
Processamento Transaccional (parte II)

Transacções lisas (*flat transactions*)

- Exibem as quatro propriedades ACID
- Bem adaptadas a transacções de curta duração
- Com transacções de longa duração apresentam os seguintes problemas:
 - *Maior probabilidade de se abortar uma transacção, tendo como consequência uma degradação do desempenho (maior volume de trabalho desfeito)*
 - *Inibição do acesso a recursos usados numa transacção por outras transacções durante períodos demasiado longos*
 - *Impossibilidade de se abortar apenas parte do trabalho realizado numa transacção*
 - *Não reflecte a natureza hierárquica típica de alguns processamentos*

Transacções lisas - limitações

Exemplo 1:

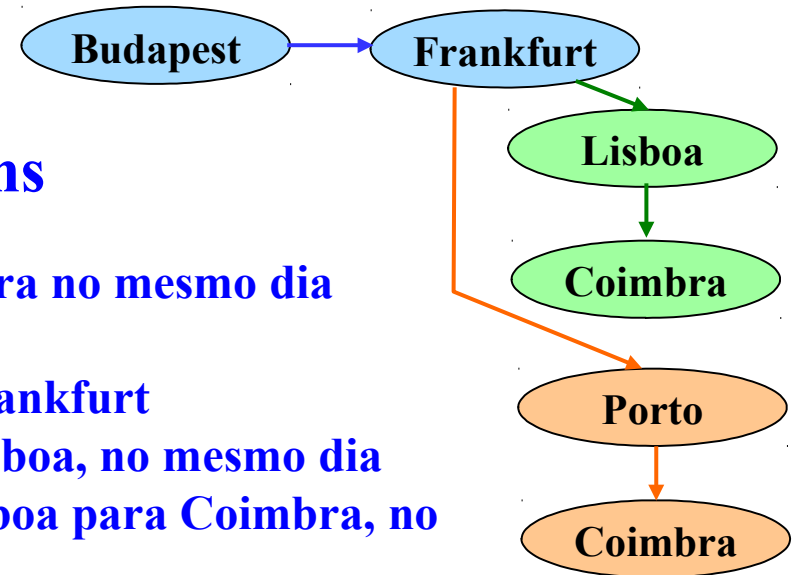
- Cenário de uma agência de viagens

» Pretendemos ir de Budapeste para Coimbra no mesmo dia

- 1: Reservar voo de Budapeste para Frankfurt
- 2: Reservar voo de Frankfurt para Lisboa, no mesmo dia
- 3: Reservar bilhete de comboio de Lisboa para Coimbra, no mesmo dia

Problema: impossível arranjar comboio de Lisboa para Coimbra no mesmo dia

- » Com transacções lisas, a única solução é abortarmos toda a transacção.
- » Mas talvez fosse possível Reservar um voo de Frankfurt para o Porto, no mesmo dia, o que apenas envolveria desistir da reserva do voo de Frankfurt para Lisboa.
- » **rollbacks parciais (savepoints) poderiam ajudar!**



Transacções lisas - limitações

Exemplo 2:

- Cenário de um banco

- » Pretendemos actualizar no fim do ano 1.000.000 de contas bancárias debitando o valor correspondente a despesas de manutenção

```
BEGIN TRANSACTION
para conta desde 1 até 1.000.000 fazer
    actualizarConta(conta)
COMMIT
```

- » Transacções muito longas aumentam a exposição da transacção a falhas, devidas a erros ou à concorrência.
- » Reiniciar uma transacção pode significar perder demasiado trabalho.
- » Recursos bloqueados durante muito tempo

- » *mini-batches e chained transactions* poderiam ajudar!

Transacções lisas - limitações

Exemplo 3:

- Numa aplicação OO

- » Suponham-se os objectos *O1* e *O2* com métodos *m1* e *m2*, respectivamente.
- » A chamada isolada de um destes métodos de um destes objectos deve lançar uma transacção, só sendo os seus efeitos visíveis no exterior após a validação da transacção (commit).
- » Como fazer se, por vezes, se pretender chamar *O2.m2()* de dentro de *O1.m1()*, mantendo a possibilidade de chamar *O2.m2()* isolado, mas pretendendo que, no primeiro caso, os efeitos de *O2.m2()*, embora visíveis na transacção associada a *O1.m1()* só sejam visíveis no exterior se esta terminar com sucesso.
- » *Manutenção de contexto transaccional entre objectos, ou transacções hierárquicas poderiam ajudar!*

Transacções com *savepoints*

- Permitem desfazer parcialmente as acções de uma transacção
- Evitam desfazer e refazer processamentos iniciais comuns a vários cursos de acção
- Constituem um método mais apropriado para lidar com erros em transacções que exibem muitas dependências entre os seus processamentos

Transacções com *savepoints* (ISO SQL 2011)

<savepoint statement> ::=
SAVEPOINT <savepoint specifier>

<savepoint specifier> ::=
<savepoint name>

<release savepoint statement> ::=
RELEASE SAVEPOINT <savepoint specifier>

<rollback statement> ::=
ROLLBACK [WORK] [AND [NO] CHAIN]
[<savepoint clause>]

<savepoint clause> ::=
TO SAVEPOINT <savepoint specifier>

Transacções com *savepoints* (SQL Server 2012)

```
ROLLBACK { TRAN | TRANSACTION }  
    [ transaction_name | @tran_name_variable  
    | savepoint_name | @savepoint_variable ]  
    [ ; ]
```

```
SAVE { TRAN | TRANSACTION }  
    { savepoint_name | @savepoint_variable }  
    [ ; ]
```


Transacções com *savepoints*

```
create table alunos (  
    numero numeric primary key,  
    nome char(10)  
)
```

```
create table inscr (  
    aluno numeric not null foreign key references alunos,  
    turma numeric not null  
    primary key (aluno, turma)  
)
```

Transacções com *savepoints*

```
create table tabLog (  
  id int identity primary key,  
  tempo datetime,  
  descr char(50)  
)
```

*Pode inserir-se um aluno mesmo que não o consigamos inscrever na turma pretendida.
Mas esse facto deve ser registado em tabLog.*

ordem temporal a manter

| id | tempo | descr |
|-----|-------------------------|--|
| 1 | 2003-09-29 17:19:02.703 | a inscrever aluno 100 na turma 1 |
| 2 | 2003-09-29 17:19:02.710 | inscrito aluno 100 na turma 1 |
| ... | ... | ... |
| 9 | 2003-09-29 17:19:02.717 | falhou a inscricao do aluno 400 na turma 1 |

Transacções com *savepoints*

O procedimento inscrever já existe e, para se garantir a preservação da integridade da lógica de negócios, deve ser usado para inscrever um aluno numa turma

```
create procedure inscrever(@al numeric, @t numeric)
as
  declare @dimt numeric
  set @dimt = (select count(*) from inscr where turma = @t)
  if (@dimt < 20) begin -- podemos inserir mais um aluno
    insert into inscr values (@al, @t)
    insert into tabLog values(getdate(), 'inscrito aluno ' +
                               cast(@al as varchar) +
                               ' na turma ' + cast(@t as varchar))

    return 0
  end
  else begin
    return 1
  end
end
```

Foram ignorados os possíveis erros na execução das instruções SQL

Transacções com *savepoints*

```
create procedure insAlComInscr (@al numeric, @nome varchar(20),
                                @t numeric)
as
  declare @res numeric, @dt datetime
  begin transaction
  insert into alunos values(@al, @nome)
  save transaction svPt      <- -----
  insert into tabLog values(getdate(), 'a inscrever aluno '+
                           cast(@al as varchar)+' na turma '+cast(@t as varchar))
  exec @res = inscrever @al, @t
  if (@res = 1)
  begin
    rollback transaction svPt -----
    insert into tabLog values(getdate(), 'falhou a inscricao do aluno '+
                             cast(@al as varchar)+' na turma '+cast(@t as varchar))
  end
  commit transaction
```

*repõe o estado da transacção,
mas não o do programa!*

*Foram ignorados os possíveis erros na
execução das instruções SQL*

Transacções com *savepoints*

Neste caso simples, poderíamos, em alternativa, fazer:

```
create procedure insAlComInscr (@al numeric, @nome varchar(20),  
                                @t numeric)  
as  
    declare @res numeric, @dt datetime  
    begin transaction  
    insert into alunos values(@al, @nome)  
    insert into tabLog values(getdate(), 'a inscrever aluno '+  
                                cast(@al as varchar)+' na turma '+cast(@t as varchar))  
    exec @res = inscrever @al, @t  
    if (@res = 1)  
    begin  
        delete from tabLog where descr like 'a inscrever aluno '+  
                                cast(@al as varchar)+'%';  
        insert into tabLog values(getdate(), 'falhou a inscricao do aluno '+  
                                cast(@al as varchar)+' na turma '+cast(@t as varchar))  
    end  
    commit transaction
```

Transacções com *savepoints*

Mas se o SP inscrever for mais complexo, ou não conhecermos os pormenores da sua implementação? Mo exemplo seguinte, quando retorna o valor 1, o SP inscrever também realiza operações sobre os dados, pelo que não conhecendo os pormenores da sua implementação, seria impossível conseguir o mesmo sem usar savepoints ou abortar toda a transacção

```
create procedure inscrever(@al numeric, @t numeric)
as
  declare @dimt numeric
  set @dimt = (select count(*) from inscr where turma = @t)
  if (@dimt < 20) begin -- podemos inserir mais um aluno
    insert into inscr values (@al, @t)
    insert into tabLog values(getdate(), 'inscrito aluno ' +
                             cast(@al as varchar) + ' na turma ' + cast(@t as varchar))

    return 0
  end
  else begin
    insert into tabLog values(getdate(), 'impossível inscrever aluno ' +
                             cast(@al as varchar) + ' na turma ' + cast(@t as varchar))

    return 1
  end
end
```

