

# Banco de Dados

## Pós-Graduação em Ciência da Computação

**Prof. Dr. Ronaldo Celso Messias Correia**  
[ronaldo.correia@unesp.br](mailto:ronaldo.correia@unesp.br)



UNIVERSIDADE ESTADUAL PAULISTA  
“JÚLIO DE MESQUITA FILHO”



# Stored Procedure e Triggers MySQL

# Stored Procedure

- São funções escritas usando SQL e ficam armazenadas no servidor
- Podem receber parâmetros e executar funções complexas, retornando ou não as informações para o usuário
- Vantagens:
  - Minimizam o tráfego da rede
  - São mais rápidos, aproveitam a capacidade do servidor e da otimização do SGBD
  - Facilitam a manutenção. Pode ser acessada por diversos programas, se houver alguma modificação a ser feita, basta fazê-la no BD
  - A resposta curta é, sempre que você puder. Não existem desvantagens em se usar stored procedures.
- Limitações:
  - Passar qualquer informação variável para a stored procedure como parâmetros ou colocá-las em uma tabela que a stored procedure possa acessar.
  - A linguagem de escrita de stored procedures e triggers pode ser muito limitada para operações mais complexas.

# Stored Procedures - Sintaxe

```
CREATE PROCEDURE sp_name ([parameter[,...]])  
[characteristic ...]  
BEGIN  
<corpo da rotina>  
END  
  
[parameter:  
 [ IN | OUT | INOUT ] nome_parametro tipo  
tipo:  
 Qualquer tipo de dados válidos no MySQL
```

characteristic:

```
LANGUAGE SQL  
[NOT] DETERMINISTIC  
SQL SECURITY {DEFINER | INVOKER}  
COMMENT string
```

Corpo:  
Declarações de procedimento em SQL válida

- <nome> - é o nome da store procedure
- Param1, Param2 – parâmetros
  - VendaDatas(Data1 Date, Data2 Date)
- Cada parâmetro é um parâmetro IN por default
  - IN – Parâmetro de entrada
  - OUT – Parâmetro de saída (será assumido NULL como valor de entrada)
  - INOUT – Entrada e Saída

Select \* from INFORMATION\_SCHEMA.ROUTINES

# Stored Procedure - Exemplo 1

```
DELIMITER $$  
DROP PROCEDURE IF EXISTS simpleproc $$  
CREATE PROCEDURE simpleproc()  
BEGIN  
    SELECT 'OLA';  
END $$  
  
DELIMITER ;
```

- Para executar o procedimento
- CALL simpleproc()

O exemplo usa o comando delimiter para alterar o delimitador de instrução para antes da definição do procedure. Isto permite que o delimitador ';' (padrão) usado no corpo de procedure seja passado para o servidor em vez de ser interpretado pelo MySQL

# Stored Procedure - Exemplo 2

```
DELIMITER //  
CREATE PROCEDURE inserecliente(v_nome VARCHAR(60), v_endereco VARCHAR(20))  
BEGIN  
    IF ((v_nome != "") AND (v_endereco != "")) THEN  
        INSERT INTO cliente (nome, endereco) VALUES (v_nome, v_endereco);  
    ELSE  
        SELECT 'NOME e ENDEREÇO devem ser fornecidos para o cadastro!' AS Msg;  
    END IF;  
END;
```

Para executar o procedimento

- CALL inserecliente ("jose da silva", "Rua das flores");

```
CREATE TABLE cliente (  
    id int auto_increment primary  
    key,  
    nome varchar(60) not null,  
    endereco varchar(40) not null,  
    genero varchar(1)  
)
```

# Stored Procedure - Exemplo 3

```
DELIMITER //
```

```
CREATE PROCEDURE updatecliente(v_id int, v_nome VARCHAR(60), v_endereco VARCHAR(20))
BEGIN
    IF ((v_id > 0) AND (v_id != "") AND (v_nome != null) AND (v_endereco != ""))
    THEN
        UPDATE cliente SET nome = v_nome, endereco=v_endereco WHERE id = v_id;
    ELSE
        SELECT 'NOME e ENDEREÇO devem ser fornecidos para o cadastro!' AS Msg;
    END IF;
END;
```

Para executar o procedimento

- CALL updatecliente (10,'jose da silva xavier', 'Rua do Amor Perfeito');

# Stored Procedure - Exemplo 4

```
DELIMITER //
CREATE PROCEDURE deletecliente(v_id int)
BEGIN
    IF ((v_id > 0) AND (v_id !='')) THEN
        DELETE FROM cliente WHERE id = v_id;
    ELSE
        SELECT 'ID não informado' AS Msg;
    END IF;
END;
```

Para executar o procedimento

- CALL deletecliente (10);

# Stored Procedure - Exemplo 5

## Exemplo de utilização de parâmetro tipo IN

```
CREATE PROCEDURE sp_in (p VARCHAR(10))
    SET @X = P;
```

## Exemplo de utilização de parâmetro tipo OUT

```
CREATE PROCEDURE sp_out (OUT p VARCHAR(10))
    SET P = 'ola';
```

## Exemplo de utilização de parâmetro tipo INOUT

```
CREATE PROCEDURE sp_inout (INOUT p int)
BEGIN
    SET @X = P * 2;
    SET P = @X;
END;
```

# Atividade Prática - CRUD Tabela Cliente

```
CREATE TABLE cliente (
    cod_cliente INTEGER(3) PRIMARY KEY NOT NULL,
    nome_cliente VARCHAR (40) NOT NULL,
    endereco_cliente VARCHAR(40) NOT NULL,
    cidade_cliente VARCHAR(40) NOT NULL,
    cidade_cep VARCHAR(30) NOT NULL,
    uf_cliente VARCHAR(2) NOT NULL,
    cgc_cliente VARCHAR(30) NOT NULL,
    ie_cliente VARCHAR(4) NULL
);
```

[script banco de dados pedido - Documentos Google](#)

# Atividade Prática - CRUD Tabela Cliente

```
CREATE PROCEDURE inserecliente(.....)
```

```
CREATE PROCEDURE updatecliente(v_id int, .....)
```

```
CREATE PROCEDURE deletecliente(v_id int)
```

# Atividade Prática - CRUD Tabela Vendedor

```
CREATE TABLE vendedor (
    cod_vendedor INTEGER(3) PRIMARY KEY NOT NULL,
    nome_vendedor VARCHAR(40) NOT NULL,
    salario_vendedor INTEGER NOT NULL,
    comissao_vendedor CHAR NOT NULL
);
```

[script banco de dados pedido - Documentos Google](#)

# Atividade Prática - CRUD Tabela Vendedor

```
CREATE PROCEDURE inserevendedor(.....)
```

```
CREATE PROCEDURE updatevendedor(v_id int, .....)
```

```
CREATE PROCEDURE deletevendedor(v_id int)
```

# Atividade Prática - CRUD Tabela Produto

```
CREATE TABLE produto (
    cod_produto INTEGER(3) PRIMARY KEY NOT NULL,
    unidade_produto VARCHAR(3) NOT NULL,
    desc_produto VARCHAR(40) NOT NULL,
    valor_unitario INTEGER(8) NOT NULL
);
```

[script banco de dados pedido - Documentos Google](#)

# Atividade Prática - CRUD Tabela Produto

```
CREATE PROCEDURE insereproduto(.....)
```

```
CREATE PROCEDURE updateproduto(v_id int, .....)
```

```
CREATE PROCEDURE deleteproduto(v_id int)
```

# Variáveis de Usuário

- O MySQL suporta variáveis específicas da conexão com a sintaxe @nomevariável
- As variáveis não precisam ser inicializadas.
- Elas contém NULL por padrão e podem armazenar um valor inteiro, real ou uma string.
- Todas as variáveis de uma thread são automaticamente liberadas quando uma thread termina.
- Variáveis de usuários devem ser utilizadas em expressões onde são permitidas.
- O tipo padrão de uma variável é baseada no tipo da variável no início da instrução. (Assume-se que uma variável não atribuída possui o valor NULL e é do tipo STRING).

```
SET @variável= { expressão inteira | expressão real | expressão string }  
[,@variável= ...]
```

# Variáveis de Usuário

- Para atribuir um valor a uma variável em outras instruções diferentes de SET utilizar o operador de atribuição := em vez de =, porque = é reservado para comparações em instruções diferentes de SET:

```
SET @t1=0, @t2=0, @t3=0;  
mysql> SELECT @t1:=(@t2:=1)+@t3:=4,@t1,@t2,@t3;
```

@t1:=(@t2:=1)+@t3:=4	@t1	@t2	@t3
5	5	1	4

```
mysql> SET @X='oi';  
mysql> SELECT @X
```

# Variáveis Locais

- Uma variável local somente será válida durante a execução de um procedimento armazenado, seja ele uma Stored Procedure, uma Trigger ou Stored Function, sendo que, após o término da execução de tais procedimentos, esta variável é destruída da memória, juntamente com seu respectivo valor.
- Para ser declarada, precisa estar entre os chamados compound statements, ou comandos aninhados dentro de um procedimento qualquer, que por sua vez também são chamados de Stored Routines.
- Para se declarar uma variável local, é necessário estarmos posicionados entre BEGIN ... END

```
DELIMITER //
CREATE PROCEDURE SP_TEST(IN num INT)
BEGIN
    DECLARE x INT DEFAULT 0;
    SET x = num;
END;
```

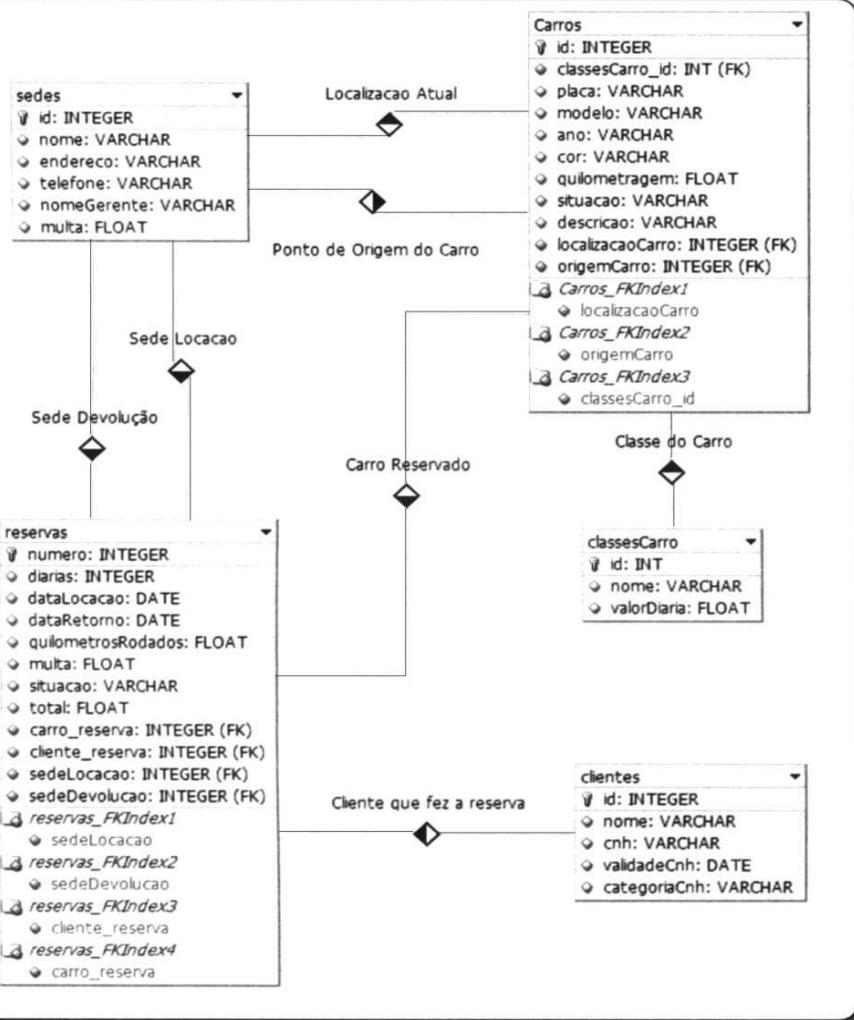
[MySQL :: Manual de Referência do MySQL 8.4 :: 15.6.4.2 Escopo e Resolução de Variáveis Locais](#)

# Variáveis Locais

- Variáveis locais não são case sensitives, ou seja, VAR, var, VaR e vAr são a mesma coisa. O que devemos nos atentar é quanto ao seu escopo.
- O escopo das variáveis locais no MySQL é definido pela sua posição, aninhada em blocos de um procedimento armazenado no servidor de bancos de dados MySQL.

```
CREATE PROCEDURE SP_ESCOPO()
BEGIN
    DECLARE x INT DEFAULT 0;
    BEGIN
        DECLARE x INT DEFAULT 0;
        BEGIN
            DECLARE x INT;
            END;
            SET x = NULL;
        END;
        SELECT x AS Var;
    END;
```

