ACH2025

Laboratório de Bases de Dados

Aula 16

Transações

Professora:

- **Fátima L. S. Nunes**

Conceitos

- ✓ Transação: unidade de execução do programa que acessa/atualiza vários itens de dados.
  - Em geral, iniciada por um programa do usuário, escrita em linguagem de alto nível.
  - Delimitada pelas instruções: ***begin transaction*** e ***end transaction***
  - Para garantir integridade dos dados, SGBD deve manter as propriedades ACID: Atomicidade, Consistência, Isolamento e Durabilidade.

Conceitos

- ✓ Propriedades ACID:
  - **Atomicidade**: todas as operações executadas ou nenhuma delas.
  - **Consistência**: execução de transação isolada (sem outra transação simultânea) deve manter consistência dos dados.
  - **Isolamento**: uma transação não “percebe” outra transação – para uma determinada transação, ou transação terminou antes dela ou começou depois.
  - **Durabilidade**: após uma transação completada, mudanças persistem no BD, mesmo se houver falhas no sistema.

Conceitos

- ✓ Propriedades ACID:
  - Exemplo: transações acessando dados com as operações:

    read (x): transfere dado do BD para o buffer

    write (x): transfere dado do buffer para o BD
  - Exemplo clássico de transferência de dinheiro entre 2 contas bancárias:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

Conceitos

- ✓ Propriedades ACID:
  - Exemplo clássico de transferência de dinheiro entre 2 contas bancárias:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

Consistência?

Conceitos

- ✓ Propriedades ACID:
  - Exemplo clássico de transferência de dinheiro entre 2 contas bancárias:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

Consistência:

-soma de A e B permanecem inalteradas no final

-garantir a consistência de uma transação individual é dever do programador da aplicação

Conceitos

- ✓ Propriedades ACID:
  - Exemplo clássico de transferência de dinheiro entre 2 contas bancárias:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

Atomicidade?

Conceitos

- ✓ Propriedades ACID:
  - Exemplo clássico de transferência de dinheiro entre 2 contas bancárias:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

Atomicidade:

-ou faz todas as operações ou desfaz tudo se houver falha no SGBD

- tarefa do SGBD (componente de gerenciamento de transação)

Conceitos

- ✓ Propriedades ACID:
  - Exemplo clássico de transferência de dinheiro entre 2 contas bancárias:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

Durabilidade?

Conceitos

- ✓ Propriedades ACID:
  - Exemplo clássico de transferência de dinheiro entre 2 contas bancárias:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

Durabilidade:

-quando a transação terminar, os valores da nova conta persistirão no BD, mesmo se houver falha do sistema

- tarefa do SGBD (componente de gerenciamento de recuperação)

Conceitos

- ✓ Propriedades ACID:
  - Exemplo clássico de transferência de dinheiro entre 2 contas bancárias:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

Isolamento?

Conceitos

- ✓ Propriedades ACID:
  - Exemplo clássico de transferência de dinheiro entre 2 contas bancárias:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

Isolamento:

-se houver outra transação sendo executada ao mesmo tempo, esta não influenciará na transação atual

- tarefa do SGBD (componente de controle de concorrência)

Estados de transação

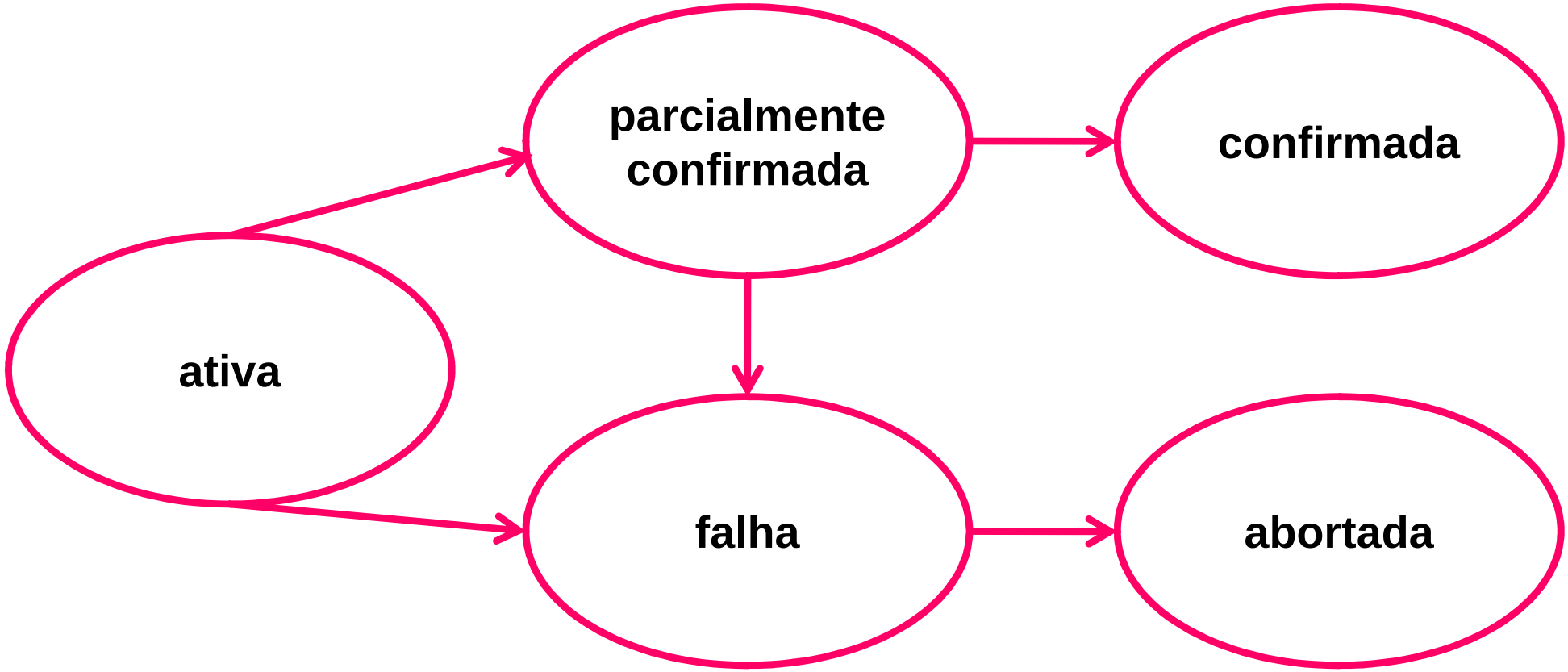
- ✓ Se SGBD não apresentar falhas: transação completada com sucesso ➔ transação confirmada (***commited***).
- ✓ Em caso de falhas: transação abortada➔ transação revertida (***rolled back***).
- ✓ Após confirmada, transação somente poderá ser revertida se houve uma *transação de compensação* ➔ responsabilidade do usuário.

Estados de transação

- ✓ Estados possíveis de uma transação:
  - ativa: enquanto está executando;
  - parcialmente confirmada: após execução instrução final;
  - falha: quando descobre-se que execução normal não pode prosseguir;
  - abortada: depois que transação foi revertida e o BD foi restaurado ao estado anterior ao início da transação;
  - confirmada: após término bem sucedido.

Estados de transação

- ✓ Diagrama de estados:



Estados de transação

- ✓ SGBD pode:
  - reiniciar a transação: somente se tiver sido abortada como resultado de algum erro de hw ou sw que não foi criado por meio da lógica interna da transação.
    - transação reiniciada é considerada nova transação.
  - matar a transação: devido a algum erro lógico interno que só pode ser corrigido com reescrita do programa de aplicação.
    - exemplos de motivos: entrada defeituosa de dados, dados não encontrados no BD.
  - maioria dos SGBDs não permite escritas externas observáveis em transações (impressões, caixa eletrônico)...

Por quê ???

Estados de transação

- ✓ SGBD pode:
  - reiniciar a transação; somente se tiver sido abortada como resultado de algum erro de hw ou sw que não foi criado por meio da lógica interna da transação.
    - transação reiniciada é considerada nova transação.
  - matar a transação: devido a algum erro lógico interno que só pode ser corrigido com reescrita do programa de aplicação.
    - exemplos de motivos: entrada defeituosa de dados, dados não encontrados no BD.
  - maioria dos SGBDs não permite escritas externas observáveis em transações (impressões, caixa eletrônico)...