Transacções com *savepoints*

BEGIN TRAN T1

op1

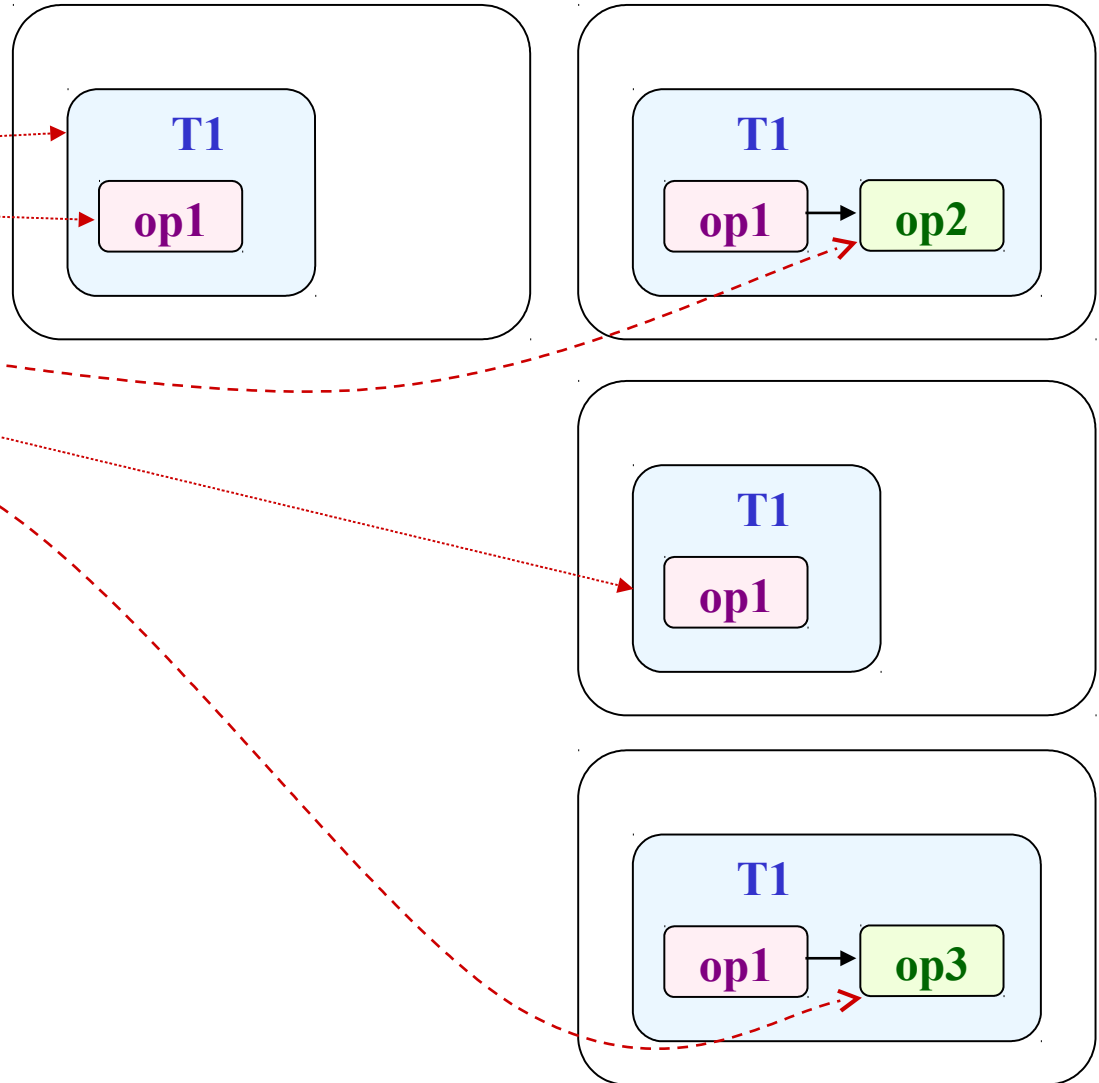
SP1:

op2

ROLLBACK SP1

op3

COMMIT TRAN T1



Transacções com *savepoints* persistentes

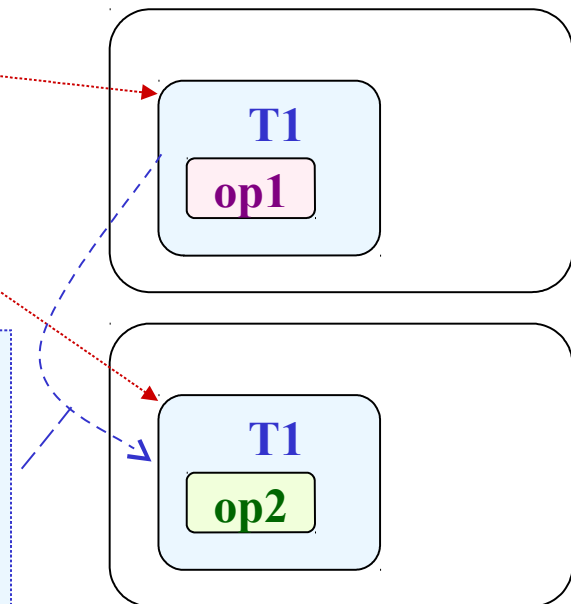
- Alternativa, na qual os *savepoints* persistem a falhas do sistema, isto é, o estado da BD na altura do *savepoint* é gravado em memória persistente
- Em caso de reinício da transacção (devido a falhas), poder-se-ia proceder do seguinte modo:
 - O intervalo após o último *savepoint* é anulado (*rolled back*)
 - O estado do último *savepoint* gravado em memória persistente é restaurado como estado corrente da transacção
- Pode permitir ganhos de desempenho significativos por, tendencialmente, reduzir o volume de trabalho perdido quando existe reinício de uma transacção.
- Mas, como restaurar o estado de execução do programa que controla a transacção, dado que o “regresso” a um *savepoint* é automático e não controlado pelo programador?

Transacções encadeadas (*chained transactions*)

- Exploram a ideia complementar dos *savepoints*, permitindo validar resultados parciais, mas, mantendo os objectos aos quais já se acedeu.
- Podem, por exemplo, manter um cursor, mesmo após a realização de um *commit* parcial
- Podem libertar-se os objectos não necessários, mas manter o restante contexto transaccional (uma forma de se lidar com transacções longas)
- Destroem a propriedade “atomicidade”

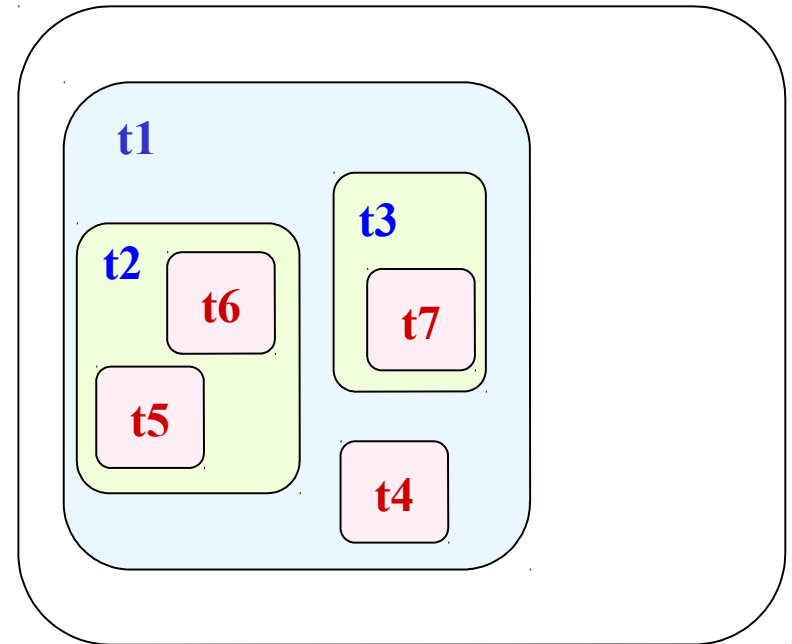
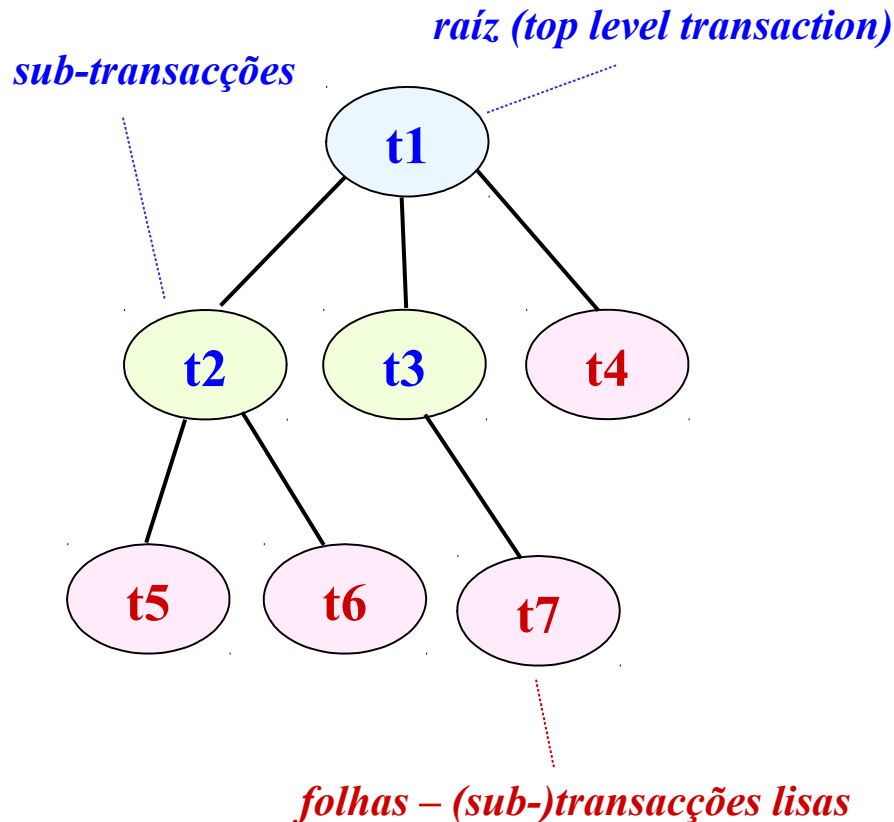
BEGIN TRAN T1
op1
partial commit
op2
COMMIT TRAN T1

