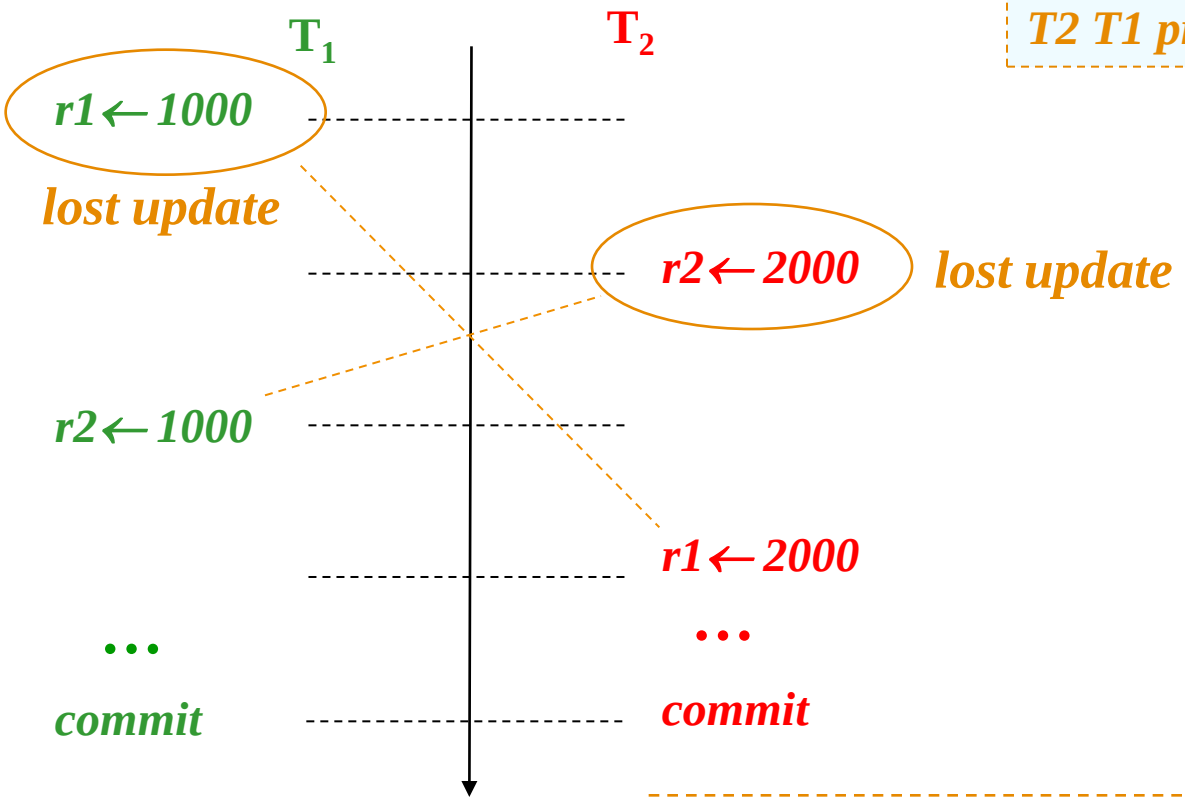
S5 = < r(t2,x1), r(t1,x1),w(t2,x1),w(t1,x1),c(t2),r(t1,x2),c(t1)>