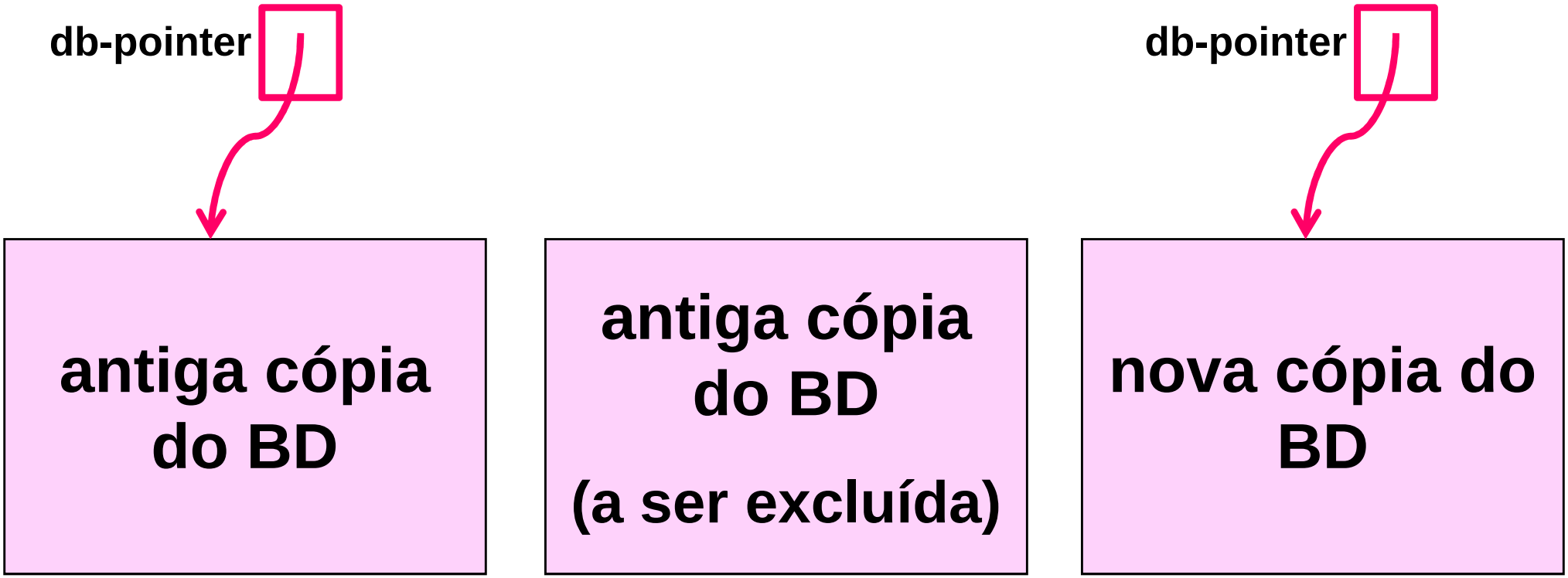
Por quê ??? Garantia de atomicidade !!!

Implementação de atomicidade e durabilidade

- ✓ Responsabilidade: SGBD – componente de gerenciamento de recuperação
- ✓ Como poderiam ser implementadas essas propriedades ???

Implementação de atomicidade e durabilidade

- ✓ Responsabilidade: SGBD – componente de gerenciamento de recuperação
- ✓ Esquema mais simples: cópia de sombra



Implementação de atomicidade e durabilidade

- ✓ Esquema mais simples: cópia de sombra
  - considera somente uma transação ativa ao mesmo tempo;
  - considera BD como simplesmente um arquivo em disco;
  - transação que quer atualizar BD cria cópia completa dele;
  - todas atualizações feitas sobre a cópia;
  - se transação abortada, a nova cópia é excluída;
  - transação concluída:
    - SO confirma se todas páginas da cópia foram gravadas em disco;
    - atualiza *db-pointer* para apontar para nova cópia;
    - exclui cópia antiga.

Implementação de atomicidade e durabilidade

- ✓ Esquema mais simples: cópia de sombra
  - tratamento de falhas:
    - falha de transação: exclui a nova cópia do BD
    - falha do sistema:
      - se *db-pointer* não foi atualizado no disco ➔ quando sistema for reiniciado, lerá o *db-pointer* e verá conteúdo original do BD;
      - se *db-pointer* foi atualizado ➔ todas páginas foram gravadas no disco ➔ quando sistema for reiniciado, lerá o *db-pointer* e verá conteúdo depois de todas atualizações serem realizadas pela transação.
    - gravação do *db-pointer* tem que ser atômica: todos os seus bytes escritos no disco! Sistemas de discos garantem isso (atualizações atômicas de blocos ou setores).

Implementação de atomicidade e durabilidade

- ✓ Esquema mais simples: cópia de sombra
  - Desvantagens ???

Implementação de atomicidade e durabilidade

- ✓ Esquema mais simples: cópia de sombra
  - Desvantagens ???
    - cópia inteira do BD;
    - não permite transações simultâneas.
  - Outros esquemas: mais para frente, em recuperação de falhas.

Execuções simultâneas

- ✓ Executar transações serialmente é mais fácil, mas há motivos para permitir concorrência. Quais ???

Execuções simultâneas

- ✓ Executar transações serialmente é mais fácil, mas há motivos para permitir concorrência. Quais ???
  - Melhor *throughput* (número de transações executadas em determinada quantidade de tempo);
  - Melhor utilização de recursos (processador, E/S);
  - Tempo de espera reduzido (transações curtas e longas).
- ✓ Qual é o problema de execução simultânea de transações?

Execuções simultâneas

- ✓ Executar transações serialmente é mais fácil, mas há motivos para permitir concorrência. Quais ???
  - Melhor *throughput* (número de transações executadas em determinada quantidade de tempo);
  - Melhor utilização de recursos (processador, E/S);
  - Tempo de espera reduzido (transações curtas e longas).
- ✓ Qual é o problema de execução simultânea de transações?
  - Consistência!

Execuções simultâneas

- ✓ Schedule???

Execuções simultâneas

- ✓ Schedule: ordem cronológica de executar transações

T1:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

T2:

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A)
read (B);
B := B + temp;
write (B);
```

Como garantir consistência ao final da execução das duas transações?

Execuções simultâneas

- ✓ Schedule: ordem cronológica de executar transações

T1:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

T2:

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A)
read (B);
B := B + temp;
write (B);
```

Como garantir consistência ao final da execução da duas transações?

Supondo saldo atual = 1000 e 2000 para A e B, respectivamente. Soma final deve ser = 3000.

Execuções simultâneas

- ✓ Alternativa 1: uma transação após a outra

T1:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

T2:

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A)
read (B);
B := B + temp;
write (B);
```

Execuções simultâneas

- ✓ Alternativa 1: uma transação após a outra

T1:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

T2:

Valores iniciais:

A=1000

B=2000

Valores finais de A e B ?

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A)
read (B);
B := B + temp;
write (B);
```

Execuções simultâneas

- ✓ Alternativa 1: uma transação após a outra

T1:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

T2:

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A)
read (B);
B := B + temp;
write (B);
```

Valores finais de A e B ?

855 e 2145

Execuções simultâneas

- ✓ Alternativa 1: uma transação após a outra

T1:	T2:

read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
	read(A);
	temp := A * 0,1;
	A := A – temp;
	write (A)
	read (B);
	B := B + temp;
	write (B);

Execuções simultâneas

- ✓ Alternativa 1: uma transação após a outra

T1:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

T2:

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A)
read (B);
B := B + temp;
write (B);
```

Valores finais de A e B ?

Execuções simultâneas

- ✓ Alternativa 1: uma transação após a outra

T1:

```
read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
```

T2:

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A)
read (B);
B := B + temp;
write (B);
```

Valores finais de A e B ?

850 e 2150

Execuções simultâneas

- ✓ Schedule:
  - ordem cronológica (sequencial) de executar transações;
  - precisa consistir todas as instruções da transação;
  - precisa preservar a ordem das instruções em cada transação;
  - os *schedules* vistos são seriais;
  - Quantos *schedules* seriais válidos existem para um conjunto de ***n*** transações ???

Execuções simultâneas

- ✓ Schedule:
  - ordem cronológica de executar transações;
  - precisa consistir todas as instruções da transação;
  - precisa preservar a ordem das instruções em cada transação;
  - os *schedules* vistos são seriais
  - Quantos *schedules* seriais válidos existem para um conjunto de ***n*** transações ??? ***n!***

Execuções simultâneas

- ✓ Schedule:
  - ordem cronológica de executar transações;
  - precisa consistir todas as instruções da transação;
  - precisa preservar a ordem das instruções em cada transação;
  - os *schedules* vistos são seriais
  - Quantos *schedules* seriais válidos existem para um conjunto de ***n*** transações ??? ***n!***
  - *Schedule* serial executa transações simultaneamente?

Execuções simultâneas

- ✓ Schedule:
  - ordem cronológica de executar transações;
  - precisa consistir todas as instruções da transação;
  - precisa preservar a ordem das instruções em cada transação;
  - os *schedules* vistos são seriais
  - Quantos *schedules* seriais válidos existem para um conjunto de ***n*** transações ??? ***n!***
  - *Schedule* serial executa transações simultaneamente? **NÃO!**

Execuções simultâneas

- ✓ Se SGBD executa transações simultaneamente:
  - *schedule* não precisa ser serial;
  - SO pode comutar entre transações: fatia de tempo para cada uma delas.
  - Vantagem?

Execuções simultâneas

- ✓ Se SGBD executa transações simultaneamente:
  - *schedule* não precisa ser serial;
  - SO pode comutar entre transações: fatia de tempo para cada uma delas.
  - Vantagem? Compartilhamento de recursos.
  - Várias sequências de execução possíveis ➔ intercalação de instruções:
    - difícil prever quantas instruções de cada transação serão executadas em sua fatia de tempo;
    - quantos *schedules* são possíveis para ***n*** transações?

Execuções simultâneas

- ✓ Se SGBD executa transações simultaneamente:
  - *schedule* não precisa ser serial;
  - SO pode comutar entre transações: fatia de tempo para cada uma delas.
  - Vantagem? Compartilhamento de recursos.
  - Várias sequências de execução possíveis ➔ intercalação de instruções:
    - difícil prever quantas instruções de cada transação serão executadas em sua fatia de tempo;
    - quantos *schedules* são possíveis para ***n*** transações?

    Muito maior que ***n!***

Execuções simultâneas

- ✓ Alternativa 2: intercalando instruções

T1:	T2:

`read(A);`
`A := A – 50;`
`write (A)`	

	`read(A);`
	`temp := A * 0,1;`
	`A := A – temp;`
	`write (A)`

`read (B);`
`B := B + 50;`
`write (B);`	

	`read (B);`
	`B := B + temp;`
	`write (B);`

Execuções simultâneas

- ✓ Alternativa 2: intercalando instruções

T1:

```
read(A);
A := A – 50;
write (A)
```

```
read (B);
B := B + 50;
write (B);
```

T2:

Início: A=1000, B=2000

Valores finais de A e B ?

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A)
```

```
read (B);
B := B + temp;
write (B);
```

Execuções simultâneas

- ✓ Alternativa 2: intercalando instruções

T1:

```
read(A);
A := A – 50;
write (A)
```

```
read (B);
B := B + 50;
write (B);
```

Início: A=1000, B=2000

Valores finais de A e B ?

855 e 2145

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A)
```

```
read (B);
B := B + temp;
write (B);
```

Execuções simultâneas

- ✓ Todas as execuções concorrentes resultam em um estado correto ?

Execuções simultâneas

- ✓ Todas as execuções concorrentes resultam em um estado correto ?

T1:

```
read(A);
A := A – 50;
```

```
write (A);
read (B);
B := B + 50;
write (B);
```

T2:

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A);
read(B);
```

Valores finais de A e B ?

```
B := B + temp;
write (B);
```

Execuções simultâneas

- ✓ Todas as execuções concorrentes resultam em um estado correto ?

T1:

```
read(A);
A := A – 50;
```

```
write (A);
read (B);
B := B + 50;
write (B);
```

T2:

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A);
read(B);
```

Valores finais de A e B ?

950 e 2100

```
B := B + temp;
write (B);
```

Execuções simultâneas

- ✓ Todas as execuções concorrentes resultam em um estado correto ?

T1:

Onde está o erro?

```
read(A);
A := A – 50;
```

```
write (A);
read (B);
B := B + 50;
write (B);
```

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A);
read(B);
```

```
B := B + temp;
write (B);
```

Execuções simultâneas

- ✓ Todas as execuções concorrentes resultam em um estado correto ?

T1:

Onde está o erro?

```
read(A);
A := A – 50;
```

```
write (A);
read (B);
B := B + 50;
write (B);
```

```
read(A);
temp := A  * 0,1;
A := A – temp;
write (A);
read(B);
```

```
B := B + temp;
write (B);
```

Execuções simultâneas

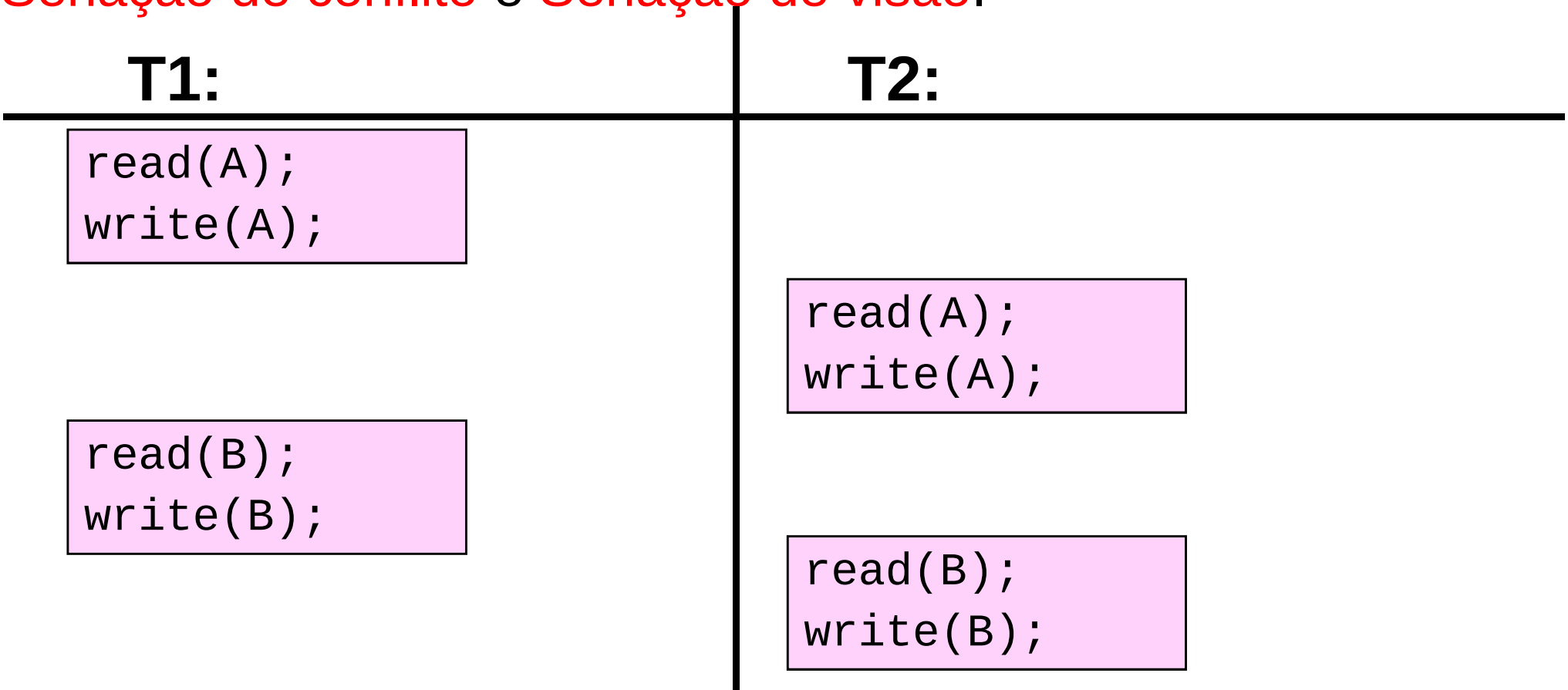
- ✓ Se o controle da execução ficar a cargo do SO, muitos *schedules* errados serão possíveis.
- ✓ Por isso, é tarefa do SGBD garantir que qualquer *schedule* executado deixe o BD em estado consistente.
  - ✓ o ***schedule* precisa ser equivalente** a um *schedule* serial.
- ✓ Responsável por isso: **componente de controle de concorrência.**

– Então, a questão se resume a: obter um *schedule* **equivalente** a um *schedule* serial.

– Há estratégias para isso.

Execuções simultâneas - Seriação

- Difícil saber quais são realmente as operações de uma transação.
- Para efeitos de escalonamento, interessam somente as instruções: read (Q) e write (Q).
- Seriação de conflito e Seriação de visão.



Execuções simultâneas – Seriação de conflito

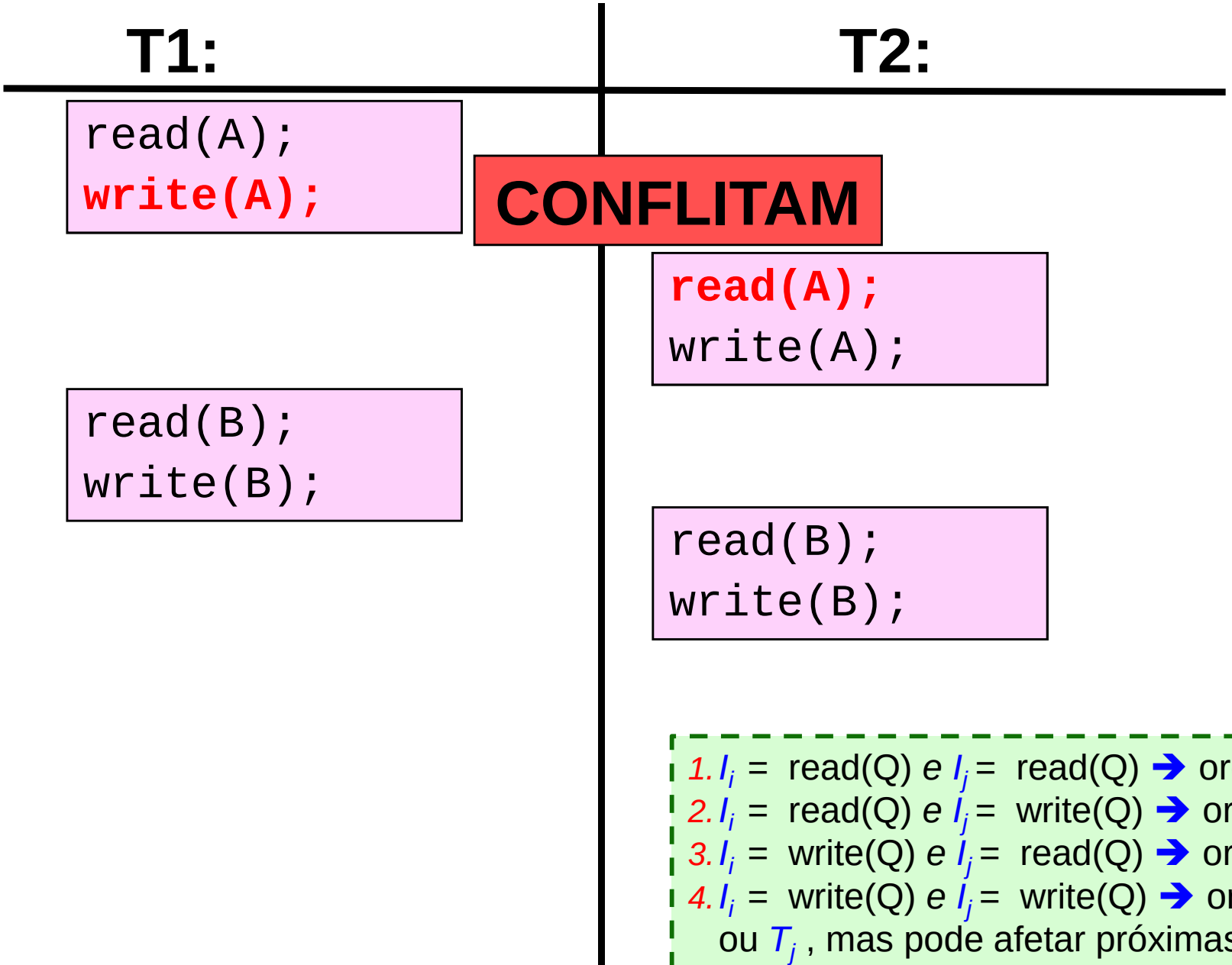
- Schedule *S* com duas instruções consecutivas I_i *e* I_j, pertencentes às transações T_i *e* T_j , respectivamente.
- Se I_i *e* I_j se referem a diferentes itens de dados ➔ podemos inverter instruções I_i *e* I_j consecutivas sem afetar resultados de qualquer instrução do *schedule.*
- Se I_i *e* I_j se referem ao mesmo item dado ➔ ordem das duas etapas pode importar.

Execuções simultâneas – Seriação de conflito

- 4 casos a considerar:
  1. I_i = read(Q) *e* I_j = read(Q) ➔ ordem não importa
  2. I_i = read(Q) *e* I_j = write(Q) ➔ ordem importa:
     - Se I_i vem antes de I_j, T_i não lê o valor de Q escrito por T_j
     - Se I_j vem antes de I_i, T_i lê o valor de Q escrito por T_j
  3. I_i = write(Q) *e* I_j = read(Q) ➔ ordem importa: pelos mesmos motivos
  4. I_i = write(Q) *e* I_j = write(Q) ➔ ordem não afeta T_i ou T_j , mas:
     - valor obtido pela próxima instrução read(Q) do *schedule S* é afetado;
     - se não houver outra instrução write (Q) após I_i e I_j em *S,* a ordem das instruções afeta diretamente o valor final de Q no estado do banco de dados que resulta do *schedule S.*

Execuções simultâneas – Seriação de conflito

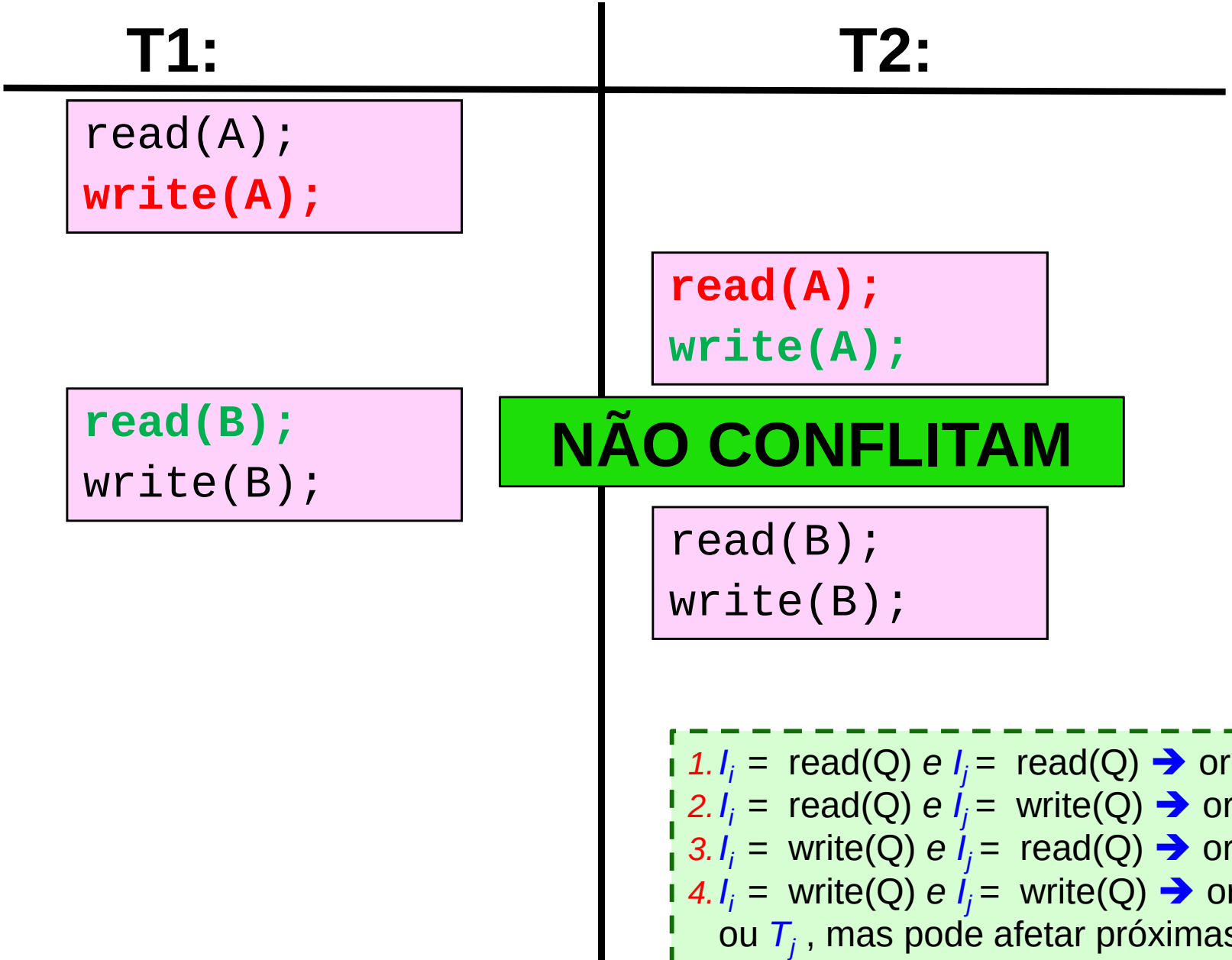
– I_i e I_j **conflitam** se houver pelo menos uma operação write



1. I_i = read(Q) *e* I_j = read(Q) ➔ ordem não importa
2. I_i = read(Q) *e* I_j = write(Q) ➔ ordem importa
3. I_i = write(Q) *e* I_j = read(Q) ➔ ordem importa
4. I_i = write(Q) *e* I_j = write(Q) ➔ ordem não afeta T_i ou T_j , mas pode afetar próximas instruções