# I de Carros



# Stored Procedure - Exemplo 6

## Sistema para aluguel de carros - LOCAÇÃO

DELIMITER //

```
CREATE PROCEDURE registrarlocacao(v_diarias int, v_datalocacao DATE, v_carro int, v_sede int)
```

```
BEGIN
```

```
// Inserindo registro de nova reserva
```

```
INSERT INTO reservas (diarias, datalocacao, carro_reserva, sedelocacao) VALUES (v_diarias, v_datalocacao,  
v_carro, v_sede)
```

```
// Atualizando a situação do carro
```

```
UPDATE carros SET situacao = "alugado" WHERE id = v_carro
```

```
END;
```

# Stored Procedure - Exemplo 7

## Sistema para aluguel de carros - DEVOLUÇÃO

```
DELIMITER //
```

```
CREATE PROCEDURE registrardevolucao(v_reserva int, v_dataretorno DATE, v_quilometrosrodados float(8,2), v_multa float(10,2), v_total float(10,2), v_sedelocacao int, v_sededevolucao int)
```

```
BEGIN
```

```
// Atualizando registro de nova reserva
```

```
UPDATE reservas SET dataretorno = v_dataretorno, quilometrosrodados = v_quilometrosrodados, multa = v_multa, total = v_total,  
sededevolucao = v_sededevolucao WHERE id = v_reserva;
```

```
// Atualizando a situação do carro
```

```
IF (v_sedelocacao != v_sededevolucao) THEN
```

```
    UPDATE carros SET situacao = "fora do ponto de origem" WHERE id = v_carro;
```

```
ELSE
```

```
    UPDATE carros SET situacao = "disponivel" WHERE id = v_carro;
```

```
END IF
```

```
END;
```

# Desvios Condicionais e Iterações

## ➤ IF THEN ELSE

```
create procedure sp_lista_clientes(in opcao integer)
begin
    if opcao = 0 then
        select * from cliente where genero= 'F';
    else
        if opcao = 1 then
            select * from cliente where genero = 'M';
        else
            select * from cliente;
        end if;
    end if;
end $$
```

IF condition THEN  
statement;/s  
ELSE statement/s;  
ENDIF

# Desvios Condicionais e Iterações

## ➤ CASE

```
create procedure sp_lista_clientes(in opcao integer)
begin
    CASE opcao
        WHEN 0 THEN select * from cliente where sexo = 'F';
        WHEN 1 THEN select * from cliente where sexo =
'M';
        else
            select * from cliente;
    END CASE;
END
```

CASE variable  
WHEN condição1 statement/s;  
WHEN condição2 statement/s;  
ELSE  
 statement/s  
END CASE

# Desvios Condicionais e Iterações

## ➤ REPEAT UNTIL

```
create procedure sp_repeat(in var1 integer)
begin
    REPEAT
        SELECT var1;
        set var1 = var1 + 1;
        UNTIL var1 > 5
    END REPEAT;
END;
```

REPEAT  
statement/s;  
UNTIL condição  
END REPEAT

# Desvios Condicionais e Iterações

## ➤ WHILE

```
create procedure sp_while(in var1 integer)
begin
    WHILE (var1 < 20) DO
        SELECT var1;
        set var1 = var1 + 1;
    END WHILE;
END;
```

WHILE condição DO  
 statement/s;  
END WHILE

# Tratamento de Erros

```
CREATE TABLE cliente (
```

```
    id int primary key,
```

```
    nome varchar(60) not null,
```

```
    cpf varchar(11) not null unique,
```

```
    email varchar(50) not null unique,
```

```
    genero varchar(1)
```

```
);
```

```
insert into cliente(id, nome, cpf, email) values
```

```
(2, 'maria ', '123456', 'maria@gmail.com');
```

# Manipuladores e Cursores

- Com o advento das stored procedures, certas condições podem exigir tratamento específico. Estas condições podem ser relacionadas a erros, bem como controle de fluxo geral dentro da rotina.
- Manipuladores permitem executar declarações caso certa condição esteja presente
- Cursores permitem iterar através de um resultset, processando-o linha a linha.
- Cursores: O termo é um acrônimo para CURrent Set Of Records (conjunto de registros corrente)
  - São utilizados para posicionar um ponteiro em uma linha específica e podem permitir atualizações para as linhas com base na posição atual (O MySQL não suporta)

# Tratamento de Erros - Condições (Condition)

[MySQL :: Manual de Referência do MySQL 4.1 :: 13.1 Erros Retornados \(must.na\)](#)

- O tratamento de erros dentro de procedimentos armazenados no MySQL é baseado em **condições** .
- Caso uma determinada condição for atendida, um erro que podemos personalizar, baseado nos tipos de condições existentes, será disparado.
  - Erros: violação de uma chave primária ou índice único, violação de integridade referencial ou mesmo um WARNING em meio ao processamento do procedimento.

```
DECLARE condition_name CONDITION FOR condition_value
```

```
condition_value:
```

```
SQLSTATE [VALUE] sqlstate_value | mysql_error_code
```

- Permite associar um nome simbólico a uma condição de erro (MySQL error code ou SQLSTATE).
- Facilita a compreensão e reutilização em múltiplos handlers.
- Deve ser declarada antes dos cursores e handlers.

## ➤ SQLSTATE: padrão SQL ANSI/ISO

- String de 5 caracteres (ex: '23000', '42S02')
- Definido pelo padrão SQL (ANSI/ISO)
- Portável entre diferentes bancos de dados
- Representa categorias de erro
  - SQLSTATE '23000' -- Violação de restrição (ex: chave primária duplicada)
  - SQLSTATE '42S02' -- Tabela não existe

## ➤ error\_code: código específico do MySQL

- Número inteiro (ex: 1062, 1051)
- Definido exclusivamente pelo MySQL
- Usado nas mensagens de erro exibidas pelo MySQL
- Não é portável (outros SGBDs não usam os mesmos códigos).
- Pode ser usado diretamente em DECLARE HANDLER, mas não em SIGNAL.

# Tratamento de Erros

- Usando `error_code`
  - `DECLARE duplicate_entry CONDITION FOR 1062;`
- Usando `SQLSTATE`
  - `DECLARE duplicate_entry CONDITION FOR SQLSTATE '23000';`
- Um `error_code` MySQL geralmente tem um `SQLSTATE` correspondente
  - `1062 (error_code) → '23000' (SQLSTATE)`
  - Mensagem: Entrada '%s' duplicada para a chave %d
- Ambos funcionam, mas o `SQLSTATE` é mais portável

[MySQL :: MySQL 8.0 Reference Manual :: 15.6.7.1 DECLARE ... CONDITION Statement](#)

# Tratamento de Erros - Manipuladores

- Permite capturar erros, warnings ou condições específicas
- Rotina executada automaticamente para tratar condições específicas (como erros, avisos ou eventos) que ocorrem durante a execução de procedimentos armazenados, funções ou triggers, permitindo um controle personalizado sobre como o sistema deve responder a essas situações.

```
DECLARE handler_type HANDLER FOR condition_value[,...] sp_statement
```

[MySQL :: MySQL 8.0 Reference Manual :: 13.6.7.2 DECLARE ... HANDLER Statement](#)

# Tratamento de Erros - Manipuladores

- Tipos de Ação: indica o que o MySQL deve fazer depois de executar o handler
  - CONTINUE: permite ao processo continuar depois que as ações do manipulador (handler) foram executados
  - EXIT: encerram imediatamente o bloco atual BEGIN/END

Tipo	Exemplo	Significado
Código de erro	1051, 1062	Erros MySQL específicos.
SQLSTATE	'23000', '42S02'	Valores padrão ANSI SQL.
Nome da condição	no_such_table	Declarado com <code>DECLARE CONDITION</code> .
SQLWARNING		Qualquer aviso (SQLSTATE inicia com '01').
NOT FOUND		SQLSTATE '02000' (cursor sem dados).
SQLEXCEPTION		Qualquer erro (SQLSTATE ≠ '00', '01', '02').

# Tratamento de Erros - Manipuladores

- Tipos de Ação: indica o que o MySQL deve fazer depois de executar o handler
  - CONTINUE: permite ao processo continuar depois que as ações do manipulador (handler) foram executados
  - EXIT: encerram imediatamente o bloco atual BEGIN/END

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000'  
BEGIN  
    -- tratamento de erro  
END;
```

---

```
DECLARE chave_duplicada CONDITION FOR SQLSTATE '23000';  
DECLARE CONTINUE HANDLER FOR chave_duplicada  
BEGIN  
    -- Código de tratamento  
END;
```

# Tratamento de Erros - Exemplo 1

```
CREATE PROCEDURE manipula1()
BEGIN
    DECLARE coluna_desconhecida CONDITION FOR SQLSTATE '42S22';
    DECLARE EXIT HANDLER FOR coluna_desconhecida
    BEGIN
        SELECT 'erro de coluna desconhecida';
    END;
    SELECT coluna;
    SELECT 'continua';
END;
```

- É declarada uma condição chamada coluna\_desconhecida, que irá surgir quando for atingido o SQLSTATE 42S22, que ocorre quando uma coluna é desconhecida
- O manipulador EXIT exibe a mensagem de erro.
- No corpo da procedure a declaração SELECT coluna (para ativar o código de erro) e SELECT ' continua', que nunca será executada, pois a procedure será encerrada assim que a condição estiver presente

# Tratamento de Erros - Exemplo 1

```
CREATE TABLE cliente (
    id int auto_increment primary key,
    nome varchar(60) not null,
    endereco varchar(40) not null,
    cpf varchar(11) not null unique,
    email varchar(50) not null unique,
    genero varchar(1)
);
```

# Tratamento de Erros - Exemplo 1

```
DELIMITER $$  
CREATE PROCEDURE inserecliente(v_nome VARCHAR(60), v_endereco VARCHAR(20),  
v_cpf VARCHAR(11), v_email VARCHAR(50))  
BEGIN  
    DECLARE EXIT HANDLER FOR SQLSTATE '23000'  
    BEGIN  
        SELECT 'Violação de chave duplicada!' AS Msg;  
    END;  
    IF ((v_nome != "") AND (v_endereco != "")) THEN  
        INSERT INTO cliente (nome, endereco, cpf,email) VALUES (v_nome, v_endereco, v_cpf,  
v_email);  
    ELSE  
        SELECT 'NOME e ENDEREÇO devem ser fornecidos para o cadastro!' AS Msg;  
    END IF;  
END $$
```

# Tratamento de Erros - Exemplo 2

```
CREATE PROCEDURE violacao_chave(IN v_id INT)
BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE '23000'
    BEGIN
        SELECT 'Violação de chave primária!' AS Msg;
    END;
    INSERT INTO cliente SET id =v_id;
END;
```

- O SQLSTATE aparecerá sempre que um erro for enviado a um usuário quando este executa uma operação ilegal ou que viole a integridade dos dados.
- Quando tentar inserir uma informação duplicada em uma coluna que é chave primária de uma tabela, o erro 1062 com o SQLSTATE 23000 é retornado

# Tratamento de Erros - Exemplo 3

```
CREATE PROCEDURE test.sp_2(IN num CHAR(1))
BEGIN
    DECLARE EXIT HANDLER FOR SQLWARNING
    BEGIN
        SELECT 'O dado foi truncado!' AS Msg;
    END;
    INSERT INTO test.tbl_1 SET id =num;
END;
```

- Ao enviar um dado do tipo CHAR para ser inserido em uma coluna do tipo INT, causará um WARNING e uma mensagem de truncamento dos dados, condição declarada no HANDLER com a condição EXIT que encerrará o procedure.

# Cursos em MySQL

- O MySQL não suporta todas as recursos dos cursores
- Os Cursos no MySQL são não sensitivos (não devemos atualizar uma tabela enquanto estamos usando um cursor); são somente leitura (não podemos fazer atualizações usando a posição do cursor); e não rolantes (só podemos avançar para o próximo registro e não para traz e/ou para frente)
- São usados para acessar um resultset que possa recuperar uma ou mais linhas. São usados para posicionar um ponteiro em uma linha específica

```
DECLARE cursor_name CURSOR FOR sql_statement
```

- Vários cursores podem ser declarados num mesmo procedure

# Cursos em MySQL - Exemplo 1

- OPEN sp\_cursos1: ativa o cursor previamente declarado
- FETCH: retorna a próxima linha do resultset atual. Os resultados devem ser armazenados em algum lugar
  - As variáveis x e y armazenam as duas colunas retornadas pelo SELECT id, nome FROM cliente que compõe o cursor
- CLOSE sp\_cursos1: fecha o cursor
- No exemplo acima apenas a primeira linha do resultset é retornada

```
CREATE PROCEDURE exemplo_cursor1 (OUT rid INT, OUT rnome INT)
BEGIN
    DECLARE x,y INT;
    DECLARE sp_cursor1 CURSOR
        FOR SELECT id, nome FROM cliente;
    OPEN sp_cursor1;
        FETCH sp_cursor1 INTO x, y;
    CLOSE sp_cursor1;
    SET rid = x;
    SET rnome = y;
END $
```