- *libertação de alguns recursos*
- *manutenção de outros (p. ex: preservação de cursores)*
- *menos trabalho perdido em caso de falhas*
- *último commit parcial marca ponto de reinício em caso de falha*



Transacções hierárquicas

Generalização da ideia de *savepoints* para uma estrutura em árvore de transacções



Transacções hierárquicas

- **Regra de commit** – *a terminação com sucesso de uma transacção coloca os seus resultados visíveis apenas no contexto da sua transacção-pai. Logo os resultados só serão tornados visíveis definitivamente se a transacção de topo terminar com sucesso.*
- **Regra de rollback** – *a acção rollback de uma transacção implica a realização de rollback sobre todas as suas sub-transacções, independentemente de estas terem terminado com sucesso*
- **Regra de visibilidade** – *os objectos detidos por uma transacção podem ser tornados visíveis nas suas sub-transacções. A terminação com sucesso de uma transacção torna os seus resultados visíveis na sua transacção-pai*

Transacções hierárquicas? (SQL Server)

**BEGIN TRAN [SACTION] [*transaction_name* | @*tran_name_variable*
[WITH MARK ['*description*']]]**

ROLLBACK [WORK]

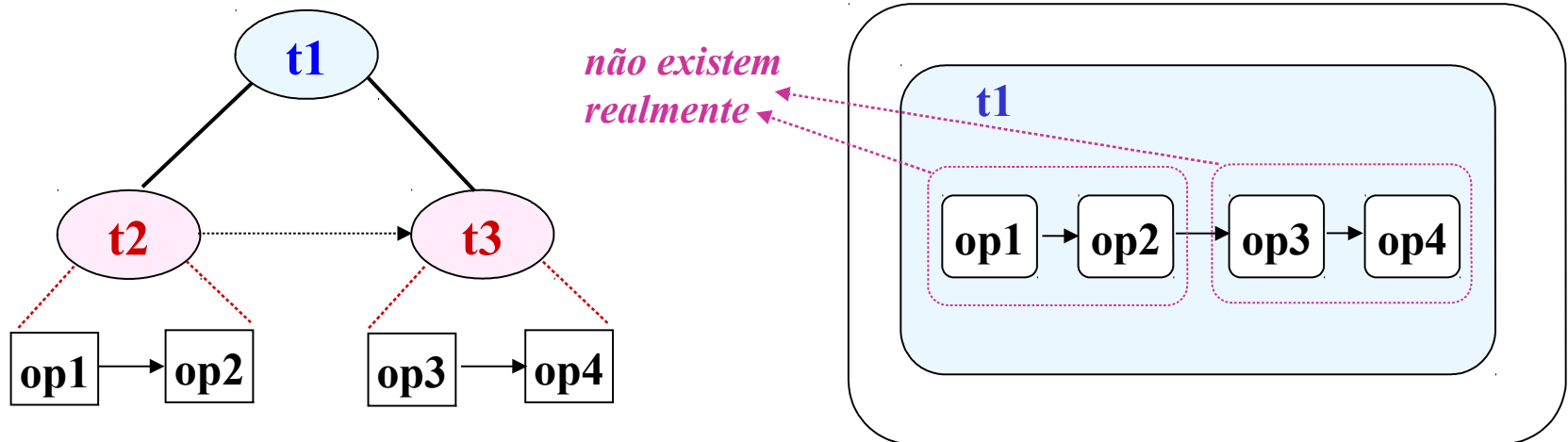
**ROLLBACK [TRAN [SACTION] [*transaction_name*
| @*tran_name_variable*
| *savepoint_name*
| @*savepoint_variable*]]**

COMMIT [WORK]

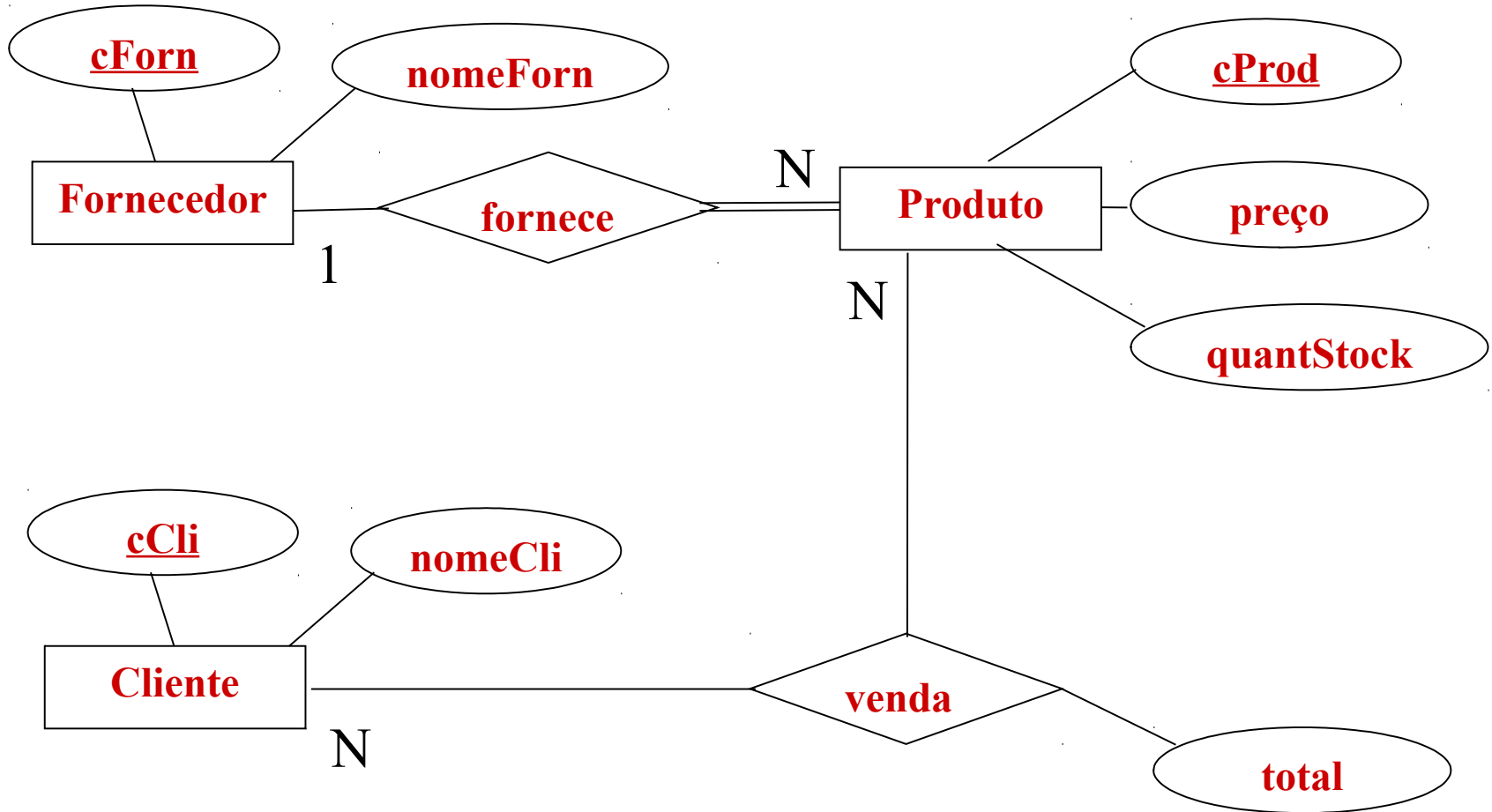
**COMMIT [TRAN [SACTION] [*transaction_name* |
@*tran_name_variable*]]**

Transacções hierárquicas (SQL Server)

- Forma muito limitada de transacções hierárquicas
- Destinada apenas a permitir que um procedimento armazenado possa iniciar uma transacção se não estiver uma em curso
- ROLLBACK aplica-se sempre à transacção de topo
- COMMIT numa sub-transacção é ignorado
- **Em rigor, existe sempre apenas a transacção de topo**



Transacções hierárquicas (simulação com *savepoints*)



Transacções hierárquicas (simulação com *savepoints*)

```
create table Fornecedor (  
  cForn char(6) primary key,  
  nomeForn char(20)  
)
```

```
create table Cliente (  
  cCli char(6) primary key,  
  nomeCli char(20) not null  
)
```


```
create table Vendas (  
  cProd char(6) foreign key references Produto,  
  cCli char(6) foreign key references Cliente,  
  total real not null  
)
```


```
create table Produto (  
  cProd char(6) primary key,  
  preco decimal(5,2) not null,  
  quantStock int not null,  
  cForn char(6) foreign key references fornecedor  
)
```

Pretende-se a seguinte funcionalidade transaccional:

1. Para clientes comuns, venda de um produto dado o seu código e a quantidade. Deve-se ficar a saber o custo total da venda
2. Para clientes institucionais, venda de um produto a um cliente, dado o código de produto, o código de cliente e a quantidade. Deve-se ficar a saber o custo total da venda. Deve registar-se o total de vendas de um produto a um cliente

Transacções hierárquicas (simulação com *savepoints*)

```
create proc venderProd (@cProd char(6), @quant int, @custo real output
as
begin transaction TrVenderProd 



if (select quantStock from Produto where cProd = @cProd) >= @quant
begin
    update Produto set quantStock = quantStock - @quant where cProd = @cProd
    select @custo = preco*@quant from Produto where cProd = @cProd
    commit tran TrVenderProd
    return 0
end
else
begin
    rollback transaction TrVenderProd 

    return 1
end
```

Foram ignorados os possíveis erros na execução das instruções SQL

Transacções hierárquicas (simulação com *savepoints*)

```
create proc VenderProdInst @cProd char(6), @cCli char(6), @quant int,  
                           @custo real output  
  
as  
declare @res int  
begin transaction  
exec @res = venderProd @cProd, @quant, @custo output  
if @res <> 0  
begin  
    rollback  
    return @res  
end  
if exists (select * from vendas where cCli = @cCli and cProd = @cProd)  
    update Vendas set total = total + @quant where cCli = @cCli and  
                                                cProd = @cProd  
else  
    insert into Vendas values (@cProd, @cCli, @quant)  
commit  
return 0
```



