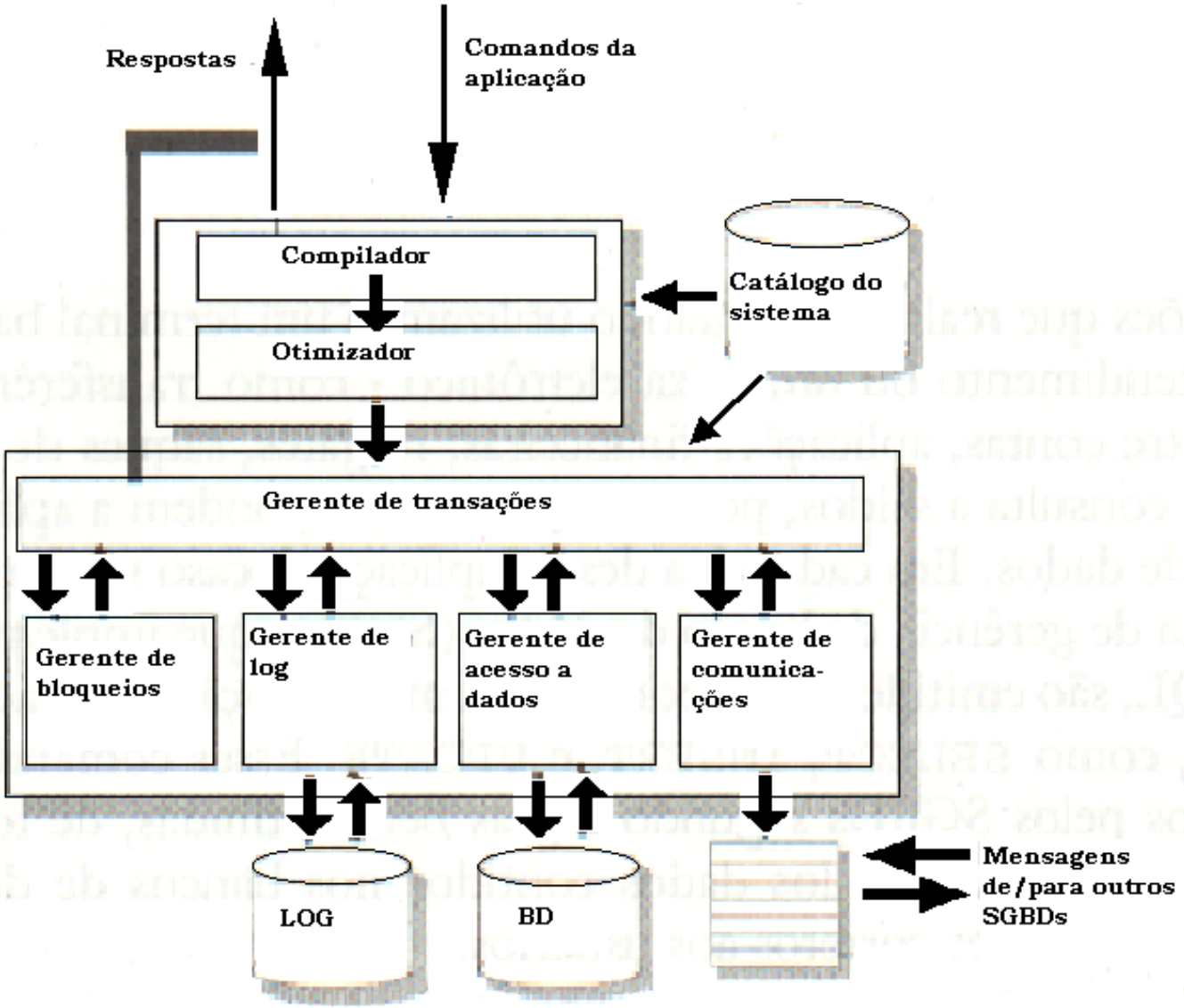
Transações

- Arquitetura dos SGBDs
- Gerente de Transações
- Conceito de Transação
- Propriedades ACID
- Definição de Transação no Oracle
- Estado da Transação
- Equilíbrio de Performance
- Escalas *(Schedules)* Concorrentes
- Serialização
- Recuperação
- Implementação de Isolação
- Fenômenos a serem evitados
- Níveis de Isolação da SQL-92

Arquitetura dos SGBDs



Gerente de transações

- Módulo ***gerente de transações*** é o que gerencia todo o processo de execução do plano otimizado elaborado previamente. É quem libera as ações para serem executadas na ordem definida pelo plano e ainda supervisiona o trabalho feito pelos demais componentes funcionais.