Transacções e concorrência - Anomalias

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

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

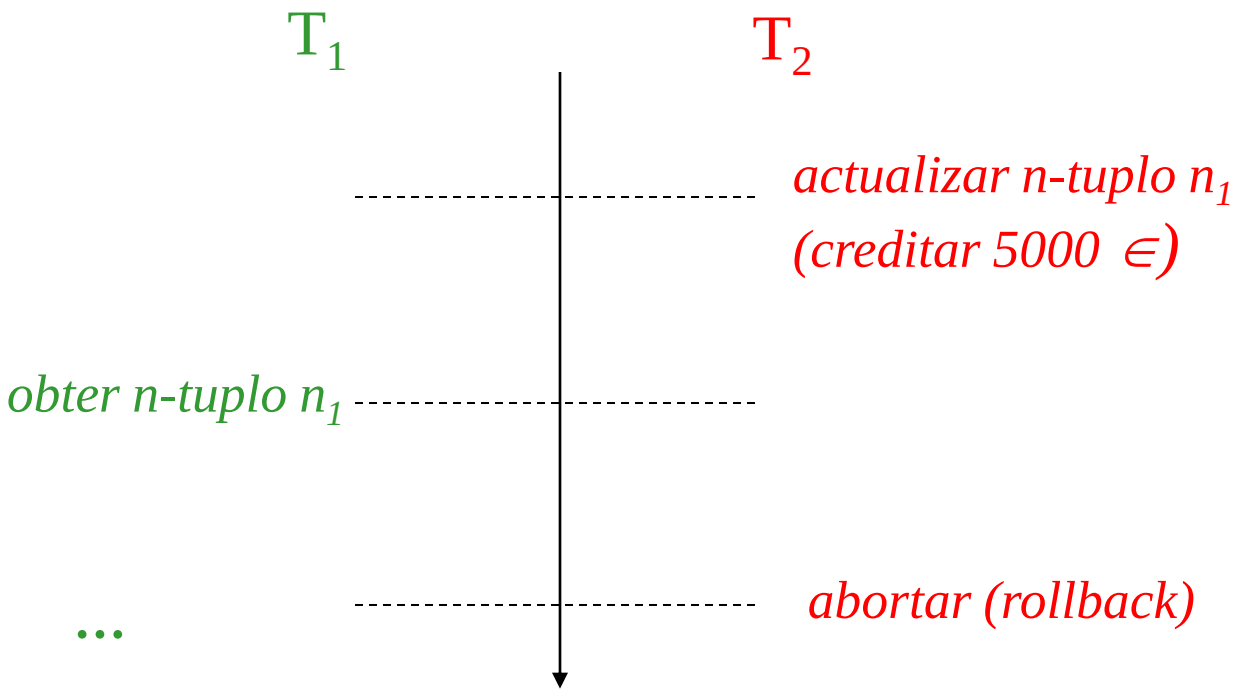


Não ocorre com o SQL 92

Escalonamentos não estritos

Transacções e concorrência - Anomalias

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

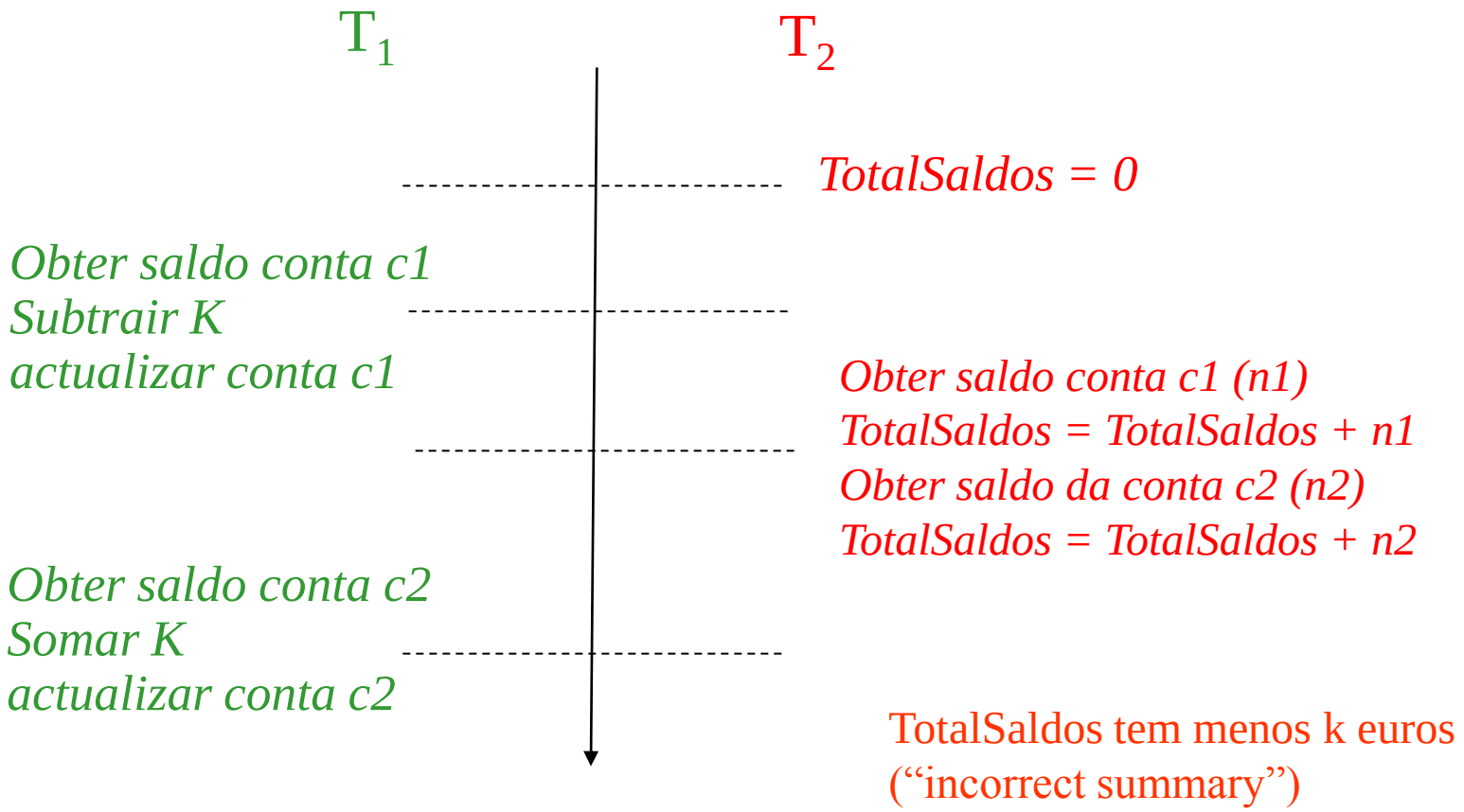


"dirty read" ou "temporary update"

Escalonamentos que exibem cascading rollback

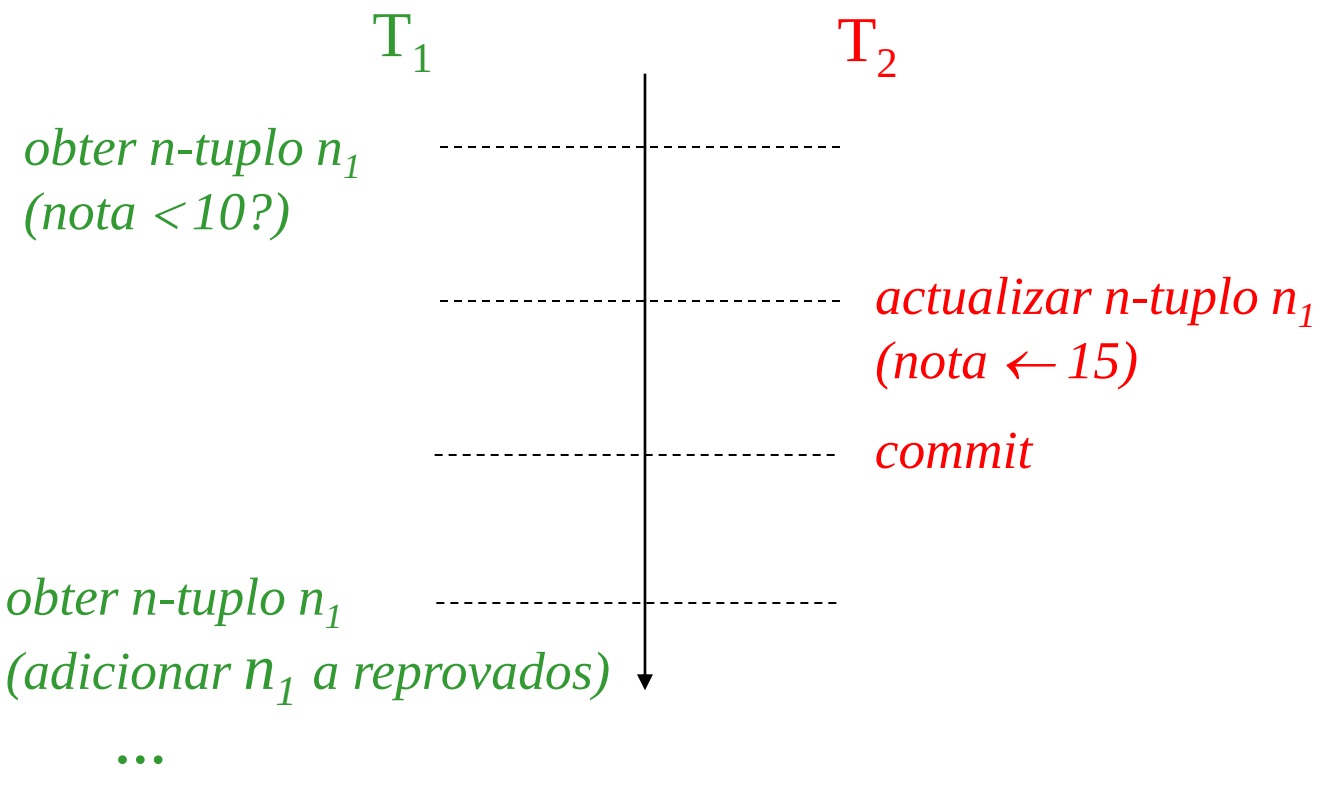
Transacções e concorrência - Anomalias

"uncommited dependency"



Transacções e concorrência - Anomalias

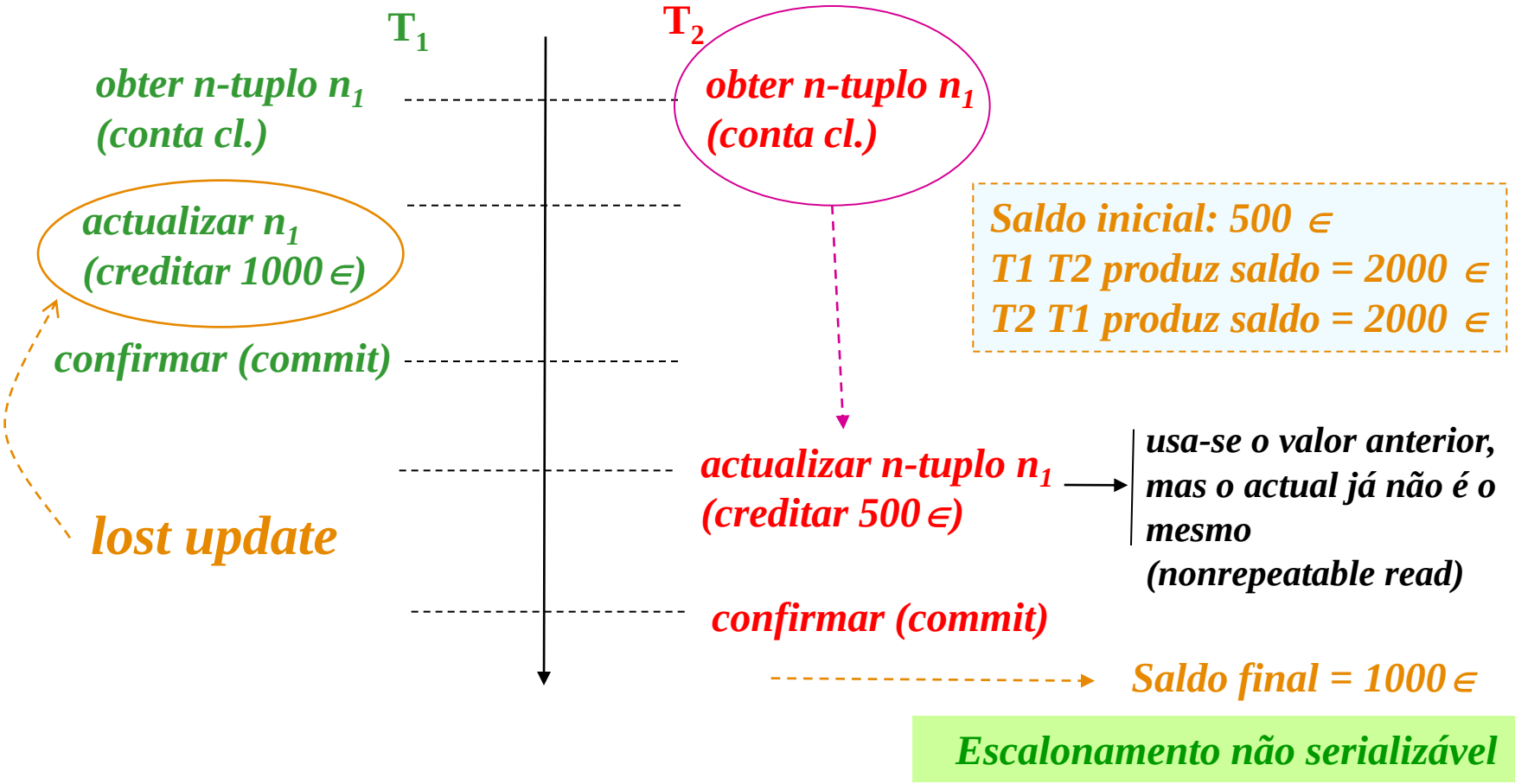
"nonrepeatable read", ou "inconsistent analysis" *(c. R/W)*



