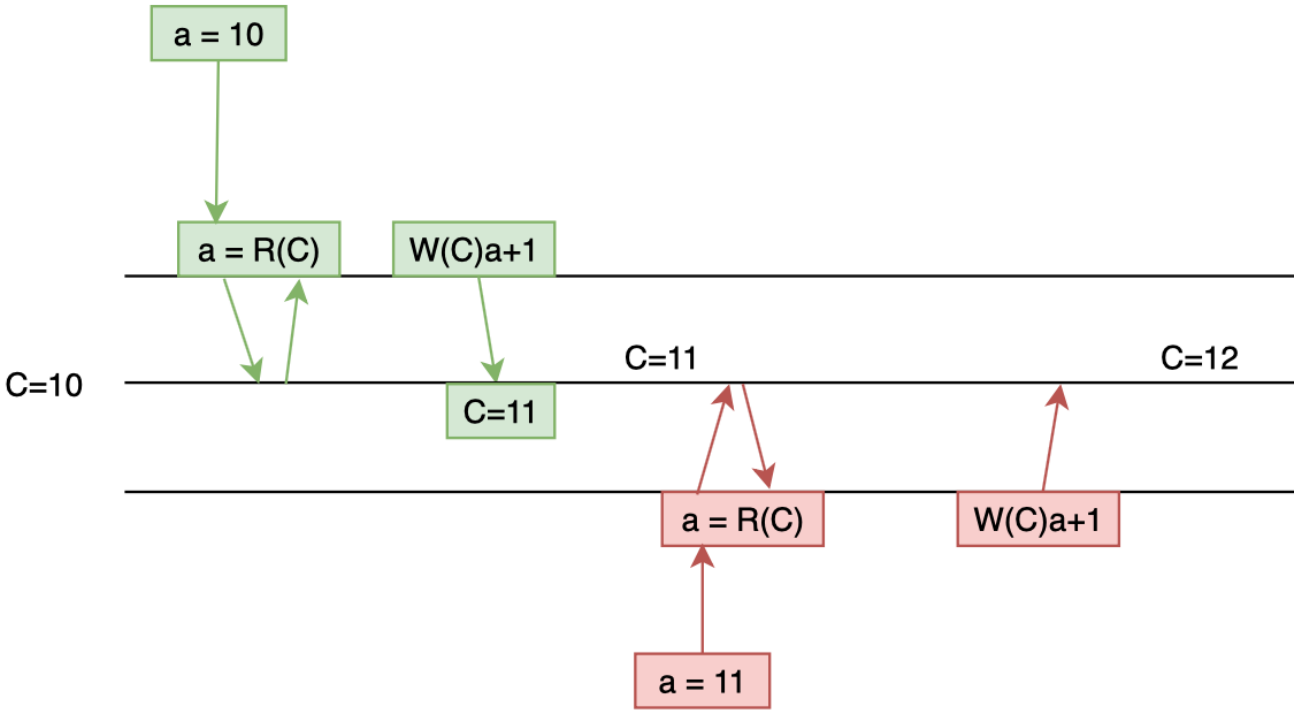
GBC074 – Sistemas Distribuídos

Bancos de dados distribuídos

Introdução

- Banco de dados e transações:

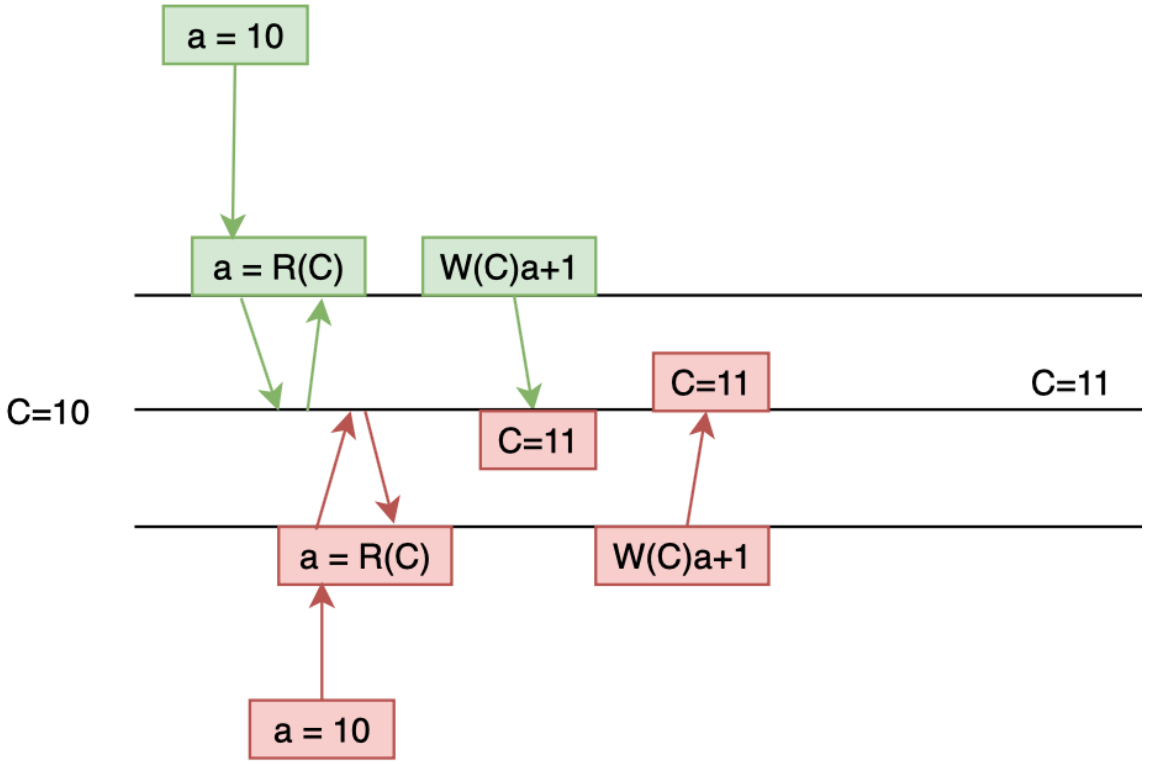
$$a = R(C); W(C)a + 1$$



Introdução

- Banco de dados e transações:

$$a = R(C); W(C)a + 1$$



Introdução

- Bancos de dados tradicionais
  - ACID: **Atomicidade, Consistência, Isolamento e Durabilidade**.
- **Atomicidade**
  - Tratamento das operações como um conjunto indivisível
  - Ou todas as operações no conjunto são executadas ou nenhuma é
- **Consistência**
  - Transições devem **respeitar restrições nos seus dados**, Exemplo: os tipos de cada entrada no banco e integridade referencial.
  - Para não confundir com a consistência estudada anteriormente, podemos renomear esta propriedade para **corretude**

Introdução

- Bancos de dados tradicionais
  - ACID: **Atomicidade, Consistência, Isolamento e Durabilidade**.
- **Isolamento**
  - Como e quando os efeitos de uma transação passam a ser visíveis para outras transações
  - Corresponde à consistência estudada anteriormente:
    - Exemplo: **consistência eventual** ou ainda **seriabilidade estrita**
- **Durabilidade**
  - Garantia de que os resultados de uma transação são permanentemente gravados no sistema, a despeito de falhas

Dirty reads

- Considere as duas transações a seguir:

T1(a,b)
$sB = R(b)$
$W(b)sB * 1.1$
$sA = R(a)$
$W(a)sA - (sB * 0.1)$

T2([a,b])
$sA = R(a)$
$sB = R(b)$
$sT = sA + sB$

Dirty reads

- Se *a* = 50 e *b* = 100, qual o valor final desta execução?

T1(a,b)	T2([a,b])
$sB = R(b)$	
	$sA = R(a)$
$W(b)sB * 1.1$	
	$sB = R(b)$
$sA = R(a)$	
$W(a)sA - (sB * 0.1)$	
	$sT = sA + sB$

Dirty reads

- Se a = 50 e b = 100, qual o valor final desta execução?
- Dados sendo modificados "vazaram" de T1 para T2
- Nível de isolamento = nenhum

T1(a,b)	T2([a,b])
$sB = R(b)$ = 100	
	$sA = R(a) = 50$
$W(b) sB * 1.1$ = 110	
	$sB = R(b) = 110$
$sA = R(a) = 50$	
$W(a) sA - (sB * 0.1) = 40$	
	$sT = sA + sB = 160$

Lost update

- Considere a execução em paralelo da mesma transação T1
- Se *a* = 50 e *b* = 100, qual o valor final desta execução?

`T1(a,b)`	`T1(a,b)`
$sB = R(b)$	
	$sB = R(b)$
	$W(b) sB * 1.1$
$W(b) sB * 1.1$	
	$sA = R(a)$
	$W(a) sA - (sB * 0.1)$
$sA = R(a)$	
$W(a) sA - sB * 0.1$	

Lost update

- Considere a execução em paralelo da mesma transação T1
- Se *a* = 50 e *b* = 100, qual o valor final desta execução?
- Atualização de *b* foi perdida

T1(a,b)	T1(a,b)
$sB = R(b) = 100$	
	$sB = R(b) = 100$
	$W(b) sB * 1.1 = 110$
$W(b) sB * 1.1$ = 110	
	$sA = R(a) = 50$
	$W(a) sA - (sB * 0.1) = 40$
$sA = R(a) = 40$	
$W(a) sA - sB * 0.1 = 30$	

Execução serial

- Funciona, mas perdemos **concorrência**

- O que queremos na prática:
  - Execução de transações semelhante à serial
  - **Não** queremos uma execução serial
  - Queremos uma execução **equivalente** a uma execução serial

Equivalência serial

- Duas execuções de transações são **equivalentes** se:
  - são execuções das mesmas transações (mesmas operações)
  - Quaisquer duas operações conflitantes são executadas na mesma ordem nas duas execuções
- Duas operações são **conflitantes** se:
  - Pertencem a transações diferentes,
  - Operam no mesmo dado, e
  - Pelo menos uma delas é escrita.
- Uma execução tem **equivalência serial** se:
  - É equivalente a alguma execução serial das transações
  - Para obter tanto desempenho advindo da concorrência quanto corretude advinda da serialização, escalone as operações de forma a garantir equivalência serial

Equivalência serial

- Como obter equivalência serial?

- Precisamos garantir **por construção** a equivalência serial
- Considere seguinte restrição
  - A execução de duas transações tem Equivalência Serial se todos os pares de operações conflitantes entre as transações são executados na mesma ordem.
  - Uma execução qualquer tem equivalência serial se todos os pares de transações tem equivalência serial.

Equivalência serial

- Revisitemos o exemplo do *lost update*. Quais operações conflitam nesta execução?

Operação	T1(a,b)	T1(c,b)
1	$sB = R(b)$	
2		$sB = R(b)$
3		$W(b) sB * 1.1$
4	$W(b) sB * 1.1$	
		$sC = R(c)$
		$W(c) sC - sB * 0.1$
	$sA = R(a)$	
	$W(a) sA - sB * 0.1$	

Equivalência serial

- Revisitemos o exemplo do *lost update*. Quais operações conflitam nesta execução?

Operação	T1(a,b)	T1(c,b)
2		$sB = R(b)$
3		$W(b)sB * 1.1$
1	$sB = R(b)$	
4	$W(b)sB * 1.1$	
		$sC = R(c)$
		$W(c)sC - sB * 0.1$
	$sA = R(a)$	
	$W(a)sA - sB * 0.1$	

Abort em cascata

- E se no exemplo anterior a transação da direita abortasse?
  - Teríamos um *dirty read*
  - Para que este dirty read não leve a inconsistências, a transação da esquerda deve também abortar

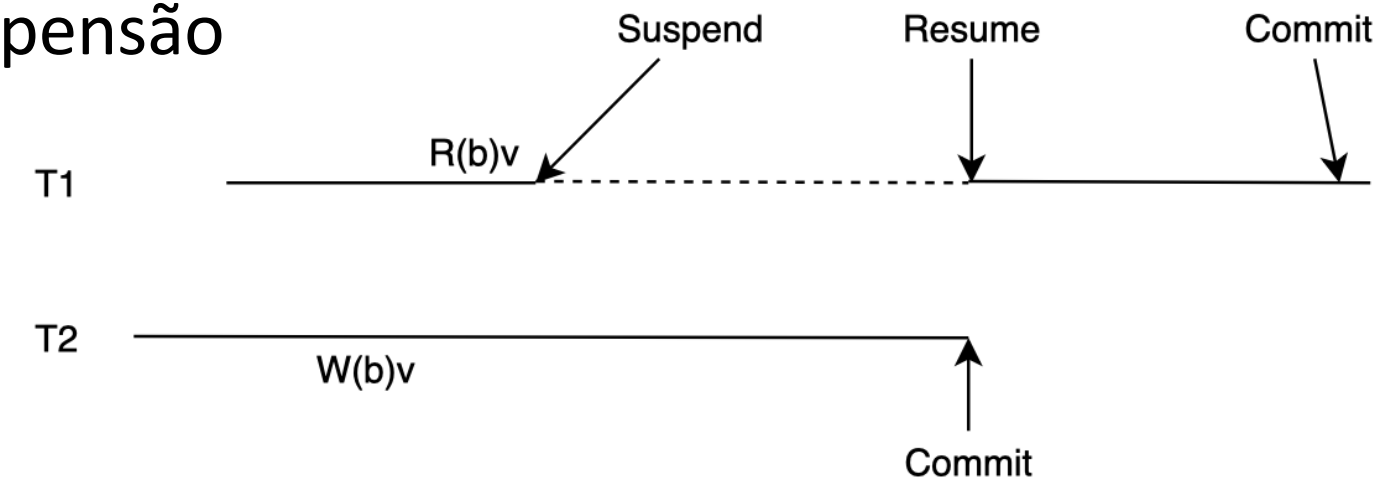
Operação	T1(a,b)	T1(c,b)
2		$sB = R(b)$
3		$W(b)sB * 1.1$
1	$sB = R(b)$	
4	$W(b)sB * 1.1$	
		$sC = R(c)$
		$W(c)sC - sB * 0.1$
	$sA = R(a)$	
	$W(a)sA - sB * 0.1$	

Abort em cascata

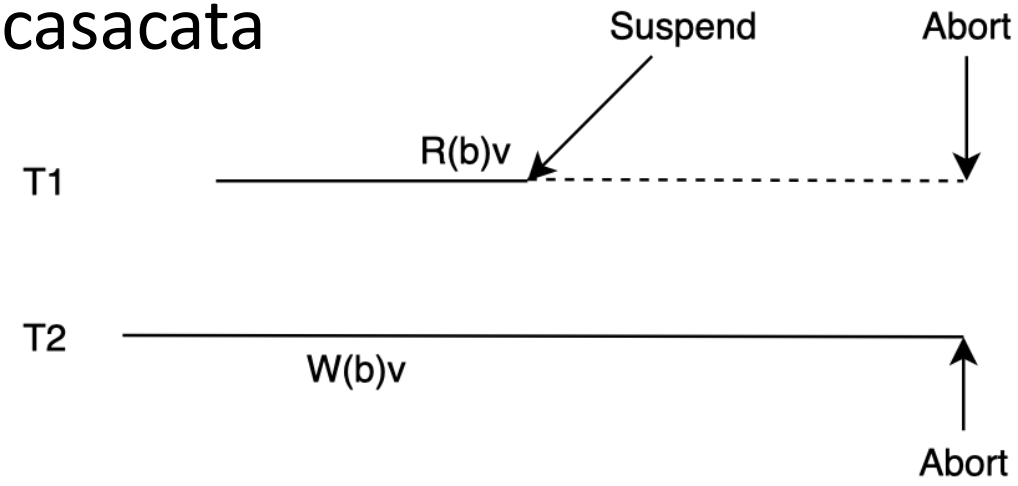
- Implementada da seguinte forma:
- Se uma transação lê um dado atualizado por uma transação não "*comitada*", suspenda a transação executando a leitura
- Se transação que atualizou o dado foi abortada, todas as suspensas que leram dela devem ser abortadas
- repita passo anterior

Abort em cascata

Suspensão

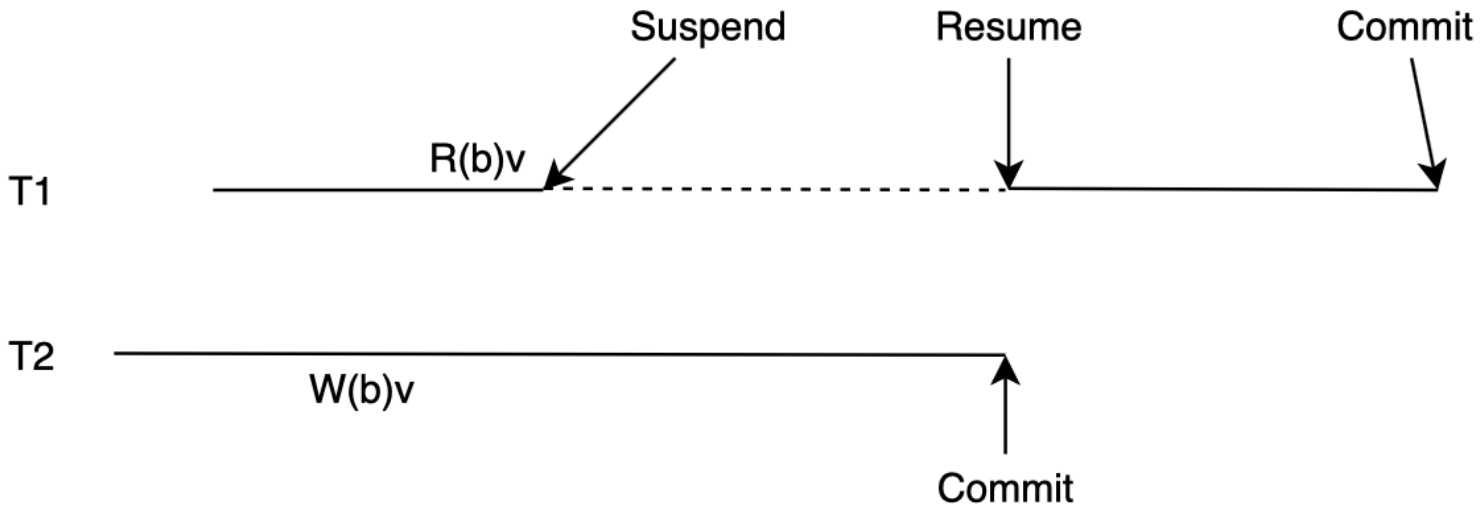


Abort em casacata



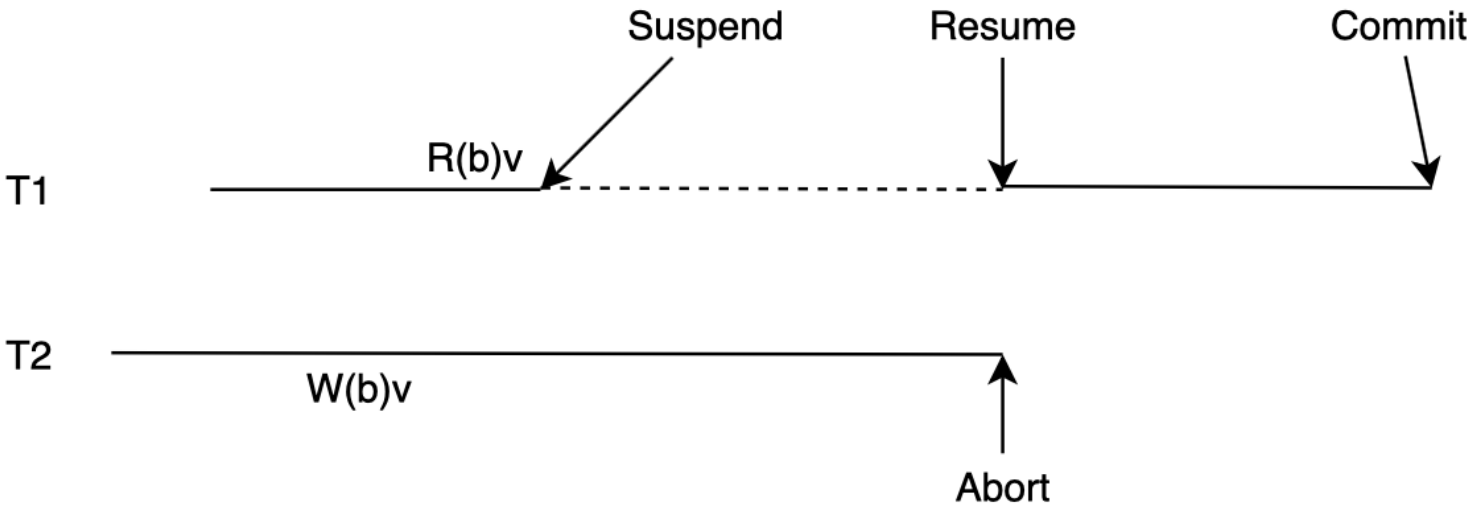
Abort em cascata

- E se evitarmos *dirty reads* em vez de tratarmos?
- Quando um transação T1 tenta ler um dado "sujo" escrito por T2, suspenda a execução da transação T1, **antes** da leitura acontecer
- Quando transação T2 for terminada, continue a execução de T1



Abort em cascata

- E se evitarmos *dirty reads* em vez de tratarmos?
- Quando um transação T1 tenta ler um dado "sujo" escrito por T2, suspenda a execução da transação T1, **antes** da leitura acontecer
- Quando transação T2 for terminada, continue a execução de T1



Abort em cascata

- E se evitarmos *dirty reads* em vez de tratarmos?
- Neste caso, temos a **execução estrita**

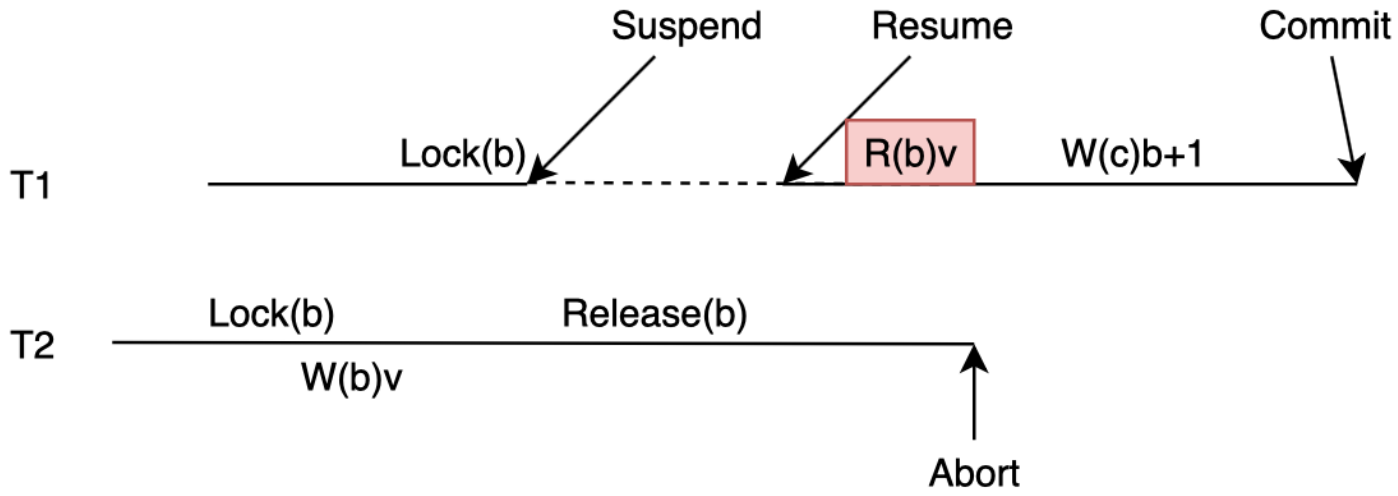
- **Como implementar execuções estritas eficientes**?
- A resposta está no controle de concorrência das transações

Controle de concorrência

- 3 abordagens:
- ***Locking***
  - Abordagem pessimista
  - Paga um alto preço de sincronização mesmo quando as transações não interferem umas nas outras
- multi-versão
  - Abordagem otimista
  - Tem algo custo quando há muitos conflitos entre as transações
- *Timestamp*
  - Abordagem mais complexa de se implementar

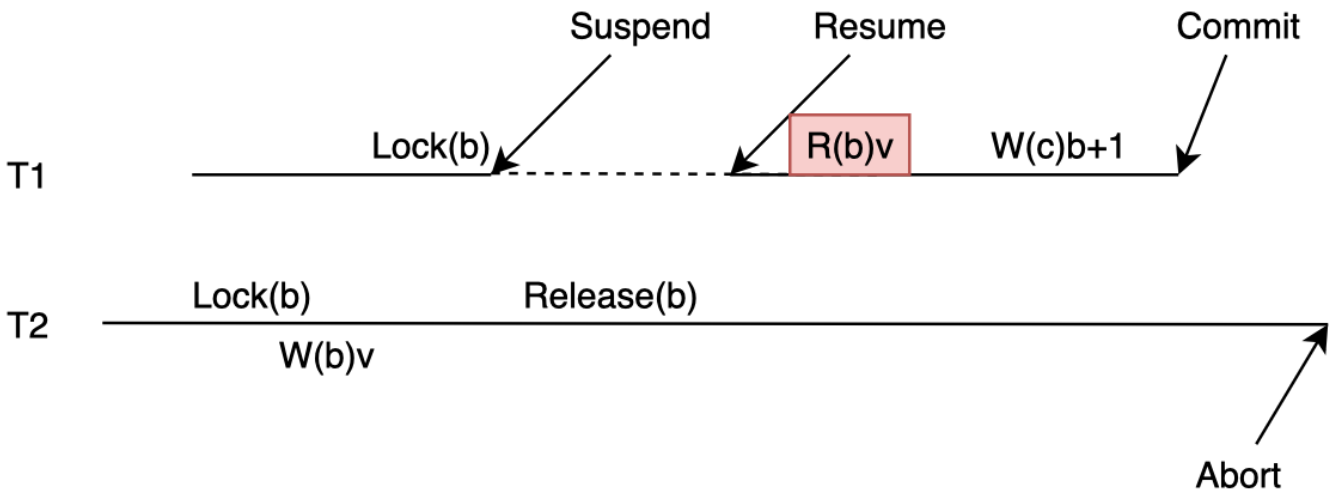
Controle de concorrência - *Locking*

- Todos os objetos acessados pela transação são "trancados" até que não sejam mais usados
- Abordagem deve ser usada da maneira correta para evitar *dirty read:*



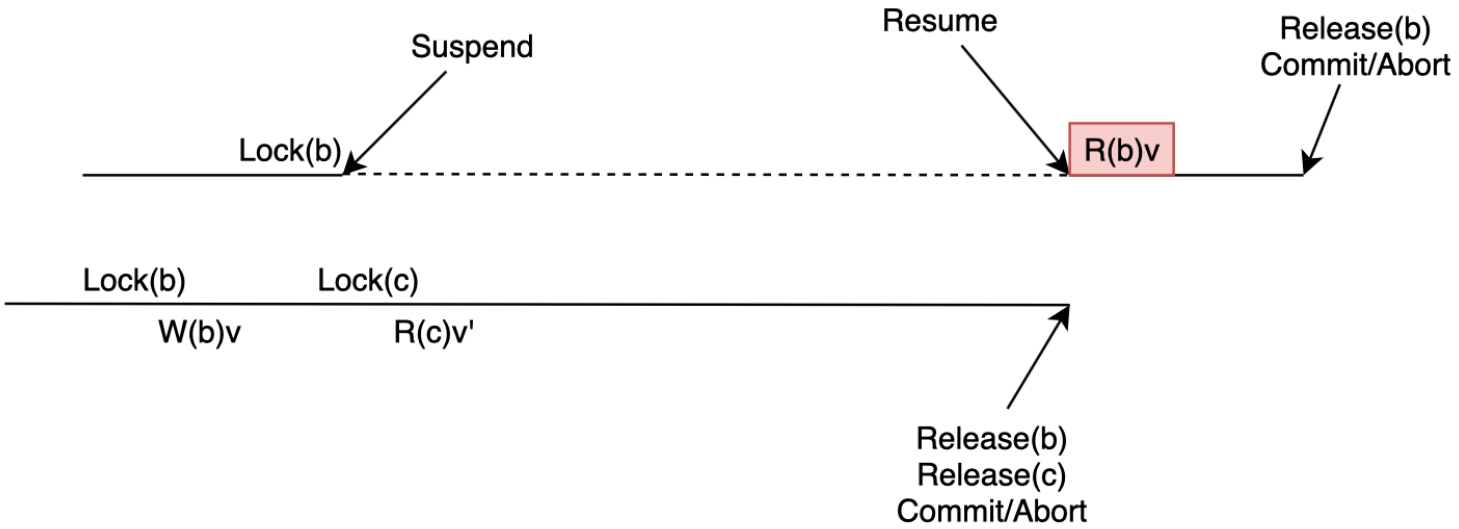
Controle de concorrência - *Locking*

- Todos os objetos acessados pela transação são "trancados" até que não sejam mais usados
- Abordagem deve ser usada da maneira correta para evitar *dirty read*
  - *Abort* em cascata pode não resolver:



Controle de concorrência - *Locking*

- Todos os objetos acessados pela transação são "trancados" até que não sejam mais usados
- Uso de ***strict two phase locking*** resolve:
  - Transações **trancam o objeto quando primeiro acessado** e só **destrancam ao final da transação**



Controle de concorrência - *Locking*

- Uso de ***strict two phase locking*** reduz concorrência
- Alternativas:
  - Read/Write locks
  - múltiplos leitores/único escritor
  - Diferentes granularidades:
    - lock em uma coluna de uma linha do banco, de toda a linha, de toda a relação, ou mesmo de todo o banco de dados

Controle de concorrência - Multi-versão

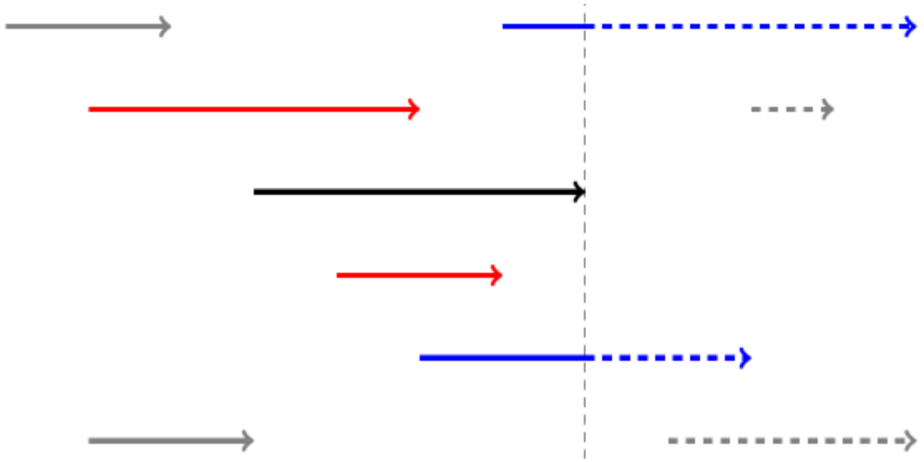
- Controle de concorrência multi-versão (MVCC, do termo em inglês)
- Mantém uma cópia privada dos dados acessados pela transação
- Ao final da execução, na fase de **validação:**
  - se cópia pública não tiver sido modificada, a transação é bem sucedida e atualizações são feitas nas cópias públicas
- Técnica conhecida como ***deferred update:***
  - Atrasa a atualização da cópia pública até o final da transação
  - Baixo *overhead*, se não houver conflitos
  - Se houver muitos conflitos, o trabalho da transação é todo desperdiçado
    - Transação será abortada na **validação**

Controle de concorrência - Multi-versão

- Validação
  - Consiste em verificar se os *read* e *write sets* de quaisquer transações concorrentes são disjuntos
  - Dadas as transações t1 e t2:
    - t1 não deve ler dados escritos por t2
    - t2 não deve ler dados escritos por t1
    - t1/t2 não deve escrever dados escritos por t2/t1

Controle de concorrência - Multi-versão

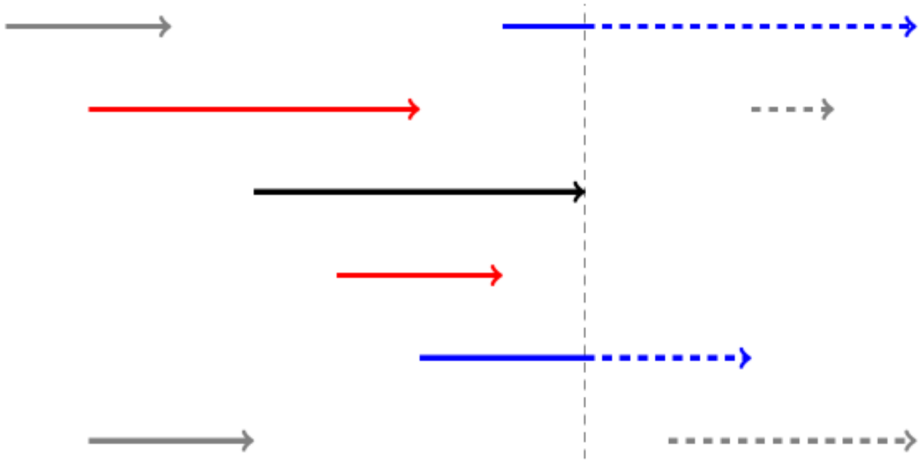
- Validação



- Transações já comitadas (***backward validation***):
  - t1: transação sendo validada (seta preta)
  - t2: transação já comitada (setas vermelhas)
  - t1 não deve ler dados escritos por t2

Controle de concorrência - Multi-versão

- Validação



- Transações ainda em execução (***forward validation***):
  - t1: transação sendo validada (seta preta)
  - t2: transação ainda em execução (setas azuis)
  - t2 não deve ler dados escritos por t1
  - pode levar a cenário em que nenhuma transação é jamais comitada, pois uma cascata de aborts pode ocorrer

Controle de concorrência - *Timestamp*

- Atribui-se um *timestamp* a cada transação
- Execução deve ser equivalente à execução serial de acordo com os *timestamps*
- Transação recebe um *timestamp* no início
- Operações são validadas na execução:
  - leia somente se nenhuma transação com maior *timestamp* tiver escrito e comitado
  - escreva somente se nenhuma transação com maior *timestamp* tiver lido e comitado
- Transações "executam na ordem do *timestamp*"

Bancos de dados distribuídos

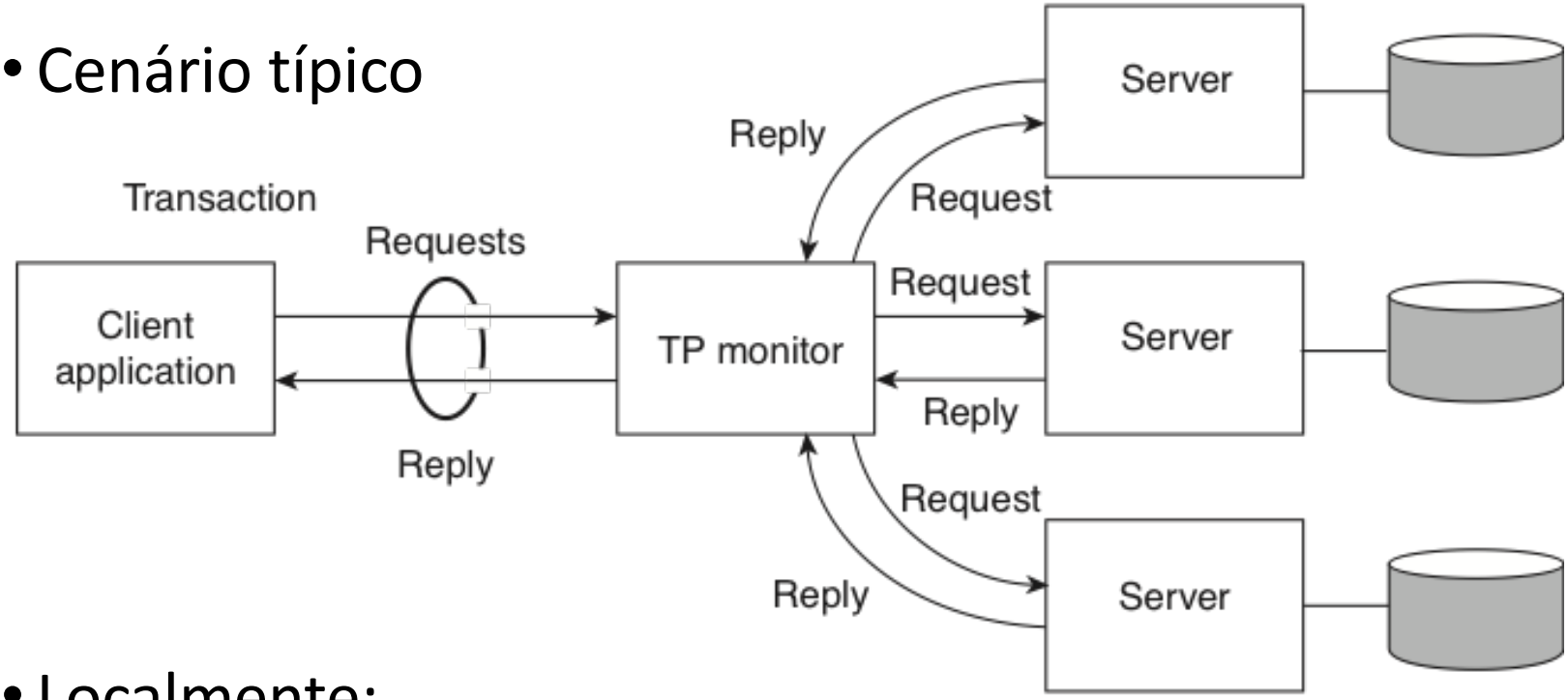
- Como implementar controle de transações em um sistema distribuído?
  - Múltiplos servidores
  - Transações em cada servidor
  - Transações distribuídas
  - Como obter equivalência serial em transações distribuídas?

Bancos de dados distribuídos

- Transação distribuída - representação
  - `begintransaction(): tid (transaction id)`
  - `operation(tid,op)`
  - `endtransaction(tid): ok/nok`
  - `aborttransaction(tid)`
- Papéis
  - Cliente
  - servidor: *resource managers*
  - servidor: *transaction monitor/manager*

Bancos de dados distribuídos

- Cenário típico



- Localmente:
  - Cada BD funciona como um sistema centralizado normal
  - Usa abordagens otimistas/pessimistas para garantir consistência
- Grande problema com BD distribuído:
  - Garantir o *acordo* na terminação

Comprometimento distribuído

- Comprometimento distribuído (*Distributed Commitment*)
  - transação ***t*** acessa recursos nos resource managers (*rm*)
  - terminar com sucesso ***t*** em todos os *rm* - *commit* - ou
  - abortar ***t*** em todos os *rm*
  - ainda que enlaces de comunicação, nós e *rm* falhem, antes ou durante a terminação da transação

Comprometimento distribuído

- Participante
  - resource manager "tocado" pela transação
- Coordenador
  - transaction manager
- Cliente decide quando iniciar o *commit*

- Cada participante faz *commit* ou *abort* da transação local
  - pode retornar ok ou nok.
- Coordenador não começa a *commit* até que a ***t*** tenha terminado em todos os participantes e cliente tenha solicitado.
- Participantes falham por parada

Comprometimento distribuído

- **1PC (*One phase commit*)**

- Cliente envia `endtransaction(tid)` para o coordenador
- Coordenador envia mensagem para participantes "comitarem"

- Mas...
  - E se um participante retornar nok enquanto outros retornam ok?
  - E se um participante não responder?

Comprometimento distribuído

- **2PC (*Two phase commit*)**

- Cliente envia `endtransaction(tid)` para o coordenador
- coordenador envia mensagem para participantes se prepararem para terminar
- coordenador espera que todos se preparem ou digam se não podem
- coordenador envia *ordem* de terminação

Comprometimento distribuído

- **2PC - Comprometimento**
- Um participante ***p*** está pronto para *commit* se:
  - Tiver todos os valores modificados por ***t*** em memória estável, e
  - Nenhuma razão para abortar a transação
- Coordenador não pode começar a terminação até que **todos** os participantes estejam prontos
- Se algum participante aborta, o coordenador deve abortar

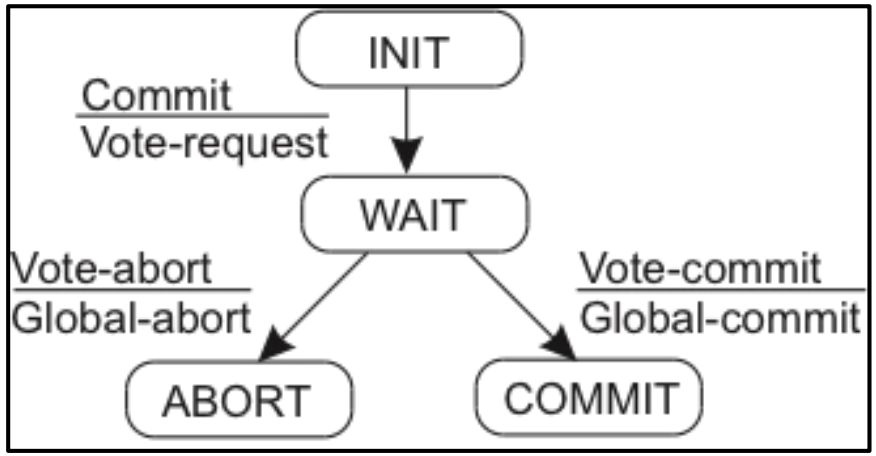
- Problema de acordo: igual ao consenso?

Comprometimento distribuído

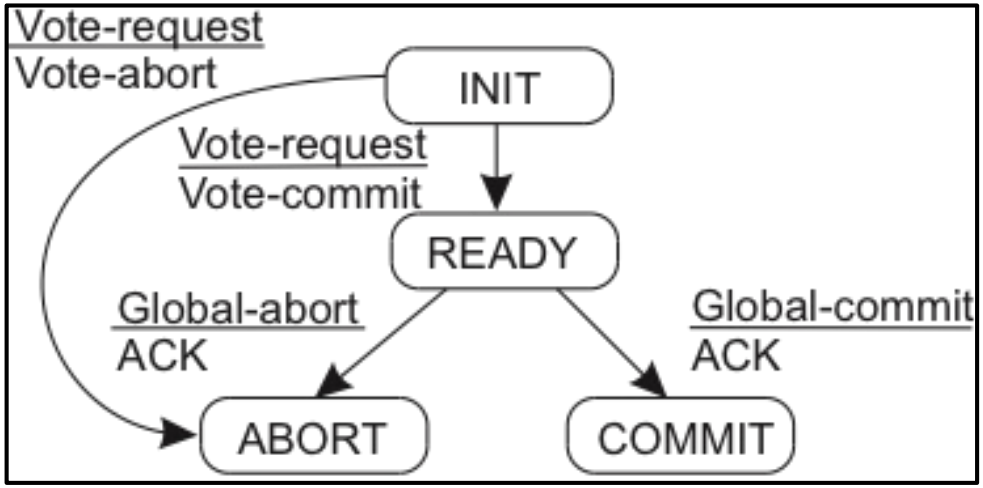
- **2PC - Protocolo**
- Fase 1
  - a: coordenador envia `vote-request` para participantes
  - b: participante responde com `vote-commit` ou `vote-abort` para o coordenador
    se `vote-abort`, então aborta localmente
- Fase 2
  - a: coordenador coleta votos de todos os processos
    se forem todos `vote-commit`,
    então envia `global-commit` para os participantes
    e `ok` para o cliente
  - b: participantes esperam por `global-commit` ou `global-abort`

Comprometimento distribuído

- **2PC – Protocolo**



Coordenador



Participante

Comprometimento distribuído

- **2PC – Falha no participante (e posterior recuperação)**
- Se no estado:
  - INIT:
  - ABORT:
  - COMMIT:
  - READY:

Comprometimento distribuído

- **2PC – Falha no participante (e posterior recuperação)**
- Se no estado:
  - INIT:
    em sabia que a terminação começou → Aborta
  - ABORT:
    havia votado `abort` recebido `global-abort` → Continua
  - COMMIT:
  - estava pronto para terminar a transação → Continua
  - READY:
    estava esperando por `commit/abort` → Consulta coordenador

        E se coordenador não estiver presente?

Comprometimento distribuído

- **2PC – Falha no coordenador (e posterior recuperação)**
- E se ninguém ouviu a decisão final do coordenador?
  - O protocolo não pode continuar enquanto o coordenador não retornar
    - Se os RM abortarem, podem estar contradizendo algo dito ao cliente, por exemplo, "Sim, ATM, pode entregar o dinheiro”
    - Se comitarem, podem estar executando um comando que o cliente vê como anulado, como "Reenvie o pedido de mais 27 carros à fábrica"

Comprometimento distribuído

- **2PC – Falha no coordenador (e posterior recuperação)**
- Ao se recuperar, o coordenador:
  - sabe se começou a terminação de alguma transação
  - sabe se já enviou alguma resposta final para as transações inacabadas
  - sabe se já recebeu a confirmação de todos os participantes (se transação não estiver em aberto)
  - reenvia a última mensagem das transações em aberto

Comprometimento distribuído

- **2PC – Otimizações**
- Participantes "somente-leitura"
  - Não se importa com a decisão; termina após fase 1
  - Responde com `vote-commit-ro`
- *Abort* presumido
  - Se ocorrer timeout, coordenador envia `global-abort` a todos e esquece transação
  - Se questionado, responde com `global-abort`

Comprometimento distribuído

- **2PC – Coleta de Lixo**
- Após receber decisão, o participante pode concluir e esquecer a transação
- Mas e se um participante falho precisar se recuperar e todos os outros envolvidos tiverem esquecido a transação?
  - Coleta de lixo só pode ser feita quando todos tiverem confirmado a execução da transação e, por isso, Fase 2b é necessária.

Comprometimento distribuído

- **Paxos Commit**
  - Usa instâncias de Consenso Distribuído para votar
- **O protocolo**
  - Para terminar a transação ***T***, coordenador envia `request-commit` a todos os participantes
  - Participante ***P*** propõe seu voto na instância T_P de consenso
  - Todo participante ***P*** espera pelas decisões das instâncias de consenso ***Ti*** para todos os participantes ***i***
    - Se todas as decisões forem `commit`, comita a transação
    - Se cansar de esperar por T_Q, propõe `abort` em T_Q

Comprometimento distribuído

- **Paxos Commit - Falha no Participante**
- Se participante falha antes de votar, então alguém votará `abort` por ele
- Se participante ***P*** falha (ou é suspeito de):
  - É possível que dois votos diferentes tenham sido propostos em TP
  - Não é um problema pois a decisão é a mesma para todos observando a instância (consenso garante)
- Após se recuperar:
  - Participante recupera as decisões de todas as instâncias T_i e termina apropriadamente