```
CREATE TABLE cliente1209 (
    id int auto_increment primary key,
    nome varchar(60) not null,
    endereco varchar(40) not null,
    cpf varchar(11) not null unique,
    email varchar(50) not null unique,
    genero varchar(1)
);
```

# Cursores em MySQL - Exemplo 2

- clientes (id\_cliente, nome)
- pedidos (id\_pedido, id\_cliente, data\_pedido)
- itens\_pedido (id\_item, id\_pedido, produto, quantidade, preco\_unitario)
  
- Para cada cliente que teve pedidos no último mês, calcular o total gasto e armazenar isso em uma nova tabela chamada relatorio\_gastos.

# Cursos em MySQL - Exemplo 2

```
CREATE PROCEDURE exemplo_cursor2 (OUT rid INT, OUT rnome
VARCHAR(60))
BEGIN
    DECLARE x, z INT;
    DECLARE y VARCHAR(60);
    DECLARE sp1_cursor CURSOR
        FOR SELECT id, nome FROM cliente;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET z = 1;
    OPEN sp1_cursor;
    REPEAT
        FETCH sp1_cursor INTO x, y;
        UNTIL (z=1)
    END REPEAT;
    CLOSE sp1_cursor;
    SET rid = x;
    SET rnome = y;
END
```

# Cursos em MySQL - Exemplo 2 - Funcionando

```
CREATE PROCEDURE exemplo_cursor1 (OUT rid INT, OUT rnome VARCHAR(50))
BEGIN
    DECLARE x INT;
    DECLARE y VARCHAR(50);
    DECLARE FIM INT DEFAULT 0;
    DECLARE sp_cursor1 CURSOR
        FOR SELECT cod_cliente, nome_cliente FROM cliente;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET FIM = 1;
    OPEN sp_cursor1;
    REPEAT
        FETCH sp_cursor1 INTO x, y;
        select x,y;
    UNTIL (FIM = 1)
    END REPEAT;
    CLOSE sp_cursor1;
    SET rid = x;
    SET rnome = y;
END $$
```

# Cursos em MySQL - Exemplo 2

- Para iterar pelo resultset inteiro e retornar os resultados utilizar um REPEAT UNTIL
- No exemplo 2 um manipulador é declarado com a condição NOT FOUND e atribui 1 à variável Z, sendo justamente z=1 a condição testada pelo REPEAT UNTIL
- A condição NOT FOUND inclui todos os erros com SQLSTATE que começam com 02, um dos quais é o erro NO DATA TO FETCH
- Outros tipos de condições também podem ser abordadas na criação de um HANDLER ou CONDITION:
  - Declarar explicitamente um código de SQLSTATE para tratar o erro;
  - Declarar o SQLWARNING e todos os SQLSTATES iniciados com 01 serão tratados;
  - Declarar NOT FOUND, mais comum em Cursors e Stored Functions, para tratamento de SQLSTATES iniciados com 02;
  - Declarar o SQLEXCEPTION que tratará erros de SQLWARNING ou NOT FOUND.

# Cursos em MySQL - Exemplo 3

```
CREATE PROCEDURE curdemo()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
    DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
    DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
    DECLARE a CHAR(16);
    DECLARE b,c INT;
    OPEN cur1;
    OPEN cur2;
    REPEAT
        FETCH cur1 INTO a, b;
        FETCH cur2 INTO c;
        IF NOT done THEN
            IF b < c THEN
                INSERT INTO test.t3 VALUES (a,b);
            ELSE
                INSERT INTO test.t3 VALUES (a,c);
            END IF;
        END IF;
        UNTIL done END REPEAT;
    CLOSE cur1;
    CLOSE cur2;
END
```

# Triggers - Gatilhos

- São procedimentos especiais ativados quando ocorre uma inserção, atualização ou exclusão em uma tabela ou em uma view
- Diferem da stored procedures pelo fato de não serem chamados diretamente pelo usuário: quando ocorre um determinado evento na tabela, eles são executados
- Possibilitam:
  - Manter a integridade dos dados: atualizar tabelas associadas
  - Cria logs do sistema: a cada inclusão
  - Notificações de alterações no banco aos usuários
  - Validação de restrições de integridade mais complexas que as suportadas diretamente pelo SGBD

# Triggers - Sintaxe

CREATE

[DEFINER = {user | CURRENT\_USER}]

TRIGGER nome\_trigger tempo\_trigger evento\_trigger

ON nome\_tabela FOR EACH ROW

BEGIN

declaração\_trigger

END

# Triggers - Sintaxe

- DEFINER: Quando o TRIGGER for disparado, esta opção será checada para checar com quais privilégios este será disparado. Utilizará os privilégios do usuário informado em user ('ronaldo'@'localhost') ou os privilégios do usuário atual (CURRENT\_USER). Caso essa sentença seja omitida da criação do TRIGGER, o valor padrão desta opção é CURRENT\_USER();
- Nome\_trigger: define o nome do procedimento, por exemplo, trg\_test;
- Tempo\_trigger: define se o TRIGGER será ativado antes (BEFORE) ou depois (AFTER) do comando que o disparou;
- Evento\_trigger: aqui se define qual será o evento, INSERT, REPLACE, DELETE ou UPDATE;
- nome\_tabela: nome da tabela onde o TRIGGER ficará "pendurado" aguardando o trigger\_event;
- declarações\_trigger: as definições do que o TRIGGER deverá fazer quando for disparado.

# Trigger - Exemplo 1

- Trigger que atualiza o campo VALOR da tabela VENDAS cada vez que se alterar a tabela de itens

delimiter //

```
CREATE TRIGGER InsItem AFTER INSERT ON Itens  
FOR EACH ROW  
BEGIN
```

```
    UPDATE Vendas set valor_total_venda = valor_total_venda + New.ValorTotal  
    WHERE cod_pedido = New.cod_pedido;
```

END //

Vendas (cod\_pedido, codcliente, data,  
valor\_total\_venda)  
Itens (cod\_pedido, codproduto, qtde,  
preco\_un, ValorTotal)

O alias NEW indica o registro que está  
sendo inserido

## Trigger - Exemplo 2

Sistema para aluguel de carros - Atualização da quilometragem rodada por um carro após sua devolução

DELIMITER //

CREATE TRIGGER tr\_kmrodados after update on reservas

FOR EACH ROW

UPDATE carros SET quilometragem = quilometragem + NEW.quilometrosrodados

WHERE NEW.carro\_reserva = carros.id;

# Trigger - Exemplo 3

➤ Exclusão de um item da tabela

delimiter //

CREATE TRIGGER DellItem AFTER DELETE ON Itens FOR EACH ROW

BEGIN

    UPDATE Vendas set valor\_total\_venda = valor\_total\_venda - OLD.ValorTotal

    WHERE cod\_pedido = OLD.cod\_pedido;

END //

Vendas (cod\_pedido, codcliente, data,  
valor\_total\_venda)

Itens (cod\_pedido, codproduto, qtde,  
preco\_un, ValorTotal)

O alias OLD indica o registro que está  
sendo apagado

## Trigger - Exemplo 4

➤ Trigger para atualização de dados

Vendas (pedido , codcliente, data, valor)  
Itens (pedido, codproduto, qtde, preco,  
ValorTotal)

Set term # ;

CREATE TRIGGER Atulitem AFTER UPDATE ON Itens

BEGIN

IF (New.ValorTotal <> OLD.ValorTotal) THEN

    UPDATE Vendas set valor = valor - OLD.ValorTotal +    New.ValorTotal WHERE

Pedido = OLD.Pedido;

END //

## 1 – Considerando a tabela **auditoria\_salario** com os seguintes atributos:

func\_codigo: código do funcionário

salário\_inicial: salário antes de ser alterado

salário\_alterado: novo salário do funcionário

data\_alteração: data da alteração do salário

nome\_usuário: usuário que realizou a alteração do salário do funcionário

**funcionario (cod\_func, nome\_func,  
data\_nascimento, endereco, salario)**

Criar um trigger que, ao alterar o salário de um empregado, registrar corretamente na tabela **auditoria\_salario** as atualizações.

# Exercícios

2- Criar uma stored procedure chamada alteraSalFunc, que altera o salário de um funcionário de acordo com o número de dependentes que ele possui. Por exemplo, se o funcionário possuir 1 dependente, ele terá um aumento de 10%; se o funcionário possuir 2 dependentes, ele terá um aumento de 20%; e assim por diante. Deve ser passado como parâmetro o código do funcionário, e a função deve retornar a porcentagem de aumento do salário do empregado, além de atualizar o salário do funcionário de forma adequada no banco de dados.

funcionario (idfunc, nome, salario, data\_admissao, cpf, rg)

dependentes (idfunc, nome\_dependente, parentesco, data\_nascimento)

- Instrução SQL padrão para lançar uma exceção.
- Usada em triggers, procedures e functions.
- Permite interromper uma operação quando uma regra de negócio não é atendida.
- Aborta a instrução que o disparou.
- Reverte todos os efeitos dessa instrução (atomicidade por instrução).
- **Sintaxe Básica**

```
SIGNAL SQLSTATE '45000'
```

```
SET MESSAGE_TEXT = 'Mensagem de erro';
```

# SIGNAL - Triggers

- Exemplo 1 – Validação antes de inserir - Impedir inserção de cliente menor de 18 anos.

```
DELIMITER $$
```

```
CREATE TRIGGER trg_cliente_idade
BEFORE INSERT ON clientes
FOR EACH ROW
BEGIN
    IF NEW.cli_idade < 18 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Cliente deve ter no mínimo 18 anos.';
    END IF;
END$$
```

```
DELIMITER ;
```

# SIGNAL - Triggers

## ➤ Exemplo 2 – Impedir preço abaixo do mínimo

```
DELIMITER $$
```

```
CREATE TRIGGER trg_valida_preco_produto
BEFORE UPDATE ON produtos
FOR EACH ROW
BEGIN
    IF NEW.prd_preco < NEW.prd_preco_minimo THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Preço informado está abaixo do valor mínimo permitido.';
    END IF;
END$$
```

```
DELIMITER ;
```

# SIGNAL - Procedures

## ➤ Exemplo 3 - Verificar saldo antes de saque

```
CREATE PROCEDURE sacar(IN p_conta INT, IN p_valor DECIMAL(10,2))
BEGIN
    DECLARE v_saldo DECIMAL(10,2);
    SELECT saldo INTO v_saldo FROM contas WHERE conta_id = p_conta;
    IF v_saldo IS NULL THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Conta inexistente.';
    ELSEIF v_saldo < p_valor THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Saldo insuficiente para saque.';
    ELSE
        UPDATE contas SET saldo = saldo - p_valor WHERE conta_id = p_conta;
    END IF;
END$$
DELIMITER ;
```

# SIGNAL - Procedures

## ➤ Exemplo 4 - Inserir cliente com e-mail único

```
DELIMITER $$
```

```
CREATE PROCEDURE inserir_cliente(IN p_nome VARCHAR(100), IN p_email VARCHAR(100))
BEGIN
```

```
    IF EXISTS (SELECT 1 FROM clientes WHERE cli_email = p_email) THEN
        SIGNAL SQLSTATE '45000'
```

```
        SET MESSAGE_TEXT = 'E-mail já cadastrado. Utilize outro.';
```

```
    ELSE
```

```
        INSERT INTO clientes (cli_nome, cli_email) VALUES (p_nome, p_email);
```