Escalonamento não serializável

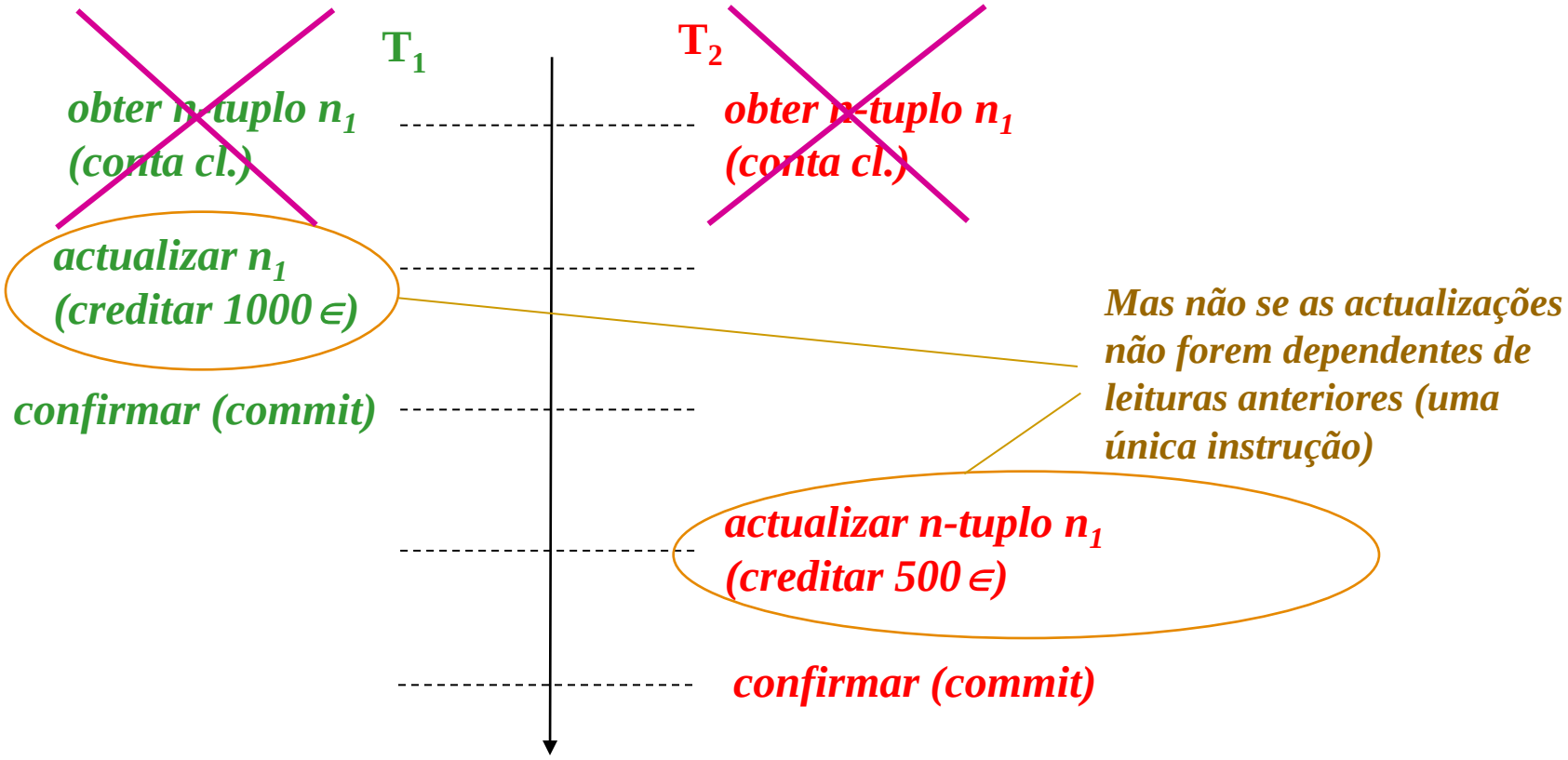
Transacções e concorrência - Anomalias

"nonrepeatable read" pode causar "lost updates"



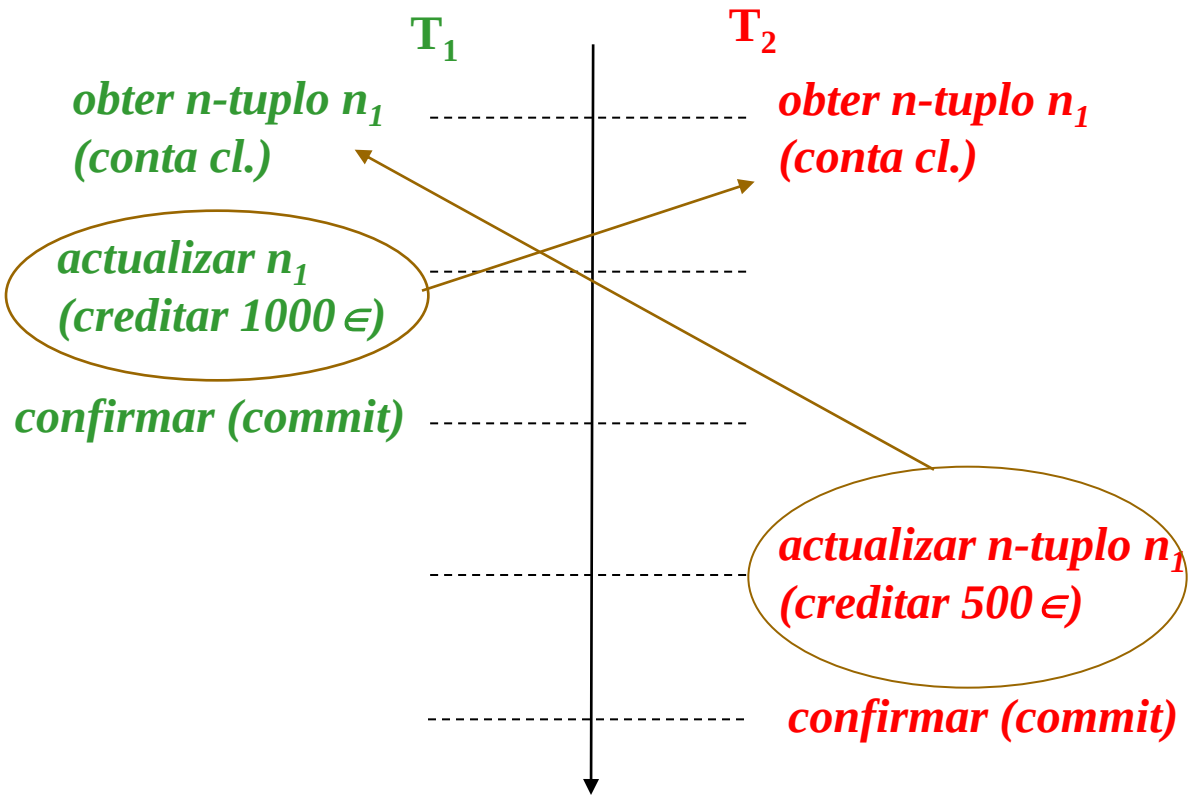
Transacções e concorrência - Anomalias

"nonrepeatable read" pode causar "lost updates"



Transacções e concorrência - Anomalias

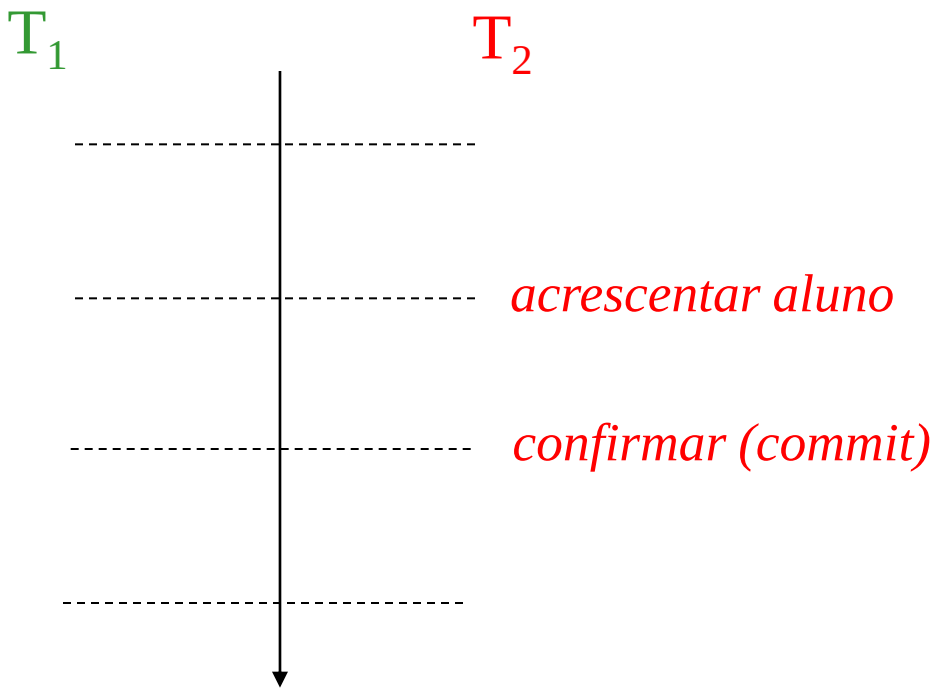
Com "repeatable read" teriamos um *deadlock*



Deadlock:
Uma das transacções seria abortada

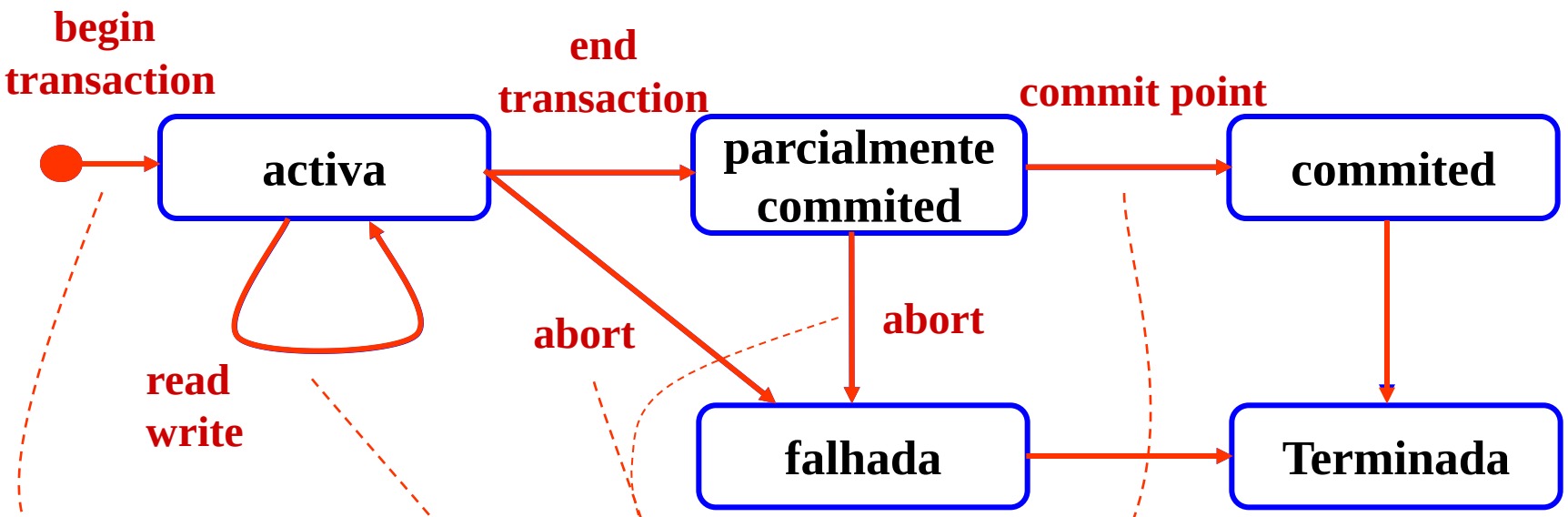
Transacções e concorrência - Anomalias

"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



Log:

[start_transaction, T]
[read_item,T,X]
ou [write_item,T,X,oldV,newV]
[abort,T]
ou [commit,T]

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

Falhada: a transacção vem para este estado se for abortada no seu estado activa ou se os testes realizados no estado parcialmente commited falharem

Terminada: a transação deixa de existir no sistema

Transacções – nível de isolamento em SQL92 e SQL Server

Modo de acesso

Nível de isolamento

```
SET TRANSACTION
  [READ ONLY | READ WRITE]
  [DIAGNOSTIC SIZE <número de condições>]
  [ISOLATION LEVEL
    {READ UNCOMMITED | READ COMMITED |
     REPEATABLE READ | SERIALIZABLE
    }
```

Também pode ser definido com a instrução START TRANSACTION

Válida apenas para a próxima transacção. Não se pode alterar o nível dentro de uma transacção

Em SQL Server 2005

```
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)

Transacções em SQL92 – 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**

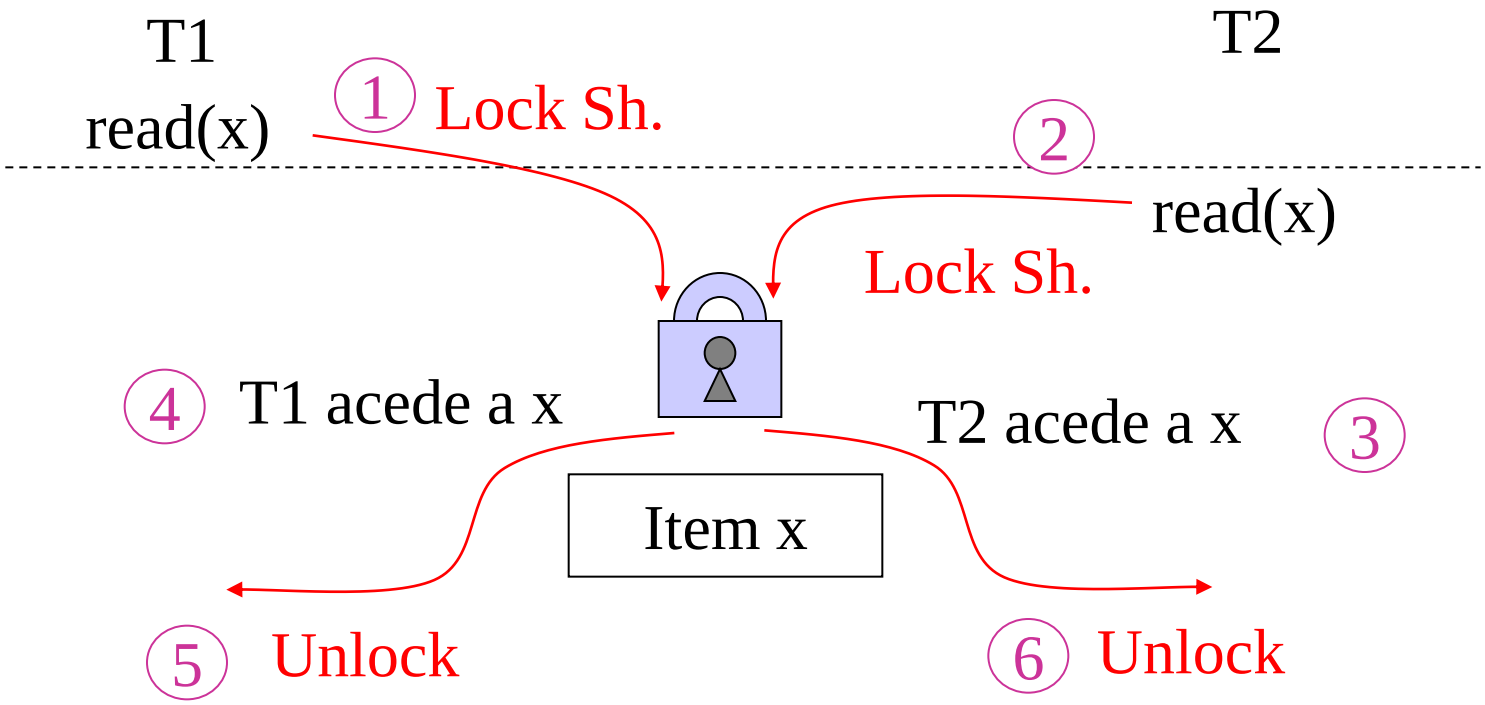
- **Uma transacção é sempre bem comportada relativamente às outras (write é bem formada e de duas fases), ou seja, não existe a anomalia *overwriting uncommited data*. O nível *read uncommited* só é possível com modo read only.**
- **Cada transacção protege-se das outras tanto quanto necessário, escolhendo 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)

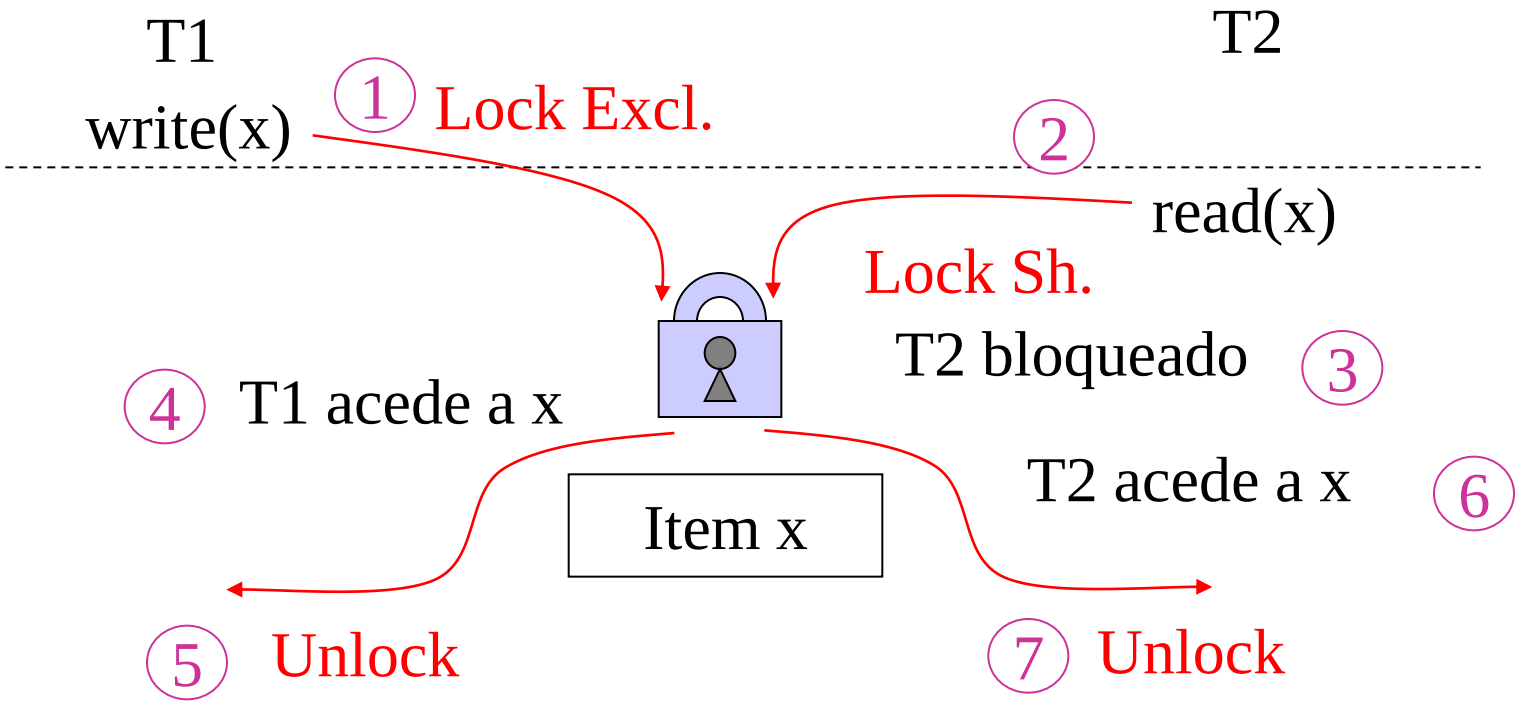
Matriz de compatibilidade

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

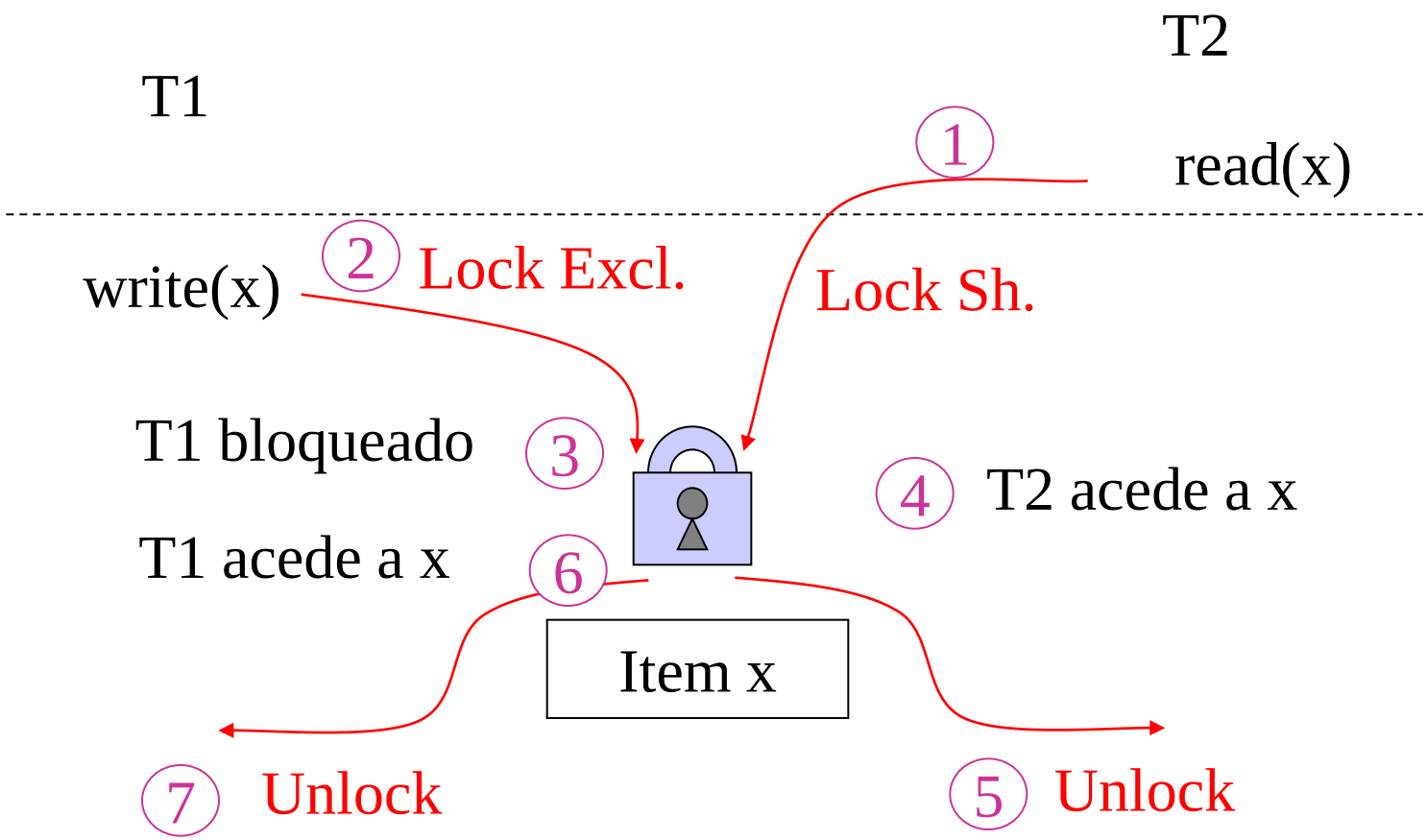
Transacções – o protocolo *two phase lock* (2pL)



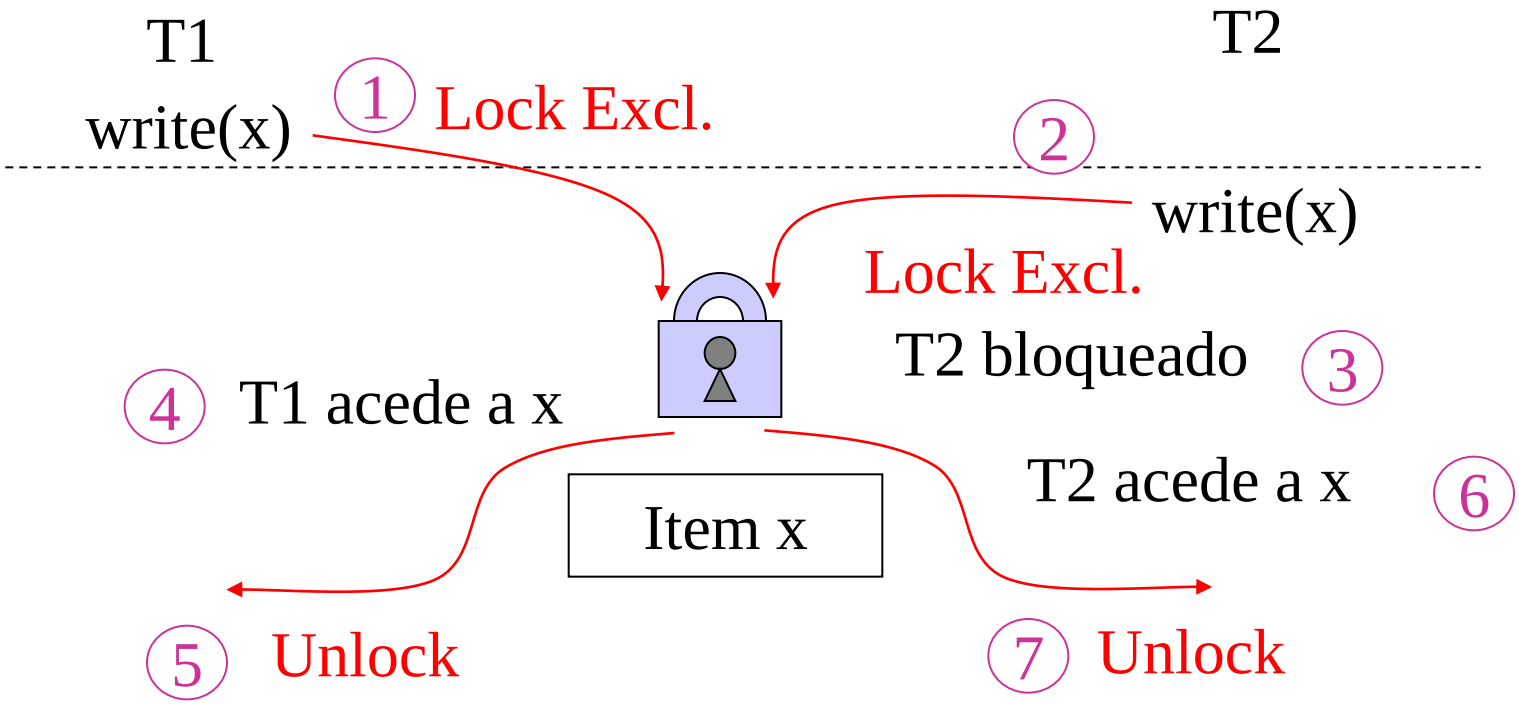
Transacções – o protocolo *two phase lock* (2pL)



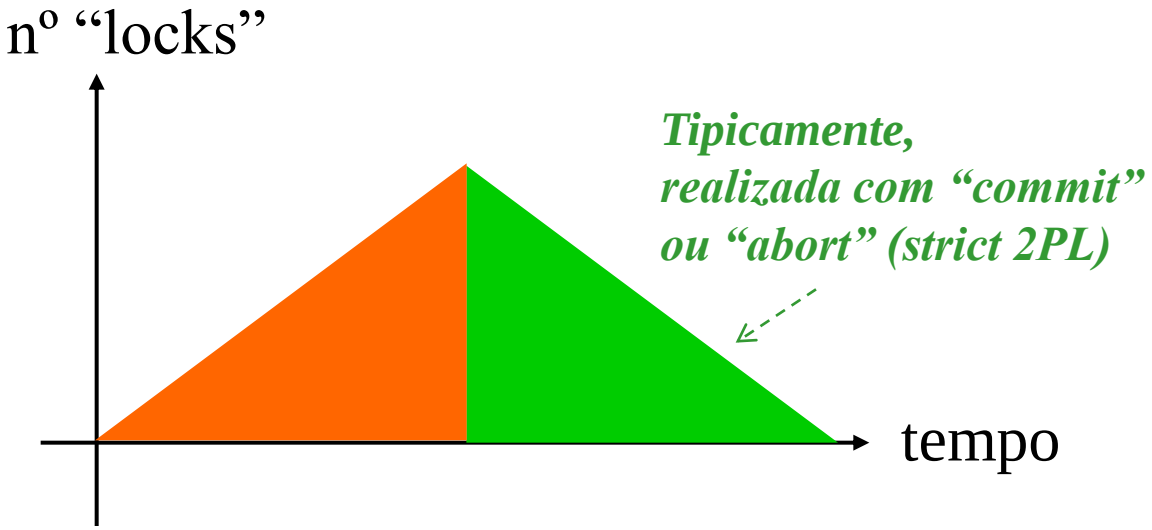
Transacções – o protocolo *two phase lock* (2pL)



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



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



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

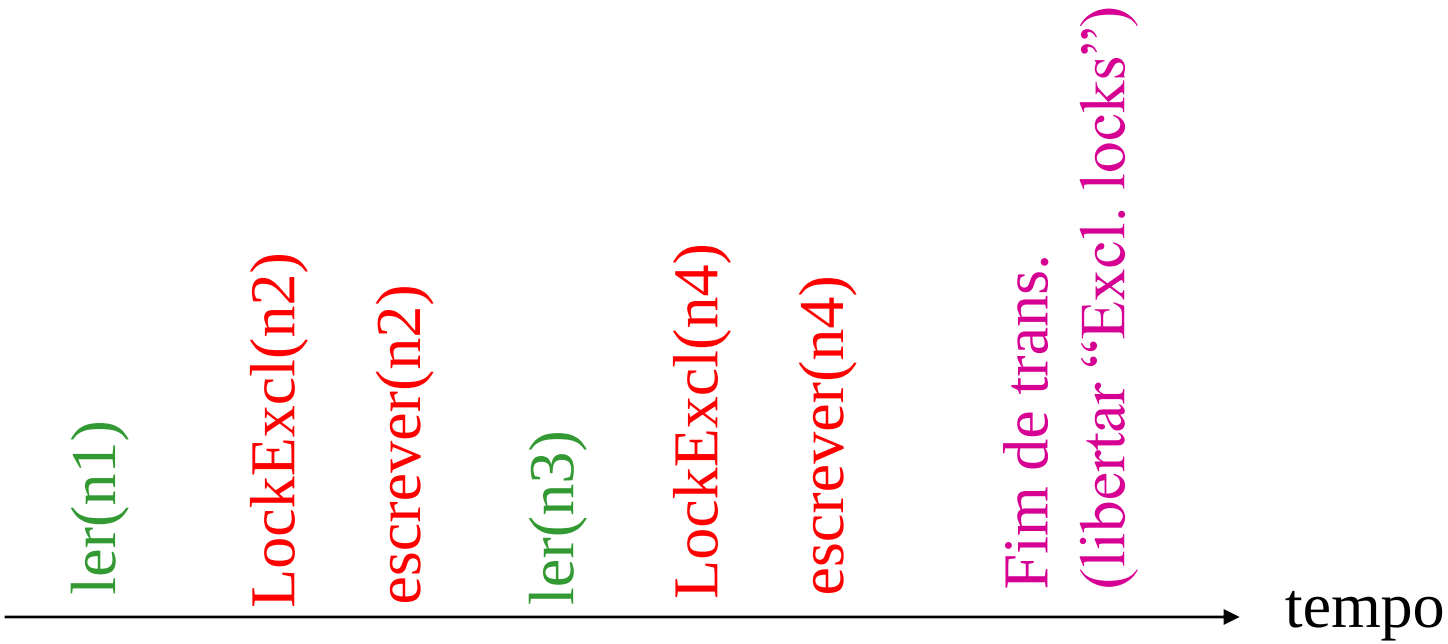
Acção de duas fases: não executa *unlock* antes de *locks* de outras acções da mesma transacção

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 92

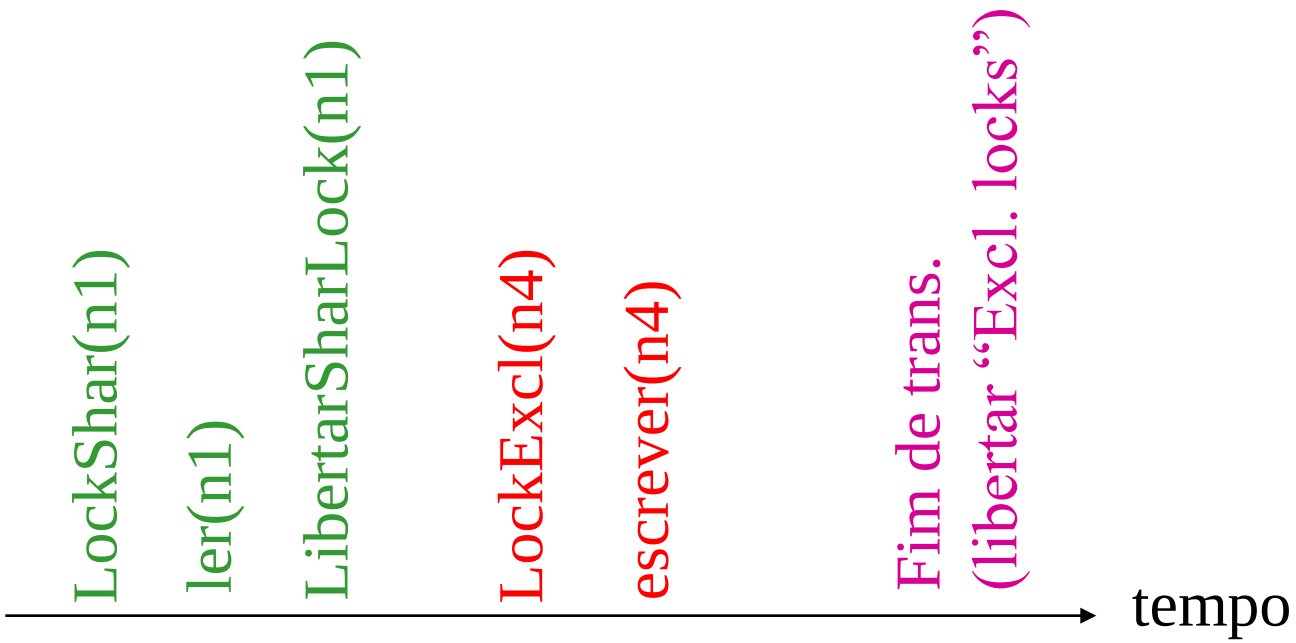
read uncommited



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

Transacções com 2PL e níveis de isolamento SQL 92

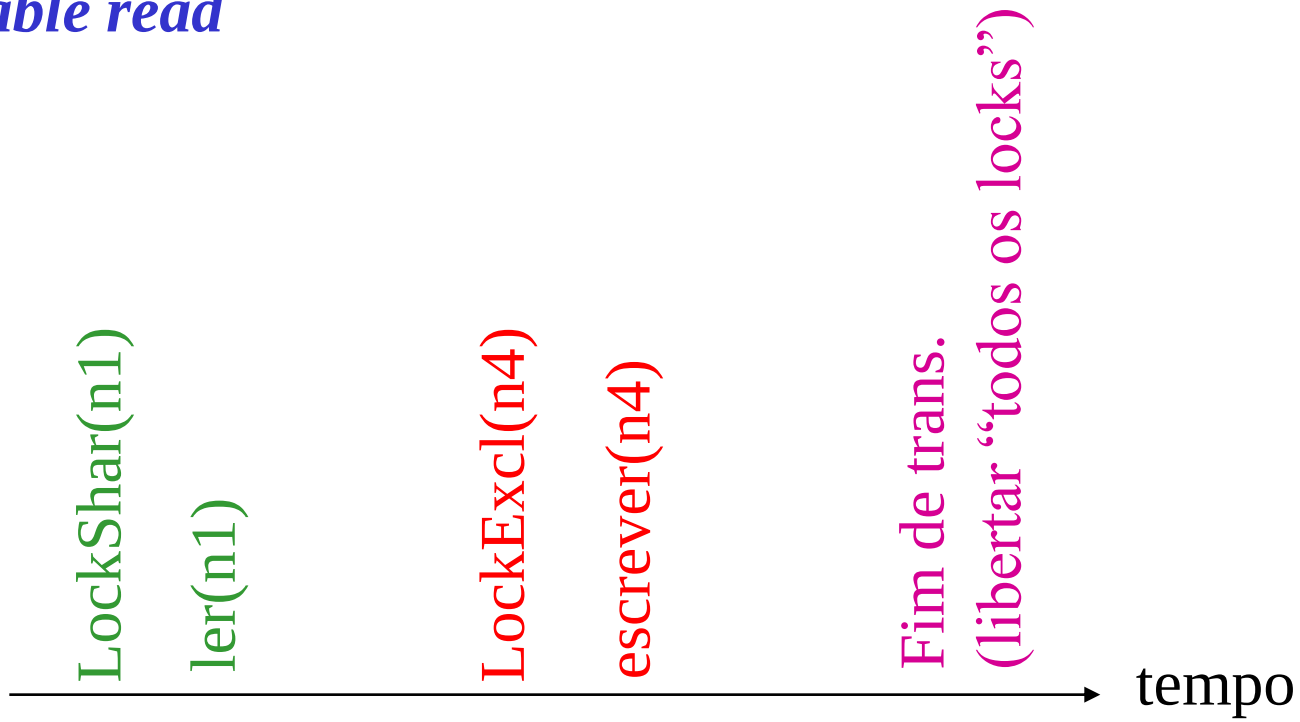
read commited



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

Transacções com 2PL e níveis de isolamento SQL 92

repeatable read

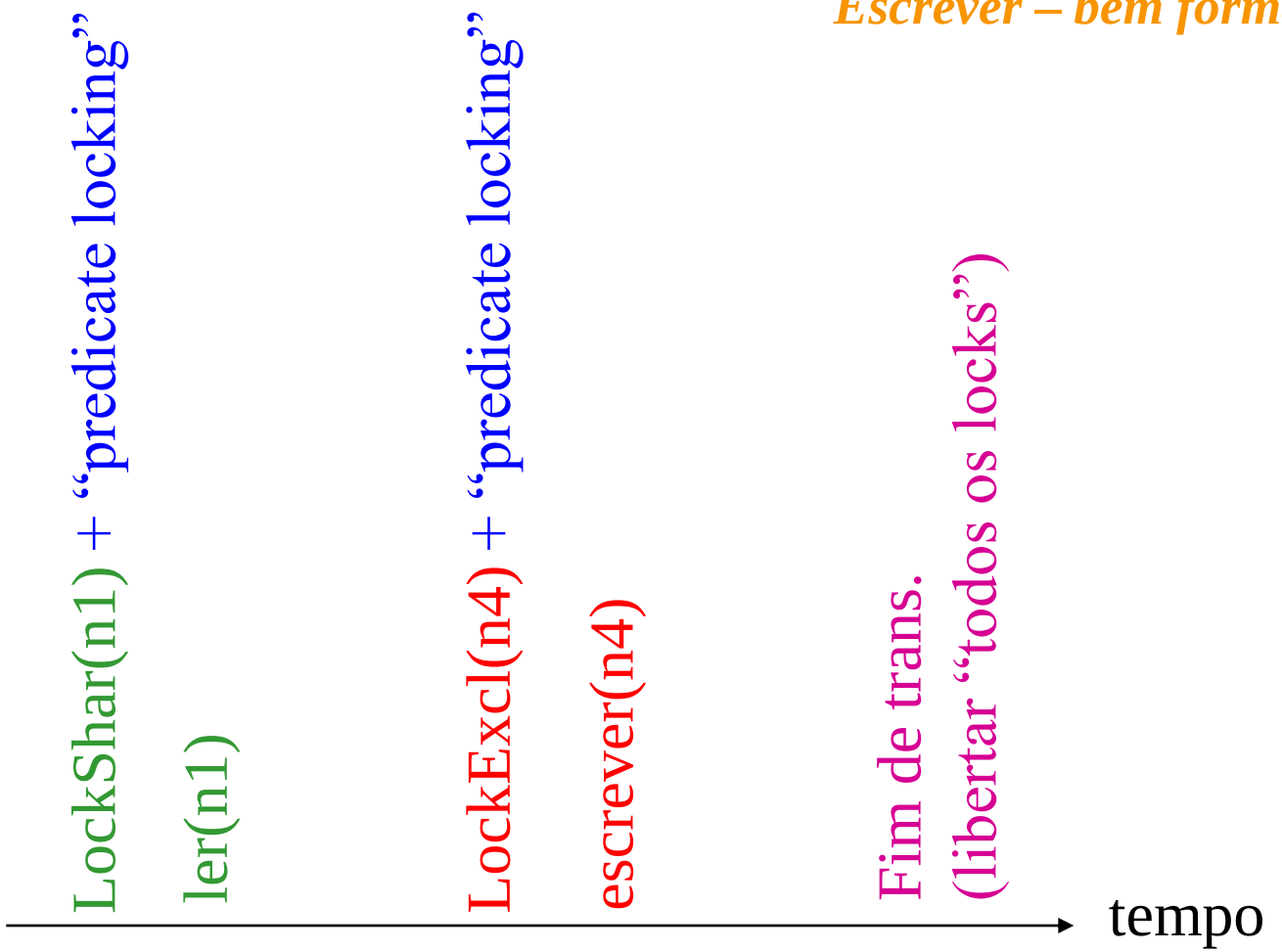


Ler – bem formada e 2 fases
Escrever – bem formada e de 2 fases

Transacções com 2PL e níveis de isolamento SQL 92

serializable

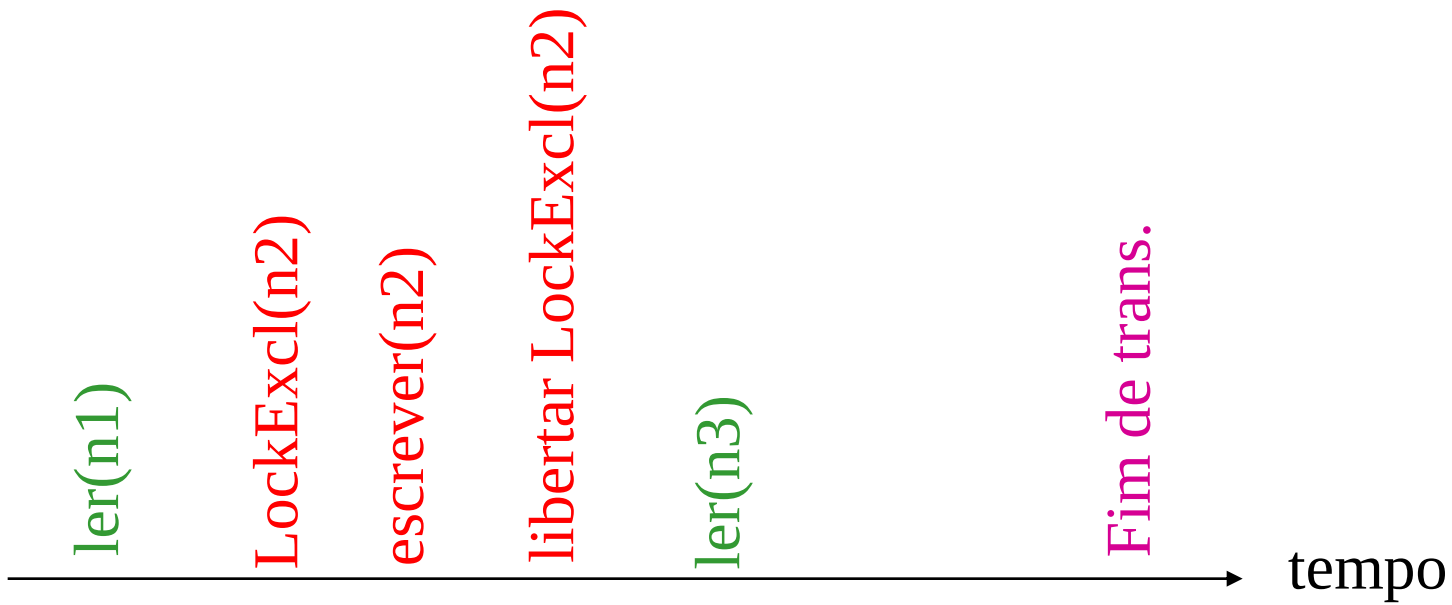
Ler – bem formada e de 2 fases
Escrever – bem formada e de 2 fases



Transacções com 2PL

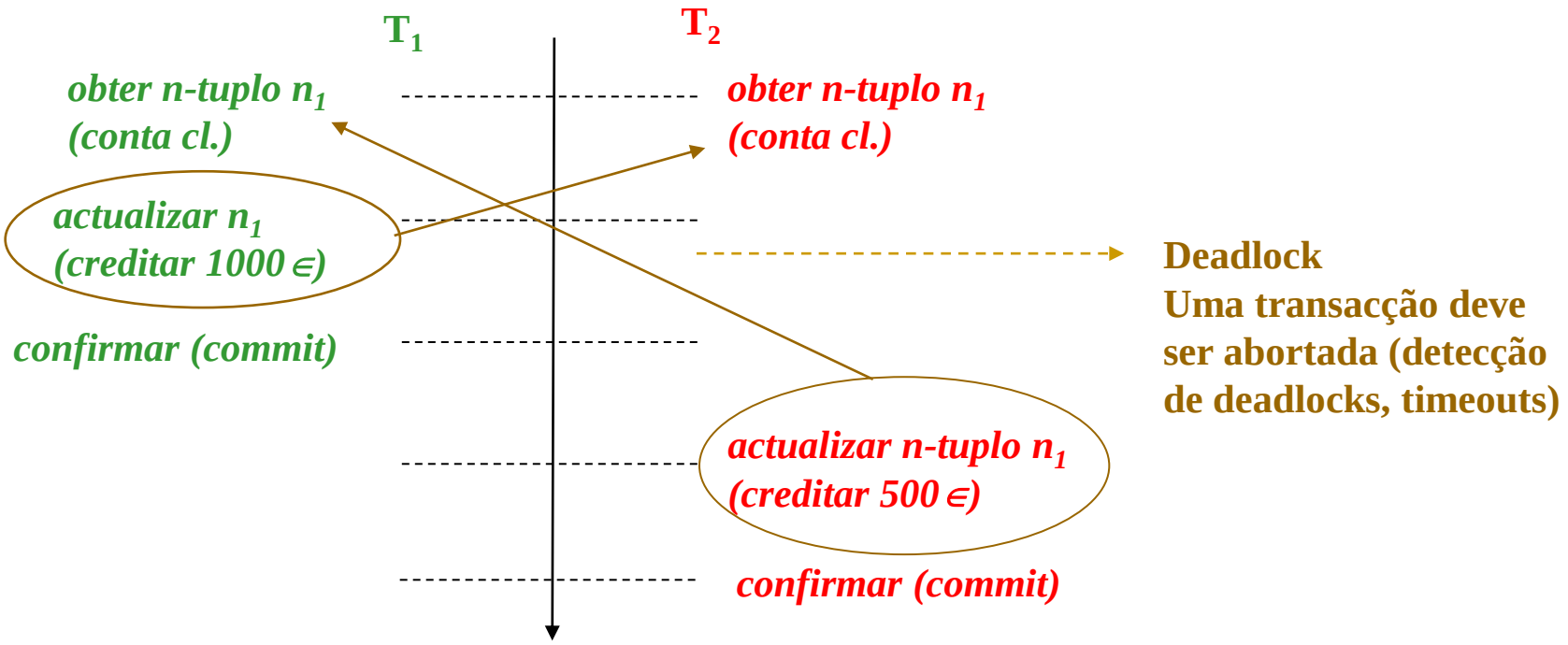
***chaos** (possível em alguns sistemas)*

Ler – não é bem formada
Escrever – é bem formada



Apresenta a anomalia "overwriting uncommited data"

Transacções com 2PL – deadlocks



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 –starvation

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

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

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 a anomalia "dirty reads") (repeatable read)*

Transacções – início e fim

Início:

- **Implícitas (SQL92)**

  No SQL-Server 2005:

  SET IMPLICIT TRANSACTIONS {ON|OFF}

- **Explícitas**

  Em ISO SQL (1999 – 2003): START TRANSACTION [modo]

  No SqlServer 2005 : BEGIN TRAN[SACTION] [*nome*]

Terminação:

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

  (também ROLLBACK TRAN[SACTION [nome]) no SQL Server 2005)
- **COMMIT [WORK]**

  (também COMMIT TRAN[SACTION [nome]) em SQL Server 2005)

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 l.eituras 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 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 conjunto 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).

```
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
```

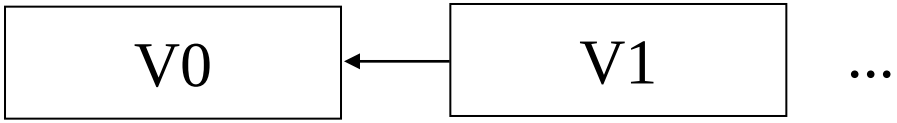
```
WRITE(T, X)
    i ← índice da última versão de X
    Enquanto tw(Xi) > ts(T) fazer
        i ← índice da versão anterior de i;
    Se tr(Xi) >  ts(T) então
        Abortar;
    Senão
        Executar escrita, inserindo nova versão (k),
            após a versão i;
        tw(Xk) ← tr(Xk) ← ts(T)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

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 – multi-versões em Sql Server 2005

O Sql Server 2005 introduz o novo nível de isolamento SNAPSHOT, o qual constitui uma alternativa ao nível READ COMMITTED e REPEATABLE READ, 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 versão 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 2005

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 – multi-versões em Sql Server 2005

Existe também 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.

Diferimento de teste de restrições

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

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

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

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

<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

begin 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

begin 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 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

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 2005 Books Online