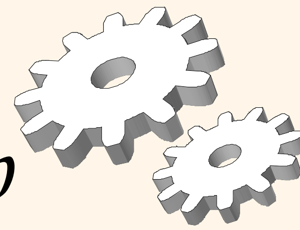
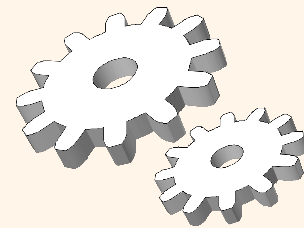


Instituto de Computação Unicamp
Disciplina de Banco de Dados

Controle de Concorrência

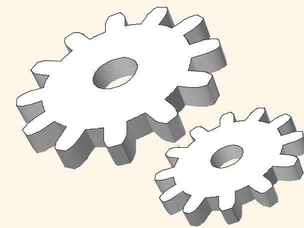
Capítulo 17



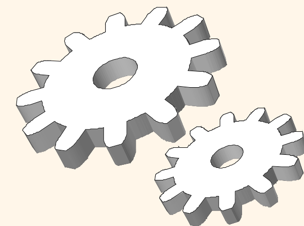
Roteiro

- Parte 1
  - Controle de Lock Baseado em Concorrência
  - Gerenciamento de Lock
- Parte 2
  - Técnicas de Locking Especializadas
  - Controle de Concorrência Sem Locking

Plano Conflito-Serializável



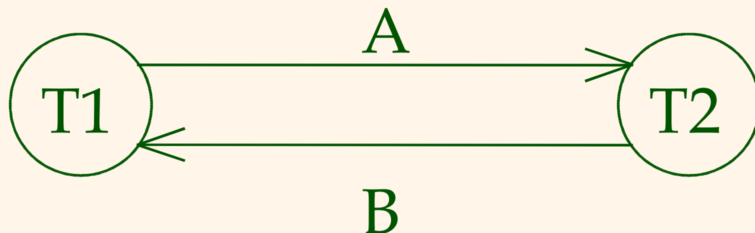
- Dois planos são conflito-equivalentes se:
  - Eles envolvem as mesmas ações das mesmas transações
  - Todo par de ações conflitantes é ordenada no mesmo caminho
- Plano S é conflito-serializável se S é conflito-equivalente em um mesmo plano serial



Exemplo

- Um plano que não é conflito-serializável:

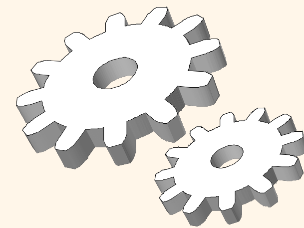
T1:	R(A), W(A),	R(B), W(B)
T2:	R(A), W(A), R(B), W(B)	



Grafo de Dependência

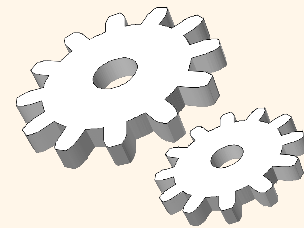
Captura potenciais conflitos entre transações

- O ciclo no grafo revela o problema. A saída de T1 depende de T2, e vice-versa.



Grafo de Dependência

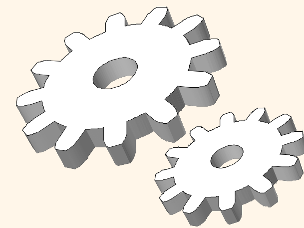
- *Grafo de Dependência*: aresta de *Ti* para *Tj* se *Tj* lê/escreve o último objeto escrito por *Ti*.
- Teorema: Plano é conflito-serializável se e somente se seu grafo de dependência é acíclico



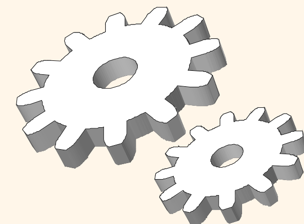
Revisão: 2PL Strict

- *Protocolo 2-Phase Locking (2PL Strict)*:
  - Cada ação X tem que obter um lock S (*compartilhado*) no objeto antes da leitura, e um lock X (*exclusivo*) no objeto antes da escrita
  - Todos os locks mantidos por uma transação são liberados quando a transação é completada
  - Se uma ação X mantém um lock X sobre um objeto, nenhuma outra ação pode conseguir um lock nesse objeto
- 2PL permite somente planos ao qual o grafo é acíclico (ou seja, conflito serializável)

Two-Phase Locking (2PL)



- Protocolo 2-Phase Locking
  - Cada ação X tem que obter um lock S (*compartilhado*) sobre o objeto antes da leitura, e um lock X (*exclusivo*) sobre o objeto antes da escrita
  - Uma transação não pode solicitar locks adicionais a não ser que ela libere locks
  - Se uma ação X mantém um lock X sobre um objeto, nenhuma outra ação pode conseguir um lock nesse objeto

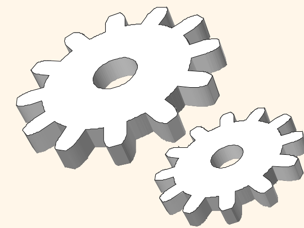


Visão Serializável

- Os planos S1 e S2 são visão-equivalentes se:
  - Se Ti lê o valor inicial de A em S1, então Ti também lê o valor inicial de A em S2
  - Se Ti lê o valor de A escrito por Tj em S1, então Ti também lê o valor de A escrito por Tj em S2
  - Se Ti escreve o valor final de A em S1, então Ti também escreve o valor final de A em S2

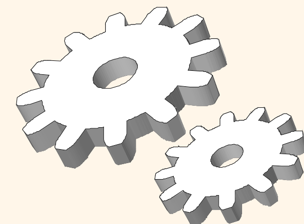
T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)



Gerenciamento de Lock

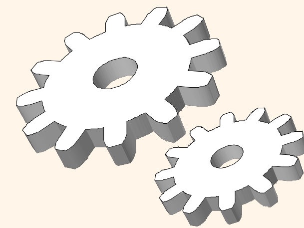
- Pedidos de Lock e UnLock são executados por um gerenciador de lock
- Entradas em uma tabela de Lock:
  - Número de transações correntemente permitidas ao lock de um objeto (se estiver em modo compartilhado)
  - Tipo do lock (compartilhado ou exclusivo)
  - Ponteiro para pilha de locks solicitados
- Atualização de Lock: transações que estão com um lock compartilhado podem ser atualizadas para permitir um lock exclusivo



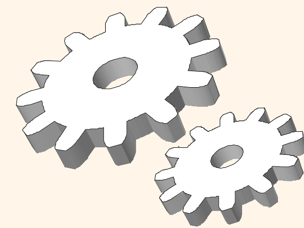
Deadlocks

- Deadlock: ciclo de transações esperando por locks serem liberados um dos outros
- Dois caminhos para gerenciar deadlocks:
  - Prevenção de Deadlock
  - Detecção de Deadlock

Prevenção de Deadlock

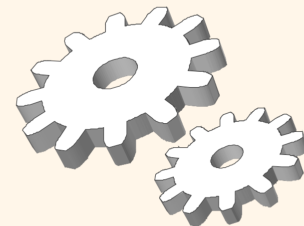


- Associar prioridades com base em timestamps. Assume-se que Ti quer um lock que Tj mantém. Duas políticas são possíveis:
  - Espera-Morrer: se Ti tem maior prioridade, Ti espera por Tj; em outro caso Ti aborta
  - Espera-Golpe: se Ti tem maior prioridade, Tj aborta; em outro caso Ti espera
- Se uma transação reinicia, faz-se garantir que ela tenha seu timestamp original



Detecção de Deadlock

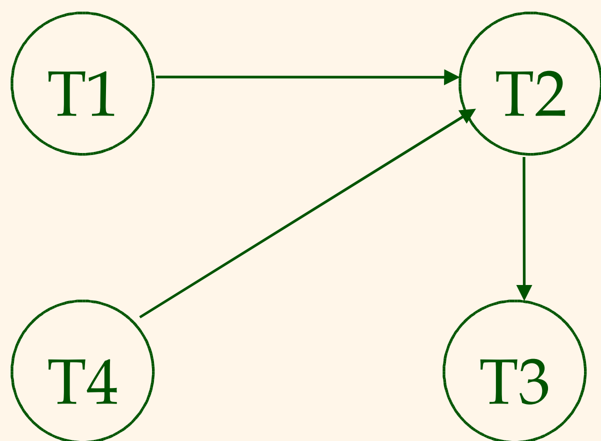
- Criar um grafo waits-for:
  - Nodos são transações
  - Há uma aresta de Ti para Tj se Ti está esperando por Tj liberar um lock
- Periodicamente procurar por ciclos no grafo waits-for

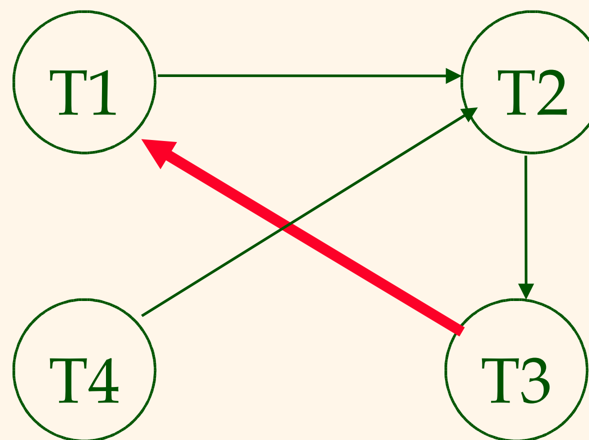


Detecção de Deadlock (Cont)

Exemplo:

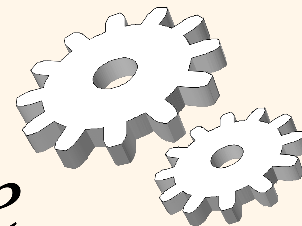
T1: S(A), R(A), S(B)
T2: X(B),W(B) X(C)
T3: S(C), R(C) X(A)
T4: X(B)



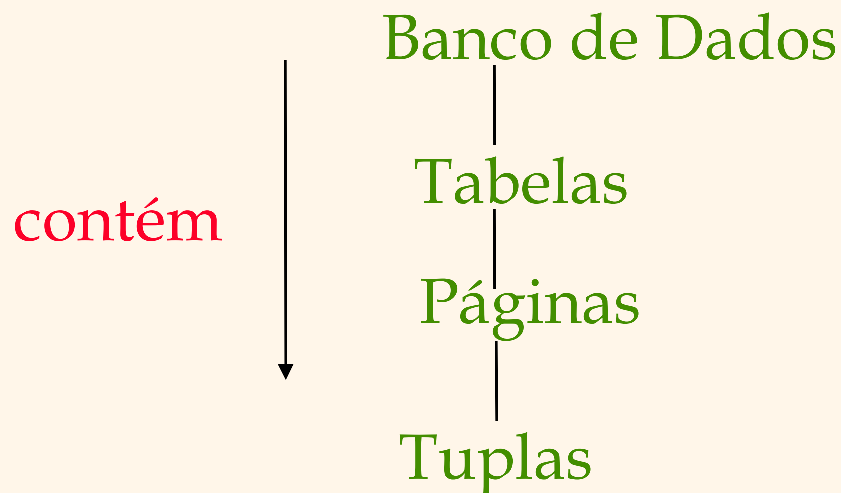


Grafo Waits-For antes e depois do deadlock

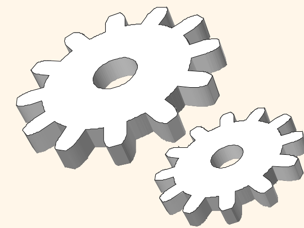
Locks de Múltipla-Granularidade



- Difícil decidir o que é granularidade para o lock (tuplas vs. páginas vs. tabelas)
- Dados são aninhados:



Solução: Novos Modos de Lock

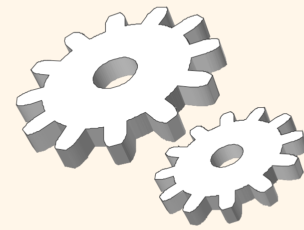


- Permite a uma ação X um lock em cada nível, mas com um protocolo especial usando "intention locks":
- Antes de dar o lock num item, a ação X tem que colocar "intention locks" sobre todos os seus ancestrais
- No unlock, caminhamos do específico para o geral (i.e., bottom-up).

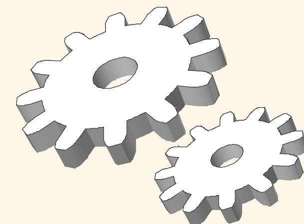
	IS	IX	S	X
IS	√	√	√	
IX	√	√		
S	√		√	
X				

- Modo SIX: S e IX ao mesmo tempo (conflita com locks que conflitam com S e IX).

Protocolo Lock de Múltiplo-Granularidade



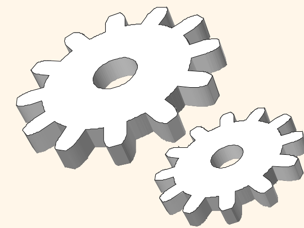
- Cada ação X inicia da raiz da hierarquia.
- Para conseguir o lock S ou IS sobre um nodo, tem que manter IS no nodo pai.
- Para conseguir X ou IX ou SIX sobre um nodo, tem que manter IX ou SIX no nodo pai.
- Tem que liberar locks em ordem bottom-up.



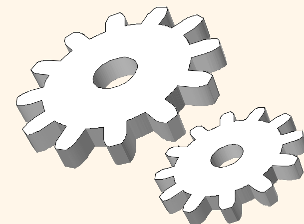
Exemplos

- T1 varre R, e atualiza algumas poucas tuplas:
  - T1 consegue um lock SIX sobre R, então repetidamente consegue um lock S sobre as tuplas de R, e ocasionalmente atualiza para o lock X
- T2 usa um índice para ler somente parte de R:
  - T2 consegue um lock IS sobre R, e repetidamente consegue um lock S nas tuplas de R
- T3 lê todos de R:
  - T3 consegue um lock S sobre R.
  - Ou, T3 poderia comportar-se como T2 ou;
    pode usar lock escalável para decidir

Bancos de Dados Dinâmicos



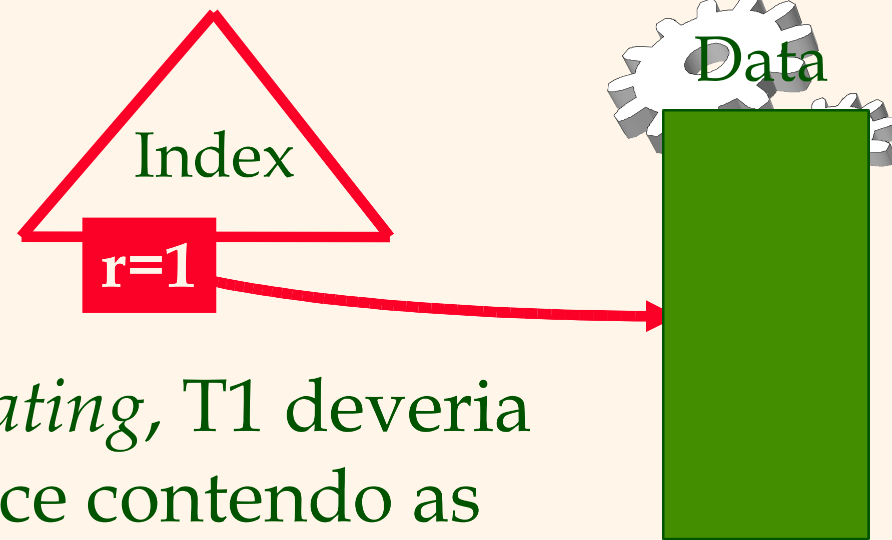
- Se nós relaxamos a hipótese que o BD é uma coleção fixa de objetos, até mesmo 2PL não garantirá seriabilidade:
  - T1 dá lock em todas as páginas contendo registros sailor com *rating* = 1, e encontra o sailor mais velho (*age* = 71)
  - Depois, T2 insere um novo sailor; *rating* = 1, *age* = 96
  - T2 também elimina o sailor mais velho com *rating* = 2 (*age* = 80), e confirma (commit)
  - T1 agora dá lock em todas as páginas contendo registros sailor com *rating* = 2, e encontra o mais velho (*age* = 63).
- Não há um estado consistente do BD onde T1 é correto



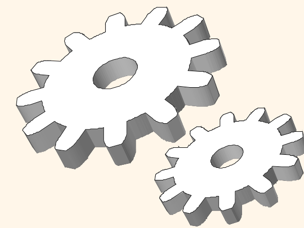
Problema

- T1 implicitamente assume que ele tem locked o conjunto de todos os registros sailor com *rating* = 1.
  - A hipótese somente é mantida se nenhum registro sailor é adicionado enquanto T1 é executado
  - São necessários alguns mecanismos para reforçar essa hipótese (Index locking e Predicate locking).
- Pesquisas mostram que o conflito-serializável garante seriabilidade somente se o conjunto de objetos é fixo

Index Locking

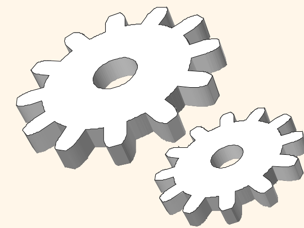


- Se há um índice no campo *rating*, T1 deveria dar lock nas páginas de índice contendo as entradas de dados com *rating* = 1
  - Se não há registros com *rating* = 1, T1 tem que dar lock nas páginas de índice onde uma entrada de dados *estaria*, se ela existisse
- Se não há um índice adequado, T1 tem que dar lock em todas as páginas, e dar lock no arquivo/tabela para prevenir que novas páginas sejam adicionadas, para garantir que nenhum novo registro com *rating* = 1 seja adicionado



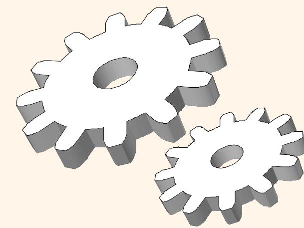
Predicate Locking

- Permissão de lock a todos os registros que satisfazem algum predicado lógico, e.g. *age > 2*salary*
- Index locking é um caso especial de predicate locking ao qual um índice suporta implementação eficiente do predicate lock
- Em geral, predicate locking tem muitos locking overhead



Locking em Árvores B+

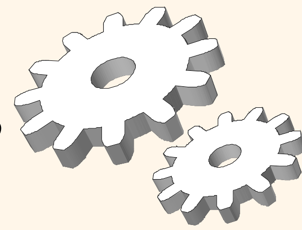
- Como nós podemos eficientemente dar lock a um particular nodo-folha?
- Uma solução: Ignorar a estrutura da árvore, simplesmente dar lock a páginas enquanto caminhando pela árvore, seguindo 2PL
- Isso tem uma performance muito ruim:
  - Nodo raiz (e os nodos de nível mais alto) tornam-se gargalos, porque toda árvore acessa inicialmente a raiz



Duas Observações Úteis

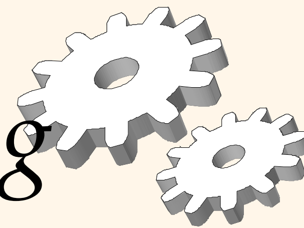
- Níveis mais altos da árvore somente direcionam as pesquisas para páginas-folha
- Em inserções, um nodo tem que estar locked (em modo X) - similar na eliminação
- Nós podemos explorar essas observações para projetar protocolos de locking eficientes que garantam seriabilidade *apesar de violarem a 2PL*

Algoritmo Simples de Locking de Árvores

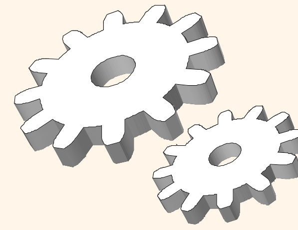


- Procura: inicia na raiz e desce; repetidamente dá lock no filho e unlock no pai
- Inserção/Eliminação: inicia na raiz e desce, obtendo quantos locks forem necessários. Uma vez o filho estando locked, verifica se ele é seguro:
  - Se o filho é seguro, libera todos os locks dos antecessores
- Nodo seguro: nodo em que as mudanças não serão propagadas acima desse nodo

Um Melhor Algoritmo de Locking em Árvores (Bayer-Schkolnick)

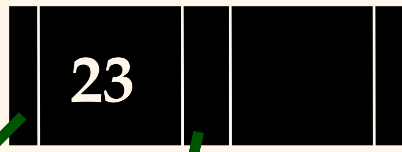


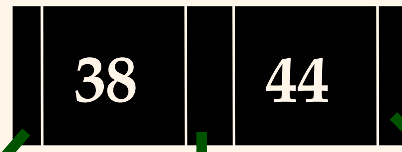
- Procura: como antes
- Inserção/Eliminação:
  - Dá o lock para procura, consegue a folha, e atribui o lock para a folha
  - Se a folha não é segura, libera todos os locks, e reinicia a ação X usando protocolos de Inserção/Eliminação
- Aposta que somente o nodo folha será modificado; senão, dá o lock para a primeira folha que passar. Na prática, é melhor que o algoritmo anterior.

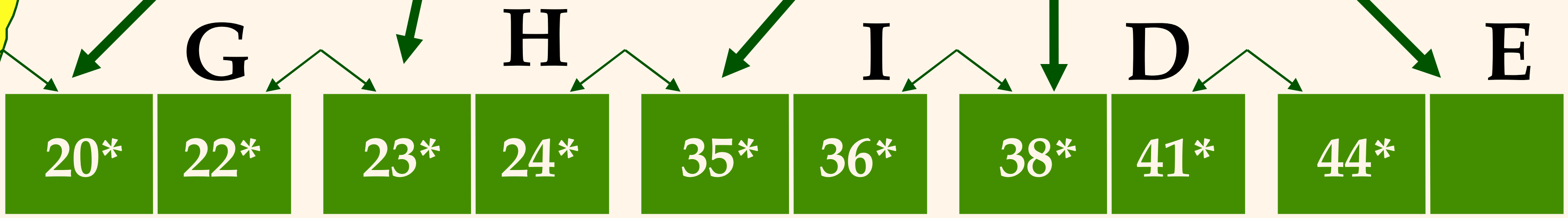


ROOT

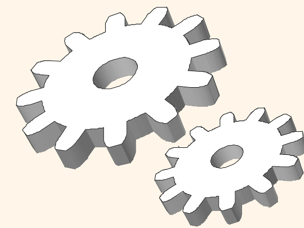
Examplo







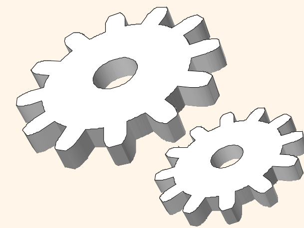
Faça:
1) Delete 38*
2) Insert 25*
4) Insert 45*
5) Insert 45*, then 46*



Um Algoritmo Ainda Melhor

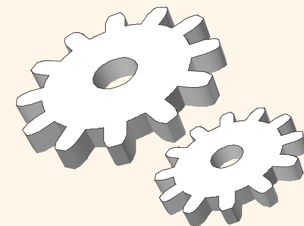
- Procura: como antes.
- Insert/Delete:
  - Usa o protocolo original de Insert/Delete, mas atribui locks IX em vez de locks X em todos os nós.
  - Cada nó-folha recebe um lock, os locks IX são convertidos para locks X top-down: i.e., iniciando dos nós-folhas próximos do root. (Top-down reduz chances de ocorrer deadlock.)

(Contraste no uso de locks IX aqui com o uso dele no protocolo de lock com múltiplo-granularidade.)



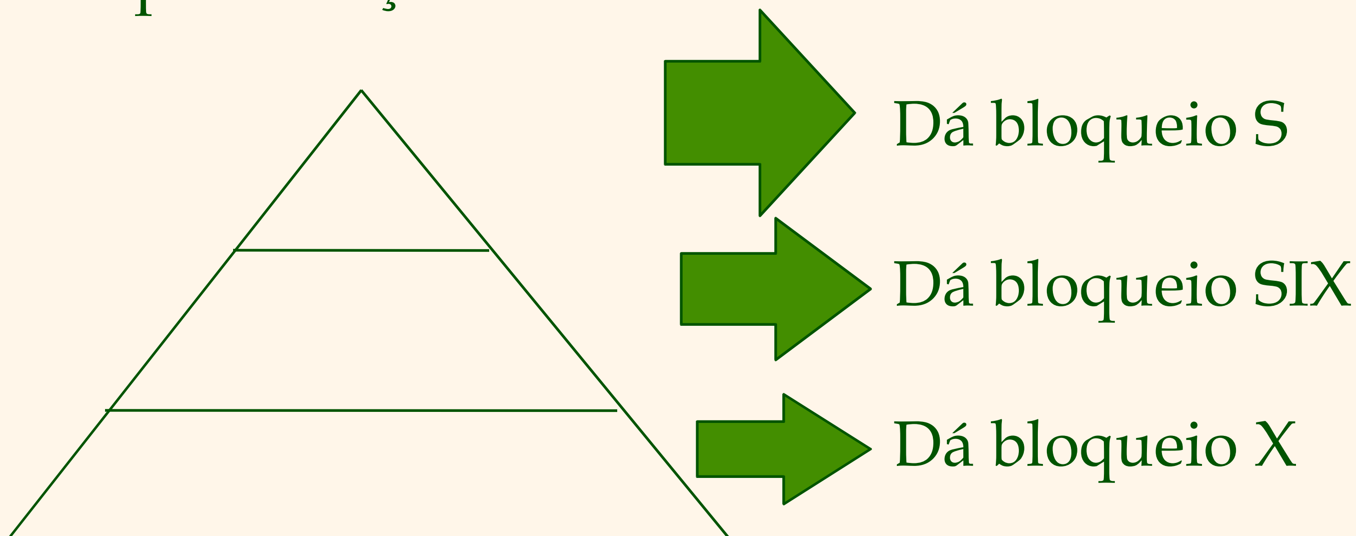
Roteiro

- Parte 1
  - Controle de Lock Baseado em Concorrência
  - Gerenciamento de Lock
- Parte 2
  - Técnicas de Locking Especializadas
  - Controle de Concorrência Sem Locking

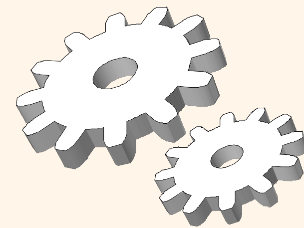


Algoritmo Híbrido

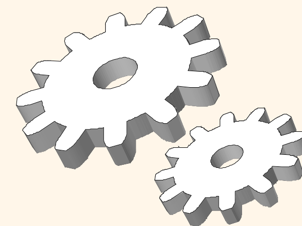
- A possibilidade que nós realmente precisarmos de um bloqueio exclusivo decrementa a medida que nós nos movimentamos para cima na árvore.
- Aproximação Híbrida:



CC Otimista (Kung-Robinson)

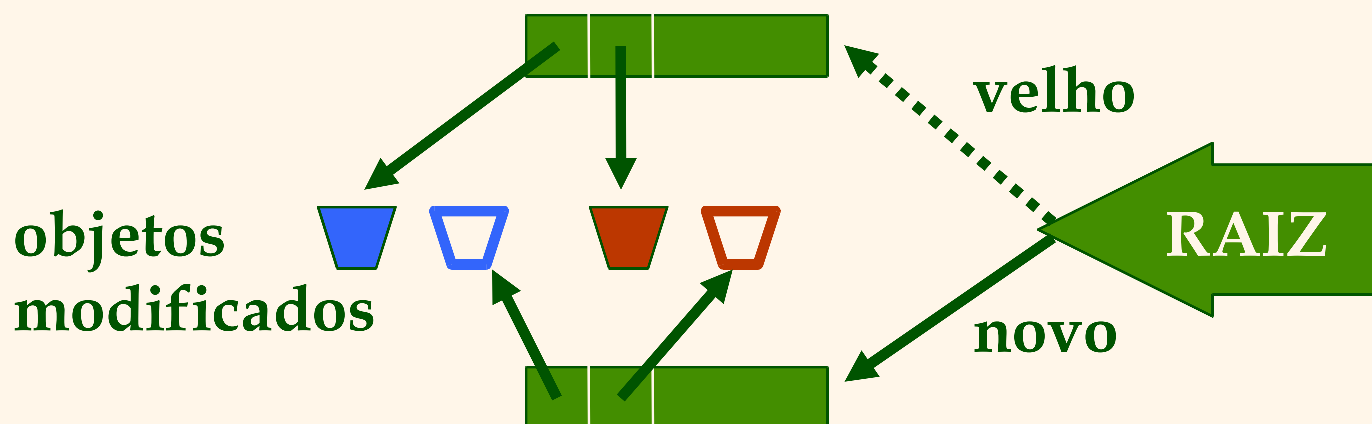


- Protocolo de Bloqueio tem uma visão pessimista sobre as transações,   na qual os conflitos são prevenidos. Desvantagens:
  - Gerenciamento de Locking (Bloqueio) .
  - Detecção/Resolução de Deadlock.
  - Contenção de Lock para uso de objetos.
- Se os conflitos são raros, nós podemos supor que não ocorrerão conflitos e optar pelo não-locking. Ao invés de procurar por conflitos antes da ação "Exclusiva" confirmar (commit), deixamos este passo para depois.

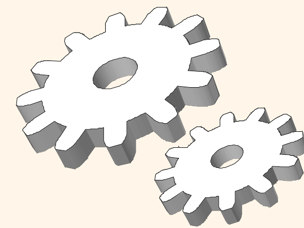


Modelo Kung-Robinson

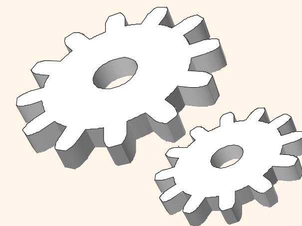
- A ação exclusiva tem três fases:
  - READ: a ação exclusiva lê o banco de dados, mas faz mudanças para cópias privadas de objetos.
  - VALIDATE: procura por conflitos.
  - WRITE: salva no banco de dados as cópias locais.



Validação

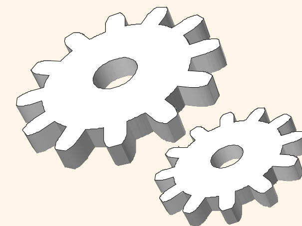


- Condições dos testes que são suficientes para garantir que não ocorram conflitos.
- A cada ação exclusiva é associado um ***id*** numérico.
  - Usa-se um **timestamp**.
- Os ***ids*** das ações exclusivas são associados no final da fase READ, antes do início da validação.
- ReadSet(Ti): conjunto de objetos lidos por Ti.
- WriteSet(Ti): conjunto de objetos modificados por Ti.



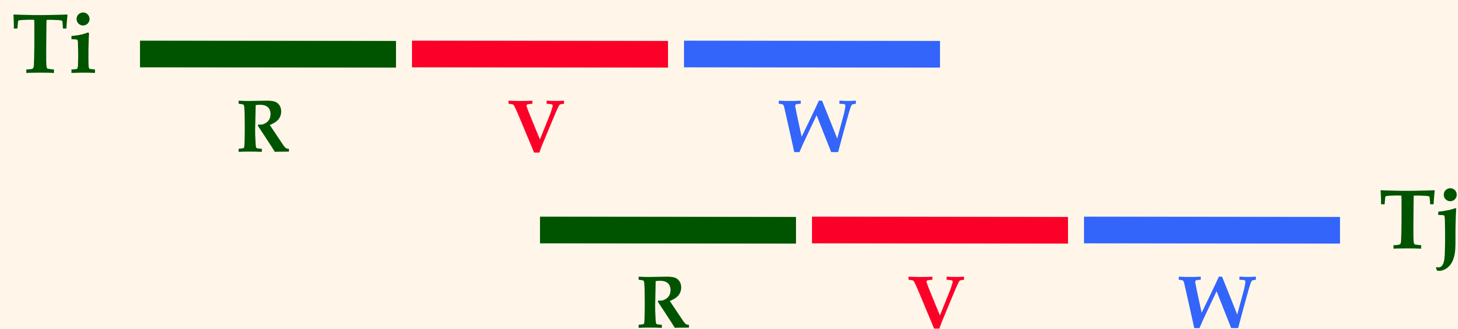
Teste 1

- Para todo i e j tal que Ti < Tj, verificar que Ti completa antes de Tj começar.

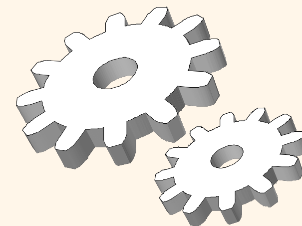


Teste 2

- Para todo i e j tal que Ti < Tj, verificar que:
  - Ti completa antes de Tj começar sua fase de Write

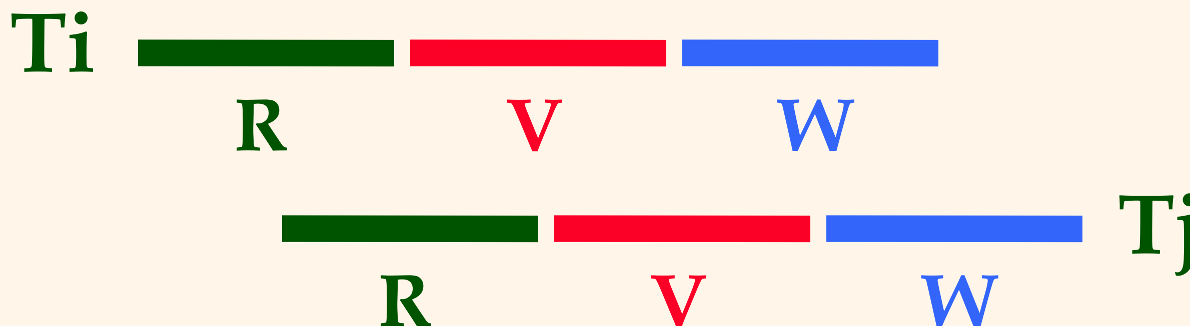


Tj lê o dado sujo? Ti sobreescreve as escritas de Tj?



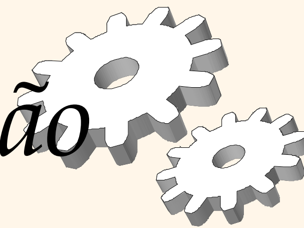
Teste 3

- Para todo i e j tal que Ti < Tj, verificar que:
  - Ti completa a fase de Read antes que Tj.



Tj lê o dado sujo? Ti sobreescreve as escritas de Tj?

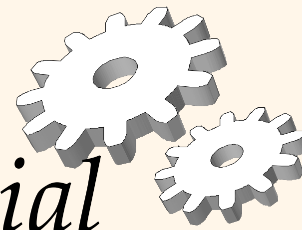
Aplicando os Testes 1 e 2: Validação Serial



- Para validar a ação exclusiva:

```
valid = true;
/* S = conjunto das ações exclusivas que são committed
depois de Begin(T) */
< foreach  Ts in S do {
  if ReadSet(Ts) does not intersect WriteSet(Ts)
        then valid = false;
  }
  if valid then { install updates; // Fase de Write
                  Commit T }
          else Restart T>
```

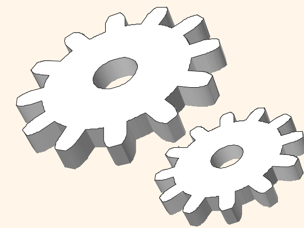
fim da seção crítica



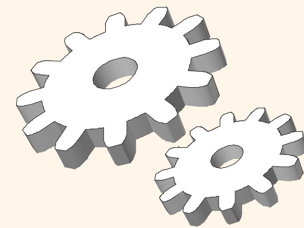
Comentários sobre Validação Serial

- Aplicando o Teste 2, com T representando a regra de Tj e cada ação exclusiva em Ts.
- Associação do ***id*** a ação exclusiva, validação, e a fase de Write estão dentro de uma **seção crítica**
- Otimização para as ações de Leitura:
  - Não há a necessidade de seção crítica (porque não há fase de Write).

Validação Serial (Cont)

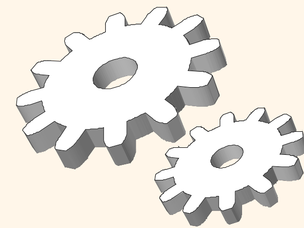


- Validação Serial MultiEstágio: valida em estágios, em cada estágio validando T, contra um subconjunto de ações exclusiva que são committed depois de Begin(T).
  - Somente no último estágio é preciso estar em seção crítica
- Starvation: executa starving da ação exclusiva em uma seção crítica
- Espaço para WriteSets: para validar Tj, temos que ter WriteSets para todos os Ti, onde Ti < Tj e Ti foi ativado quando Tj iniciou.
  - Não há problemas se os ***ids*** das ações exclusiva são associados no início da fase Read.



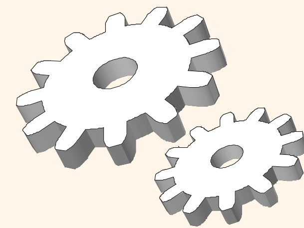
Overheads em CC Otimista

- Registrar as atividades de read/write em ReadSet e WriteSet por ação exclusiva.
  - Criar e destruir esses conjuntos quanto necessário.
- Buscar por conflitos durante a validação.
  - Seções críticas podem reduzir concorrência.
- CC otimista reinicia as ações exclusivas que falharam na validação.



2PL "Otimista"

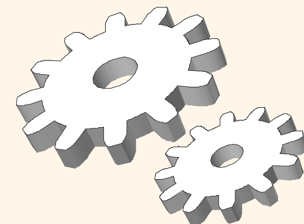
- É possível fazer o seguinte:
  - Conjunto de locks compartilhados como usual
  - Mudanças para cópias privadas de objetos.
  - Obter todos os locks exclusivos no final da ação exclusiva, fazer a escrita, e então liberar todos os locks.
- Em contraste do CC Otimista com o Kung-Robinson, este esquema resulta em ações exclusivas sendo bloqueadas, esperando por locks.
  - Entretanto, não há fase de validação, não há reinicialização (módulo deadlocks).



CC Timestamp

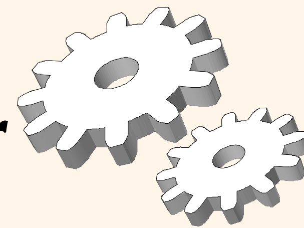
- **Idéia:** dar a cada objeto um read-timestamp (RTS) e um write-timestamp (WTS), dar a cada ação transação um timestamp (TS) quando ela inicia:
  - Se a ação “a_i” das ações exclusivas em Ti dá conflito com a ação “a_j” das ações exclusivas em Tj, e TS(Ti) < TS(Tj), então “a_i” tem que ocorrer antes de “a_j”.

Quando a transação quer ler o Objeto O



- Se TS(T) < WTS(O), isso viola a ordem do timestamp de T para o escritor de O.
  - Então, aborta T e reinicia ele com um novo e maior TS (se reiniciar com o mesmo TS, T falhará novamente).
- Se TS(T) > WTS(O):
  - Permitir T para ler O.
  - Reinicia RTS(O) para max(RTS(O), TS(T))

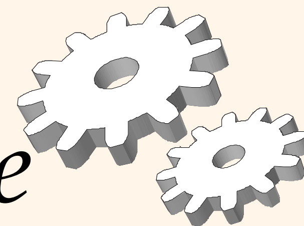
Quando a transação quer escrever o Objeto O



- Se TS(T) < RTS(O), isso viola a ordem do timestamp de T para o escritor de O; aborta e reinicia T.
- Se TS(T) < WTS(O), viola a ordem do timestamp de T para o escritor de O.
  - **Regra de Thomas Write: Nós podemos seguramente ignorar escritas antigas; não sendo necessário reiniciar T (a escrita em T é efetivamente seguida por outra escrita, sem a intervenção de leituras).**
- Senão, permite T para escrever O.

T1	T2
R(A)	
	W(A) **Commit**
W(A) **Commit**	

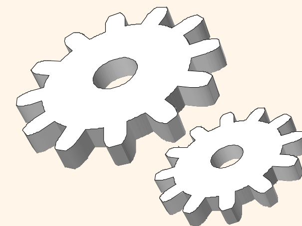
CC Timestamp e Recuperabilidade



- Planos de não-recuperabilidade são permitidos:

T1	T2
W(A)	
	R(A)
	W(B)
	Commit

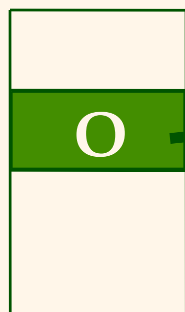
- CC Timestamp pode ser modificado para permitir somente planos recuperáveis:
  - Buffer sempre escrito até que o escritor dê o commit (o WTS (O) é atualizado quando a escrita é permitida)
  - Leitores de Blocos T (onde TS(T) > WTS(O)) até que o escritor de O commit
- Similar aos escritores que mantém os locks exclusivos até commit, mas ainda não é um completo 2PL

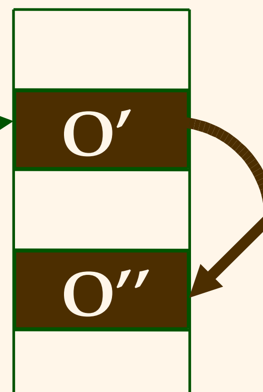


CC Timestamp MultiVersão

- **Idéia:** deixar os escritores fazerem uma nova cópia enquanto os leitores usam uma cópia antiga apropriada:

Segmento Principal (Versões Correntes de Objetos de BD)

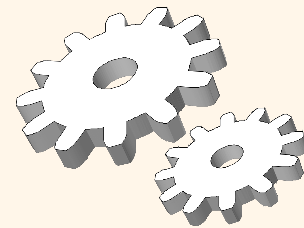




Versão Pool (Versões antigas que podem ser úteis para alguns leitores ativos)

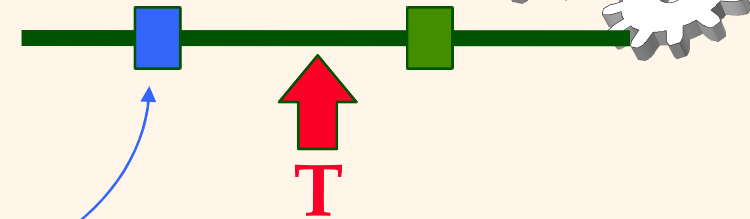
- Leitores são sempre permitidos para proceder.
  - Mas podem ser bloqueados até o escritor commit

CC Multiversão (Cont)

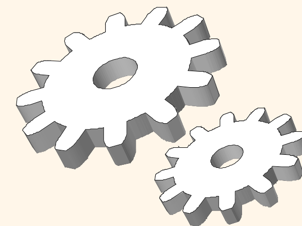


- Cada versão de um objeto tem seus escritores TS como seu WTS, e o TS da transação que são os mais recentemente lidos dessa versão como seu RTS.
- É proibido atrasos nas versões; nós podemos discartar versões que são "muitos antigas para serem interessantes".
- Cada transação é classificada como Reader or Writer.

Ação X Reader

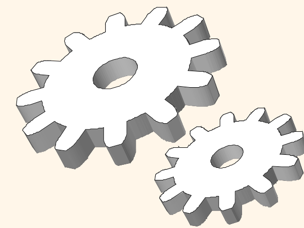


- Para cada objeto ser lido:
  - Encontra **versão mais nova** com WTS < TS (T) (inicia com a corrente versão no segmento principal através das versões mais novas).
- Assumindo que alguma versão de todo objeto existe no início do tempo, Ações X Reader nunca são reiniciadas.



Ação X Writer

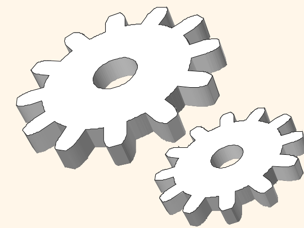
- Para ler um objeto, seguir o protocolo de leitura.
- Para escrever um objeto:
  - Encontrar a **versão mais nova de V**, cujo WTS < TS(T).
  - Se RTS(V) < TS(T), T faz uma cópia CV de V, com um ponteiro para V, com WTS(CV) = TS(T), RTS(CV) = TS(T) (escrita é buffered até T commit; outra ação X pode ver os valores de TS, mas não pode ler a versão CV).
  - Senão, rejeita a escrita.



Transaction Support in SQL-92

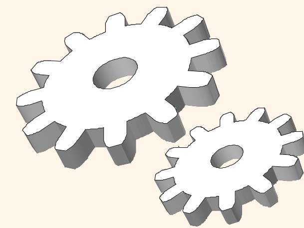
- Cada transação tem um modo de acesso, um tamanho de diagnóstico, e um nível de isolamento.

Nível de Isolamento	Leitura Suja	Leitura Sem Repetição	Problema Fantasma
Read Sem Commit	Talvez	Talvez	Talvez
Read Com Commit	Não	Talvez	Talvez
Reads Repetidos	Não	Não	Talvez
Serializável	Não	Não	Não



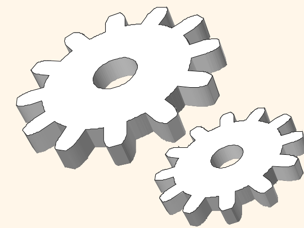
Sumário

- Há diversos esquemas de controle de concorrência baseados em bloqueio (2PL Strict, 2PL). Conflitos entre transações podem ser detectados no grafo de dependência
- O gerenciador de bloqueio mantém o caminho dos bloqueios. Deadlocks podem ser prevenidos ou detectados



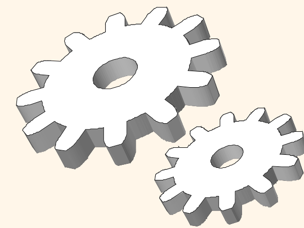
Sumário (Cont)

- Index locking é comum, e afeta a performance significativamente.
  - Necessário quando acessamos registros via índice.
  - Necessário para conjuntos lógicos locking de registros (index locking/predicate locking).
- Índices Árvores-estruturados:
  - Uso direto de 2PL é ineficiente.
  - Bayer-Schkolnick ilustra o potencial para melhora.
- Na prática, as melhores técnicas agora conhecidas; fazem nível-registro, mais do que o locking em nível-página.



Sumário (Cont)

- Locking de múltipla granularidade reduzem o overhead envolvido no conjunto de bloqueios para coleções aninhadas de objetos (e.g., um arquivo de páginas); não deve ser confundida com locking tree-index
- CC otimista objetiva minimizar CC overheads em um ambiente “otimista”, onde as leituras são comuns e as escritas são raras
- CC otimista tem seu próprio overhead, entretanto; a maioria dos sistemas reais usam locking



Sumário (Cont)

- CC Timestamp é uma outra alternativa para 2PL; permite alguns planos serializáveis que a 2PL não permite
- Garantir recuperabilidade com CC Timestamp requer habilidade para bloquear transações, ao qual é similar ao bloqueio
- CC Timestamp Multiversão é uma variante ao qual garante que transações somente-leitura nunca sejam reiniciadas; elas podem sempre ler uma versão mais antiga adequada