```
    END IF;
```

```
END$$
```

```
DELIMITER ;
```

# Banco de Dados

## Pós-Graduação em Ciência da Computação

**Prof. Dr. Ronaldo Celso Messias Correia**  
[ronaldo.correia@unesp.br](mailto:ronaldo.correia@unesp.br)



UNIVERSIDADE ESTADUAL PAULISTA  
“JÚLIO DE MESQUITA FILHO”



# Transações

# Transações

- Conceito de transação
- Estado da transação
- Execuções simultâneas
- Seriação
- Facilidade de recuperação
- Implementação do isolamento
- Definição de transação na SQL
- Testando a seriação

# Conceito de transações

- Uma transação é uma unidade da execução de programa que acessa e possivelmente atualiza vários itens de dados.
- Uma transação precisa ver um banco de dados consistente.
- Durante a execução da transação, o banco de dados pode ser temporariamente inconsistente.
- Quando a transação é completada com sucesso (é confirmada), o banco de dados precisa ser consistente.
- Após a confirmação da transação, as mudanças que ele faz no banco de dados persistem, mesmo se houver falhas de sistema.
- Várias transações podem ser executadas em paralelo.
- Dois problemas principais para resolver:
  - Falhas de vários tipos, como falhas de hardware e quedas de sistema
  - Execução concorrente de múltiplas transações

# Propriedades ACID

- Uma transação é unidade da execução do programa que acessa e possivelmente atualiza vários itens de dados
- Para preservar a integridade dos dados, o banco deve assegurar:
  - **Atomicidade** . Ou todas as operações da transação são refletidas corretamente no banco de dados ou nenhuma delas é.
  - **Consistência** . A execução de uma transação isolada preserva a consistência do banco de dados.
  - **Isolamento** . Embora várias transações possam ser executadas simultaneamente, cada transação precisa estar desinformada das outras transações que estão sendo executadas ao mesmo tempo. Os resultados da transação intermediária precisam estar ocultos das outras transações sendo executadas simultaneamente.
    - Ou seja, para cada par de transações,  $T_i$  e  $T_j$ , parece para  $T_i$  que  $T_j$  terminou a execução antes que  $T_i$  começasse ou  $T_j$  iniciou a execução depois que  $T_i$  terminou.
  - **Durabilidade** . Depois que uma transação for completada com sucesso, as mudanças que ela fez ao banco de dados persistem, mesmo que existem falhas no sistema.

# Operações de acesso

- O Acesso ao banco de dados é obtido pelas seguintes operações:
  - `read(X)` – que transfere o item de dados X do banco de dados para um buffer local alocado à transação que executou a operação **read**
  - `write(X)` – que transfere o item de dados X do buffer local da transação que executou a **write** de volta ao banco de dados
- Em um sistema de banco de dados real, a operação write (escrita) não resulta necessariamente na atualização imediata dos dados no disco

# Exemplo de transferência de fundos

- 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 atomicidade — Se a transação falhar após a etapa 3 e antes da etapa 6, o sistema deve garantir que suas atualizações não sejam refletidas no banco de dados, ou uma inconsistência irá resultar.
- Requisito de consistência — A soma de A e B é inalterada pela execução da transação.

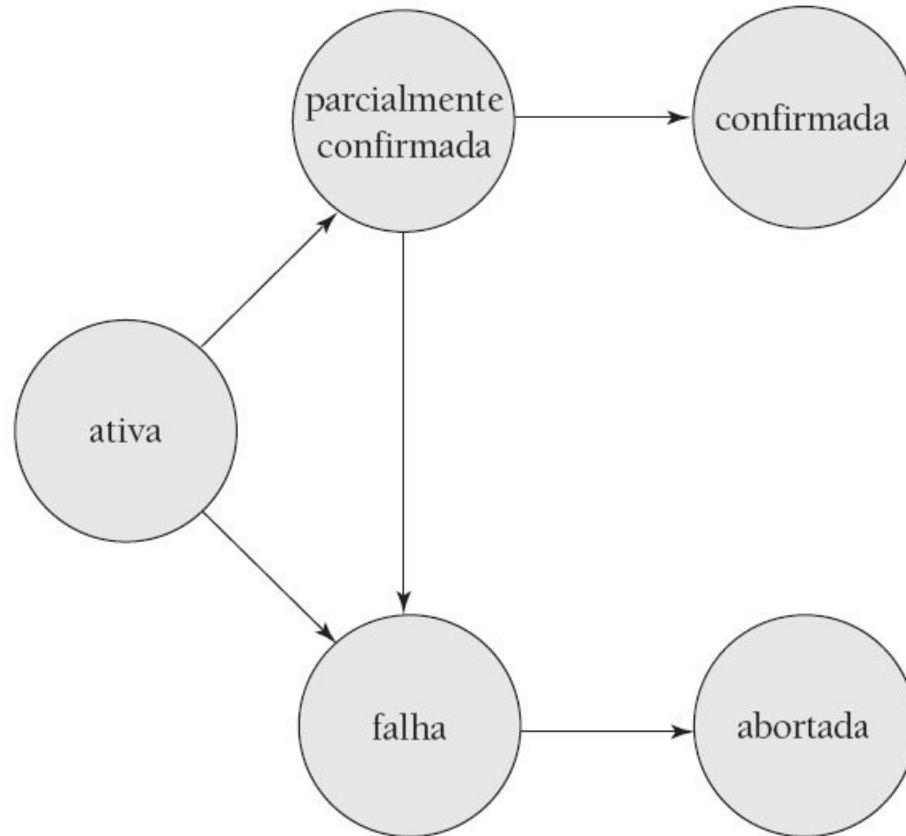
## Exemplos de transferência de fundos (cont.)

- Requisito de isolamento — Se entre as etapas 3 e 6, outra transação receber permissão de acessar o banco de dados parcialmente atualizado, ele verá um banco de dados inconsistente (a soma A + B será menor do que deveria ser).
  - Isso pode ser trivialmente assegurado executando transações serialmente (serializadas), ou seja, uma após outra. Entretanto, executar múltiplas transações simultaneamente oferece vantagens significativas, como veremos mais adiante.
- Requisito de durabilidade — Quando o usuário é notificado de que a transação está concluída (ou seja, a transação dos \$ 50 ocorreu), as atualizações no banco de dados pela transação precisam persistir apesar de falhas.

# Estado da transação

- **Ativa** – O estado inicial; a transação permanece nesse estado enquanto está executando
- **Parcialmente confirmada** – Depois que a instrução final foi executada
- **Falha** – Depois da descoberta de que a execução normal não pode mais prosseguir
- **Abortada** – Depois que a transação foi revertida e o banco de dados foi restaurado ao seu estado anterior ao início da transação. Duas opções após ter sido abortada:
  - Reiniciar a transação; pode ser feito apenas se não houver qualquer erro lógico interno
  - Excluir a transação – erro lógico interno
- **Confirmada** (Committed)– Após o término bem sucedido

## Estado da transação (cont.)



# Execuções simultâneas (Concorrentes)

- Várias transações podem ser executadas simultaneamente no sistema. As vantagens são:
  - **Melhor utilização do processador e do disco**, levando a um melhor throughput de transação: uma transação pode estar usando a CPU enquanto outra está lendo ou escrevendo no disco
  - **Tempo de médio de resposta reduzido para transações**: as transações curtas não precisam esperar atrás das longas
- **Esquemas de controle de concorrência** – mecanismos para obter isolamento; ou seja, para controlar a interação entre as transações concorrentes a fim de evitar que elas destruam a consistência do banco de dados
- **Throughput**: número de transações executadas em determinada quantidade de tempo

# Escalonamento

- Escalonamento – Sequências de instruções que especificam a ordem cronológica em que as instruções das transações concorrentes são executadas
  - Um escalonamento para um conjunto de transações precisa consistir em todas as instruções dessas transações
  - Precisam preservar a ordem em que as instruções aparecem em cada transação individual
- Uma transação que completa com sucesso sua execução terá uma instrução commit como a última instrução (será omitida se for óbvia)
- Uma transação que não completa com sucesso sua execução terá instrução abort como a última instrução (será omitida se for óbvia)

# Escalonamento 1

- Suponha que T1 transfere \$ 50 de A para B e T2 transfere 10% do saldo de A para B.
- A seguir está um escalonamento serial em que T1 é seguido de T2 .

$T_1$	$T_2$
<code>read(A)</code> $A := A - 50$ <code>write (A)</code> <code>read(B)</code> $B := B + 50$ <code>write(B)</code>	<code>read(A)</code> $temp := A * 0.1$ $A := A - temp$ <code>write(A)</code> <code>read(B)</code> $B := B + temp$ <code>write(B)</code>

## Escalonamento 2

- Sejam  $T_1$  e  $T_2$  as transações definidas anteriormente. O escalonamento a seguir não é um escalonamento serial, mas é equivalente ao escalonamento 1.

$T_1$	$T_2$
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$	$\text{read}(A)$ $temp := A * 0.1$ $A := A - temp$ $\text{write}(A)$
$\text{read}(B)$ $B := B + 50$ $\text{write}(B)$	$\text{read}(B)$ $B := B + temp$ $\text{write}(B)$

# Escalonamento 3

- O seguinte escalonamento concorrente 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)

# Seriação/Serialização

- Suposição básica – Cada transação preserva a consistência do banco de dados
- Portanto, a execução serial de um conjunto de transações preserva a consistência do banco de dados
- Um escalonamento (possivelmente simultâneo) é serializável se for equivalente a um escalonamento serial. Diferentes formas de equivalência de escalonamento ensejam as noções de:
  1. Seriação de conflito
  2. Seriação de view
- Ignoramos as operações exceto read e write, e consideramos que as transações podem realizar cálculos arbitrários sobre dados em buffers locais entre reads e writes. Nossos escalonamentos simplificados consistem apenas em instruções read e write.

# Instruções conflitantes

- As instruções  $l_i$  e  $l_j$  das transações  $T_i$  e  $T_j$  respectivamente, estão em conflito se e somente se algum item  $Q$  acessado por  $l_i$  e por  $l_j$  e pelo menos uma destas instruções escreveram  $Q$ .
  1.  $l_i = \text{read}(Q)$ ,  $l_j = \text{read}(Q)$ .  $l_i$  e  $l_j$  não estão em conflito
  2.  $l_i = \text{read}(Q)$ ,  $l_j = \text{write}(Q)$ . Estão em conflito
  3.  $l_i = \text{write}(Q)$ ,  $l_j = \text{read}(Q)$ . Estão em conflito
  4.  $l_i = \text{write}(Q)$ ,  $l_j = \text{write}(Q)$ . Estão em conflito
- Intuitivamente, um conflito entre  $l_i$  e  $l_j$  força uma ordem temporal (lógica) entre eles. Se  $l_i$  e  $l_j$  são consecutivos em um escalonamento e não entram em conflito, seus resultados permanecem inalterados mesmo se tiverem sido trocados no escalonamento.

# Seriação de conflito

- Se um escalonamento  $S$  puder ser transformado em um escalonamento  $S'$  por uma série de trocas de instruções não conflitantes, dizemos que  $S$  e  $S'$  são equivalentes em conflito.
- Dizemos que um escalonamento  $S$  é serial de conflito se ele for equivalente em conflito a um escalonamento serial

## Seriação de conflito (cont.)

- O Escalonamento 3 abaixo pode ser transformado em Escalonamento 1 (slide a seguir), um escalonamento onde  $T_2$  segue  $T_1$ , por uma série de trocas de instruções não conflitantes. Portanto, o escalonamento 3 é serial de conflito.

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

## Seriação de conflito (cont.)

- A instrução write(A) de  $T_2$  não conflita com a instrução read(B) de  $T_1$
- Continuando a inverter instruções não conflitantes:
  - Trocar a instrução read(B) de  $T_1$  pela instrução read(A) de  $T_2$
  - Trocar a instrução write(B) de  $T_1$  pela instrução write(A) de  $T_2$
  - Trocar a instrução write(B) de  $T_1$  pela instrução read(A) de  $T_2$

# Escalonamento 1

$T_1$	$T_2$
<b>read(A)</b> $A := A - 50$ <b>write (A)</b> <b>read(B)</b> $B := B + 50$ <b>write(B)</b>	<b>read(A)</b> $temp := A * 0.1$ $A := A - temp$ <b>write(A)</b> <b>read(B)</b> $B := B + temp$ <b>write(B)</b>

## Seriação de conflito (cont.)

- Exemplo de um escalonamento que não é serial de conflito:
- Não podemos trocar instruções no escalonamento acima para obter o escalonamento serial  $\langle T_3, T_4 \rangle$ , ou o escalonamento serial  $\langle T_4, T_3 \rangle$ .

$T_3$	$T_4$
read(Q)	
write(Q)	write(Q)

# Seriação de Visão

- Sejam  $S$  e  $S'$  dois escalonamento com o mesmo conjunto de transações.  $S$  e  $S'$  são equivalentes em visão se as três condições a seguir forem satisfeitas:
  1. Para cada item de dados  $Q$ , se a transação  $T_i$  ler o valor inicial de  $Q$  no escalonamento  $S$ , então,  $T_i$  precisa, no escalonamento  $S'$ , também ler o valor inicial de  $Q$ .
  2. Para cada item de dados  $Q$ , se a transação  $T_i$  executar  $\text{read}(Q)$  no escalonamento  $S$ , e se esse valor pela transação  $T_j$  (se houver), então a transação  $T_i$ , no escalonamento  $S'$ , também precisa ler o valor de  $Q$  que foi produzido pela transação  $T_j$ .
  3. Para cada item de dados  $Q$ , a transação (se houver) que realiza a operação  $\text{write}(Q)$  final no escalonamento  $S$  precisa realizar a operação  $\text{write}(Q)$  final no escalonamento  $S'$ .
- As condições 1 e 2 garantem que cada transação lê os mesmos valores nos dois schedules e, portanto, realiza a mesma computação. A condição 3, junto com as condições 1 e 2, garante que os dois schedules resultam no mesmo estado final
- Como podemos ver, a equivalência em visão também é baseada unicamente em reads e writes isolados.

## Seriação de Visão (cont.)

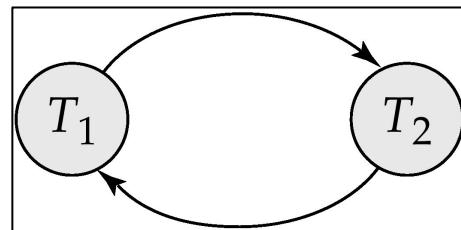
- Um escalonamento S é serial de visão se ele for equivalente em visão a um escalonamento serial
- Todo escalonamento serial de conflito também é serial de visão
- A seguir está um escalonamento que é serial de visão mas não serial de conflito

$T_3$	$T_4$	$T_6$
read( $Q$ )		
write( $Q$ )	write( $Q$ )	write( $Q$ )

- Todo escalonamento serial de visão que não é serial de conflito possui escritas cegas

# Testando a seriação

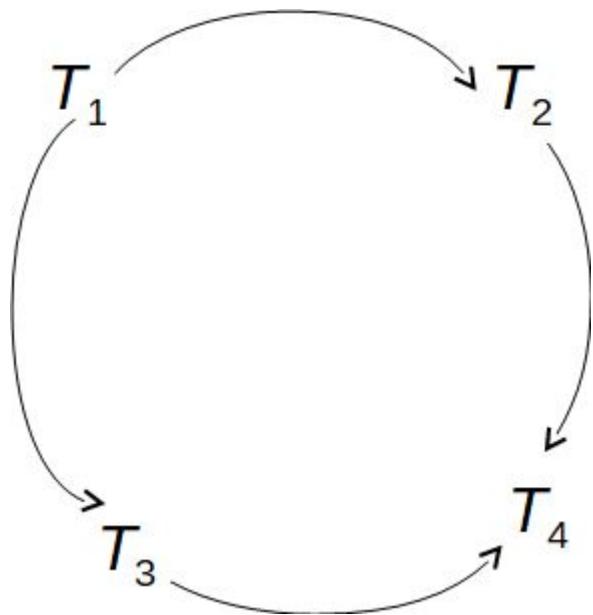
- Considere um escalonamento de um conjunto de transações  $T_1, T_2, \dots, T_n$
- Grafo de precedência — Um grafo direcionado onde os vértices são as transações (nomes)
- Desenhamos um arco (aresta) de  $T_i$  até  $T_j$  se a transação entra em conflito e  $T_i$  acessou o item de dados no qual surgiu o conflito anteriormente
- Podemos rotular o arco pelo item que foi acessado
- Exemplo 1



# Exemplo de escalonamento (Escalonamento A)

T1	T2	T3	T4	T5
read(Y) read(Z)	read(X)			