Foram ignorados os possíveis erros na execução das instruções SQL

Transacções hierárquicas (simulação com *savepoints*)

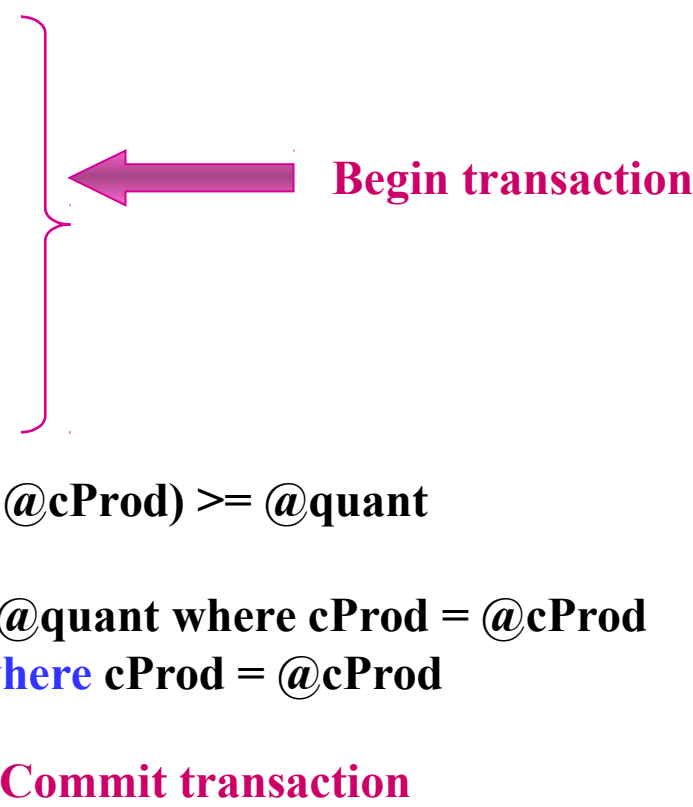
```
create proc venderProd (@cProd char(6), @quant int, @custo real output
as
begin transaction TrVenderProd }
save transaction TrVenderProd }
if (select quantStock from Produto where cProd = @cProd) >= @quant
begin
    update Produto set quantStock = quantStock - @quant where cProd = @cProd
    select @custo = preco*@quant from Produto where cProd = @cProd
    commit tran TrVenderProd
    return 0
end
else
begin
    rollback transaction TrVenderProd }
    commit tran TrVenderProd
    return 1
end
```

Mas, quando se executa isolado, do ponto de vista do SGBD, a transação termina com sucesso

Foram ignorados os possíveis erros na execução das instruções SQL

Transacções hierárquicas (simulação com *savepoints*)

```
create proc venderProd @cProd char(6), @quant int, @custo real output
as
declare @savePoint varchar(36) = null
if @@TRANCOUNT = 0
begin tran
else
begin
set @savePoint = cast(newid() as varchar(36))
save tran @savePoint
end
if (select quantStock from Produto where cProd = @cProd) >= @quant
begin
update Produto set quantStock = quantStock - @quant where cProd = @cProd
select @custo = preco*@quant from Produto where cProd = @cProd
if @savePoint is NULL
commit
return 0
end
```



The diagram illustrates the transaction flow. A large magenta arrow points left from the text "Begin transaction" to a magenta bracket that groups the lines from "if @@TRANCOUNT = 0" to "save tran @savePoint". Another magenta arrow points left from the text "Commit transaction" to a magenta bracket that groups the lines from "if @savePoint is NULL" to "commit".

Transacções hierárquicas (simulação com *savepoints*)

```
else
begin
  if @savePoint is NULL
    rollback
  else
    rollback tran @savePoint
  return 1
end
```

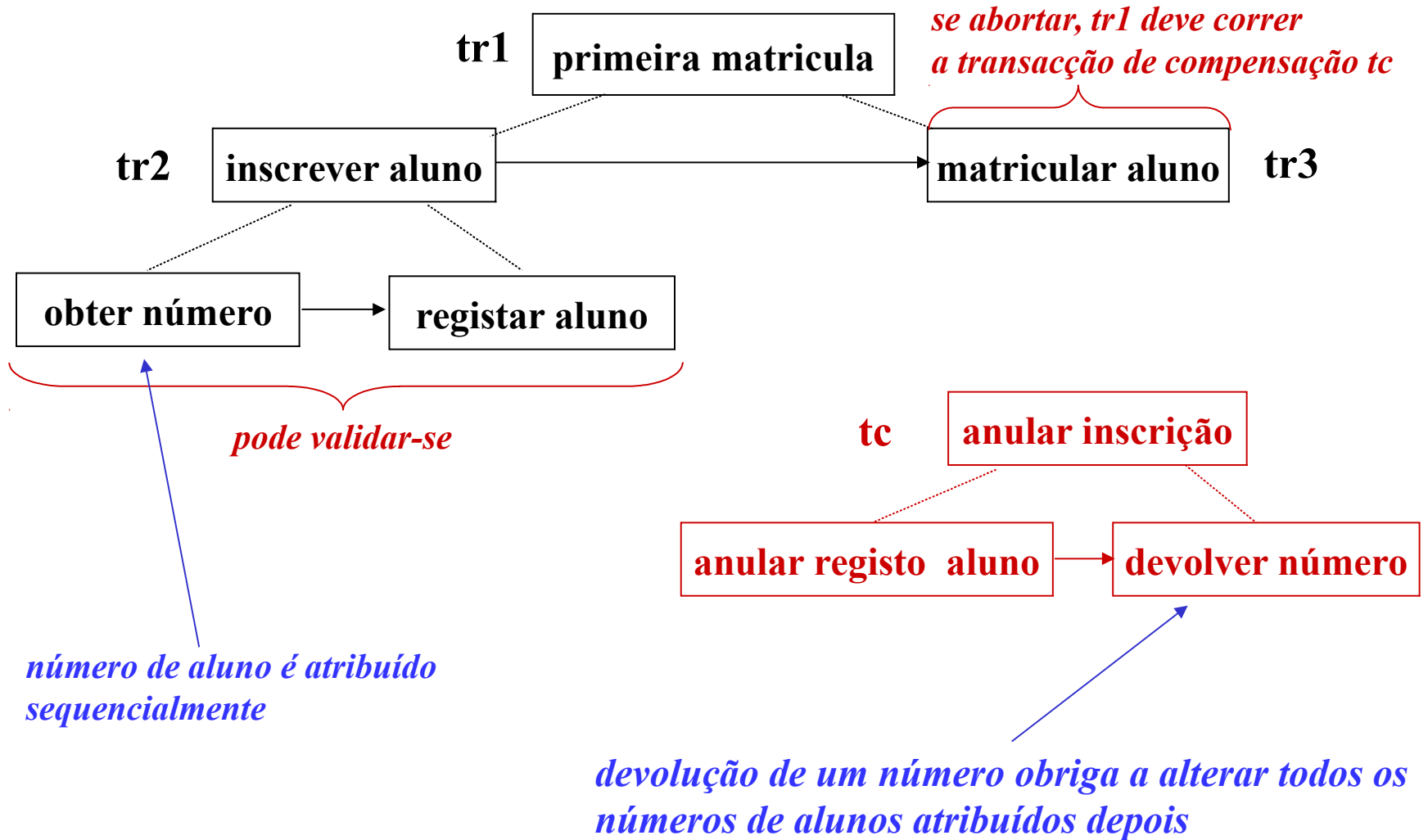


Transacções multi-nível

Mantêm a estrutura hierárquica na organização das transacções, mas:

- *Cada sub-transacção pode validar os seus efeitos de forma permanente, havendo transacções de compensação para corrigir os seus efeitos caso o nível superior realize rollback*
- *Associadas a operações sobre hierarquias de objectos*
 - *Objectos da camada n são implementados usando operações de objectos da camada $n-1$*
 - *Objectos da camada n apenas acedem a objectos da camada $n-1$*
- *Outra forma de se lidar com transacções longas*

Transacções multi-nível (Exemplo)



Transacções hierárquicas abertas (*open nested transactions*)

- Variante (anárquica) das transacções multi-nível na qual as sub-transacções podem abortar ou validar os seus resultados, independentemente das transacções-pai (não existindo, de forma explícita no modelo, transacções de compensação).
- Em rigor, constituem apenas uma forma de colocar em execução outras transacções

Transacções – Sagas

Utilizam a ideia de *commit* parcial das transacções encadeadas, conjuntamente com a ideia de transacções de compensação das transacções multi-nível:

- A atomicidade da cadeia é garantida pela utilização de transacções de compensação

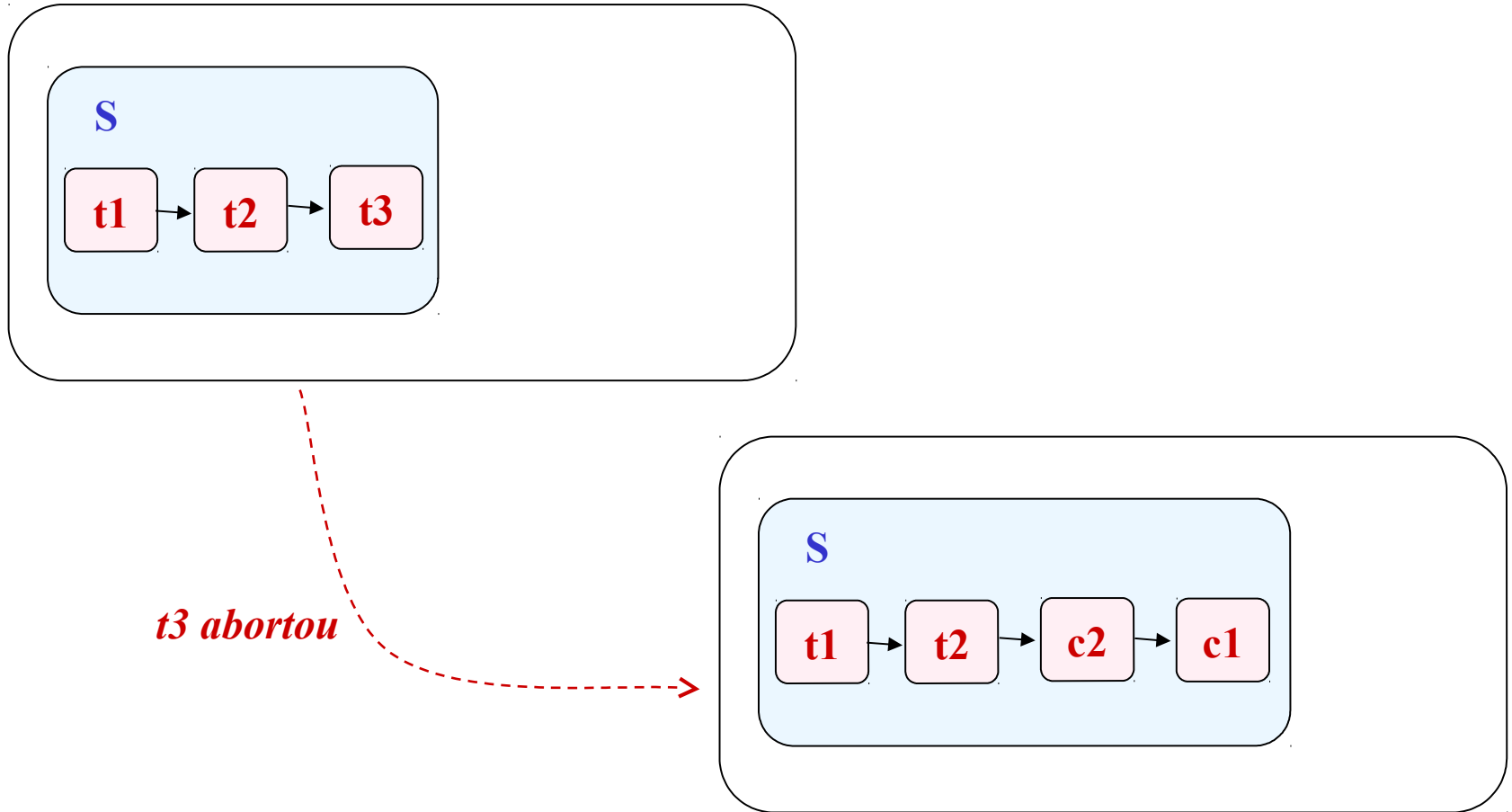
Seja t_1, \dots, t_n uma sequência de transacções lisas, e $\{c_1, \dots, c_{n-1}\}$ o conjunto das transacções de compensação respectivas.

O resultado de execução da saga é:

O resultado final da execução de t_1, \dots, t_n se não existirem erros, ou o resultado final da execução de $t_1, t_2, \dots, t_{i-1}, c_{i-1}, \dots, c_1$ se a transacção t_i abortar

Transacções – Sagas

Saga $S = (<t1,t2,t3,t4>, \{c1,c2,c3\})$



Transacções – Sagas

Muitas das ideias das sagas podem ser usadas em soluções programadas, como mostra este exemplo.

Sejam a “saga” transferir baseada nas transacções:

T1 : debitar o valor na conta origem

T2: creditar o valor na conta destino

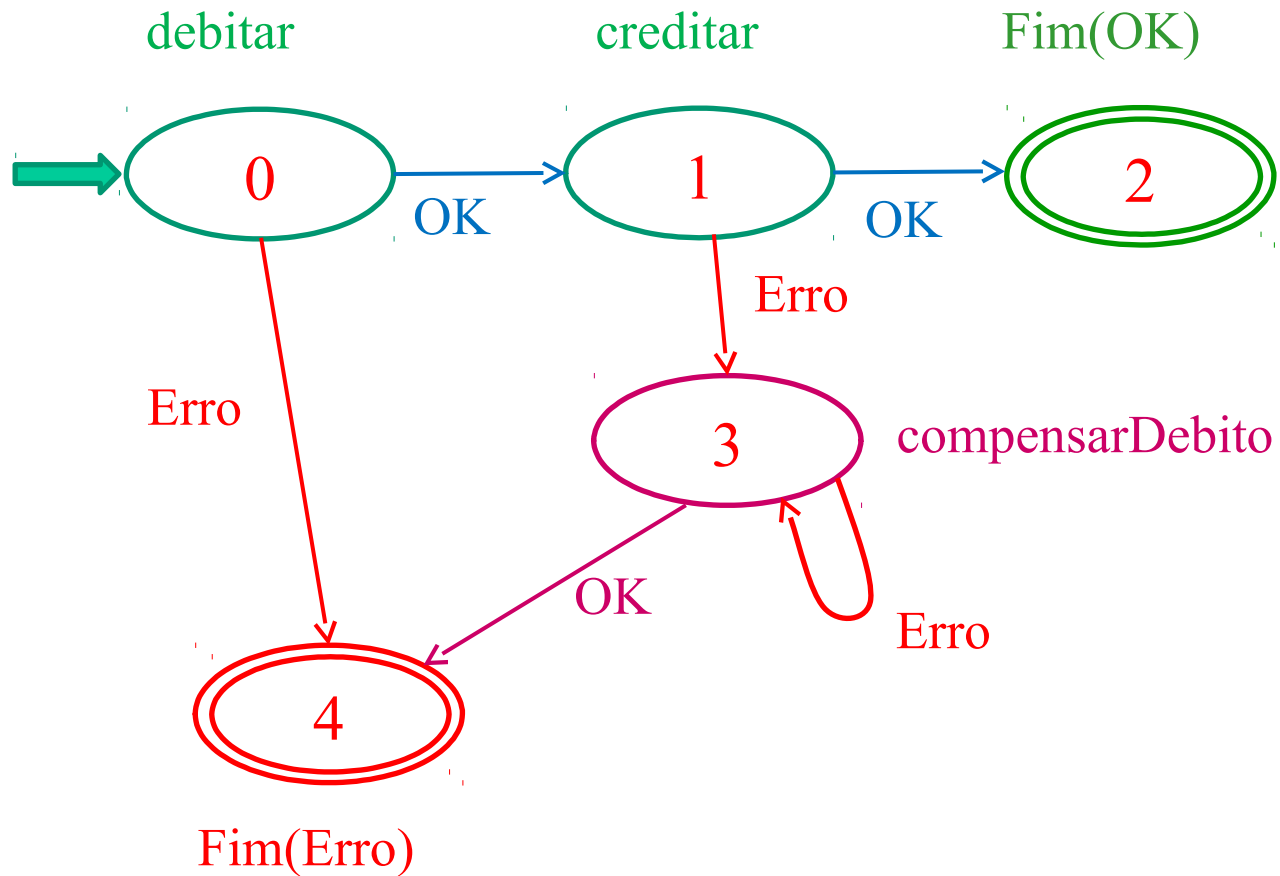
C1: transacção de compensação de T1, isto é, realizar o estorno para a conta origem se T2 falhar

Podemos usar uma tabela que nos permita manter o estado de execução da “saga” (workflow). Sempre que o processamento da “saga” é reiniciado, as transacções a executar serão ditadas por este estado (contexto)

```
create table conta (  
    id int primary key,  
    saldo real  
)
```

```
create table ContextoTransf(  
    idTransf int primary key,  
    estado int  
)
```

Transacções – Sagas



Transacções – Sagas

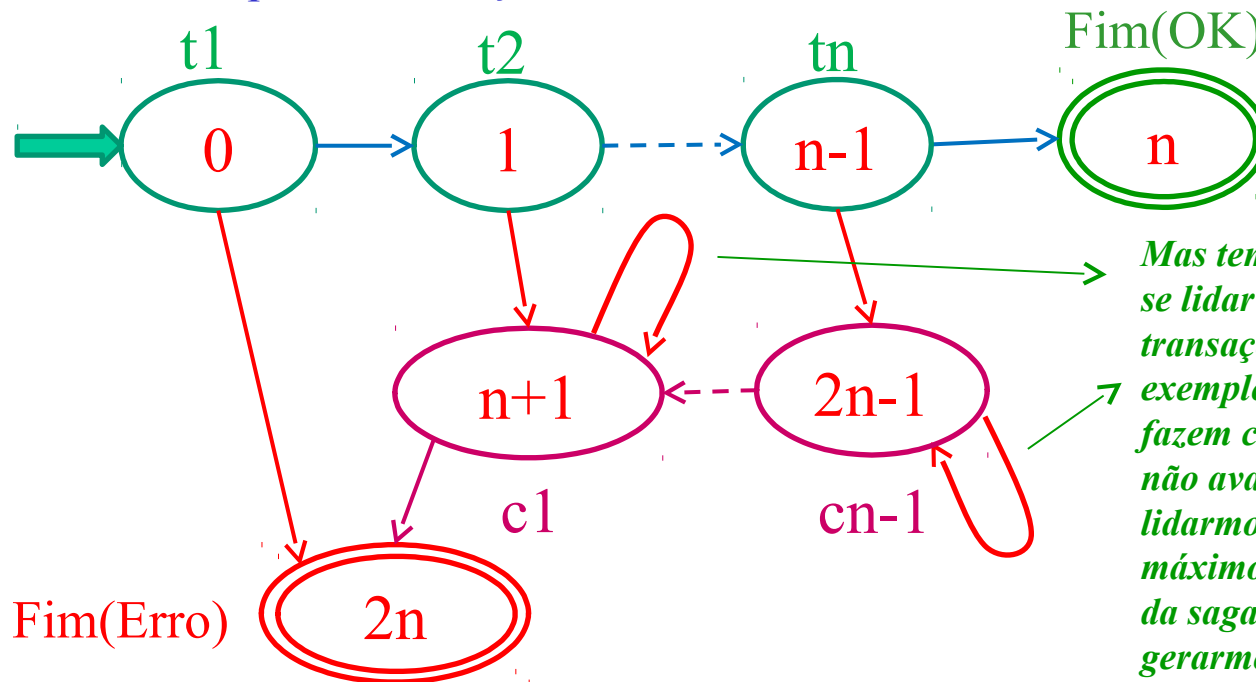
Em geral, para n transações, teremos $2n+1$ estados:

n estados para execução de cada transação

1 estado para terminação correcta

$(n-1)$ estados para execução de cada transação de compensação

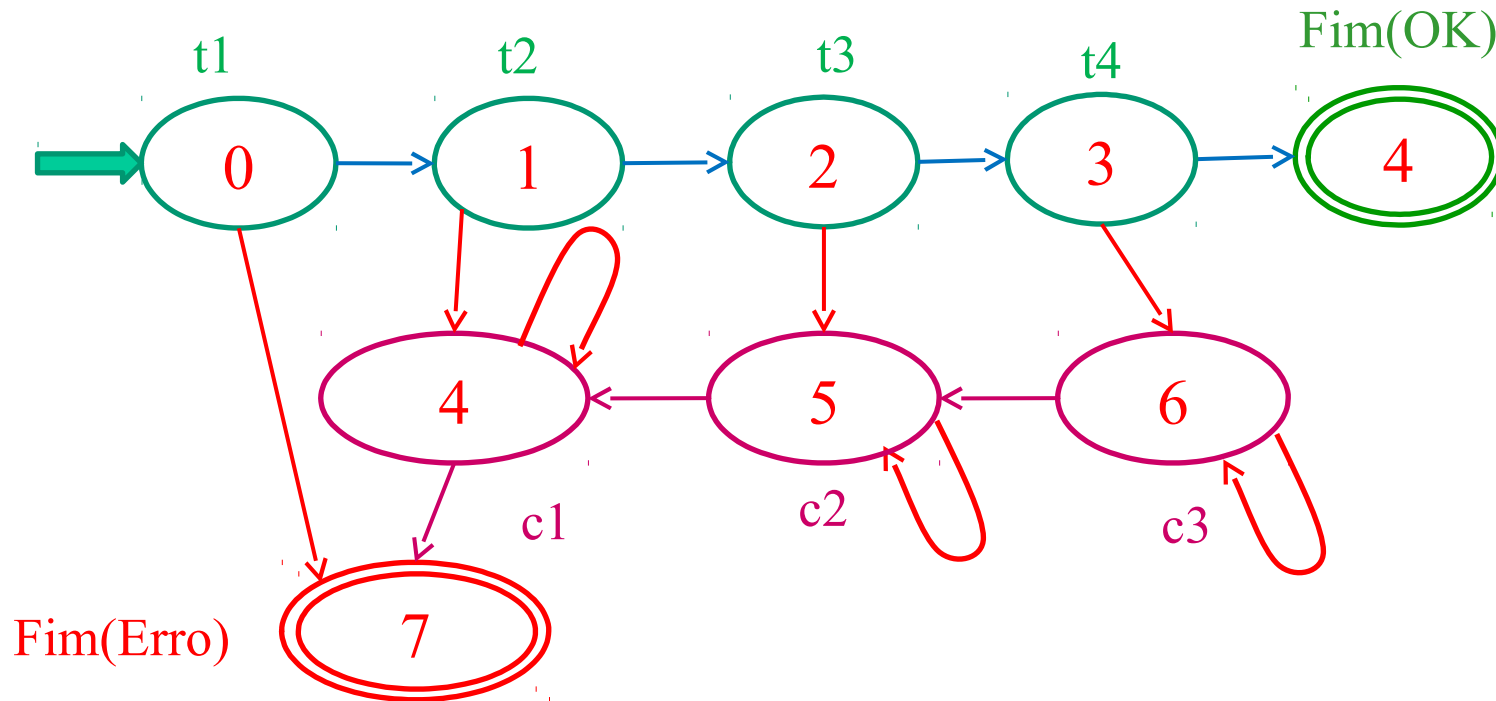
1 estado para terminação incorrecta



Mas tem de existir uma forma de se lidar com erros sistemáticos nas transações de compensação. No exemplo anterior, esses erros fazem com que o processamento não avance, sendo uma forma de lidarmos com isso limitar a um máximo de tentativas a execução da saga e, caso ela não termine, gerarmos um erro (eventualmente com registo numa tabela de log dessa ocorrência)

Transacções – Sagas

Exemplo para a saga $S = \langle t1, t2, t3, t4, c1, c2, c3 \rangle$



Transacções – Sagas

Controlo da “saga”:

```
create proc sagaTransf(@idTransf int,@c1 int, @c2 int, @montante float)
```

```
As
```

```
declare @estado int, @estadoSeg int
```

```
declare @erro int
```

```
set transaction isolation level read committed
```

```
begin tran
```

```
if not exists (select idTransf from ContextoTransf  
               where idTransf = @idTransf)
```

*Não deveria ser **SERIALIZABLE**?*

```
begin
```

```
insert into ContextoTransf values(@idTransf,0) -- debitar
```

```
end
```

```
commit
```

```
select @estado = estado from ContextoTransf  
       where idTransf = @idTransf
```

```
while @estado <> 2 and @estado <> 4
```

```
begin
```

Transacções – Sagas

```
if @estado = 0 begin
  begin tran
  exec @erro = debitar @idTransf, @c1, @montante -- correr T1 (debitar)

  if @erro = 0
    set @estadoSeg = 1 -- creditar
  else
    set @estadoSeg = 4 -- Fim (Erro)

  update ContextoTransf set estado = @estadoSeg where idTransf = @idTransf
  if @@ROWCOUNT <> 1
  begin
    rollback
    return -1 -- erro ao actualizar o contexto (não se pode fazer nada)
  end
  commit
end
```

Transacções – Sagas

```
if @estado = 1 begin
    begin tran
    exec @erro = creditar @idTransf, @c2, @montante -- correr T2 (creditar)
    if @erro = 0
        set @estadoSeg = 2 -- Fim (OK)
    else
        set @estadoSeg = 3 -- Erro (compensDebitar)

    update ContextoTransf set estado = @estadoSeg where idTransf = @idTransf
    if @@ROWCOUNT <> 1
        begin
            rollback
            return -1 -- erro ao actualizar o contexto (não se pode fazer nada)
        end
    commit
end
```


Transacções – Sagas

```
if @estado = 3 begin
    begin tran
    exec @erro = compensDebitar @idTransf, @c1, @montante -- correr
                                                    -- compensDebitar

    if @erro = 0
        set @estadoSeg = 4 -- Fim (Erro)
    else
        set @estadoSeg = 3 -- manter estado

    update ContextoTransf set estado = @estadoSeg where idTransf = @idTransf
    if @@ROWCOUNT <> 1
    begin
        rollback
        return -1 -- erro ao actualizar o contexto (não se pode fazer nada)
    end
    commit
end
```

Transacções – Sagas

```
if @estado not in (0,1,3)
begin
    print 'estado indeterminado'
    return -4 -- estado indeterminado
end
set @estado = @estadoSeg
end -- while
if @estado = 2
begin
    print 'transferência já concluída'
    return @estado
end
if @estado = 4
begin
    print 'Compensação já concluída'
    return @estado
end
```

Transacções – Sagas

Transacção T1:

```
create proc debitar(@idTransf int, @c int, @montante float)
As
    update conta set saldo = saldo-@montante
        where id = @c and saldo >= @montante
    if @@ROWCOUNT <> 1
        return -1 -- erro ao actualizar o saldo (conta inexistente ou saldo insuficiente)
    return 0
```

Transacção T2:

```
create proc creditar(@idTransf int, @c int, @montante float)
As
    update conta set saldo = saldo+@montante
        where id = @c
    if @@ROWCOUNT <> 1
        return -1 -- erro ao actualizar o saldo (conta inexistente)
    return 0
```

Transacções – Sagas

Transacção C1:

```
create proc compensDebitar(@idTransf int, @c int, @montante float)
as
    update conta set saldo = saldo + @montante
        where id = @c
    if @@ROWCOUNT <> 1
        return -1 -- erro ao actualizar a conta (conta inexistente)
    return 0
```

Transacções longas com mini lotes (*mini-batches*)

Cenário:

- No final de cada período, pretende-se fazer a actualização dos montantes de 1000000 de contas bancárias (depósitos a prazo) de acordo com a taxa de juro praticada.
- Deve garantir-se que todas as contas são alteradas.

Soluções possíveis (?):

- Fazer a alteração das 1000000 contas numa transacção lisa
 - Transacção muito longa, logo maior exposição a falhas
 - Recursos (contas) indisponíveis por demasiado tempo reduzem nível de concorrência
- Chained transactions
 - Não se garante que todas as contas são actualizadas
- Sagas
 - As trans. de compensação o que fazem? Anulam o processamento já realizado? Se sim, como fazê-lo se as contas tiverem sido alteradas entretanto? Senão, que outro tipo de acção podem desencadear?

Transacções longas com *mini-batches*

- O ideal seria o sistema manter o contexto de execução (incluindo contexto transaccional e o estado da aplicação)
- Em caso de reinício da aplicação, devido a falhas, a transacção e a aplicação reiniciar-se-iam no ponto determinado pelo contexto
- De difícil realização
- Na prática, muitas vezes, tem de ser a aplicação a manter o contexto transaccional
 - No caso do exemplo da alteração das 1000000 contas bancárias, pode ser utilizada uma técnica designada *mini-batches*

Transacções longas com *mini-batches*

```
create table conta(  
  numero int primary key,  
  titular char(20) not null,  
  valor real  
)
```

```
create table contexto (  
  job int primary key,  
  num int not null  
)
```

```
create proc alterar @taxa real, @numJob int  
as  
  declare @num int  
  declare @nextNum int  
  declare @continuar int = 1  
  set tran isolation level repeatable read  
  if not exists (select * from contexto where job = @numJob)  
    insert into contexto values (@numJob,1)  
  select @num = num from contexto where job = @numJob  
  if(@num < 0)  
  begin  
    print 'processamento já terminado'  
    return @num  
  end
```

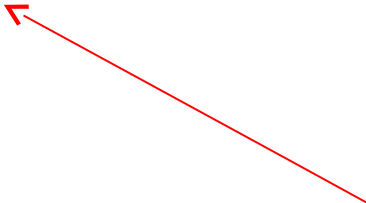
Transacções longas com *mini-batches*

```
while (@continuar = 1)
begin
    begin tran
    begin try
        exec processarLote @num, 2, @taxa, @nextNum output
    end try
    begin catch
        rollback;
        throw
    end catch
    if @nextNum > @num
    begin
        update contexto set num = @nextNum where job = @numJob
        I f(@@ROWCOUNT = 0)
        begin
            rollback
            return -1 -- não se pode fazer nada
        end
    end
end
commit work
```


Transacções longas com *mini-batches*

```
if @nextNum = @num
  set @continuar = 0
  set @num = @nextNum
end -- while
update contexto set num = -2 where job = @numJob
if(@@ROWCOUNT = 0)
  return -1 -- não se pode fazer nada

return 0
end
```



*No fim de se ter percorrido
todas as contas marca-se o job
como concluído*

Transacções longas com *mini-batches*

```
create proc processarLote (@num int, @dimLt int, @taxa real, @nextNum int output)
as
begin

    set @nextNum = 1+(select max(Numero) from
                        (select Numero from conta where numero >= @num
                         ORDER BY numero ASC OFFSET 0 ROWS FETCH NEXT @dimLt
                                                                    ROWS ONLY) as t)

    if @nextNum is null
        set @nextNum = @num
    update top (@dimLt) conta set valor = valor*(1+@taxa)
        where numero >= @num

end
```

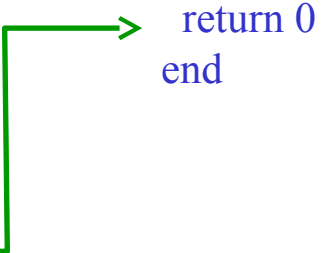
Mas pode ser mais simples se usarmos uma tabela auxiliar

Transacções longas com *mini-batches*

```
create table contasAalterar(  
    job int,  
    numero int references conta,  
    primary key (job, numero)  
)  
  
create table contexto_v2 (  
    job int primary key,  
    done int not null  
)  
  
create proc alterar_v2 @taxa real, @numJob int  
as  
begin  
    declare @done int  
    set tran isolation level repeatable read  
    begin tran  
    if not exists (select * from Contexto_v2 where job = @numJob)  
    begin  
        insert into contasAalterar select @numJob as job, Numero from conta  
        insert into Contexto_v2 values (@numJob, 0)  
    end  
    end tran  
    commit
```

Transacções longas com *mini-batches*

```
select @done = done from Contexto_v2 where job = @numJob
if @done = 1
begin
    print 'processamento já terminado'
    return -2
end
while (@done = 0)
begin
    begin tran
    begin try
        exec processarLote_v2 2, @taxa, @numJob
    end try
    begin catch
        rollback
    end catch
    commit work
    if not exists (select * from contasAalterar where job = @numJob)
    begin
        set @done = 1
        update contexto_v2 set done = 1 where job = @numJob
    end
end
end
```



```
return 0
end
```

Transacções longas com *mini-batches*

```
create proc processarLote_v2 (@dimLt int, @taxa real, @job int)
as
begin

    update conta set valor = valor*(1+@taxa)
        where numero in
            (select top(@dimLt) Numero from ContasAalterar
                where job = @job)

    delete top(@dimLt) from ContasAalterar where job = @job

end
```

Transacções longas com *mini-batches*

Exemplo de uso:

```
declare @res int
declare @done int = 0
declare @nTent int = 4
while (@done = 0 and @nTent > 0)
begin
    begin try
        exec @res = alterar .01, 1
        if @res = 0 or @res = -2
            set @done = 1
    end try
    begin catch
        set @nTent = @nTent - 1
    end catch
end
if @done = 1
    print 'OK'
else
    print 'Esgotado número de tentativas. Tentar mais tarde.
```

*Solução idêntica se pode
adoptar para o caso das
Sagas*

Se persistir o problema, falar com os técnicos de informática'

Transacções longas com *pseudo-conversações*

- Uma transacção diz-se conversacional (uma “conversação”) se envolver interação com um utilizador durante a sua execução.
- Como estas transacções poderiam ser muito longas, uma técnica é dividi-las em várias transacções menores, mantendo o contexto da conversação de forma adequada (pseudo-conversações).
- Não é frequente existir suporte para este esquema nos SGBDs correntes
- Uma alternativa consiste em implementá-lo em software, o que é particularmente facilitado em domínios caracterizados por pares de operações reservar/libertar.
- A técnica pode ser usada como forma de se implementar estratégias de controlo transaccional pessimista offline em ambientes que não têm de envolver interação com os utilizadores

Transacções longas com *pseudo-conversações* (exemplo)

Cenário:

Num sistema de gestão de funcionários, vários utilizadores podem editar dados dos funcionários.

Não pode haver interferências entre as edições dos utilizadores, mas também não é desejável a manutenção de transacções durante a duração de cada edição (muitas transacções, necessidade de se manterem conexões activas para a BD, sobrecarga do SGBD, etc.).

Devem prever-se situações típicas de interacções com utilizadores (abandono do posto de trabalho durante muito tempo, ida para fim-de-semana sem terminar o trabalho, etc.)

Transacções longas com *pseudo-conversações* (exemplo)

```
create table funcionario (  
  numero int primary key,  
  nome varchar(40),  
  salario int  
)
```

```
create table controlo (  
  numero int primary key,  
  dataEntrada datetime,  
  token uniqueidentifier,  
  foreign key (numero) references funcionario  
)
```

Transacções longas com *pseudo-conversações* (exemplo)

```
create proc obter @numero int, @nome varchar(40) output, @salario int output, @token  
uniqueidentifier output
```

```
as  
delete from controlo where GetDate() > dateadd(mi,2,dataEntrada)
```

```
set transaction isolation level read committed
```

```
begin tran
```

```
if exists (select * from controlo where numero = @numero)
```

```
begin
```

```
rollback tran
```

```
return 1 /* erro – funcionário em edição */
```

```
end
```

```
select @nome=nome, @salario=salario from funcionario where numero = @numero
```

```
if (@@RowCount = 0)
```

```
begin
```

```
rollback tran
```

```
return 2 /* erro – funcionario nao existe */
```

```
end
```

```
set @token = newid()
```

```
insert into controlo values(@numero, getdate(),@token)
```

```
commit tran
```

*Podem ser horas, dias, meses,
anos...*

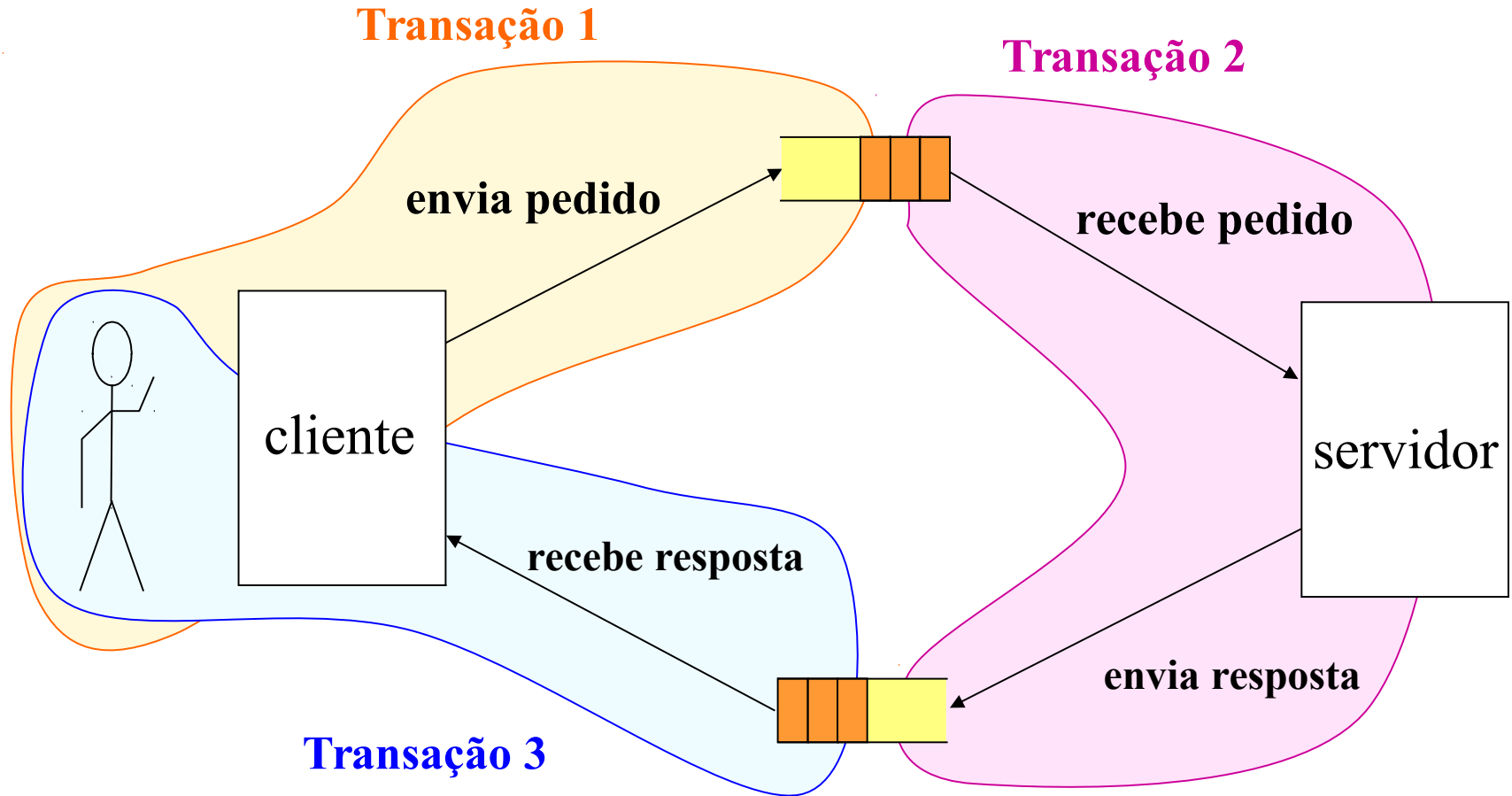
*Desde que se mantenha o token,
será possível propagar as
alterações, mesmo que as
aplicações ou o SGBD tenha
crashes recuperáveis*

```
return 0 /* OK */
```

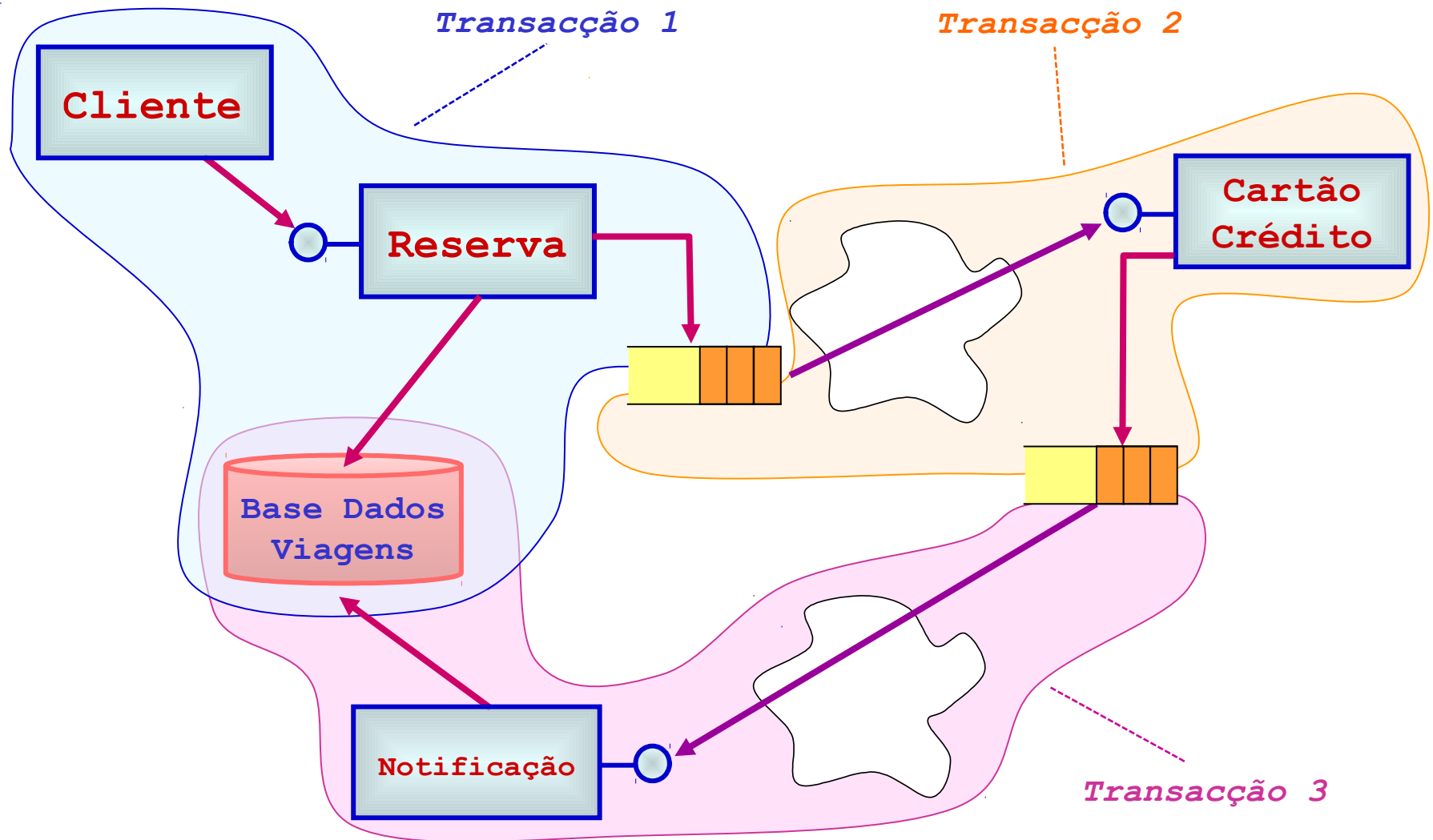
Transacções longas com *pseudo-conversações* (exemplo)

```
create proc actualizar @numero int, @nome varchar(40), @salario int,
                                @token uniqueidentifier
as
declare @oldToken uniqueidentifier
delete from controlo where GetDate() > dateadd(mi,2,dataEntrada)
set transaction isolation level read committed
begin tran
select @oldToken=token from controlo where numero=@numero
if (@@RowCount = 0) or (@oldToken<>@token)
begin
    rollback tran
    return 1 /* Erro - funcionario nao detido por este utilizador*/
end
update funcionario set nome=@nome,salario=@salario
                                where numero=@numero
delete from controlo where numero = @numero
commit tran
return 0
```

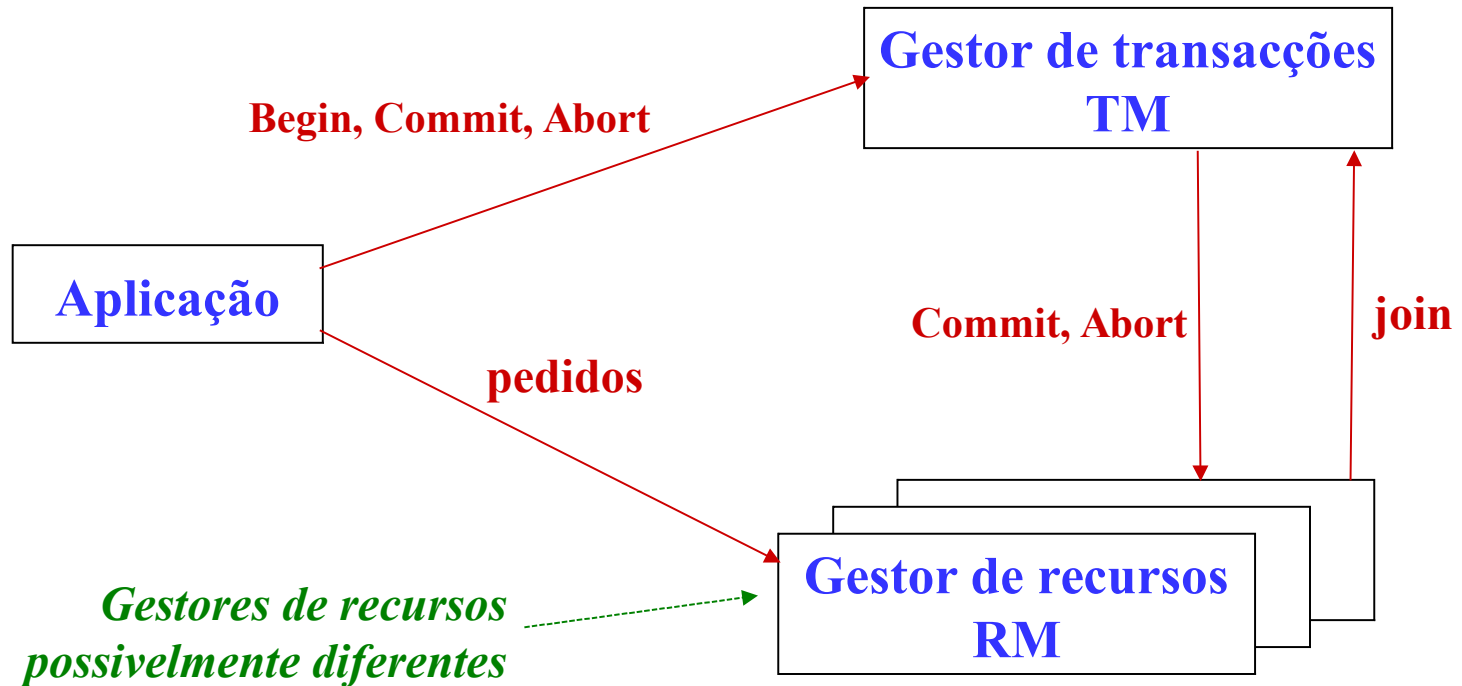
Transacções longas com filas de mensagens (trans. assíncronas)



Transacções assíncronas - Exemplo



Sistema transaccional



Demarcação transaccional

➤ *como associar o trabalho realizado por vários componentes distribuídos a uma transacção global?*

- **Associação explícita:**
 - da responsabilidade da aplicação
- **Associação implícita:**
 - realizada pela monitorização das mensagens trocadas pelos vários componentes

Alistamento transaccional directo

```
string cs;  
SqlConnection cn;  
SqlCommand cmd;  
SqlTransaction tr;
```

```
cs = ConfigurationManager.ConnectionStrings["base dados"].ConnectionString;  
cn = new SqlConnection(cs);
```

① **cn.Open();**

② **tr = cn.BeginTransaction();**
cmd = cn.CreateCommand();
cmd.CommandText = "update contas set saldo = 2000 where numero = 1111";
cmd.Transaction = tr;
cmd.ExecuteNonQuery();
cmd.CommandText = "delete from contas where numero = 2222";
cmd.ExecuteNonQuery();
//tr.Rollback();
tr.Commit();
cn.Dispose();

Alistamento transacional invertido

```
using (TransactionScope ts =  
    new TransactionScope(TransactionScopeOption.Required)) {  
  
    SqlConnection cn1 = new SqlConnection(cs);  
    SqlCommand cmd1 = new SqlCommand();  
    ② cn1.Open();  
    cmd1.CommandText = "update contas set saldo = 2002 where numero = 3333";  
    cmd1.Connection = cn1;  
    cmd1.ExecuteNonQuery();  
    cmd1.CommandText = "delete from contas where numero = 4444";  
    cmd1.ExecuteNonQuery();  
    cn1.Dispose();  
    ts.Complete();  
}
```


Demarcação transaccional implícita

As características transaccionais são associadas às classes (ou componentes) que implementam a funcionalidade. O ambiente de execução inicia e termina as transações de forma transparente para os programadores.

Uma forma habitual, consiste na definição das características transaccionais de forma declarativa.

Exemplo com WCF:

```
[ServiceBehavior(
    TransactionIsolationLevel =
        System.Transactions.IsolationLevel.ReadCommitted,
    TransactionAutoCompleteOnSessionClose = false)]
public class Servico : IServico
{
    [OperationBehavior(TransactionAutoComplete = false,
        TransactionScopeRequired = true
    )]
    public void alterarConta(int nc, float valor) { .... }
    ...
}
```

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

Informação sobre System.Transactions

<http://technet.microsoft.com/en-us/library/ms131084.aspx>