Conceito de Transação

- A ***transação*** é a unidade lógica de um programa em execução que lê e/ou atualiza linha(s) de tabela(s).
  - É uma inseparável lista de operações que devem ser executadas inteiramente e nunca parcialmente.
  - Transação deve terminar com o comando COMMIT (salvar tudo) ou ROLLBACK (desfazer tudo).
- No início, a transação encontra um banco de dados consistente.
- Durante a execução da transação os dados podem ficar temporariamente inconsistentes.
- Quando a transação termina, os dados voltam a ficar consistentes.
- Duas questões principais dificultam isso:
  - Falhas de vários tipos, tais como falhas de hardware e crashes de sistema
  - Execução concorrente de múltiplas transações

Propriedades ACID

Para preservar a integridade, o SGBD deve garantir:

- **Atomicidade.** Ou todas as operações da transação são devidamente salvas no banco de dados ou nenhuma é.
- **Consistência.** A execução da transação preserva a consistência do banco de dados. A transação leva os dados de um estado consistente para outro.
- **Isolação.** Embora múltiplas transações possam executar concorrentemente, cada transação deve estar desavisada das outras. Resultados intermediários da transação devem estar escondidos das outras executadas concorrentemente.
  - ★ Isto é, para todo par de transações T_i e T_j, parece para T_i que ou T_j, terminou a execução antes que T_i começou, ou T_j começou a execução depois que T_i terminou.
- **Durabilidade.** Depois que a transação completou com sucesso, as mudanças que ela fez no banco de dados persistem, mesmo que venham a ocorrer falhas posteriores de sistema.

Definição de transação no Oracle

- Uma nova transação começa implicitamente, ao início da sessão e ao final de cada transação.
- Comandos da Linguagem de Definição de Dados (DDL) possuem Commit implícito, ou seja, cada comando é uma transação.
- Comandos da Linguagem de Manipulação de Dados (DML) devem incluir uma forma para especificar o fim do conjunto de ações que compõem a transação.
- Termina-se a transação em DML com:
  - **Commit** salva a transação corrente, gravando todas as ações e começa uma nova.
  - **Rollback** aborta a transação corrente, desfazendo suas ações.
- Cuidado com aplicativos que tenham a opção *autocommit*, pois siginifica que cada comando DML será uma transação.
- Equilibre performance X segurança

Exemplo de Transferência de Fundo

- transação para transferir $50 da conta *A* para a conta *B*:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- Requisito de Consistência – a soma de *A* e *B* não é modificada pela execução da transação.
- Requisito de Atomicidade — se a transação falhar depois do passo 3 e antes do passo 6, o sistema deverá garantir que a atualização não é refletida no banco de dados, senão resultará uma inconsistência.

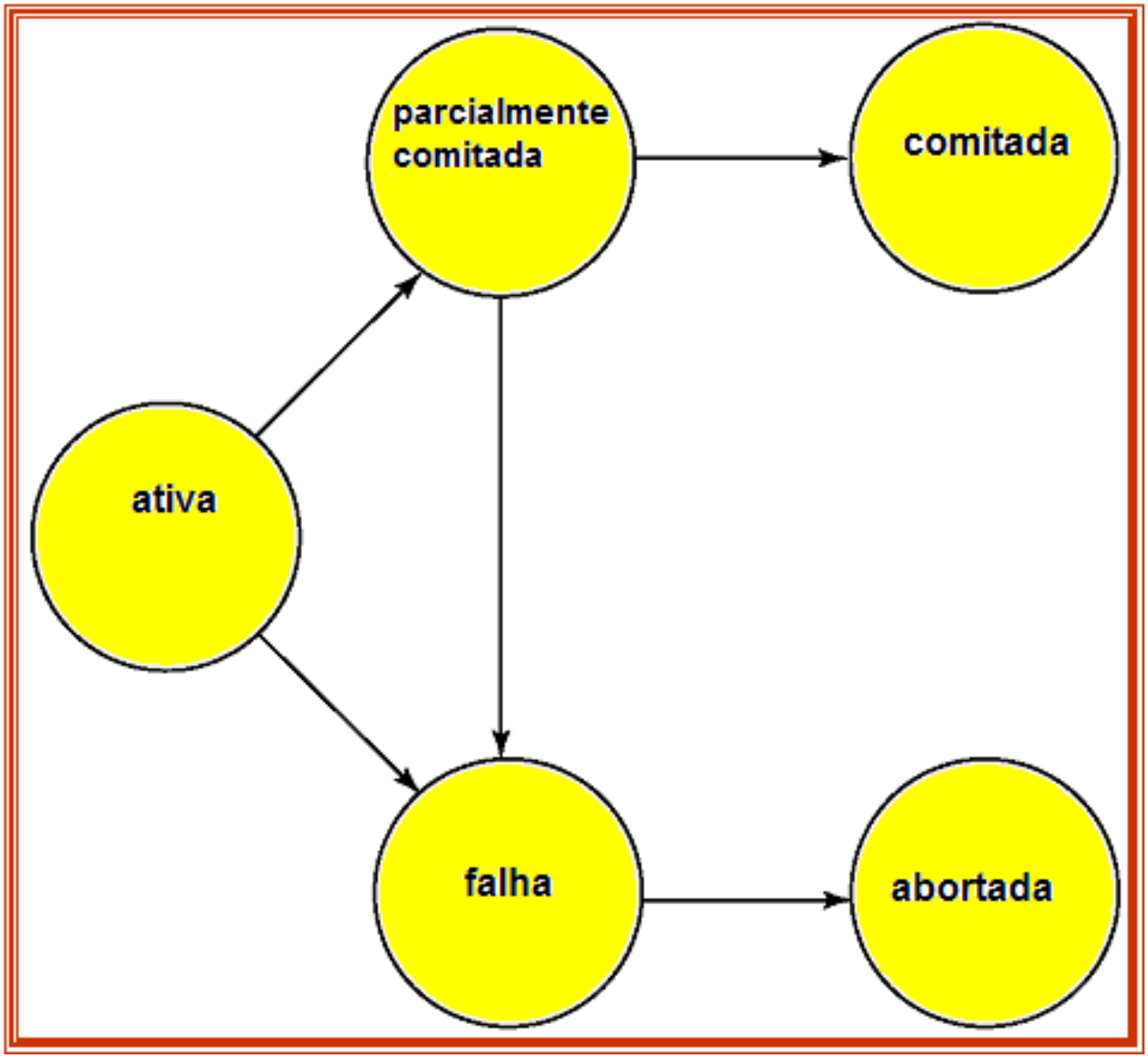
Exemplo de Transferência Fundo (Cont.)

- Requisito de Durabilidade — uma vez que o usuário tenha sido notificado que a transação completou (a transferência de $50 ocorreu), a atualização do banco de dados pela transação deve persistir a despeito de falhas.
- Requisito de Isolação — se entre os passos 3 e 6, outra transação conseguisse ler os dados parcialmente atualizados, ela veria valores inconsistentes (a soma *A* + *B* seria menos do que deveria ser). Isso pode ser garantido trivialmente rodando-se as transações ***serialmente***, isto é uma depois da outra. Contudo, executar múltiplas transações concorrentemente tem significantes benefícios, conforme veremos.

Estado da Transação

- **Ativa,** estado inicial; a transação fica nesse estado enquanto é executada
- **Parcialmente comitada,** depois que o comando final foi executado.
- **Falha,** depois de descoberto que a execução normal não pode mais prosseguir.
- **Abortada,** depois de ser feito *roll back* e o banco de dados ser restaurado ao seu estado antes do início da transação.
- **Comitada,** depois de completada com *sucesso*.

Estado da Transação (Cont.)