read(U)	read(Y) write(Y)	write(Z)		read(V) read(W) read(W)
read(U) write(U)			read(Y) write(Y) read(Z) write(Z)	

# Gráfo de precedênciā para o escalonamento A



# Teste para seriação de conflito

- Um escalonamento é serial de conflito se e somente se seu grafo de precedência for acíclico
- Existem algoritmos de detecção de ciclo que assumem a ordem cronológica  $n^2$ , onde  $n$  é o número de vértices no grafo. (Algoritmos melhores assumem a ordem  $n + e$ , onde  $e$  é o número de arestas.)
- Se o grafo de precedência for acíclico, a ordem de seriação pode ser obtida por meio da classificação topológica do grafo. Essa é a ordem linear com a ordem parcial do grafo. Por exemplo, uma ordem de seriação para o escalonamento A seria  $T_5 \ T_1 \ T_3 \ T_2 \ T_4$ .

## Teste para seriação de View

- O teste do grafo de precedência precisa ser modificado para se aplicar a um teste para seriação de view
- O problema de verificar se um escalonamento é serial de view entra na classe dos problemas não procedurais completos. Portanto, embora a existência de um algoritmo eficiente seja improvável, os algoritmos práticos que simplesmente verificam algumas condições suficientes para a seriação de view ainda podem ser usados.

# Facilidade de recuperação

- É necessário tratar do efeito das falhas de transação nas transações sendo executadas simultaneamente.
- Escalonamento recuperável — Se uma transação  $T_j$  lê um item de dados anteriormente escrito por uma transação  $T_i$ , então a operação commit de  $T_i$  aparece antes da operação commit de  $T_j$ .
- O escalonamento (Escalonamento 11) não é recuperável se  $T_9$  for confirmado imediatamente após o read

$T_8$	$T_9$
read( $A$ )	
write( $A$ )	
read( $B$ )	read( $A$ )

- Se  $T_8$  abortasse,  $T_9$  teria lido (e possivelmente mostrado ao usuário) um estado inconsistente. Portanto, o banco de dados precisa garantir que Escalonamentos sejam recuperáveis

## Facilidade de recuperação (cont.)

- Rollback em cascata – Uma única falha de transação leva a uma série de rollbacks de transação. Considere o seguinte Escalonamento onde nenhuma das transações ainda foi confirmada (portanto, o Escalonamento é recuperável)

$T_{10}$	$T_{11}$	$T_{12}$
read( $A$ ) read( $B$ ) write( $A$ )	read( $A$ ) write( $A$ )	read( $A$ )

- Se  $T_{10}$  falhar,  $T_{11}$  e  $T_{12}$  também precisam ser revertidos
- Pode chegar a desfazer uma quantidade de trabalho significativa

## Facilidade de recuperação (cont.)

---

- Escalonamentos não em cascata — Rollbacks em cascata não podem ocorrer; para cada par de transações  $T_i$  e  $T_j$  tal que  $T_j$  leia um item de dados escrito anteriormente por  $T_i$ , a operação commit de  $T_i$  apareça antes da operação read de  $T_j$ .
- Todo escalonamento não em cascata também é recuperável
- É desejável restringir os escalonamentos aos não em cascata

# Aspectos da implementação

- Um banco de dados (SGBD) precisa fornecer um mecanismo que garanta que todos os escalonamentos possíveis sejam seriais de conflito ou de view, e sejam recuperáveis e, preferivelmente, não em cascata
- Uma política em que apenas uma transação pode ser executada de cada vez gera escalonamentos seriais, mas fornece um menor grau de concorrência
- Os esquemas de controle de concorrência conciliam entre a quantidade de concorrência que permitem e a quantidade de sobrecarga a que ficam sujeitos

# Testes de controle de serialização

- Testar a serialização de um escalonamento após ele ter sido executado é um pouco tarde demais!
- Objetivo – Desenvolver os protocolos de controle de concorrência que garantirão a capacidade de serialização. Eles normalmente não examinam o gráfo de precedência enquanto está sendo criado, em vez disso, um protocolo imporá uma regra que evita escalonamentos não serializáveis.
- Os testes para a serialização ajudam a entender por que um protocolo de controle de concorrência está correto

# Banco de Dados

## Pós-Graduação em Ciência da Computação

**Prof. Dr. Ronaldo Celso Messias Correia**  
[ronaldo.correia@unesp.br](mailto:ronaldo.correia@unesp.br)



UNIVERSIDADE ESTADUAL PAULISTA  
“JÚLIO DE MESQUITA FILHO”



# Concorrência

# Controle de concorrência

- Protocolos baseados em bloqueio
- Protocolos baseados em timestamp
- Granularidade múltipla
- Esquemas multiversão
- Tratamento de deadlock

# Protocolos baseados em bloqueio

- Um bloqueio é um mecanismo para controlar o acesso concorrente a um item de dados
- Bloqueio Binário:
  - Pode ter dois estados ou valores: locked (bloqueado) ou unlocked (desbloqueado)
  - No máximo uma transação pode manter o bloqueio de um determinado item
  - Duas transações não podem acessar o mesmo item de maneira concomitante
- Bloqueio Múltiplo: podem ser bloqueados em dois modos:
  - 1. Modo exclusivo (X). O item de dados pode ser lido e também escrito. O bloqueio X é solicitado pela instrução lock-X.
  - 2. Modo compartilhado (S). O item de dados só pode ser lido. O bloqueio S é solicitado pela instrução lock-S.
- As solicitações de bloqueio são feitas ao gerenciador de controle de concorrência. A transação só pode prosseguir após a concessão da solicitação.

# Protocolos baseados em bloqueio (cont.)

- Uma transação pode receber um bloqueio sobre um item se o bloqueio solicitado for compatível com os bloqueios já mantidos sobre o item por outras transações
- Matriz de compatibilidade de bloqueio

	S	X
S	true	false
X	false	false

- Qualquer quantidade de transações pode manter bloqueios compartilhados sobre um item, mas se qualquer transação mantiver um bloqueio exclusivo sobre um item, nenhuma outra pode manter qualquer bloqueio sobre o item.
- Se um bloqueio não puder ser concedido, a transação solicitante deve esperar até que todos os bloqueios incompatíveis mantidos por outras transações tenham sido liberados. O bloqueio é então concedido.

# Protocolos baseados em bloqueio (cont.)

- Exemplo de uma transação realizando bloqueio:

T <sub>1</sub>	T <sub>2</sub>
lock-X(B) read(B) $B := B - 50$ write(B) unlock(B) lock-X(A) read(A) $A := A + 50$ write(A) unlock(A)	lock-S(A) read(A) unlock(A) lock-S(B) read(B) unlock(B) display(A + B)

- O bloqueio acima não é suficiente para garantir a seriação - se A e B fossem atualizados entre a leitura de A e B, a soma exibida estaria errada.
- Um protocolo de bloqueio é um conjunto de regras seguidas por todas as transações enquanto solicita e libera bloqueios. Os protocolos de bloqueio restringem o conjunto de escalonamentos possíveis.

# Protocolos baseados em bloqueio (cont.)

- Transações  $T_1$  e  $T_2$  executadas concorrentemente

$T_1$	$T_2$	Gerenciador de Controle de concorrência
lock-X( $B$ )  read( $B$ ) $B := B - 50$ write ( $B$ ) unlock( $B$ )	lock-S( $A$ )  read( $A$ ) unlock( $A$ ) lock-S( $B$ )  read( $B$ ) unlock( $B$ ) display( $A + B$ )	grant-X( $B, T_1$ )  grant-S( $A, T_2$ )  grant-S( $B, T_2$ )  grant-X( $A, T_2$ )
lock-X( $A$ )  read( $A$ ) $A := A + 50$ write ( $A$ ) unlock( $A$ )		

# Armadilhas dos protocolos baseados em bloqueio

- Considere o escalonamento parcial

$T_3$	$T_4$
lock-X( $B$ )	
read( $B$ )	
$B := B - 50$	
write( $B$ )	
	lock-S( $A$ )
	read( $A$ )
	lock-S( $B$ )
lock-X( $A$ )	

- Nem  $T_3$  e nem  $T_4$  podem ter progresso - a execução de lock-S( $B$ ) faz com que  $T_4$  espere que  $T_3$  libere seu bloqueio sobre  $B$ , enquanto a execução de lock-X( $A$ ) faz com que  $T_3$  espere que  $T_4$  libere seu bloqueio sobre  $A$ .
- Essa situação é chamada de deadlock
  - Para controlar o deadlock, um dentre  $T_3$  ou  $T_4$  precisa ser revertido (rollback) e seus bloqueios liberados.

## Armadilhas dos protocolos baseados em bloqueio (cont.)

- A maioria dos protocolos de bloqueio têm potencial para gerar deadlocks. Deadlocks são um mal necessário.
- Inanição também é possível se o gerenciador de controle de concorrência for mal projetado. Por exemplo:
  - Uma transação pode estar esperando por um X-lock em um item, enquanto uma sequência de outras transações solicitam e recebem um S-lock no mesmo item.
  - A mesma transação sofre repetidos rollbacks devido a deadlocks.
- O gerenciador de controle de concorrência pode ser projetado para prevenir a inanição (starvation).

# O protocolo de bloqueio em duas fases

- Esse é um protocolo que garante escalonamentos seriáveis por conflito.
- Requer que cada transação emita solicitações de bloqueio e desbloqueio em duas fases:
  - Fase 1: Fase de expansão / crescimento
    - transação pode obter bloqueios
    - transação não pode liberar bloqueios
  - Fase 2: Fase de Encolhimento
    - transação pode liberar bloqueios
    - transação não pode obter bloqueios
- O protocolo garante serialização. Pode-se provar que as transações podem ser serializáveis na ordem dos seus pontos de bloqueio (lock points) (ou seja, o ponto onde a transação adquire o seu último bloqueio).

# O protocolo de bloqueio em duas fases (cont.)

- Exemplo de bloqueio em duas fases

T <sub>1</sub>	T <sub>2</sub>
Lock-X (Aplic); Read (Aplic); Aplic.Saldo = Aplic.Saldo - 500; Write (Aplic); Lock-X (Conta); Unlock (Aplic); // Inicia 2 <sup>a</sup> fase Read (Conta);	
Conta.Saldo = Conta.Saldo + 500; Write (Conta); Unlock (Conta);	Lock-S (Conta); <i>Bloqueada</i>
	Read (Conta); Lock-S (Aplic); Unlock (Conta); // Inicia 2 <sup>a</sup> fase Read (Aplic); Print (Conta.Saldo + Aplic.Saldo); Unlock (Aplic);

# O protocolo de bloqueio em duas fases (cont.)

- O bloqueio em duas fases não garante ausência de deadlocks.

$T_1$	$T_2$
Lock-X (Aplic); Read (Aplic); Aplic.Saldo = Aplic.Saldo - 500;	
	Lock-S (Conta); Read (Conta); Lock-S (Aplic);
Write (Aplic); Lock-X (Conta);	Bloqueada
Bloqueada	Bloqueada
<b>Não executa:</b> Unlock (Aplic); Read (Conta); Conta.Saldo = Conta.Saldo + 500; Write (Conta); Unlock (Conta);	<b>Não executa:</b> Unlock (Conta); Read (Aplic); Print (Conta.Saldo + Aplic.Saldo); Unlock (Aplic);

# O protocolo de bloqueio em duas fases (cont.)

- Bloqueio em duas fases não evita rollback em cascata

$T_1$	$T_2$
Lock-X (Aplic); Read (Aplic); $\text{Aplic.Saldo} = \text{Aplic.Saldo} - 500;$ Write (Aplic); Lock-X (Conta); Unlock (Aplic); Read (Conta);	
	Lock-S (Aplic); Read (Aplic); Lock-S (Conta);
Conta.Saldo = Conta.Saldo + 500; Write (Conta); // ABORTA Unlock(Conta);	Bloqueada
	Read (Conta); // ABORTA Unlock (Conta); Print (Conta.Saldo + Aplic.Saldo); Unlock (Aplic);

# Variantes do bloqueio em duas fases (cont.)

- Rollback em cascata é possível sob o bloqueio em duas fases. Para evitar isso, usa-se um protocolo modificado chamado bloqueio em duas fases severo (strict two-phase locking). Aqui, uma transação deve manter todos os seus bloqueios exclusivos até que ela execute o commit ou o abort.
- O bloqueio em duas fases rigoroso é ainda mais restrito: todos os bloqueios são mantidos até que a transação execute commit ou abort. Nesse protocolo, as transações podem ser serializadas na ordem em que elas executam commit.

T <sub>1</sub>	T <sub>2</sub>
Lock-X (Aplic); Read (Aplic); Aplic.Saldo = Aplic.Saldo - 500; Write (Aplic); Lock-X (Conta); Read (Conta);	Lock-S (Aplic);  Bloqueada
Conta.Saldo = Conta.Saldo + 500; Write (Conta); <b>Unlock (Aplic);</b> Unlock(Conta);	
	Read (Aplic); Lock-S (Conta); <b>Unlock (Aplic);</b> Read (Conta); Print (Conta.Saldo + Aplic.Saldo); Unlock (Conta);

T <sub>1</sub>	T <sub>2</sub>
Lock-X (Aplic); Read (Aplic); Aplic.Saldo = Aplic.Saldo - 500; Write (Aplic); Lock-X (Conta); Read (Conta);	Lock-S (Aplic);  Bloqueada
Conta.Saldo = Conta.Saldo + 500; Write (Conta); <b>Unlock (Aplic);</b> Unlock(Conta);	
	Read (Aplic); Lock-S (Conta); Read (Conta); Print (Conta.Saldo + Aplic.Saldo); <b>Unlock (Aplic);</b> Unlock (Conta);

# Conversões de bloqueio

- Bloqueio em duas fases com conversões de bloqueio:
  - Primeira fase:
    - pode adquirir um bloqueio-S sobre o item
    - pode adquirir um bloqueio-X sobre o item
    - pode converter um bloqueio-S para um bloqueio-X (upgrade)
  - Segunda fase:
    - pode liberar um bloqueio-S
    - pode liberar um bloqueio-X
    - pode converter um bloqueio-X para um bloqueio-S (downgrade)
- Esse protocolo garante a seriação. Mas ainda conta com o programador para inserir as diversas instruções de bloqueio.

# Aquisição automática de bloqueios

- Uma transação  $T_i$  emite a instrução de leitura/escrita padrão, sem chamadas de bloqueio explícitas.
- A operação  $\text{read}(D)$  é processada como:

```
if  $T_i$  tem um bloqueio sobre D  
    then  
        read(D)  
    else  
        begin  
            se necessário, espera até que nenhuma outra  
            transação tenha um bloqueio-X sobre D  
            concede a  $T_i$  um bloqueio-S sobre D;  
            read(D)  
        end
```

# Aquisição automática de bloqueios (cont.)

- write(D) é processado como:

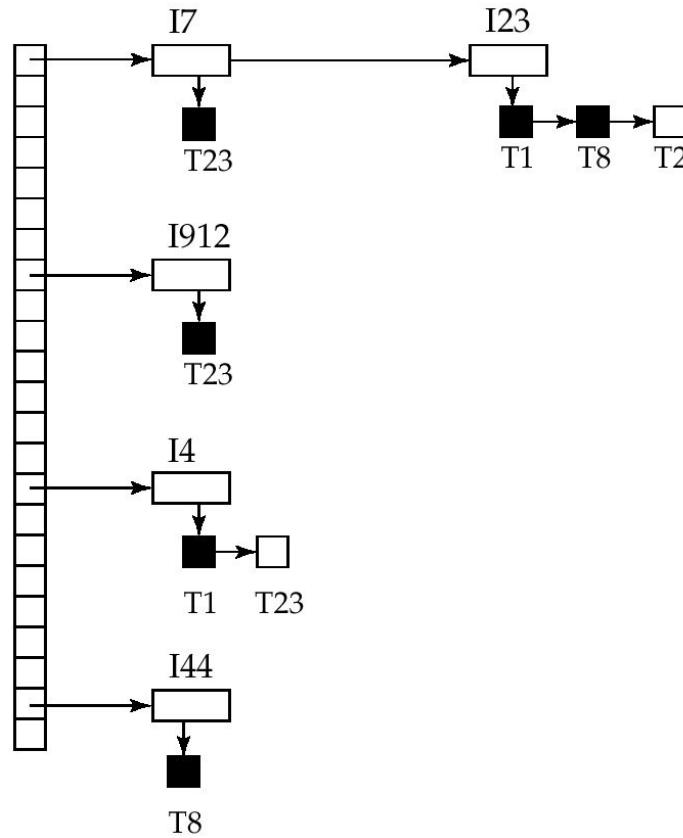
```
if Ti tem um bloqueio-X sobre D  
    then  
        write(D)  
    else  
        begin  
            se for preciso, esperar até que nenhuma outra transação tenha um bloqueio sobre D,  
            if Ti tem um bloqueio-S sobre D  
                then  
                    upgrade do bloqueio sobre D para bloqueio-X  
                else  
                    concede a Ti um bloqueio-X sobre D  
            write(D)  
        end;
```

- Todos os bloqueios são liberados após o commit ou abort

# Implementação do bloqueio

- Um Gerenciador de Bloqueios pode ser implementado como um processo separado, ao qual as transações enviam pedidos de bloqueios e desbloqueios.
- O gerenciador de bloqueios responde a um pedido de bloqueio enviando uma mensagem de concessão de bloqueio (ou uma mensagem pedindo à transação para executar rollback, em caso de um deadlock).
- A transação solicitante espera até que o seu pedido seja respondido.
- O gerenciador de bloqueios mantém uma estrutura de dados chamada tabela de bloqueios para registrar os bloqueios concedidos e os pedidos pendentes.
- A tabela de bloqueios normalmente é implementada como uma tabela hash em memória, indexada pelo nome do item de dados que está sendo bloqueado.

# Tabela de bloqueio



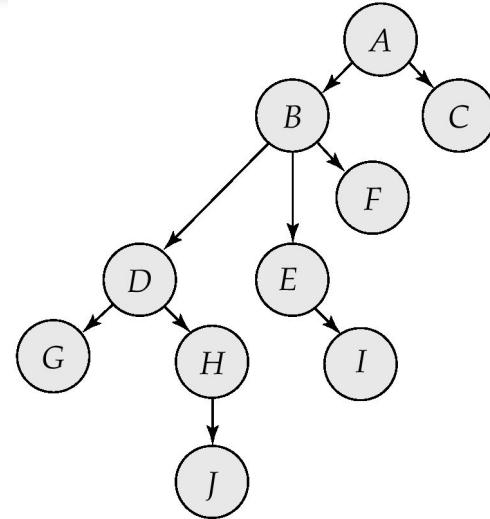
- Retângulos pretos indicam bloqueios concedidos, brancos indicam solicitações aguardando
- A tabela de bloqueio também registra o tipo de bloqueio concedido ou solicitado
- A nova solicitação é acrescentada ao final da fila de solicitações para o item de dados, e concedida se for compatível com todos os bloqueios anteriores
- As solicitações de desbloqueio resultam na solicitação sendo excluída e solicitações posteriores são verificadas para saber se agora podem ser concedidas
- Se a transação abortar, todas as solicitações aguardando ou concedidas da transação são excluídas
  - o gerenciador de bloqueio pode manter uma lista de bloqueios mantidos por cada transação, para implementar isso de forma eficiente

# Protocolos baseados em grafos

- Os protocolos baseados em grafos são uma alternativa ao bloqueio em duas fases
- Imponha uma ordenação parcial → sobre o conjunto  $D = \{d_1, d_2, \dots, d_h\}$  de todos os itens de dados.
  - Se  $d_i \rightarrow d_j$  então qualquer transação acessando  $d_i$  e  $d_j$  precisa acessar  $d_i$  antes de acessar  $d_j$ .
  - Implica que o novo conjunto  $D$  agora pode ser visto como um grafo acíclico direcionado, chamado grafo de banco de dados.
- O protocolo de árvore é um tipo simples de protocolo de grafo.
  - Somente bloqueios exclusivos são permitidos.

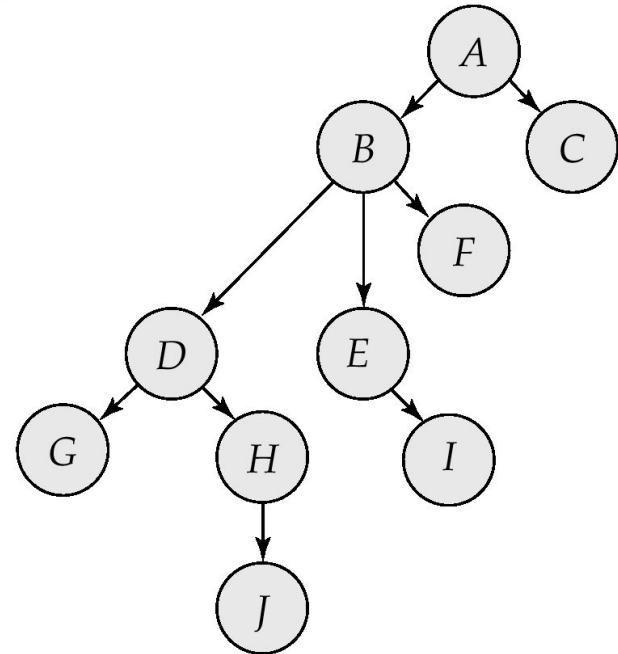
# Protocolo de árvore

- A única instrução de bloqueio permitida é lock-x.
- Cada transação  $T_i$  pode bloquear um item de dados no máximo uma vez, e precisa observar as seguintes regras:
  - O primeiro bloqueio por  $T_i$  pode ser sobre qualquer item de dados.
  - Subsequentemente, um item de dado  $Q$  pode ser bloqueado por  $T_i$  somente se o pai de  $Q$  for
  - Os itens de dados podem ser desbloqueados a qualquer momento.
  - Um item de dados que foi bloqueado e desbloqueado por  $T_i$  não pode mais tarde ser bloqueado novamente por  $T_i$
- Apresenta a vantagem de realizar o desbloqueio mais cedo do que é feito no protocolo em duas fases (reduz o tempo de espera e aumenta a concorrência)
- Todos os escalonamentos que são legais sob o protocolo de árvore são séries de conflito



# Schedule seriável sob o protocolo de árvore

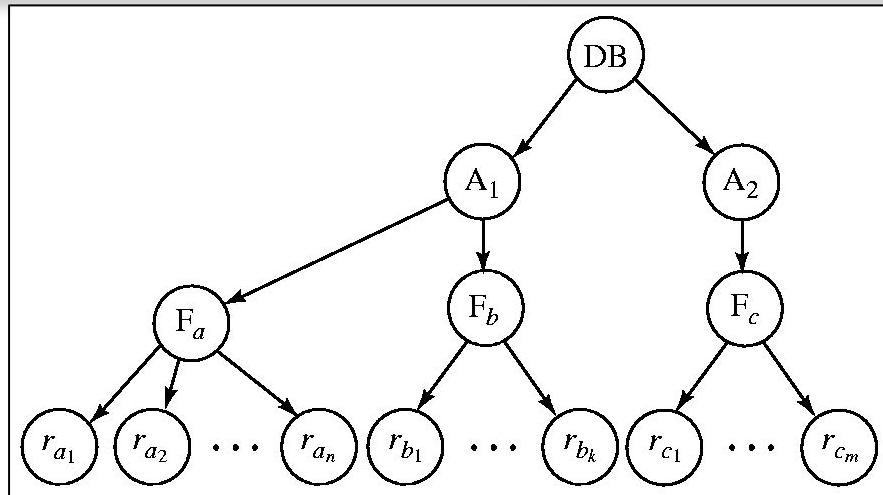
$T_{10}$	$T_{11}$	$T_{12}$	$T_{13}$
lock-X( $B$ )			
lock-X( $E$ ) lock-X( $D$ ) unlock( $B$ ) unlock( $E$ )	lock-X( $D$ ) lock-X( $H$ ) unlock( $D$ )		
lock-X( $G$ ) unlock( $D$ )	unlock( $H$ )	lock-X( $B$ ) lock-X( $E$ )	lock-X( $D$ ) lock-X( $H$ ) unlock( $D$ ) unlock( $H$ )
unlock ( $G$ )		unlock( $E$ ) unlock( $B$ )	



# Granularidades múltiplas

- Permitem que os itens de dados sejam de vários tamanhos e definem uma hierarquia de granularidades de dados, onde as granularidades pequenas são aninhadas dentro das maiores.  
Agrupar vários itens de dados e tratá-los como uma unidade de sincronismo individual
- Podem ser representadas graficamente como uma árvore (mas não confundir com o protocolo de bloqueio em árvore)
- Quando uma transação bloqueia um nó na árvore explicitamente, ela implicitamente bloqueia os descendentes do nó no mesmo modo.
- Ao invés de bloquear um item de dados, podemos bloquear tuplas, tabelas, blocos de disco ou DBs
- Granularidade do bloqueio (nível na árvore onde o bloqueio é feito):
  - granularidade menor (mais baixo na árvore): alta concorrência, alta sobrecarga de bloqueio
  - granularidade maior (mais alto na árvore): baixa sobrecarga de bloqueio, baixa concorrência

# Exemplo de hierarquia de granularidade



- O nível mais alto na hierarquia de exemplo é o banco de dados inteiro.
- Os níveis abaixo são do tipo área, arquivo e registro, nessa ordem.

# Modos de bloqueio de intenção

- Além dos modos de bloqueio S e X, existem três modos de bloqueio adicionais com granularidade múltipla:
  - Intenção de compartilhamento (IS): indica o bloqueio explícito em um nível inferior da árvore, mas apenas com bloqueios compartilhados.
  - Intenção de exclusividade (IX): indica o bloqueio explícito em um nível mais baixo com bloqueios exclusivos ou compartilhados
  - Compartilhamento com Intenção de exclusividade (SIX): a sub-árvore com raiz nesse nó é bloqueada explicitamente no modo compartilhado e o bloqueio explícito está sendo feito em um nível inferior com bloqueios no modo exclusivo.
- os bloqueios de intenção permitem que um nó de nível mais alto seja bloqueado no modo S ou X sem ter que verificar todos os nós descendentes.

## Matriz de compatibilidade com modos de bloqueio de intenção

- A matriz de compatibilidade para todos os modos de bloqueio é:

	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
S IX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

# Protocolos baseados em Timestamp

- Cada transação tem uma timestamp emitido quando entra no sistema. Se uma transação antiga  $T_i$  tem timestamp  $TS(T_i)$ , uma nova transação  $T_j$  recebe a timestamp  $TS(T_j)$  de modo que  $TS(T_i) < TS(T_j)$ .
- O protocolo gerencia a execução concorrente tal que os timestamp determinam a ordem de seriação.
- Existem dois mecanismos simples:
  - Clock do Sistema
  - Contador Lógico
- Para garantir esse comportamento, o protocolo mantém para cada dado  $Q$  dois valores timestamp:
  - $W\text{-timestamp}(Q)$  é o maior timestamp de qualquer transação que executou  $\text{write}(Q)$  com sucesso.
  - $R\text{-timestamp}(Q)$  é a maior timestamp de qualquer transação que executou  $\text{read}(Q)$  com sucesso.

# Protocolos baseados em Timestamp

- O protocolo de ordenação por timestamp garante que quaisquer operações read e write em conflito sejam executadas por ordem de timestamp.
  - Suponha que uma transação  $T_i$  emita um  $\text{read}(Q)$
1. Se  $\text{TS}(T_i) < \text{W-timestamp}(Q)$ , então  $T_i$  precisa ler um valor de  $Q$  que já foi modificado. Logo, a operação read é rejeitada, e  $T_i$  é revertida.
  2. Se  $\text{TS}(T_i) > \text{W-timestamp}(Q)$ , então a operação read é executada, e  $\text{R-timestamp}(Q)$  é definido como o máximo de  $\text{R-timestamp}(Q)$  e  $\text{TS}(T_i)$ .

# Protocolos baseados em Timestamp

- Suponha que a transação  $T_i$  emita  $\text{write}(Q)$ .
  - Se  $\text{TS}(T_i) < R\text{-timestamp}(Q)$ , então o valor de  $Q$  que  $T_i$  está produzindo foi necessário anteriormente, e o sistema considerou que esse valor nunca seria produzido. Logo, a operação  $\text{write}$  é rejeitada, e  $T_i$  é revertida.
  - Se  $\text{TS}(T_i) < W\text{-timestamp}(Q)$ , então  $T_i$  está tentando escrever um valor obsoleto de  $Q$ . Logo, essa operação  $\text{write}$  é rejeitada, e  $T_i$  é revertida.
  - Caso contrário, a operação  $\text{write}$  é executada, e  $W\text{-timestamp}(Q)$  é definida como  $\text{TS}(T_i)$ .

# Exemplo de uso do protocolo

- Um escalonamento parcial para vários itens de dados para transações com timestamp 1, 2, 3, 4, 5

$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
read( $Y$ )	read( $Y$ )			read( $X$ )
read( $X$ )	read( $X$ ) abort	write( $Y$ ) write( $Z$ )		read( $Z$ )  write( $Y$ ) write( $Z$ )

## Exatidão do protocolo de ordenação por timestamp

- O protocolo de ordenação por timestamp garante a seriação, pois todos os arcos no grafo de precedência são da forma:



Assim, não haverá ciclos no gráfico de precedência

- O protocolo de timestamp garante liberdade de impasse, pois nenhuma transação precisa esperar.
- Mas o escalonamento pode não ser livre de cascata, e pode nem sequer ser recuperável.

# Facilidade de recuperação e liberdade de cascata

- Problema com protocolo de ordenação por timestamp:
  - Suponha que  $T_i$  aborte, mas  $T_j$  tenha lido um item de dados escrito por  $T_i$
  - Então,  $T_j$  precisa abortar; se  $T_j$  tivesse permitido o commit anterior, o escalonamento não seria recuperável.
  - Além do mais, qualquer transação que tenha lido um item de dados escrito por  $T_j$  precisa abortar
  - Isso pode levar ao rollback em cascata - ou seja, uma cadeia de rollbacks
- Solução:
  - Uma transação é estruturada de modo que suas escritas sejam todas realizadas no final de seu processamento
  - Todas as escritas de uma transação formam uma ação atômica; e nenhuma transação pode ser executada enquanto uma transação estiver sendo escrita
  - Uma transação que aborta é reiniciada com um novo timestamp

# Manuseio de Deadlock

- Considere as duas transações a seguir:

T1: write (X)	T2: write(Y)
write (Y)	write (X)

- Escalonamento com deadlock

$T_1$	$T_2$
<b>lock-X on X</b> write (X)	
	<b>lock-X on Y</b> write (Y) wait for <b>lock-X</b> on X

# Manuseio de Deadlock

- O sistema está em deadlock se existe um conjunto de transações tal que cada transação no conjunto está esperando por outra transação no conjunto.
- Protocolos de Prevenção de Deadlock garantem que o sistema nunca entrará em um estado de deadlock. Algumas estratégias de prevenção são:
  - Exigir que cada transação bloqueie todos os seus itens de dados antes de iniciar a execução (pré-declaração).
  - Impor ordenação parcial de todos os itens de dados e exigir que uma transação possa bloquear itens de dados somente na ordem especificada pela ordem parcial (protocolo baseado em grafos).

# Outras Estratégias de Prevenção de Deadlock

- Os esquemas a seguir usam apenas timestamps de transação para garantir a prevenção de deadlocks.
- esquema **wait-die** (esperar-morrer) - não preemptivo
  - Transações mais antigas podem esperar que transações mais novas liberem itens de dados. Transações mais novas nunca esperam pelas mais antigas. Em vez disso, elas sofrem rollback.
  - Uma transação pode morrer várias vezes antes de adquirir um item de dado.
- esquema **wound-wait** (ferir-esperar) - preemptivo
  - Transações mais antigas ferem (forçam o rollback) de transações mais novas em vez de esperar por elas. Transações mais novas podem esperar pelas mais antigas.
  - Pode ocorrer menos rollbacks que o esquema wait-die.

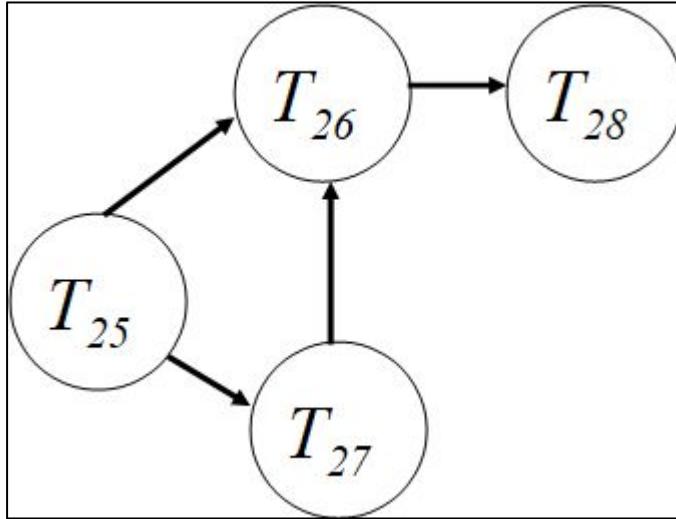
# Prevenção de Deadlock (Cont.)

- Em ambos os esquemas wait-die e wound-wait, uma transação que sofreu rollback é reiniciada com o seu timestamp original. Assim, transações mais antigas têm precedência sobre as mais novas, e a inanição é evitada.
- Esquemas com base em Timeout:
  - Uma transação espera por um bloqueio somente por uma quantidade especificada de tempo. Após esse tempo, ocorre um timeout e a transação sofre rollback.
  - Assim, deadlocks são impossíveis.
  - Simples de implementar, mas pode ocorrer inanição. Também é difícil de determinar um valor ideal para o intervalo de timeout.

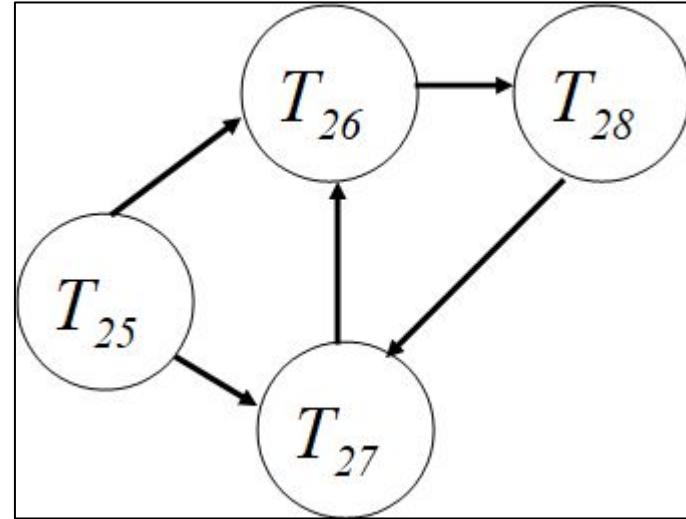
# Detecção de Deadlock (Cont.)

- Deadlocks podem ser descritos como um grafo de espera (wait-for graph), que consiste de um par  $G = (V, E)$ ,
  - $V$  é um conjunto de vértices (todas as transações do sistema)
  - $E$  é um conjunto de arestas; cada elemento é um par ordenado  $T_i \rightarrow T_j$ .
- Se  $T_i \rightarrow T_j$  está em  $E$ , então existe uma aresta direcionada de  $T_i$  para  $T_j$ , indicando que  $T_i$  está esperando que  $T_j$  libere um item.
- Quando  $T_i$  solicita um item de dado que está sendo mantido por  $T_j$ , então uma aresta  $T_i \rightarrow T_j$  é inserida no grafo de espera. Esta aresta será removida somente quando  $T_j$  não mais estiver mantendo um item de dado exigido por  $T_i$ .
- O sistema está em um estado de deadlock se e somente se o grafo de espera tiver um ciclo. Deve-se invocar um algoritmo de detecção de deadlocks periodicamente para procurar por ciclos.

# Detecção de Deadlock (Cont.)



Grafo de espera sem ciclo



Grafo de espera com ciclo

# Recuperação de Deadlocks

➤ Quando um deadlock é detectado:

- Alguma transação terá que sofrer rollback ("fazer uma vítima") para quebrar o deadlock. Escolher como vítima a transação que irá incorrer em um custo mínimo.
- Rollback - determinar quanto da transação deve ser desfeito Rollback total: Abortar a transação e então reiniciá-la.
- O mais eficiente é fazer o rollback da transação somente o necessário para quebrar o deadlock.
- Ocorrerá a inanição se a mesma transação for escolhida sempre como vítima. Incluir o número de rollbacks como fator de custo para evitar inanição.

# Banco de Dados

**Prof. Dr. Ronaldo Celso Messias Correia**  
[ronaldo.correia@unesp.br](mailto:ronaldo.correia@unesp.br)



UNIVERSIDADE ESTADUAL PAULISTA  
“JÚLIO DE MESQUITA FILHO”



# Sistema de Recuperação

# Introdução

- Um Sistema Computador está sujeito a falhas:
  - Quebra de disco, falha de energia, erro de software, fogo na sala de equipamento, sabotagem, etc.
- Em cada um destes casos, dados podem ser perdidos
- O sistema de banco de dados deve precaver-se para garantir que as propriedades de atomicidade e durabilidade de uma transação sejam preservadas
- Uma parte integrante de um sistema de banco de dados é o esquema de recuperação que é responsável pela restauração do banco de dados para um estado consistente que havia antes da ocorrência da falha

# Classificação de falha

- O tipo de falha mais simples – não resulta na perda da informação. Falhas mais difíceis de tratar – resultam em perda de informação.
- Falha de transação:
  - **Erros lógicos** : a transação não pode completar devido a alguma condição de erro interna
  - **Erros do sistema** : o sistema de banco de dados precisa terminar uma transação ativa devido a uma condição de erro (por exemplo, deadlock)
- **Falha do sistema** : uma falta de energia ou outra falha do hardware ou software faz com que o sistema falhe.
  - Suposição falhar-parar: conteúdo do armazenamento não volátil é considerado como não sendo corrompido por falha do sistema
    - Sistemas de banco de dados possuem diversas verificações de integridade para impedir adulteração de dados do disco
- **Falha do disco** : uma falha de cabeça ou falha de disco semelhante destrói todo ou parte do armazenamento de disco
  - A destruição é considerada como detectável: unidades de disco usam somas de verificação para detectar falhas

# Algoritmos de recuperação

- Algoritmos de recuperação são técnicas para garantir a consistência do banco de dados e a atomicidade e durabilidade da transação apesar das falhas.
- Algoritmos de recuperação têm duas partes:
  - Ações tomadas durante o processamento normal da transação para garantir que existem informações suficiente para recuperação de falhas
  - Ações tomadas após uma falha para recuperar o conteúdo do banco de dados a um estado que garante atomicidade, consistência e durabilidade

# Estrutura de armazenamento

- Armazenamento volátil:
  - não sobrevive a falhas do sistema
  - exemplos: memória principal, memória cache
- Armazenamento não volátil:
  - sobrevive a falhas do sistema
  - exemplos: disco, fita, memória flash, RAM não-volátil (alimentada por bateria)
- Armazenamento estável:
  - a informação residente em armazenamento estável nunca é perdida.  
Uma forma mítica de armazenamento que sobrevive a todas as falhas
  - aproximado mantendo-se várias cópias em meios não voláteis distintos



# Implementação do armazenamento estável

- Mantenha várias cópias de cada bloco em discos separados
  - as cópias podem estar em sites remotos para proteger contra desastres como incêndio ou inundação
- A falha durante a transferência de dados ainda pode resultar em cópias inconsistentes: a transferência em bloco pode resultar em
  - Término bem-sucedido
  - Falha parcial: bloco de destino possui informações incorretas
  - Falha total: bloco de destino nunca foi atualizado
- Proteção do meio de armazenamento contra falha durante a transferência de dados (uma solução):
  - Executar a operação de saída da seguinte forma (considerando duas cópias de cada bloco):
    - Escreva a informação no primeiro bloco físico.
    - Quando a primeira escrita terminar com sucesso, escreva a mesma informação no segundo bloco físico.
    - A saída só é completada depois que a segunda escrita for completada com sucesso.

# Implementação de armazenamento estável (cont.)

- Proteção do meio de armazenamento contra falha durante a transferência de dados (cont.):
- As cópias de um bloco podem diferir devido a falhas durante a operação de saída. Para recuperar-se da falha:
  - Primeiro, encontre blocos inconsistentes:
    - Solução simples: Compare as duas cópias de cada bloco do disco.
    - Solução melhor:
      - Registre as escritas em disco em andamento no armazenamento não volátil (RAM não volátil ou área especial do disco).
      - Use essa informação durante a recuperação para encontrar blocos que podem ser inconsistentes, e só compare cópias destes.
      - Usada em sistemas RAID de hardware
  - Se uma cópia de um bloco inconsistente for detectada com um erro (soma de verificação defeituosa), escreva a outra cópia em cima dela. Se ambos não têm erro, mas forem diferentes, escreva o primeiro bloco sobre o segundo.

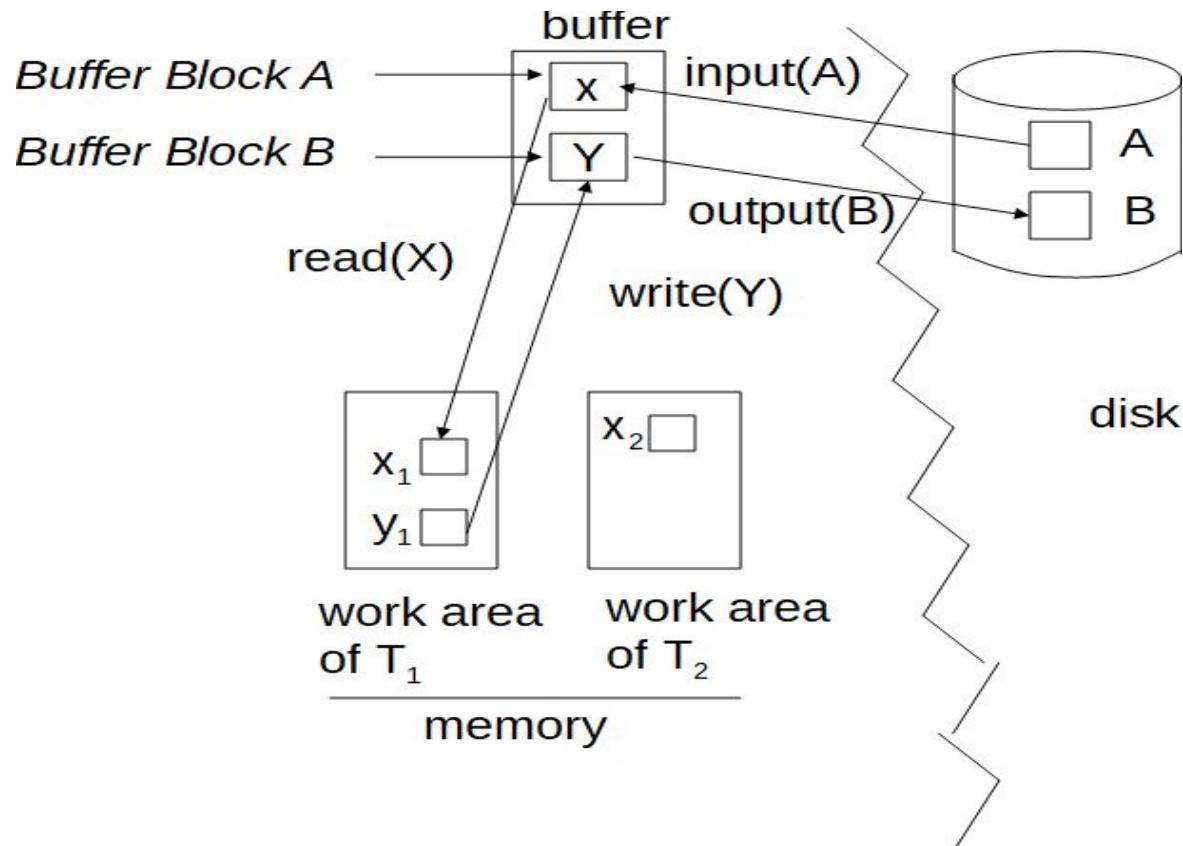
# Acesso aos dados

- Blocos físicos são aqueles blocos residindo no disco.
- Blocos de buffer são os blocos residindo temporariamente na memória principal.
- Movimentos de bloco entre disco e a memória principal são iniciados por meio das duas operações a seguir:
  - $\text{input}(B)$  transfere o bloco físico B para a memória principal.
  - $\text{output}(B)$  transfere o bloco de buffer B para o disco, e substitui o bloco físico apropriado lá.
- Cada transação  $T_i$  possui sua área de trabalho privada, onde são mantidas as cópias locais de todos os itens de dados acessados e atualizados por ela.
  - A cópia local de  $T_i$  de um item de dados X é chamada de  $x_i$ .
- Consideramos, por simplicidade, que cada item de dados cabe e é armazenado dentro de um único bloco.

# Acesso aos dados (cont.)

- A transação transfere itens de dados entre os blocos de buffer do sistema e sua área de trabalho privada usando as seguintes operações:
  - $\text{read}(X)$  atribui o valor do item de dados  $X$  à variável local  $xi$ .
  - $\text{write}(X)$  atribui o valor da variável local  $xi$  ao item de dados  $\{X\}$  no bloco de buffer.
  - esses dois comandos podem precisar da emissão de uma instrução  $\text{input}(BX)$  antes da atribuição, se o bloco  $BX$  em que  $X$  reside ainda não estiver na memória.
- Transações
  - Execute  $\text{read}(X)$  enquanto acessa  $X$  pela primeira vez;
  - Todos os acessos subsequentes são para a cópia local.
  - Após o último acesso, a transação executa  $\text{write}(X)$ .
- $\text{output}(BX)$  não precisa vir imediatamente após  $\text{write}(X)$ . O sistema pode realizar a operação  $\text{output}$  quando julgar necessário.

# Exemplo de acesso aos dados



# Recuperação e atomicidade

- Modificar o banco de dados sem garantir que a transação será confirmada pode levar o banco de dados a um estado inconsistente.
- Considere a transação Ti que transfere \$50 da conta A para a conta B; o objetivo é realizar todas as modificações do banco de dados feitas por Ti ou nenhuma delas.
- Várias operações de saída podem ser exigidas para Ti (para gerar A e B). Uma falha pode ocorrer após uma dessas modificações ter sido feita, mas antes que todas elas sejam feitas.

## Recuperação e atomicidade (cont.)

- Para garantir a atomicidade apesar das falhas, primeiro geramos informações descrevendo as modificações no armazenamento estável sem modificar o próprio banco de dados.
- Estudamos duas técnicas:
  - recuperação baseada em log, e
  - paginação de sombra
- Consideramos (inicialmente) que as transações serão executadas em série, ou seja, uma após a outra.

# Recuperação baseada em log

- Um log é mantido no armazenamento estável.
  - O log é uma sequência de registros de log, e mantém um registro das atividades de atualização no banco de dados.
- Quando a transação  $T_i$  inicia, ela se registra escrevendo um registro de log  $\langle T_i \text{ start} \rangle$
- Antes que  $T_i$  execute  $\text{write}(X)$ , um registro de log  $\langle T_i, X, V_1, V_2 \rangle$  é escrito, onde  $V_1$  é o valor de  $X$  antes do write, e  $V_2$  é o valor a ser escrito em  $X$ .
  - O registro de log observa que  $T_i$  realizou uma escrita no item de dados  $X_j$ .  $X_j$  tinha o valor  $V_1$  antes da escrita, e terá o valor  $V_2$  após a escrita.
- Quando  $T_i$  termina sua última instrução, o registro de log  $\langle T_i \text{ commit} \rangle$  é escrito.
- Consideramos, por enquanto, que os registros de log são escritos diretamente no armazenamento estável (ou seja, eles não são mantidos em buffer)
- Duas técnicas usando logs
  - Modificação de banco de dados adiada
  - Modificação de banco de dados imediata

# Modificação de banco de dados adiada

- O esquema de modificação de banco de dados adiada registra todas as modificações no log, mas adia todas as escritas para depois da confirmação parcial.
- Suponha que as transações são executadas serialmente
- A transação começa escrevendo o registro  $\langle Ti \text{ start} \rangle$  no log.
- Uma operação  $\text{write}(X)$  resulta em um registro de log  $\langle Ti, X, V \rangle$  sendo escrito, onde  $V$  é o novo valor para  $X$ 
  - Nota: o valor antigo não é necessário para esse esquema
- A escrita não é realizada em  $X$  neste momento, mas é adiada.
- Quando  $Ti$  confirma parcialmente,  $\langle Ti \text{ commit} \rangle$  é escrito no log
- Finalmente, os registros de log são lidos e usados para realmente executar as escritas previamente adiadas.

# Modificação de banco de dados adiada (cont.)

- