CES-27 e CE-288

Programação Distribuída

Prof. Celso Massaki HIRATA

Sala 116 (prédio da Computação), tel: 3947-5758
e-mail: hirata@ita.br

Capítulo 5
Transações em Banco de Dados Distribuídos

Um banco de dados distribuídos é uma coleção de dados que estão distribuídos em diferentes computadores interconectados por uma rede. Cada computador tem processamento autônomo e coopera na execução de pelo menos uma aplicação global que requer o acesso aos dados em diferentes locais.

- Propriedades
- Revisão de Recuperação em Banco de Dados Centralizados
- Recuperação em Banco de Dados Distribuídos
- Controle de Concorrência Centralizado
- Controle de Concorrência Distribuído

Propriedades de Transações

Uma transação é uma aplicação de banco de dados, por exemplo:
Transferir $1000 de uma conta A para uma conta B

Propriedades:

Atomicidade:
Ou todas ou nenhuma das operações são realizadas.

Transação Cometida: Begin Trans. ===> Commit

Transação Abortada: Begin Trans. ===> Abort

Falha de Sistema: Begin Trans. ===> System Crash

Durabilidade (ou Persistência)
Os resultados de transações cometidas não são nunca perdidos como um resultado de falhas subseqüentes.

⇨ Recuperação

Serializabilidade
Se várias transações são executadas concorrentemente, o resultado deve ser o mesmo da execução serial em alguma ordem.

⇨ **Controle de Concorrência**

Duas transações
T1: depósito de R$ 1,00 em A (inicialmente R$ 49.999,00)
T2: pagamento de juros em A (1% se Valor >= R$ 50.000,00 e 0,5% caso contrário)

Operações de T1: $R_1(A)$, $W_1(A)$ T2: $R_2(A)$, $W_2(A)$

Se a sequencia for T1 e T2, A=50.500,00
Se a sequencia for T2 e T1, A=50.249,99 (50.248,995+1)
Ambas estão corretas.

Se as operações de T1 e T2 forem executadas de forma intercalada:
$R_1(A)$, $R_2(A)$, $W_1(A)$, $W_2(A)$ A=50.248,99! Inconsistente

Isolação (ou visibilidade)
Uma transação incompleta não pode revelar seus resultados para outras transações antes do seu cometimento. Evita abortos em cascata (efeito dominó).

T1: Credita V para A (A inicialm. 0) T2: Debita W de A

Begin T1		
…		Begin T2
A:=A+V		…
…	_____	If A>W then
	A=V se sem isolação	A:= A-W
…		else abort
…		…
Falha		Commit T2

Suponha que A inicialmente é 0 e V> W, o que acontece ?
Problema: se não existir isolação T2 comete mesmo que T1 falhe.
Não se pode desfazer T2.

Sumário das Propriedades:
Atomicidade Controle concorrente Isolação Durabilidade

Recuperação em Banco de Dados Distribuídos

Modelo de Falha:
1) Falha sem perda de informação. E.g.: Divisão por 0.
2) Falha com perda de memória volátil. E.g.: pane no sistema, queda de força.
3) Falha com perda de memória não volátil

A probabilidade de 3 pode ser reduzida por implementando memória estável. Implementar memória estável consiste em replicar informação em dois ou mais discos com modos de falha independente e fazer uma atualização cuidadosa (cada cópia é atualizada e verificada por vez).

Logs:
Contém informação para desfazer e refazer operações que são realizadas por uma transação. É exigido que *desfazer* e *refazer* sejam idempotentes:

desfazer(ação); desfazer(ação); desfazer(ação) = desfazer(ação)

"Idempotência" é necessária porque a própria recuperação pode falhar.

Entrada de log típica:
<Id. Transação, Id. Registro,
ação (inserir, deletar, modificar),
valor de registro velho (necessário para desfazer),
valor de registro novo (necessário para refazer),
entrada anterior para esta transação>

<u>Exemplo:</u>
Transferir $100 de A para B (inicialmente A=1000, B=800).
Entradas de log:
1. <T1, begin-trans>
2. <T1, A, modify, 1000, 900>
3. <T1, B, modify, 800, 900>
4. <T1, commit>

Protocolo Log write-ahead
1. Antes de realizar uma atualização de banco de dados, registre pelo menos uma entrada log de desfazer na memória estável.

2. Antes de cometer uma transação, todas as entradas de log devem ser gravadas na memória estável.

Nota: Se uma atualização de banco de dados fosse realizada antes da entrada do log e uma falha ocorresse entre estas ações então seria impossível de desfazer a atualização.

Teoricamente devem-se gravar todos os registros de log para fazer a recuperação do Banco de Dados. Na recuperação, devemos considerar todos os registros de log. Contudo, existem muitos registros de transações que foram completadas há um bom tempo e não necessitam de ser tratadas.

Para evitar a recuperação total, gravamos periodicamente registros de checkpoints no arquivo de log.

Checkpoints:

1. Gravar na memória estável todos os registros de log e atualizações que ainda estão na memória volátil.
2. Gravar na memória estável a entrada de checkpoint de log:
   = conjunto de transações atualmente ativas.
   (transações ativas que não foram cometidas ou abortadas)

Procedimento de Recuperação (perda de memória volátil) usando Checkpoints:

(1) Inicialização
    conjunto desfazer = {conjunto registrado no último registro de checkpoint}
    conjunto refazer = {}
(2) da entrada de checkpoint até o fim de log faça
    se entrada = begin-trans então
        adicione trans ao conjunto desfazer
    se entrada = commit então
        mova trans do conjunto desfazer para refazer
(3) Desfaça operações no conjunto desfazer.
    Refaça operações no conjunto refazer.

Exemplo:

Vamos supor que o registro de checkpoint é gravado de 10 em 10 minutos a partir das 24 horas. Vamos supor que o sistema cai as 11h09m e o arquivo de log tem o seguinte conteúdo:

11h00m	<T5, T8, T10>	3 transações ativas
11h01m	<T12, begin-trans>	T12 é ativada
11h02m	<T8, A, modify, 1000, 900>	
11h03m	<T10, commit>	T10 completa
11h04m	<T13, begin-trans>	T13 é ativada
11h05m	<T13, D, modify, 5000, 200>	
11h06m	<T13, commit>	T13 completa
11h07m	<T12, C, modify, 110, 145>	

Recuperação usando o registro de checkpoint das 11h00m

11h00m	<T5, T8, T10>	D={T5, T8, T10} R={}
11h01m	<T12, begin-trans>	D={T5, T8, T10,**T12**} R={}
11h02m	<T8, A, modify, 1000, 900>	
11h03m	<T10, commit>	D={T5, T8,T12} R={**T10**}
11h04m	<T13, begin-trans>	D={T5, T8,T12,**T13**} R={T10}
11h05m	<T13, D, modify, 5000, 200>	
11h06m	<T13, commit>	D={T5, T8,T12} R={T10,**T13**}
11h07m	<T12, C, modify, 110, 145>	

Perda de Memória Estável:

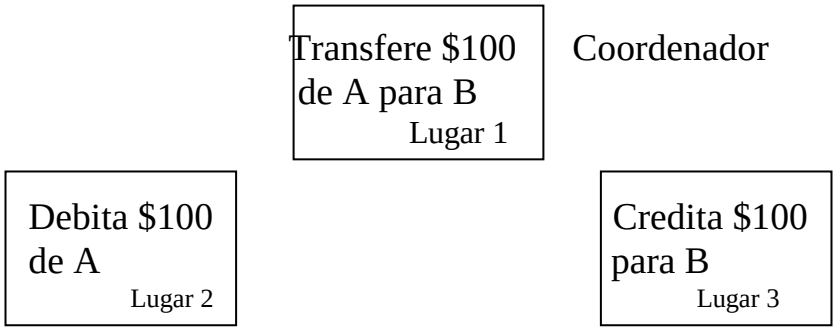
(1) Perda de informação de banco de dados
reset banco de dados para dump
refazer transações cometidas em log.

(2) Perda de informação de log
catastrófico.

Recuperação em Banco de Dados Distribuídos

(Lida com falhas em diferentes locais)

Exemplo de Transação Distribuída:



Cada agente proporciona serviços de recuperação centralizados locais (memória estável e log de transações). O coordenador de uma transação distribuída pode ficar co-hospedado com um dos agentes e usar o log deste agente ou usar outro log. O coordenador comunica com agentes por passagem de mensagem como se segue:

Início da Transação:
Coordenador:

write global-begin to log;
send begin-trans to agents;

Agentes:

receive begin-trans from coordinator;
write local-begin to log;

Efetivamente, agentes criam sub-transações locais da transação global. Todas as mensagens subseqüentes são rotuladas com o identificador da transação.

Mensagens de Aplicação:

No exemplo:
Coordenador:
send debit $100 from A to Agent 1
send credit $100 to B to Agent 2

Agent 1: (init A= 1000)
receive from coordinator
write <Tid, A, modify, 1000, 900> to log
update A

Agent 2: (init B= 800)
receive from coordinator
write <Tid, B, modify, 800, 900> to log
update B

Os agentes não precisam fazer mudanças permanentes no log ou no banco de dados até que receber um comando commit. Isso otimiza o aborto onde mudanças temporárias podem ser descartadas.

Aborto:
Coordenador:
write global-abort to log;
send abort to agents;

Agentes:
receive abort from coordinator;
write local-abort to log;
undo actions upto local-begin

Cometimento:
A implementação do cometimento distribuído é complicado pelo fato de que ou todas as transações locais devem cometer ou nenhuma. Isso deve ser válido mesmo na presença de falhas de host ou perda de mensagens.

Protocolo de Cometimento de 2 Fases

Proporciona **atomicidade** para as transações distribuídas e é resiliente a todas as falhas quando nenhuma informação de *log* é perdida.

Fase 1 — Objetivo: Alcançar uma decisão

Coordenador:

```
write prepare to log;
send prepare to agents; inicia timeout;
```

Agentes:

```
receive prepare from coordinator;
If willing to commit then
        write sub-transactions entries to log;
        write ready to log;
        send ready to coordinator;
else
        write local-abort to log;
        send abort to coordinator;
```

Fase 2

Coordenador:

```
receive ready or abort from all agents (or timeout);
if abort received (or timeout) then
        write global-abort to log;
        send abort to agents;
else { all agents reply ready }
        write global-commit to log;
        send commit to agents;
```

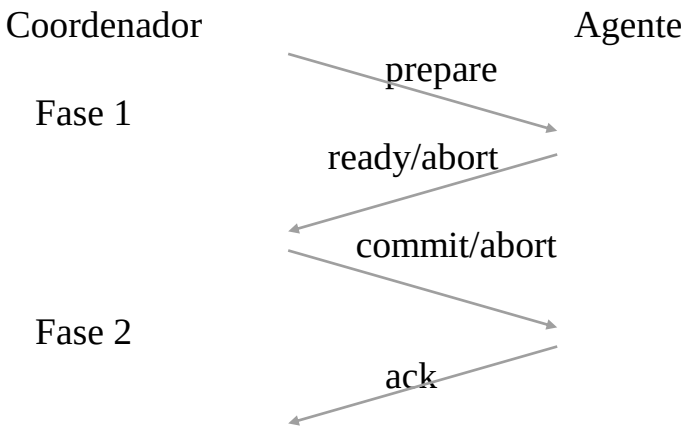
Agentes: — Objetivo: Implementar a decisão

```
receive abort or commit  from coordinator;
write to log;
send ack to coordinator;
```

Coordenador:

```
receive ack from all agents;
write complete to log;
```

Diagrama de Seqüência de Mensagem



Padrão de Comunicação: one to many

Exemplo de estágios de logs depois de um commit com sucesso:
Coordenador:

<global-begin>
<prepare, site2, site3> {records agents sites}
<global-commit>
<complete>

Agent 1:

<local-begin>
<modify, A, 1000, 900>
<ready, site1> {records coordinator site}
<local-commit>

Agent 2:

<local-begin>
<modify, B, 800, 900>
<ready, site1> {records coordinator site}
<local-commit>

Análise de Falha

Falha de Host:

a) Agente falha antes de gravar a entrada de log ready.
- Coordenador interrompe por timeout e aborta agentes ativos.
- Agente falho aborta no re-início (procedimento recuperação normal, nenhuma informação é requerida de outros locais).

b) Agente falha depois de gravar a entrada de log ready.
- Outros agentes completam (commit ou abort).
- No re-início, o agente falho deve perguntar ao coordenador ou outro agente sobre o resultado da transação.

Questão: O que acontece se os agentes falham depois de gravar ready no log mas antes de enviar a mensagem ready. (Caso *a* com falha)

c) Coordenador falha depois de gravar prepare, mas antes do global commit.
- Agentes que registraram *ready* devem esperar pela recuperação do coordenador.
- No re-início, o coordenador refaz o protocolo de commit (i.e. do prepare).
- Agentes *ready* deve reconhecer que este novo prepare é uma repetição do antigo.

d) Coordenador falha depois do global-commit (ou global-abort) mas antes de completar.
- no re-início o coordenador repete a fase 2 (i.e. enviando a decisão de commit ou abort).
- agentes que não receberam a decisão devem esperar pela recuperação do coordenador.

e) Coordenador falha depois de gravar complete.
- Nenhuma ação é requerida no re-início.

Mensagens Perdidas:

a) Mensagem de *ready* ou *abort* perdida do agente para o coordenador.
- coordenador interrompe por timeout e a transação é abortada.

b) Mensagem de prepare perdida do coordenador para agentes.
- coordenador interromperá por timeout e abortará a transação.

c) Mensagem de *commit* ou *abort* do coordenador para agente perdida.
- Apesar de não estar incluída no protocolo acima, a maneira mais fácil de lidar com isso é o agente interromper por timeout no comando e pedir para o coordenador re-enviar a decisão de commit ou abort.

d) Mensagem de ack do agente para coordenador perdida.
- Similar ao caso anterior, nesse caso, porém o coordenador deve interromper por timeout no ack e deve pedir ao agente, re-enviar a mensagens ack.

Partições de Rede:

Suponha que a partição cause que dois grupos sejam formados:

A = coordenador + alguns agentes
B = resto dos agentes

Coordenador vê falhas múltiplas de agentes em B.
(casos ***a*** e ***b*** de falha de host)

Agentes em B vêem falha do coordenador.
(casos ***c*** e ***d*** de falha de host)

Notas:

Antes que um agente tenha gravado uma entrada de log *ready*, ele pode de forma autônoma abortar sua sub-transação local.

Depois do *ready*, o agente pode ficar bloqueado se o coordenador falhar. Ele deve segurar todos os recursos da sub-transação até que o coordenador re-inicialize (e.g. locks) uma vez que ele não sabe se a transação será cometida ou abortada. Isso reduz a disponibilidade do sistema.

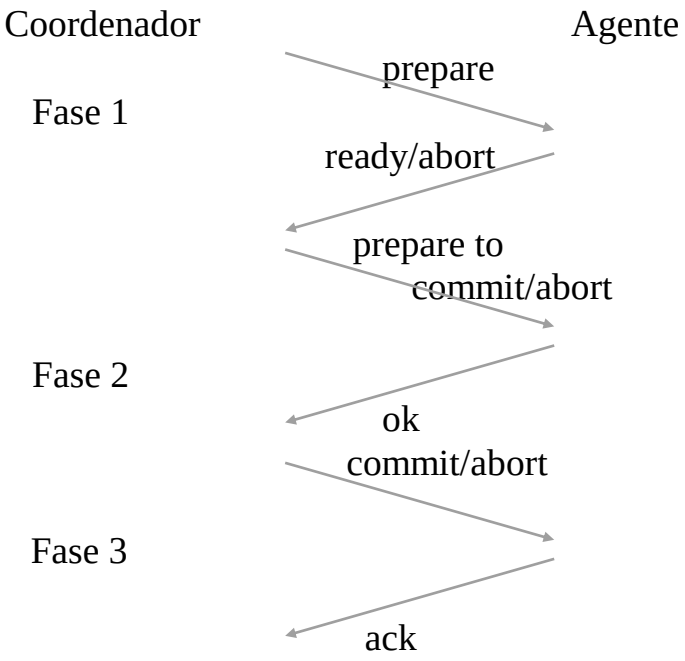
O término é **possível** apenas se um dos participantes tenha recebido o comando (abort/commit) ou nenhum dos participantes tenha recebido o comando e apenas o coordenador tenha falhado.

O término é **impossível** quando nenhum participante operacional tenha recebido o comando e o coordenador e um dos participantes tenha falhado.

Protocolo de Cometimento Não Bloqueante

(3 - phase commit)

No protocolo de 2 fases, se o coordenador falhar depois de gravar prepare, mas antes do global commit, os agentes que registraram *ready* devem esperar pela recuperação do coordenador (protocolo bloqueante). Mas se o coordenador não voltar?



Depois do *ready* os agentes não vão diretamente para o estado cometido mas para o estado *prepare-to-commit* (prepare-para-cometer).

Elimina o estado bloqueado dos agentes uma vez que:

Se nenhum agente tenha recebido a mensagem prepare-to-commit (caso do estado bloqueado para o protocolo de duas fases) os agentes operacionais podem abortar a transação porque os agentes falhos não cometeram ainda. Agentes falhos abortam no re-início.

Um **protocolo de término** é requerido para eleger um novo coordenador e completar a transação se o coordenador falhar. O protocolo deve eleger um novo coordenador e pode ser centralizado.

Protocolo de Término:

Baseado nas duas propriedades do protocolo de cometimento não bloqueante:

1) Se pelo menos um agente não entrou no estado prepare-to-commit então a transação pode ser abortada (i.e. nenhum agente recebeu a mensagem de commit)

2) Se pelo menos um agente operacional tenha alcançado o estado prepare-to-commit então a transação pode ser cometida (i.e. todos os agentes responderam previamente ready).

Perceba que os casos 1 e 2 não são mutuamente exclusivos. O protocolo que sempre comete a transação quando ambos os casos são possíveis é chamado **protocolo progressivo**.

O protocolo de término deve eleger um novo coordenador. Essa eleição pode ser centralizada ou descentralizada. Em geral, têm-se protocolos de término centralizado e não progressivo.

O protocolo de cometimento não bloqueante é catastrófico para o caso de partição de rede. Uma vez que os dois grupos podem eleger um novo coordenador e chegar a diferentes decisões por contabilizando a transação nas duas partições.

Em geral, pode ser mostrado que protocolos não bloqueantes resilientes a múltipla partição da rede não existe.

Controle de Concorrência

O protocolo de Cometimento de 2-Fases proporciona **atomicidade** com respeito a falhas e durabilidade, contudo, um controle de concorrência é requerido para proporcionar **serializabilidade** e **isolação**.

Controle de Concorrência em Banco de Dados Centralizados

Notação:
R(x,a) - Transação x lê a
W(x,a) - Transação x grava a

Exemplo:
Transação T(x) é a:=b+c;
T(x): R(x,b) < R(x,c) < W(x,a) { < é a relação precede }

Escalonamento:
Uma seqüência de operações realizada por transações. E.g.:
T(x) é a:=b+c; T(y) é d:=b+c;
S1: R(x,b)<R(y,b)<R(x,c)<R(y,c)<W(x,a)<W(y,d)

S1 é um escalonamento concorrente uma vez que as operações de T(x) e T(y) são intercaladas. T(x) e T(y) são concorrentemente ativas.

Escalonamento Serial:
Um escalonamento é serial se nenhuma transação é executada de forma concorrente. E.g.:
S2: R(x,b)<R(x,c)<W(x,a)<R(y,b)<R(y,c)<W(y,d)
pode ser escrita como
Serial(S2): T(x)<T(y)

Um escalonamento serial por definição é correto.

Um escalonamento (concorrente) é correto se ele é **serializável**.
i.e. ele é computacionalmente equivalente a um escalonamento serial.

Condições suficientes para dois escalonamentos serem computacionalmente equivalentes:

(1) Cada operação de leitura lê valor de item de dados que é produzido pela mesma operação de gravação nos dois escalonamentos.

(2) A operação de gravação final é a mesma nos dois escalonamentos.

Para verificar se um escalonamento concorrente é correto, basta verificar se ele é computacionalmente equivalente a um escalonamento serial.

Questão: São os escalonamentos S1 e S2 equivalentes?

Seja b=2 e c=3, então:
S1: R(x,b=2)<R(y,b=2)<R(x,c=3)<R(y,c=3)<W(x,a=5)<W(y,d=5)
S2: R(x,b=2)<R(x,c=3)<W(x,a=5)<R(y,b=2)<R(y,c=3)<W(y,d=5)

S1 e S2 são equivalentes.

Conflitos:
Duas operações estão em conflito se elas operam no mesmo item de dados, uma delas é a operação de gravação (a outra pode ser tanto de leitura quanto de gravação) e elas são emitidas por **transações separadas.** E.g.:

conflito read-write: R(x,a),W(y,a) no item a
ou W(y,a), R(x,a) no item a
conflito write-write: W(x,a),W(y,a) no item a

T(x) é d:=e+**a**; T(y) é **a**:=b+c;
S1: R(x,e)<R(y,b)<R(**x,a**)<R(y,c)<W(x,d)<W(**y,a**)

S2: R(y,b)<R(y,c)<W(**y,a**)<R(x,e)<R(**x,a**)<W(x,d)
(resultado diferente de S1 se todos os valores são iguais a 1)

T(x) é a:=b+c; T(y) é a:=c+d;
S1: R(x,b)<R(y,c)<R(x,c)<R(y,d)<W(**x,a**)<W(**y,a**)

S2: R(x,b)<R(y,c)<R(x,c)<R(y,d)<W(**y,a**)<W(**x,a**)
(resultado diferente de S1 se b=c=1 e d=2)

Em termos de conflito, a condição para equivalência de escalonamento é:
Dois escalonamentos S1 e S2 são equivalentes se para cada par de operações conflitantes O(x) e O(y) (lembrar que uma é de gravação)
se O(x) < O(y) em S1 então O(x) < O(y) em S2

Locks:

Lock (trava) é um mecanismo de controle de concorrência para forçar serializabilidade.

Modos de lock: modo compartilhado, modo exclusivo

Uma transação bem formada deve adquirir um lock em modo **compartilhado antes de ler** um item de dados e um **lock em modo exclusivo antes de gravar** um item de dados.

Se uma transação obteve um **lock em modo exclusivo** num item de dados, a outra transação não consegue obter **lock (exclusivo ou compartilhado**) no item de dados.

Se uma transação obteve um lock em **modo em modo compartilhado** num item de dados, a outra transação não consegue obter **lock em modo exclusivo** no item de dados. (mas consegue obter lock em modo compartilhado!)

Conflitos:

Locks resolvem conflitos (pedidos para segurar um item com modos incompatíveis) por causando as transações esperarem.

Esquema de Lock de Duas Fases:

A transação, além de bem formada, não deve pedir novos locks depois dela ter liberado um lock. Isso é para garantir serializabilidade, i.e.:

Fase de Crescimento: transação obtém locks.
Fase de Encolhimento: transação libera locks.

Nota: Para garantir **isolação,** uma transação não deve liberar nenhum lock até que ela tenha cometida.

Prova de Correteza de Lock de Duas Fases.

Quando um escalonamento não é serializável em termos de lock?

T(x) é a:=b+c; T(y) é b:=a+d;
S1: R(**x,b**)<R(**y,a**)<R(x,c)<R(y,d)<W(**x,a**)<W(**y,b**)

Em S1 quais locks serão concedidos?
Em R(x,b), T(x) pega o lock em **b** no modo compartilhado.
Em R(y,a), T(y) pega o lock em **a** no modo compartilhado.
Em R(x,c), T(x) pega o lock em **c** no modo compartilhado.
Em R(y,d), T(y) pega o lock em **d** no modo compartilhado.
Em W(x,a), T(x) tenta pegar o lock em **a** no modo exclusivo, mas não consegue.
Em W(y,b), T(y) tenta pegar o lock em **b** no modo exclusivo, mas não consegue.

T(y) aguarda T(x) em **b** e T(x) aguarda T(y) em **a! (ciclo de espera)**

De uma forma geral, para um escalonamento de n transações não ser serializável, deve existir um conjunto de n pares conflitantes num ciclo de espera tais que:
O(1,a) < O(2,a)
O(2,b) < O(3,b)
...
O(n-1,i) < O(n,i)
O(n,j) < O(1,j)

Suponha que cada transação tenha adquirido o lock no "lado esquerdo". Cada transação agora esperará indefinidamente para adquirir o lock no "lado direito" uma vez que o esquema de Duas Fases, locks não são liberados durante a fase de crescimento. Consequentemente, o sistema entra em deadlock e o escalonamento não-serializável não pode ocorrer.

Nota: Deadlocks podem ocorrer e consequentemente a transação deve incorporar algum método de detecção e quebra de deadlocks, e.g. timeout e aborto, uma vez que aborto libera todos os locks presos pela transação.

Questão:

Quais dos seguintes escalonamentos são corretos i.e. serializáveis, e para o escalonamento que é serializável, qual é o escalonamento serial equivalente?

T(x) é a:= b+c; T(y) é b:= a+c;

S3: R(x,b) < R(y,a) < R(x,c) < R(y,c) < W(y,b) < W(x,a)

S4: R(y,c) < R(x,c) < R(y,a) < W(y,b) < R(x,b) < W(x,a)

O escalonamento serial equivalente pode ser concebido como dando a ordem total de transações.

Controle de Concorrência em Banco de Dados Distribuídos

Cada transação realiza operações em vários locais. Uma execução de n transações distribuídas T(1)..T(n) em m locais é modelada por um conjunto de escalonamentos S1..Sm

Em geral, cada local tem seu escalonamento e gerencia itens de dados específicos.

Seja T(1) composta das sub-transações:
a:=a+1 no local 1
b:=b+1 no local 2
e T(2) composta das sub-transações:
a:= a/2 no local 1
b:=b/2 no local 2

O local 1 gerencia **a** enquanto o local 2 gerencia **b**.

Verifique os valores de a e b após T(1)<T(2) e T(2)<T(1). Suponha que a=b=0

T(1)<T(2) => a=b=1/2

T(2)<T(1) => a=b=1

Suponha a seguinte execução de T(1) e T(2) em dois locais

S1: R(1,a) < W(1,a) < R(2,a) < W(2,a)

S2: R(2,b) < W(2,b) < R(1,b) < W(1,b)

Cada escalonamento **local** é serializável, i.e.

S1: T(1) < T(2)

S2: T(2) < T(1)

Se ocorresse a execução, a=1/2 e b=1. **Resultado incorreto**!
Não existe um ordenação total global das transações.

Uma execução distribuída de transações para ser correta, ela precisa ser serializável.

Condição de Correteza para Transações Distribuídas

Seja E uma execução de transações T(1)..T(n) modelada por escalonamentos S1..Sm. E é correto (serializável) se existe uma ordenação total tal que para cada par de operações conflitantes O(i) e O(j) de T(i) e T(j), O(i) < O(j) em qualquer escalonamento S1..Sm se e somente se T(i) < T(j) na ordenação total.

Sem nenhuma surpresa, o esquema de Lock de Duas Fases é um mecanismo de controle de concorrência correto para transações distribuídas. A prova é como antes.

No esquema de Lock de Duas Fases, as transações distribuídas devem obter todos os locks antes de fazerem suas alterações. Se houver uma execução (distribuída) de transações distribuídas que não seja serializável, pelo esquema de Lock de Duas Fases, vai ocorrer um deadlock distribuído! Para a seguinte execução distribuída (S1 e S2) de T(1) e T(2) nos dois locais.

S1: R(1,a) < W(1,a) < R(2,a) < W(2,a) T(2) espera T(1) em a
S2: R(2,b) < W(2,b) < R(1,b) < W(1,b) T(1) espera T(2) em b
As duas esperas caracterizam o deadlock distribuído!

Questão:

Quais das seguintes execuções (distribuídas) são serializáveis?

Execução 1:
S1: R(1,a) < R(2,a) < W(2,b) < W(1,a)
S2: R(1,c) < R(2,d) < W(2,c) < W(1,c)

Execução 2:
S1: R(1,a) < R(2,a) < W(2,b) < W(1,b)
S2: W(1,d)

Execução 3:
S1: R(1,a) < R(2,a) < W(1,a) < W(2,b)
S2: R(1,d) < R(2,d) < W(2,d) < W(1,c)

Execução 4:
S1: R(1,b) < R(2,a) < W(2,a)
S2: W(2,d) < R(1,c) < R(2,c) < W(1,c)

Comentários sobre Serializabilidade

Dado T(1): transferir $10 de a para b (a, b inicialmente 100) e
T(2): transferir $20 de b para a
a execução não serializável:
S1: R(1,a) < W(1,a) < R(2,a) < W(2,a)
S2: R(2,b) < W(2,b) < R(1,b) < W(1,b)
produz o resultado correto (i.e. a= 110, b= 90)

Serializabilidade é o critério mais fraco para preservar consistência se nenhuma informação semântica sobre a transação está disponível.

Poderia-se definir um nível mais alto de operações diferentes de leitura e gravação para capturar mais conhecimento semântico, por exemplo:
Incremente I(x,d): x:= x+d;
Decremente D(x,d) x:=x-d;

Estas operações não conflitam uma vez que:
I(x,u)<D(x,w) é equivalente a D(x,w)<I(x,u)
o resultado é o mesmo: x-w+u

Podemos mostrar que nossa transação como:
S1: D(1, a, 10) < I(2, a, 20)
S2: D(2, b, 20) < I(1, b, 10)

Essa abordagem tem sido usada em alguns sistemas operacionais para ganhar mais concorrência.

Quais são os problemas dessa abordagem?

Controle de Concorrência Baseado em Rótulos de Tempo

O Esquema de Lock de Duas Fases tem a desvantagem de segurar os registros através de lock até a conclusão da transação. Se a transação for de longa duração, a demora pode resultar em um grande número de locks, levando a uma maior chance de deadlock.

O controle de concorrência **baseado em rótulos de tempo** não usa locks.

Para proporcionar uma ordenação total nas transações, um único rótulo de tempo é atribuído para cada transação. O algoritmo de relógios lógicos de Lamport é utilizado para gerar rótulos de tempo.

Mecanismo de Rótulação de Tempo Básico

1) Cada transação é atribuída um único rótulo de tempo em seu local de origem.

2) Cada operação de read e write tem o rótulo de tempo de sua transação TS.

3) Cada item de dados (x) contém a seguinte informação:
WTM(x) - o maior rótulo de tempo de uma operação write em x.
RTM(x) - o maior rótulo de tempo de uma operação read em x.

4) Para operações de read com rótulo de tempo TS:
Se TS < WTM(x) então
rejeite a operação read e reinicie a transação
senão
execute read e RTM(x):= max(RTM(x), TS);

5) Para operações write com rótulo de tempo TS:
Se (TS < RTM(x)) ou (TS < WTM(x)) então
rejeite a operação write e reinicie a transação
senão
execute write e WTM(x):= TS;

Os passos 4 e 5 garantem que operações conflitantes são executadas em ordem de rótulo de tempo em todos os locais e portanto a condição de correteza para serialização é satisfeita.

Notas:

Uma transação que é re-iniciada adquirirá um rótulo de tempo maior e assim eventualmente prosseguirá com sucesso.

Transações nunca são bloqueadas (elas são reiniciadas) e consequentemente, deadlock não pode ocorrer.

A desvantagem deste esquema é o custo de reinicialização.

Controle de Concorrência por Rótulo de Tempo e Esquema de Cometimento de Duas Fases

O esquema de cometimento de Duas Fases requer que existe um intervalo de tempo durante o qual os agentes podem cometer ou abortar transações. Com a condição de que todos os locks exclusivos são seguros até que a transação cometa garante isolação (não visibilidade de resultados intermediários).

Com rótulos de tempo, prewrites são usados em vez de locks exclusivos. Em vez de realizar uma operação de write, transações realizam operações de prewrite para bufferizar e não para atualizar. Apenas quando as transações cometem, as correspondentes atualizações são aplicadas ao banco de dados.

O mecanismo de rótulo de tempo acima deve ser modificado levemente para levar em conta os prewrites pendentes.