Sumário

Controle de Concorrência

- SGBD
  - sistema multi-usuário em geral
    - diversas transações executando simultaneamente
- Garantia de isolamento de Transações
  - 1ª solução: uma transação executa por vez
    - escalonamento serial de transações
    - solução bastante ineficiente!
      - várias transações podem esperar muito tempo para serem executadas
      - CPU pode ficar muito tempo ociosa
        - » enquanto uma transação faz I/O, por exemplo, outras transações poderiam ser executadas

Controle de Concorrência

- Solução mais eficiente
  - execução concorrente de transações de modo a preservar o isolamento
    - escalonamento (*schedule*) não-serial e íntegro
  - responsabilidade do subsistema de controle de concorrência ou *scheduler*

execução serial

T1	T2
read(X)	
X = X – 20	
write(X)	
read(Y)	
Y = Y + 20	
write(Y)	
	read(X)
	X = X + 10
	write(X)

execução não-serial ou concorrente

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
Y = Y + 20	
write(Y)	

Scheduler

- Responsável pela definição de escalonamentos não-seriais de transações
- *"Um escalonamento E define uma ordem de execução das operações de várias transações, sendo que a ordem das operações de uma transação Tx em E aparece na mesma ordem na qual elas ocorrem isoladamente em Tx"*
- Problemas de um escalonamento não-serial mal definido (inválido)
  - atualização perdida (*lost-update*)
  - leitura suja (*dirty-read*)

Atualização Perdida

- Uma transação *Ty* grava em um dado atualizado por uma transação *Tx*

T1	T2
read(X)	
X = X – 20	
	read(Z)
	X = Z + 10
write(X)	
read(Y)	
	write(X)
Y = X + 30	
write(Y)	

a atualização de X por T1 foi perdida!

Leitura Suja

- *Tx* atualiza um dado X, outras transações posteriormente lêem X, e depois *Tx* falha

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	Z = X + 10
	write(Z)
read(Y)	
abort( )	

T2 leu um valor de X que não será mais válido!

Scheduler

- Deve evitar escalonamentos inválidos
  - exige análise de operações em conflito
    - operações que pertencem a transações diferentes
    - transações acessam o mesmo dado
    - pelo menos uma das operações é *write*
  - tabela de situações de conflito de transações
    - podem gerar um estado inconsistente no BD

		Ty	
Tx		read(X)	write(X)
	read(X)		√
	write(X)	√	√

Scheduler X *Recovery*

- *Scheduler* deve cooperar com o *Recovery*!
- Categorias de escalonamentos considerando o grau de cooperação com o *Recovery*
  - recuperáveis X não-recuperáveis
  - permitem aborto em cascata X evitam aborto em cascata
  - estritos X não-estritos

Escalonamento Recuperável

- Garante que, se *Tx* realizou *commit*, *Tx* não irá sofrer UNDO
  - o *recovery* espera sempre esse tipo de escalonamento!
    - garantia de **Durabilidade**!
- Um escalonamento *E* é recuperável se nenhuma *Tx* em *E* for concluída até que todas as transações que gravaram dados lidos por *Tx* tenham sido concluídas

escalonamento não-recuperável

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
	commit( )
abort( )	

escalonamento recuperável

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
commit( )	
	commit( )

Escalonamento sem Aborto em Cascata

- Um escalonamento recuperável pode gerar abortos de transações em cascata
  - consome muito tempo de *recovery*!
- Um escalonamento *E* é recuperável e evita aborto em cascata se uma *Tx* em *E* só puder ler dados que tenham sido atualizados por transações que já concluíram
  - Evita ***dirty-read***!

escalonamento recuperável **com aborto em cascata**

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
abort( )	. . .

escalonamento recuperável **sem aborto em cascata**

T1	T2
read(X)	
X = X – 20	
write(X)	
commit( )	
	read(X)
	X = X + 10
	write(X)
	. . .

Escalonamento Estrito

- Garante que, se *Tx* deve sofrer UNDO, basta gravar a *before image* dos dados atualizados por ela
  - *UNDO(Tx)* não interferirá em atualizações posteriores de outras transações
    - Evita ***lost-update***!
- Um escalonamento *E* é recuperável, evita aborto em cascata e é estrito se uma *Tx* em *E* só puder ler ou atualizar um dado X depois que todas as transações que atualizaram X tenham sido concluídas

escalonamento recuperável sem aborto em cascata e **não-estrito**

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(Y)
	X = Y + 10
	write(X)
	commit( )
abort( )	

escalonamento recuperável sem aborto em cascata e **estrito**

T1	T2
read(X)	
X = X – 20	
write(X)	
commit( )	
	read(Y)
	X = Y + 10
	write(X)
	commit( )

Teoria da *Serializabilidade*

- Garantia de escalonamentos não-seriais válidos
- Premissa
  - *"um escalonamento não-serial de um conjunto de transações deve produzir resultado equivalente a alguma execução serial destas transações"*

entrada:
X = 50
Y = 40

execução serial

saída:
X = 40
Y = 60

T1	T2
read(X)	
X = X – 20	
write(X)	
read(Y)	
Y = Y + 20	
write(Y)	
	read(X)
	X = X + 10
	write(X)

entrada:
X = 50
Y = 40

execução não-serial ***serializável***

saída:
X = 40
Y = 60

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
Y = Y + 20	
write(Y)	

Verificação de Serializabilidade

- Duas principais técnicas
  - equivalência de conflito
  - equivalência de visão
- Equivalência de Conflito
  - *"dado um escalonamento não-serial E' para um conjunto de Transações T, E' é serializável em conflito se E' for equivalente em conflito a algum escalonamento serial E para T, ou seja, a ordem de quaisquer 2 operações em conflito é a mesma em E' e E."*

Equivalência de Conflito - Exemplo

escalonamento serial *E*

T1	T2
read(X)	
X = X – 20	
write(X)	
read(Y)	
Y = Y + 20	
write(Y)	
	read(X)
	X = X + 10
	write(X)

escalonamento não-serial *E1*

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
Y = Y + 20	
write(Y)	

escalonamento não-serial *E2*

T1	T2
read(X)	
X = X – 20	
	read(X)
	X = X + 10
write(X)	
read(Y)	
	write(X)
Y = Y + 20	
write(Y)	

- *E1* equivale em conflito a *E*
- *E2* não equivale em conflito a nenhum escalonamento serial para T1 e T2 (com T1 ou com T2 executando primeiro)
- *E1* é serializável e *E2* não é serializável

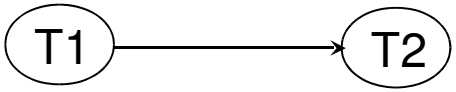
Verificação de Equivalência em Conflito

- Construção de um grafo direcionado de precedência
  - nodos são IDs de transações
  - arestas rotuladas são definidas entre 2 transações T1 e T2 se existirem operações em conflito entre elas
    - direção indica a ordem de precedência da operação
      - origem indica onde ocorre primeiro a operação
- Um grafo com ciclos indica um escalonamento não-serializável!

Grafo de Precedência

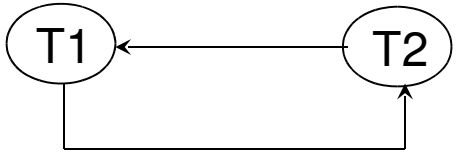
escalonamento serializável *E1*

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
Y = Y + 20	
write(Y)	



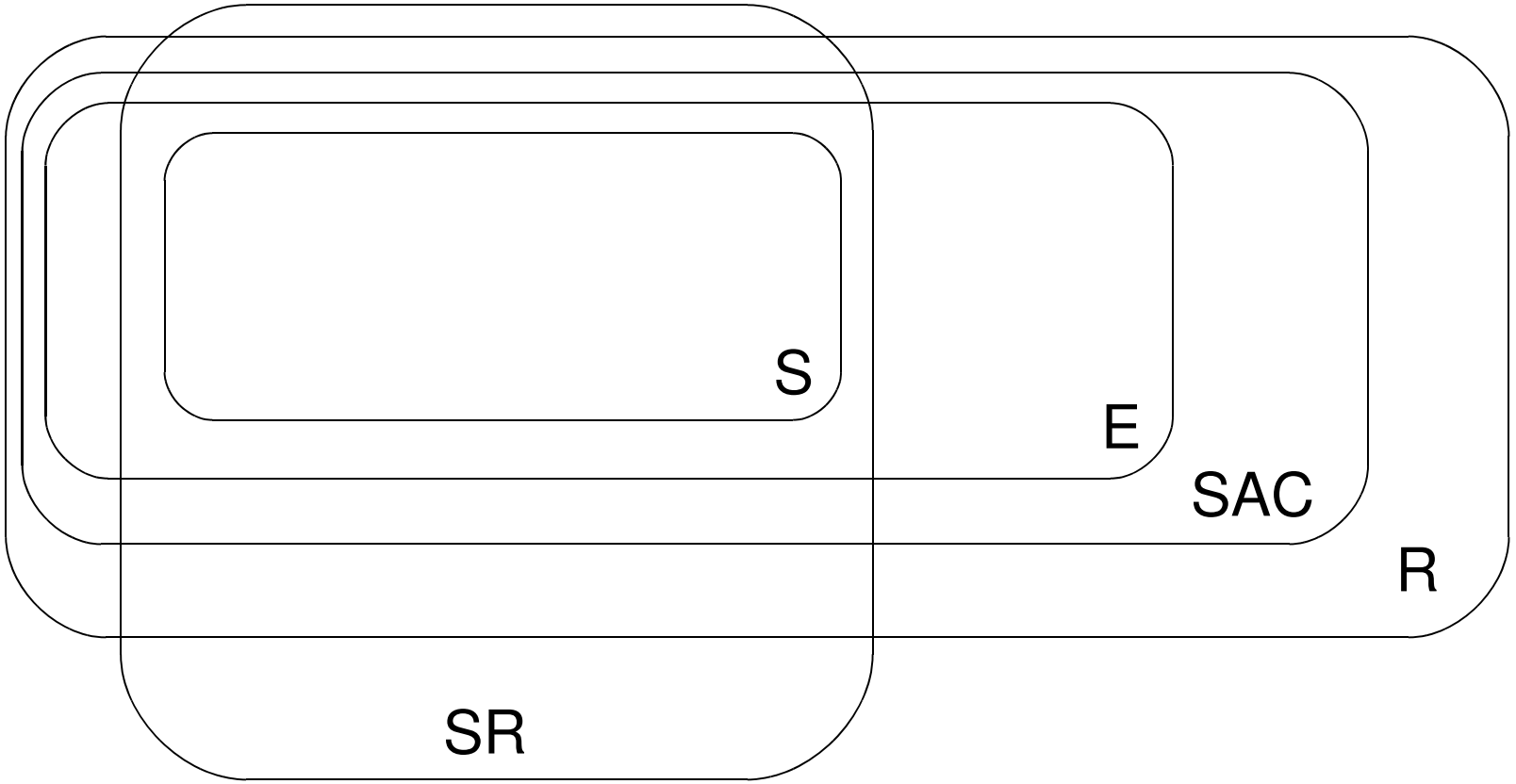
escalonamento não-serializável *E2*

T1	T2
read(X)	
X = X – 20	
	read(X)
	X = X + 10
write(X)	
read(Y)	
	write(X)
Y = Y + 20	
write(Y)	



Relação entre Escalonamentos

- SR = escalonamento serializável
- R = escalonamento recuperável
- SAC = escalonamento sem aborto em cascata
- E = escalonamento estrito
- S = escalonamento serial



História

- Representação seqüencial da execução entrelaçada de um conjunto de transações concorrentes
  - operações consideradas
    - r: *read*, w: *write*,

      c: *commit*, a: *abort*
- Exemplo

H_{E2} = r1(x) r2(x) w1(x) r1(Y) w2(x) w1(y) c1 c2

escalonamento não-serializável *E2*

T1	T2
read(X)	
X = X – 20	
	read(X)
	X = X + 10
write(X)	
read(Y)	
	write(X)
Y = Y + 20	
write(Y)	
commit( )	
	commit( )

Exercício 1

1. Dadas as transações abaixo, associe corretamente a história com o tipo de escalonamento (SR, R, SAC, E, S)

T1 = w(x) w(y) w(z) c1 T2 = r(u) w(x) r(y) w(y) c2

H_{E1} = w1(x) w1(y) r2(u) w2(x) r2(y) w2(y) c2 w1(z) c1 ( )

H_{E2} = w1(x) w1(y) w1(z) c1 r2(u) w2(x) r2(y) w2(y) c2 ( )

H_{E3} = w1(x) w1(y) r2(u) w2(x) w1(z) c1 r2(y) w2(y) c2 ( )

H_{E4} = w1(x) w1(y) r2(u) w1(z) c1 w2(x) r2(y) w2(y) c2 ( )

H_{E5} = w1(x) w1(y) r2(u) w2(x) r2(y) w2(y) w1(z) c1 c2 ( )

2. Dadas as transações ao lado, dê um exemplo, envolvendo todas elas, de uma história não-serial :

a) não-serializável

b) serializável e não-recuperável

c) sem aborto em cascata

T1

```
read(X)
X = X + 10
write(X)
Y = 20
write(Y)
```

T2

```
read(X)
Y = X + 10
write(Y)
```

T3

```
read(X)
X = X * 2
write(X)
```

Equivalência de Visão

- *"dado um escalonamento não-serial E' para um conjunto de Transações T, E' é serializável em visão se E' for equivalente em visão a algum escalonamento serial E para T, ou seja:*
  - *para toda operação read(X) de uma Tx em E', se X é lido após um write(X) de uma Ty em E' (ou originalmente lido do BD), então essa mesma seqüência deve ocorrer em E;*
  - *se uma operação write(X) de uma Tk for a última operação a atualizar X em E', então Tk também deve ser a última transação a atualizar X em E."*

Serializabilidade de Visão

- Premissas básicas
  1. enquanto cada *read(X)* de uma *Tx* ler o resultado de uma mesmo *write(X)* em *E'* e *E*, em ambos os escalonamentos, *Tx* tem a mesma visão do resultado
  2. se o último *write(X)* é feito pela mesma transação em E' e E, então o estado final do BD será o mesmo em ambos os escalonamentos
- Exemplo

$$H_{serial} = r1(X)\ w1(X)\ c1\ w2(X)\ c2\ w3(X)\ c3$$

$$H_{exemplo} = r1(X)\ w2(X)\ w1(X)\ w3(X)\ c1\ c2\ c3$$

  - $H_{exemplo}$ deve equivaler em visão à H_{serial}, pois a ordem dos *starts* de transações nele foi T1 < T2 < T3
  - $H_{exemplo}$ não é serializável em conflito, mas é serializável em visão (premissa 2 atendida!)

Serializabilidade em Conflito e de Visão

- A serializabilidade de visão é menos restritiva que a serializabilidade em conflito
  - um escalonamento E' serializável em conflito também é serializável em visão, porém o contrário nem sempre é verdadeiro
- A serializabilidade de visão é muito mais complexa de verificar que a serializabilidade em conflito
  - requer um rastreamento constante da ordem de execução de *read*s e *write*s para cada dado
    - verificar sempre se as premissas estão sendo atendidas para o escalonamento serial correspondente
    - difícil a sua utilização na prática...

Verificação de Serializabilidade

- Técnicas propostas (em conflito e de visão) são difíceis de serem testadas
  - partem do pressuposto que se está trabalhando sobre um conjunto fechado de transações para fins de verificação
- Na prática
  - conjunto de transações executando concorrentemente é muito dinâmico!
    - novas transações estão sendo constantemente submetidas ao SGBD para execução
    - custoso controlar um grafo de precedência (eq. conflito) ou seqüências válidas de *reads* e *writes* (eq. visão)
  - logo, a serializabilidade é garantida através de **técnicas** (ou protocolos) **de controle de concorrência** que não precisam testar os escalonamentos constantemente