Execuções simultâneas – Seriação de conflito

– I_i e I_j **conflitam** se houver pelo menos uma operação write



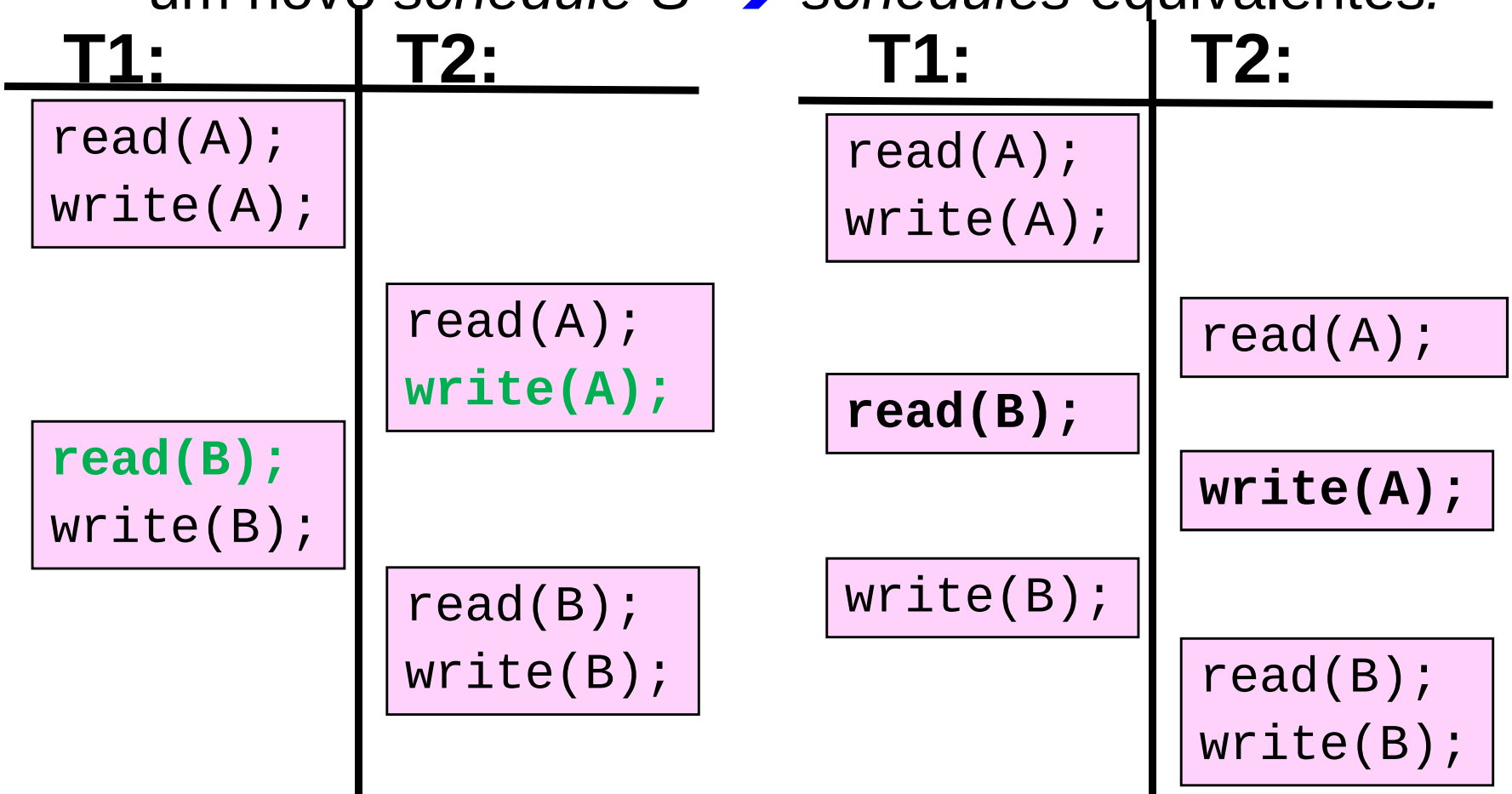
1. I_i = read(Q) *e* I_j = read(Q) ➔ ordem não importa
2. I_i = read(Q) *e* I_j = write(Q) ➔ ordem importa
3. I_i = write(Q) *e* I_j = read(Q) ➔ ordem importa
4. I_i = write(Q) *e* I_j = write(Q) ➔ ordem não afeta T_i ou T_j , mas pode afetar próximas instruções

Execuções simultâneas – Seriação de conflito

- Considerando I_i e I_j como instruções consecutivas de um *schedule* S:
  - Se I_i e I_j forem instruções de diferentes transações e não conflitarem:
    - podemos inverter a ordem de I_i e I_j , gerando um um novo *schedule S'* ➔ *schedules* equivalentes.

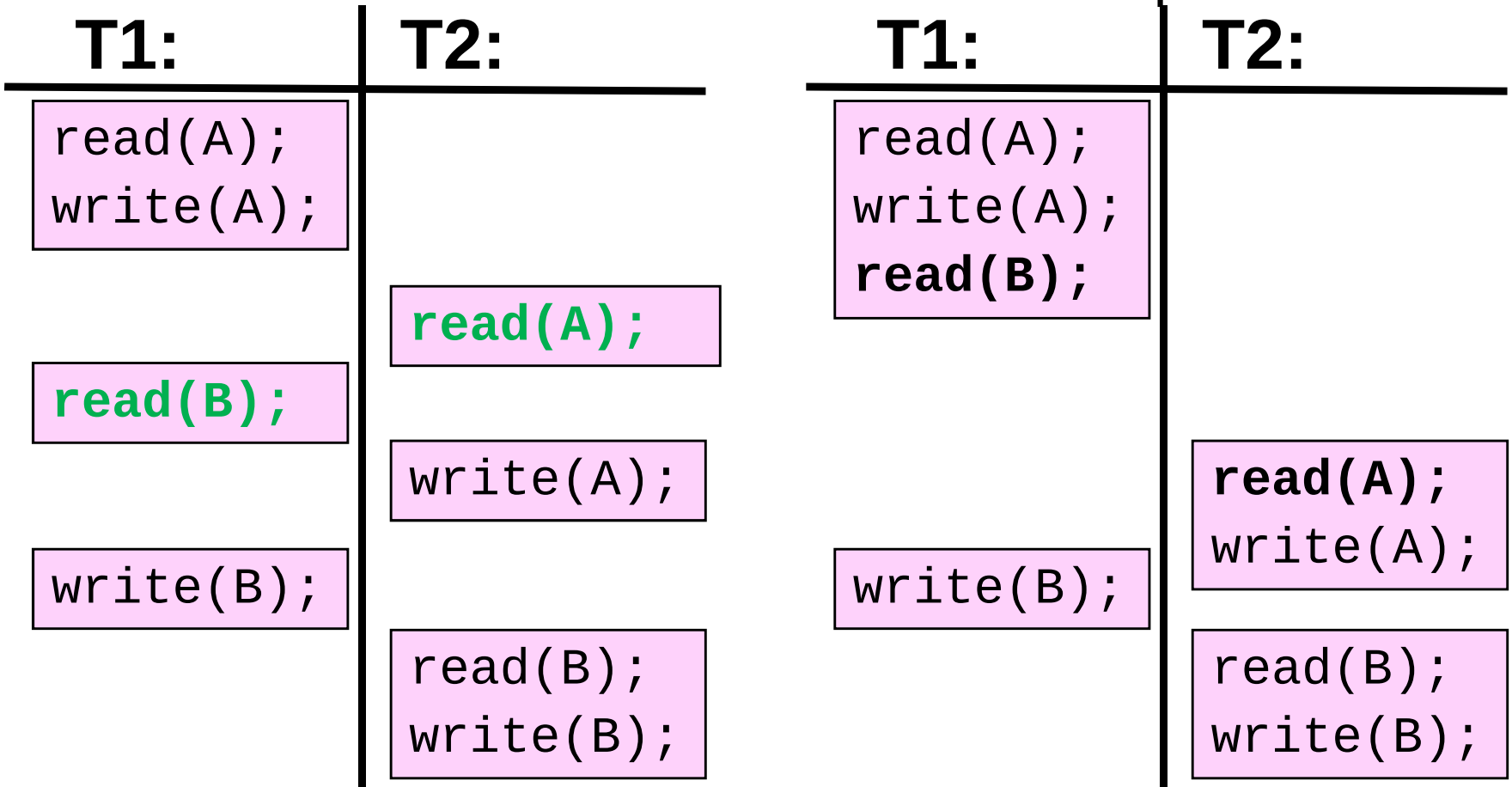
Execuções simultâneas – Seriação de conflito

- Considerando I_i e I_j como instruções consecutivas de um *schedule* S:
  - Se I_i e I_j forem instruções de diferentes transações e não conflitarem, podemos inverter a ordem I_i e I_j , gerando um um novo *schedule S'* ➔ *schedules* equivalentes.



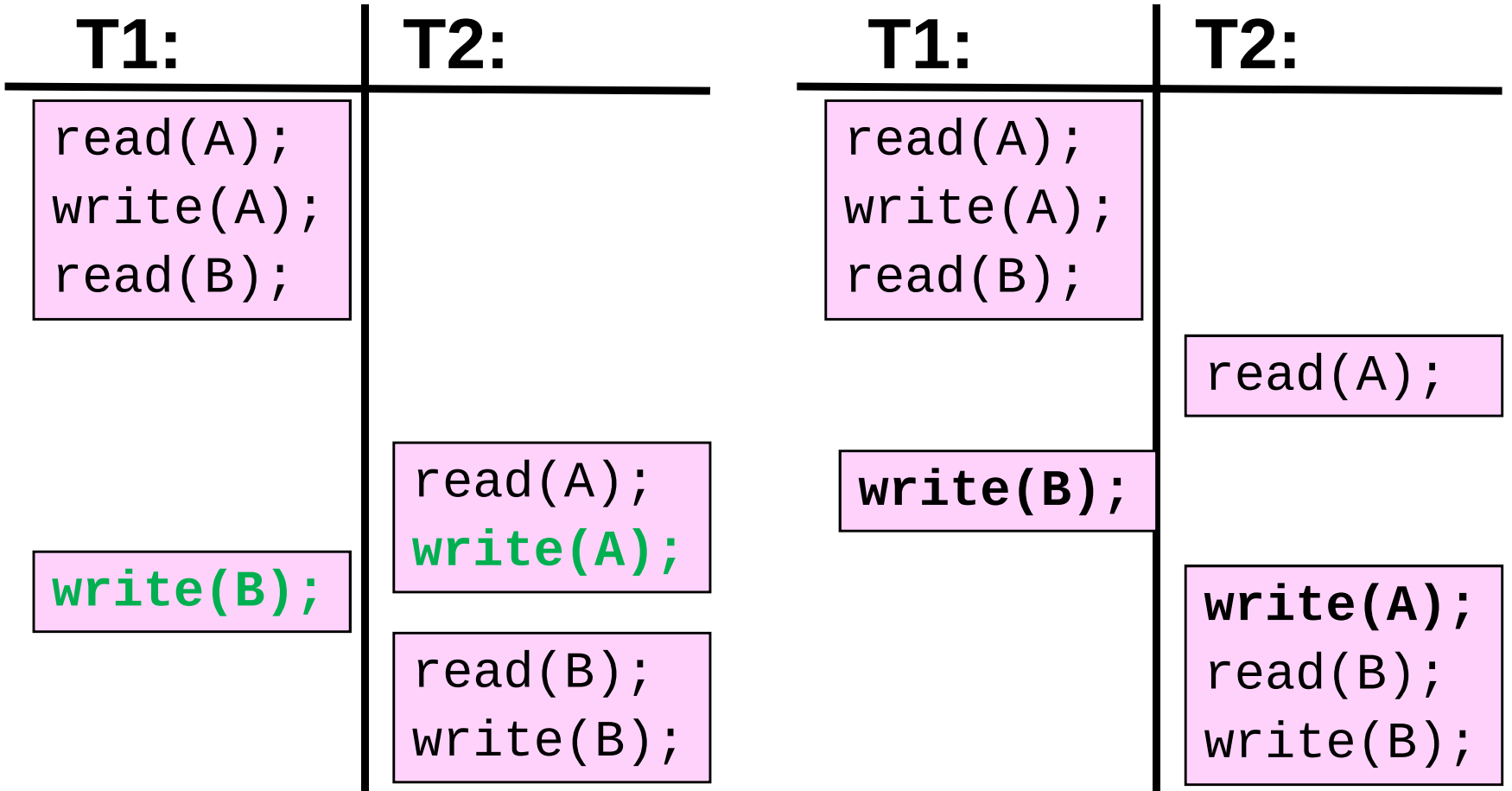
Execuções simultâneas – Seriação de conflito

- Considerando I_i e I_j como instruções consecutivas de um *schedule* S:
  - Se I_i e I_j forem instruções de diferentes transações e não conflitarem, podemos inverter a ordem I_i e I_j , gerando um um novo *schedule S'* ➔ *schedules* equivalentes.



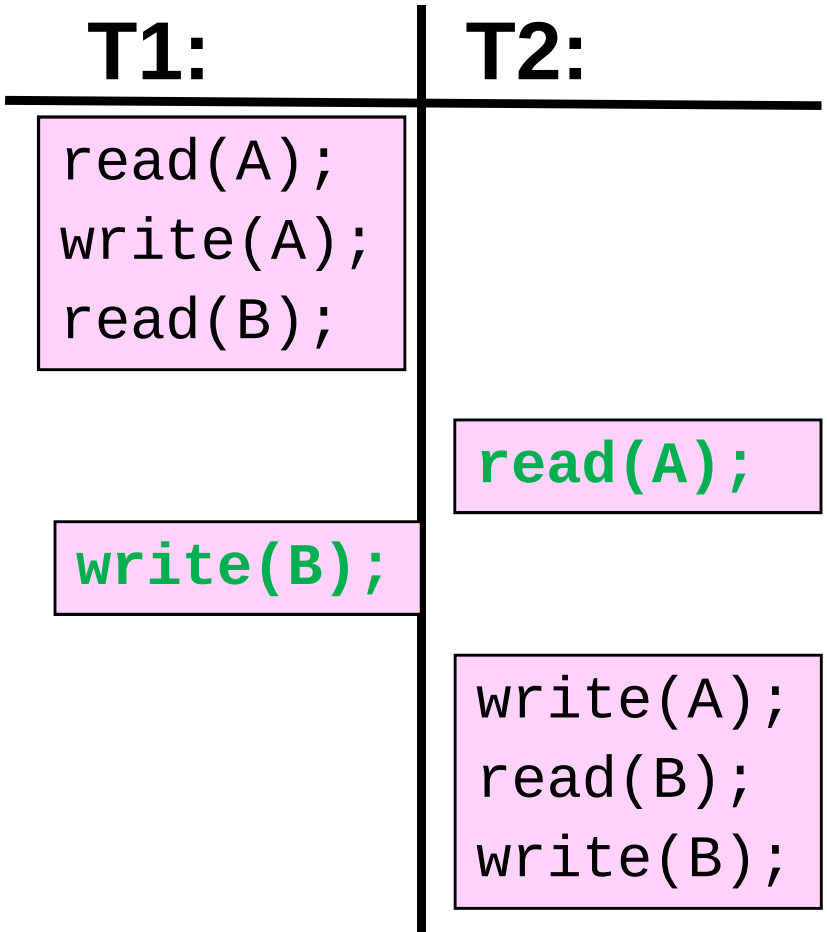
Execuções simultâneas – Seriação de conflito

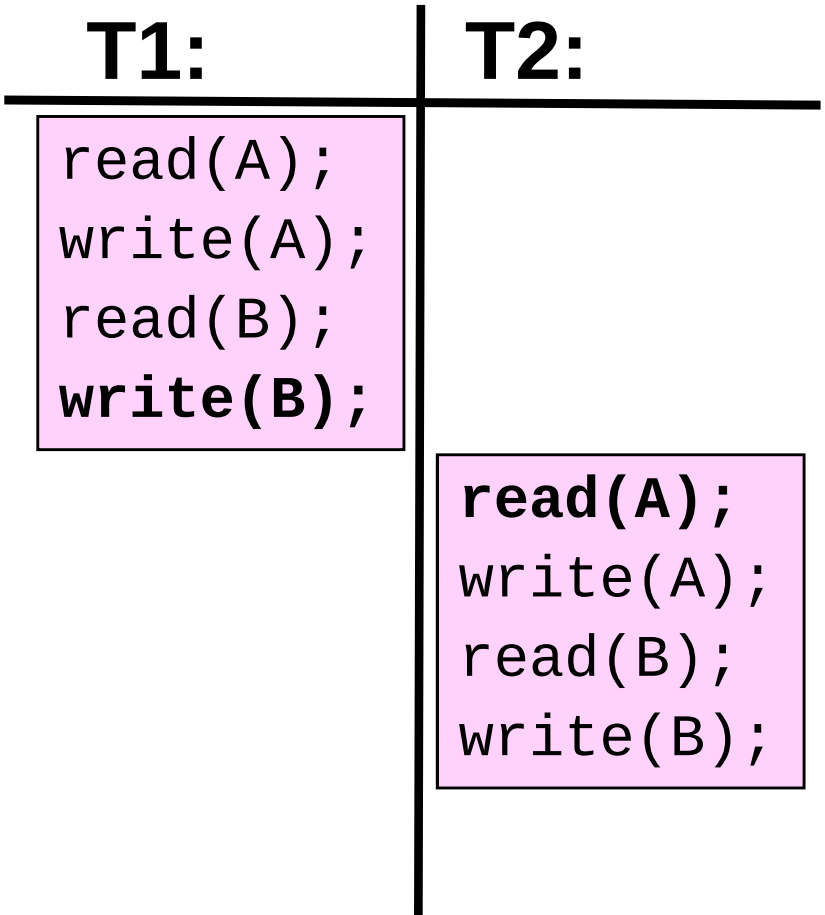
- Considerando I_i e I_j como instruções consecutivas de um *schedule* S:
  - Se continuarmos a inverter as instruções não conflitantes, obtemos um *schedule* equivalente a um *schedule serial.*



Execuções simultâneas – Seriação de conflito

- Considerando I_i e I_j como instruções consecutivas de um *schedule* S:
  - Se continuarmos a inverter as instruções não conflitantes, obtemos um *schedule* equivalente a um *schedule serial.*





Execuções simultâneas – Seriação de conflito

- Considerando I_i e I_j como instruções consecutivas de um *schedule* S:
  - Se um *schedule S* puder ser transformado em um *schedule* S´ por uma série de trocas de instruções não conflitantes, dizemos que *S* e *S'* são **equivalentes em conflito.**
  - Um *schedule* é **serial de conflito** se for equivalente em conflito a um *schedule* serial.

Execuções simultâneas – Seriação de visão

- Menos rigorosa do que equivalência em conflito.
- Dois *schedules S* e *S'* são **equivalentes em visão** se:
  1. Para cada item de dados Q, se a transação T_i ler o valor de Q no *schedule S*
     - então a transação T_i em *S'* também precisa ler o valor inicial de Q.
     - Ou seja: *schedule* inicial e final têm que ter um read (Q) correspondente

Execuções simultâneas – Seriação de visão

- Menos rigorosa do que equivalência em conflito.
- Dois *schedules S* e *S'* são **equivalentes em visão** se:

2. Para cada item de dados Q, se a transação T_i executar read(Q) no *schedule S:*
   - se este valor foi produzido por uma operação write(Q) executada pela transação T_j
     - então a operação read(Q) da transação T_i no *schedule S'* também precisa ler o valor inicial de Q que foi produzido pela mesma operação write (Q) da transação T_j .

Execuções simultâneas – Seriação de visão

- Menos rigorosa do que equivalência em conflito.
- Dois *schedules S* e *S'* são **equivalentes em visão** se:

3. Para cada item de dados Q:
   - a transação que realiza a operação write(Q) final no *schedule S* precisa realizar a operação write(Q) final no *schedule S´.*

Execuções simultâneas – Seriação de visão

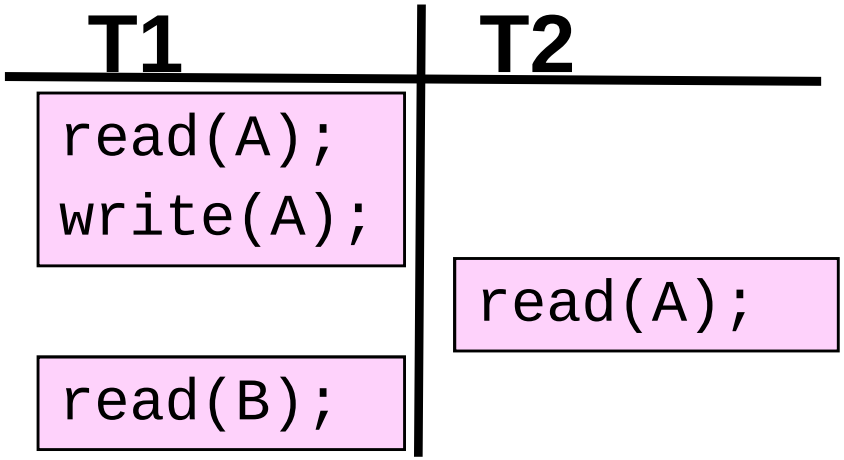
- Um *schedule S* é **serial de visão** se for equivalente em visão a um *schedule* serial.

- Exercício:
  - Verificar nos *schedules* exemplificados anteriormente, quais são seriais em visão.

Facilidade de recuperação

- Se houver falhas da transação durante a execução simultânea, é necessário desfazer o efeito dessa transação para garantir a atomicidade.
- Quais *schedules são* aceitáveis para garantir recuperação de falhas?
  - *schedules* recuperáveis
  - *schedules* não em cascata

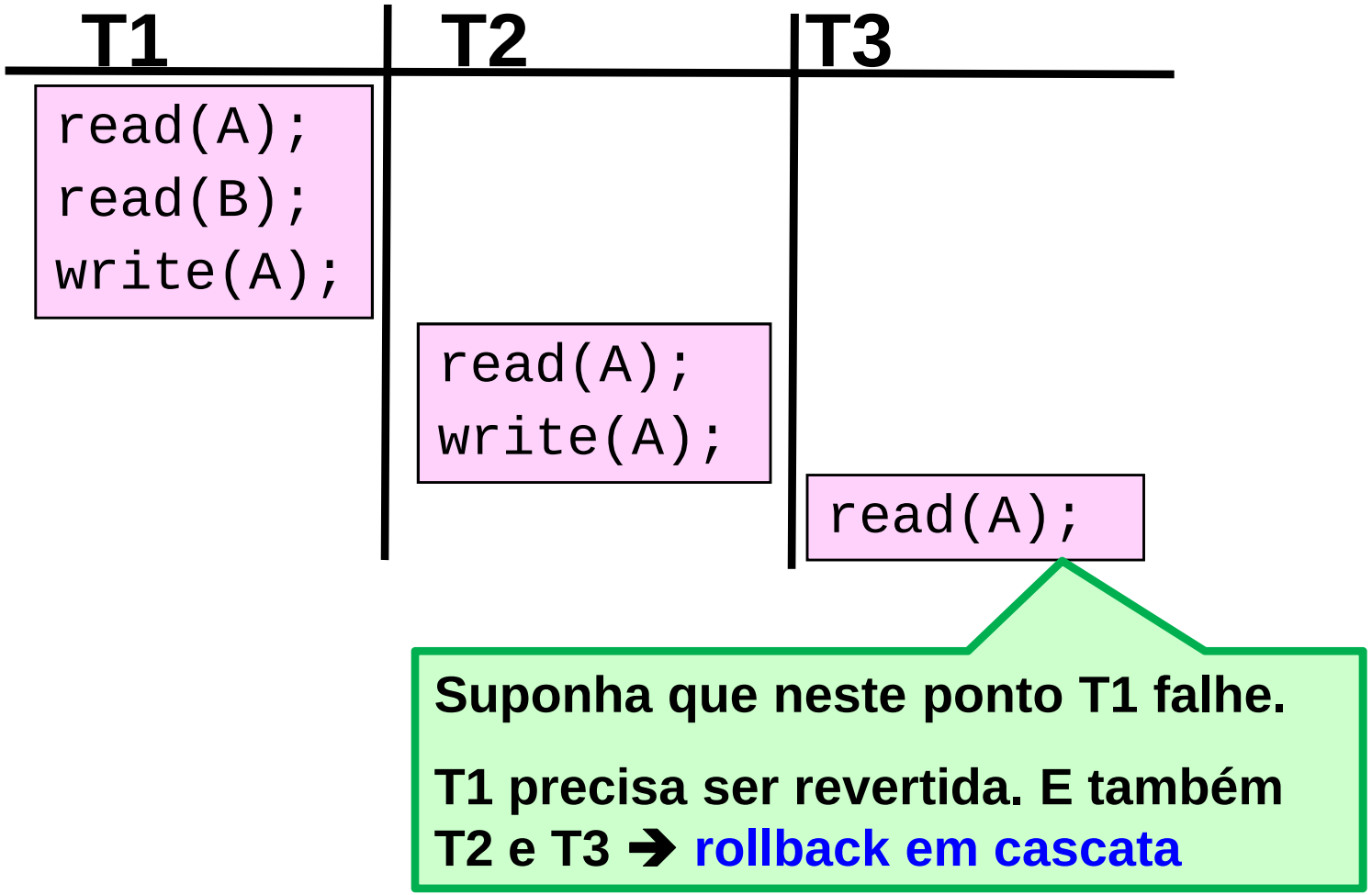
Facilidade de recuperação – *Schedules* recuperáveis



- Suponha que T2 seja confirmada imediatamente após execução de read(A) (antes de T1 ser confirmada).
- Se T1 falhar antes de ser confirmada ➔ como T2 leu o valor de A escrito por T1, teria que abortar T2 para garantir atomicidade.
- Mas T2 já foi confirmada e não pode ser abortada ➔ *schedule* não recuperável que não pode ser permitido.
- Maioria dos SGBDs exige que todos *schedulers* sejam recuperáveis:
  - ***schedule recuperável****:* aquele em que, para cada par de transações T_i e T_j, tal que T_j leia um item de dados previamente escrito por T_i, a operação ***commit*** de T_i aparece antes da operação ***commit*** de T_j.

Facilidade de recuperação – *Schedules* não em cascata

- Para que um BD se recupere corretamente de uma falha em uma transação T_i , pode ser necessário reverter várias transações ➔ isso acontece se as transações tiverem lido dados escritos por T_i .



Facilidade de recuperação – *Schedules* não em cascata

– Rollback em cascata é indesejável. Por quê???

Facilidade de recuperação – *Schedules* não em cascata

- Rollback em cascata é indesejável. Por quê???
  Trabalho desperdiçado.

- Preferível restringir *schedules* àqueles em que os rollbacks em cascata não podem ocorrer ➔ ***schedules* não em cascata.**

- ***schedule* não em cascata** ➔ aquele em que, para cada par de transações T_i e T_j, tal que T_j leia um item de dados previamente escrito por T_i, a operação de ***commit*** de T_i aparece antes da operação de ***read*** de T_j.

- ***schedule* não em cascata** também é recuperável.

Implementação de isolamento

- Diversos esquemas de controle de concorrência podem ser usados para garantir que somente *schedules* aceitáveis sejam gerados.
- Exemplo mais simples: transação adquire **bloqueio** do BD inteiro antes de iniciar e libera depois que for confirmada:
  - somente uma transação executada por vez: somente *schedules* seriais são gerados;
  - grau de concorrência baixo, apesar de garantir isolamento.
- Objetivos dos esquemas de controle de concorrência: oferecer alto grau de concorrência enquanto garante que possam ser gerados *schedules* *seriais de* conflito, seriais de visão e não em cascata.

Verificando a seriação

- Esquemas de controle de concorrência geram *schedules* ➔ esses devem ser passíveis de seriação. Como verificar?

- Construir o gráfico de precedência:
  - par **G=(V,E),** onde **V** é um conjunto de vértices e **E** é um conjunto de arestas.
  - vértices: todas as transações participantes do *schedule*
  - arestas: arestas T_i → T_j para as quais uma das condições é verdadeira:
    - T_i executa write(Q) antes que T_j execute read (Q);
    - T_i executa read(Q) antes que T_j execute write(Q);
    - T_i executa write(Q) antes que T_j execute write (Q).
  - se uma aresta T_i → T_j existir no gráfico de precedência, em qualquer *schedule* serial *S´* equivalente a *S,* T_i precisa aparecer antes de T_j

Verificando a seriação

Schedule

T1:	T2:

read(A);
A := A – 50;
write (A)
read (B);
B := B + 50;
write (B);
	read(A);
	temp := A * 0,1;
	A := A – temp;
	write (A)
	read (B);
	B := B + temp;
	write (B);

Gráfico de precedência

– arestas $T_i \rightarrow T_j$:

- T_i executa write(Q) antes que T_j execute read (Q);
- T_i executa read(Q) antes que T_j execute write(Q);
- T_i executa write(Q) antes que T_j execute write (Q).

Verificando a seriação

Schedule

T1:	T2:

	`read(A);`
	`temp := A * 0,1;`
	`A := A – temp;`
	`write (A)`

`read(A);`
`A := A – 50;`
`write (A)`
	`read (B);`
	`B := B + temp;`
	`write (B);`

`read (B);`
`B := B + 50;`
`write (B);`	

Gráfico de precedência

- arestas $T_i \rightarrow T_j$:
  - T_i executa write(Q) antes que T_j execute read (Q);
  - T_i executa read(Q) antes que T_j execute write(Q);
  - T_i executa write(Q) antes que T_j execute write (Q).

Verificando a seriação

Schedule

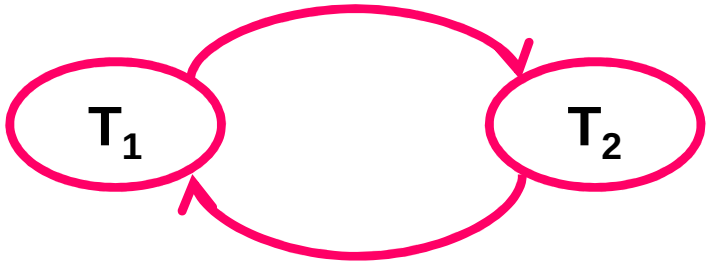
T1:	T2:
`read(A);` `A := A – 50;`	

	`read(A);`
	`temp := A * 0,1;`
	`A := A – temp;`
	`write (A);`
	`read(B);`

`write (A);`
`read (B);`
`B := B + 50;`
`write (B);`	

	`B := B + temp;` `write (B);`

Gráfico de precedência



Se gráfico tiver um ciclo, o *schedule* não é serial de conflito.

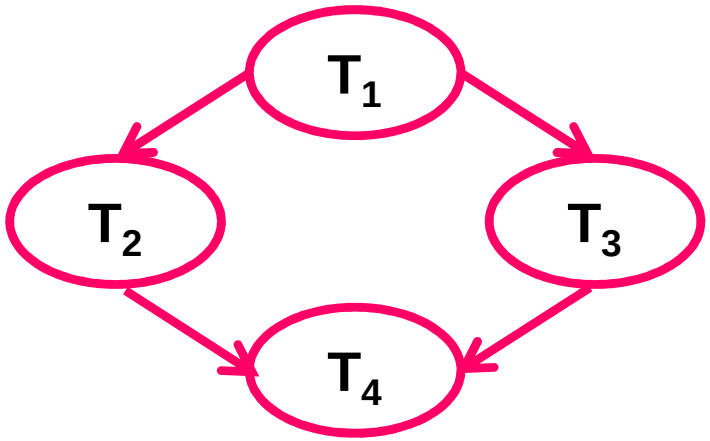
– arestas $T_i \rightarrow T_j$:

- T_i executa write(Q) antes que T_j execute read (Q);
- T_i executa read(Q) antes que T_j execute write(Q);
- T_i executa write(Q) antes que T_j execute write (Q).

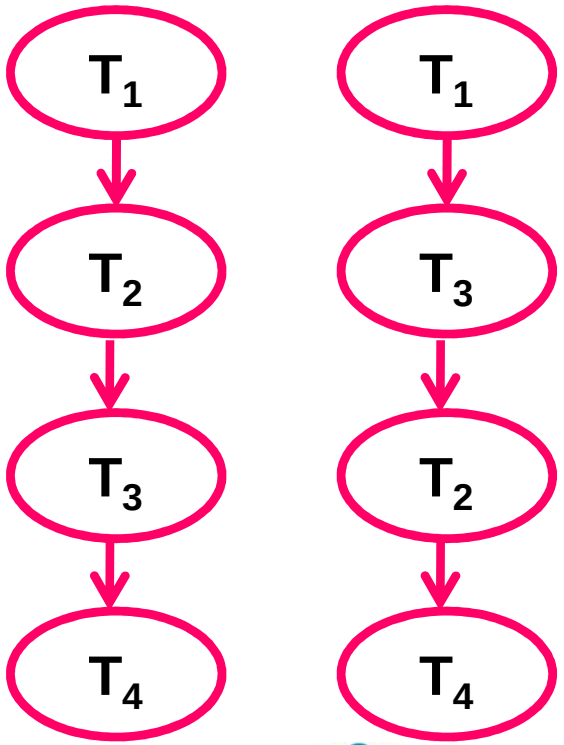
Verificando a seriação

- Ordem de seriação:
  - obtida pela classificação topológica do gráfico de precedência
  - ordem linear consistente com a ordem parcial do gráfico de precedência
  - em geral existem várias ordens de seriação possíveis.

Gráfico de precedência



Possíveis ordens de seriação



Verificando a seriação

- Para testar seriação de conflito:
  - construir gráfico de precedência;
  - executar algoritmos de detecção de ciclos (exemplo: busca em profundidade – complexidade n^2, onde *n* é o número de vértices).
- Ainda não existe algoritmo eficiente para testar seriação de visão: verifica-se as condições suficientes citadas

Bibliografia

- ✓ Silberschatz, A.; Korth, H.F.; Sudarshan, S. “Sistema de Banco de Dados”, 5a. edição, Makron Books, 2006 (capítulo 15)
- ✓ Elmasri, R.; Navathe, S.B. Fundamentals of Database Systems, Benjamin Cummings, 3a edição, 2000 (capítulo 17).
- ✓ Date, C. J. Introdução a Sistemas de Banco de Dados - Tradução da 7ª Edição, 2000 - Editora Campus (capítulo 14).

Exercícios

1. Dado o *schedule* a seguir, verifique se ele é serial de conflito por meio da troca de instruções para gerar schedule equivalente.

Transação1	Transação2	Transação3
read(X)		
write(X)		
		read(Y)
	read(Y)	
	write(Y)	
		write(Z)
	read(X)	
	read(Z)	
write(X)		
read(Y)		
		write(Y)
	write (X)	
read(X)		
		write(X)
	write(Z)	
		write(Z)

Exercícios

2. Dado o *schedule* a seguir, prove, elaborando o gráfico de precedência, se ele é ou não serial de conflito

Transação1	Transação2	Transação3	Transação4
read(X)			
write(X)			
		read(Y)	
	read(Y)		
	write(Y)		
		write(Z)	
	read(X)		
			read(X)
write(X)			
read(Y)			
		write(Y)	
			write(X)
read(X)			
		write(X)	
			write(X)
		write(Z)	

Exercícios

3. Dado o *schedule* a seguir, verifique se ele é serial de visão.

Transação1	Transação2	Transação3
read(X)		
write(X)		
		read(Y)
	read(Y)	
	write(Y)	
		write(Z)
	read(X)	
	read(Z)	
write(X)		
read(Y)		
		write(Y)
	write (X)	
read(X)		
		write(X)
	write(Z)	
		write(Z)

Exercícios

4. O que é um schedule recuperável? Dê um exemplo?
5. Dê um exemplo de schedule em cascata. Qual é o problema deste tipo de schedule?

Exercícios

- ✓ Silberschatz, A.; Korth, H.F.; Sudarshan, S. “Sistema de Banco de Dados”, 5a. edição, Makron Books, 2006 (capítulo 15) – exercícios 1 a 13
- ✓ Elmasri, R.; Navathe, S.B. Fundamentals of Database Systems, Benjamin Cummings, 3a edição, 2000 (capítulo 17) – exercícios 1 a 11, 14 a 20, 22 a 24
- ✓ Date, C. J. Introdução a Sistemas de Banco de Dados - Tradução da 7ª Edição, 2000 - Editora Campus (capítulo 14) – exercícios 1 e 2

ACH2025

Laboratório de Bases de Dados

Aula 16

Transações

Professora:

- **Fátima L. S. Nunes**