Equilíbrio de Performance

- Múltiplas transações podem rodar concorrentemente (em paralelo) no sistema. As vantagens são:
  - **Aumentar utilização de processador e disco**, levando a um melhor ***throughput*** (volume de trabalho por tempo), uma transação pode estar usando a CPU enquanto outra está lendo ou gravando em disco
  - **Reduzir tempo de resposta médio**, transações pequenas não precisam esperar por outras longas.
  - **Melhorar performance** → ***(throughput + response time)***
- *Esquemas de controle de concorrência* – mecanismos para conseguir isolação, isto é, controlar a interação entre transações concorrentes de modo a prevenir que destruam a consistência do banco de dados

Escalas (*Schedules*)

- *Escalas* – são seqüências que indicam a ordem cronológica em que instruções de transações concorrentes são executadas
  - Uma escala para um grupo de transações deve conter todas as instruções dessas transações
  - deve preservar a ordem em que as instruções aparecem em cada transação individual
  - Serial (ou seqüencial), uma transação após a outra
  - Intercalado (ou concorrente), as operações das transações são executadas intercaladamente.

Exemplo de Escala

Seja T_1 transferir \$50 de *A* para *B*, e T_2 transferir 10% do balanço de *A* para *B*. A seguir uma escala serial, em que T_1 é seguido por T_2.

T_1	T_2
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)

Exemplo de Escala(Cont.)

O Schedule 2 seguinte não é um serial, mas é *equivalente* ao Schedule 1.

T_1	T_2
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)

Em ambos Schedules, a soma A + B é preservada.

Exemplo de Escala (Cont.)

O schedule a seguir não preserva o valor da soma $A + B$.

T_1	T_2
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)

Serialização

- Consideração básica – cada transação preserva a consistência do banco de dados.
- A execução serial preserva a consistência.
- Uma escala concorrente é serializável se é equivalente a uma escala seqüencial. Há diferentes formas de equivalência:
  1. **Serialização sem conflito**
  2. **Visão serializada**
- Embora transações possam fazer cálculos arbitrários nos dados de *buffers* locais, nós assumimos escalas simplificadas contendo somente instruções **read** e **write**.

Serialização sem Conflito

- Instruções I_i e I_j de transações T_i e T_j respectivamente, **conflitam** se e somente se existir algum item Q acessado por ambas I_i e I_j, e ao menos uma dessas instruções escrever Q.

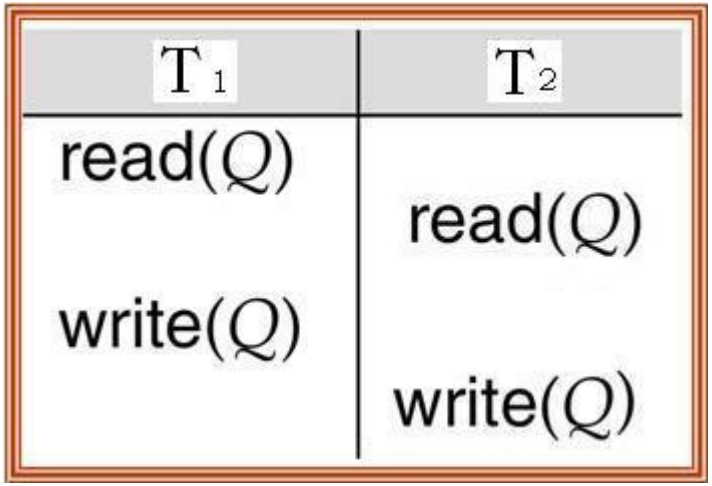
  1. I_i = **read**(Q), I_j = **read**(Q). I_i e I_j não conflitam.
  2. I_i = **read**(Q), I_j = **write**(Q). Elas conflitam.
  3. I_i = **write**(Q), I_j = **read**(Q). Elas conflitam
  4. I_i = **write**(Q), I_j = **write**(Q). Elas conflitam

  5. I_i = **read/write**(Q), I_j = **read/write**(Z). I_i e I_j não conflitam.

- Intuitivamente, um conflito entre I_i e I_j força uma ordem temporal (lógica) entre elas. Se I_i e I_j são consecutivas numa escala e não conflitam, seus resultados permaneceriam os mesmos se fossem intercambiadas (trocadas de ordem).

Serialização sem Conflito(Cont.)

- Se uma escala *S* pode ser transformada numa escala *S´* por uma série de trocas (*swaps)* de instruções não-conflitantes, dizemos que *S* e *S´* são **equivalentes por falta de conflito**.
- Dizemos que uma escala *S* é **serializável sem conflito** se for equivalente sem conflito a uma escala seqüencial (*serial schedule*)
- Exemplo de uma escala que não é serializável sem conflito:

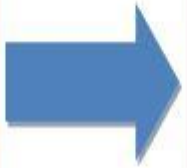


Não podemos permutar instruções no schedule acima para obtermos uma escala seqüencial < T_1, T_2 >, ou < T_2, T_1 >.

Serialização sem Conflito(Cont.)

- A escala abaixo pode ser transformada em outra seqüencial, em que ocorra toda T_1 e depois toda T_2. Apenas fazendo-se uma série de trocas de instruções não-conflitantes. Por isso a escala é serializável sem conflito.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)



T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Visão Serializada

- Sejam *S* e *S´* 2 escalas com o mesmo grupo de transações. *S* e *S´* são **equivalentes na visão** se atendidas as 3 condições:
  1. Para cada ítem de dados Q, se transação T_i lê o valor inicial de Q na escala S, então transação T_i deve também, ler na escala *S´* o valor inicial de Q.
  2. Para cada ítem de dados Q se transação T_i executa **read**(Q) na escala S, e esse valor foi produzido pela transação T_j (se alguma), então transação T_i deve também na escala *S´* ler o valor de Q que foi produzido pela transação T_j .
  3. Para cada ítem de dados *Q*, a transação (se alguma) que faz o **write**(*Q*) final na sescala *S* deve fazer o **write**(*Q*) final na escala *S´*.

  Como visto, a equivalência na visão é também baseada apenas em **reads** e **writes**.

Visão Serializada (Cont.)

- Toda escala serializável sem conflito é também visão serializável.
- Mas, há escalas visão-serializáveis que não são serializáveis sem conflito.
- Exemplo, a escala abaixo é visão-serializável mas não serializável sem conflito.

T_1	T_2	T_3
read(Q)		
	write(Q)	
write(Q)		
		write(U)
	read(T)	
write(R)		
		write(Q)

- Todo schedule visão serializável que não é serializável sem conflito possui escrita cega (***blind write***).

Fenômenos a serem evitados

- **Leitura suja (Dirty read)**, uma transação lê dados que foram escritos por outra que ainda não comitou.
- **Leitura não repetível (Nonrepeatable read)**, uma transação re-lê dados previamente lidos e descobre que outra transação comitada modificou ou deletou esses dados.
- **Leitura fantasma (Phantom read)**, uma transação re-executa uma consulta retornando um conjunto de linhas que satisfazem um critério de seleção e descobre que outra transação comitada inseriu linhas adicionais que satisfazem a condição.

Níveis de Isolamento da SQL-92

- **Leitura não comitada (*Read uncommitted*)** — mesmo registros não comitados podem ser lidos, usuário by-passa os mecanismos de *lock.*
- **Leitura comitada (*Read committed*)** — (default Oracle), cada *query* executada por uma transação vê somente dados que foram comitados antes da consulta (não da transação) começar. Somente registros comitados podem ser lidos, mas sucessivas leituras de um registro podem retornar diferentes (mas comitados) valores.
- **Leitura repetível (*Repeatable read*)** — somente registros comitados podem ser lidos, leitura repetida do mesmo registro deve retornar o mesmo valor. Contudo, a transação pode não ser serializável – pode achar alguns registros inseridos por uma outra transação.
- **Serializável (*Serializable*)** — (default ANSI), transações vêem somente as alterações que foram comitadas no momento que a transação começou, mais as alterações feitas por ela mesma com comandos INSERT, UPDATE e DELETE.

Obs. níveis mais baixos de isolação podem ser úteis para coletar informação aproximada sobre o banco de dados, por exemplo, estatísticas para otimização.

Nível de Isolamento x Problema

Problema / Nível de Isolamento	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possível	Possível	Possível
Read committed	Impossível	Possível	Possível
Repeatable read	Impossível	Impossível	Possível
Serializable	Impossível	Impossível	Impossível

Níveis de Isolamento no Oracle

- **Transação** — definido no início de cada nova transação
  - SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
  - SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  - SET TRANSACTION READ ONLY;
- **Sessão** — altera todas as subsequentes transações.
  - ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE;
  - ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED;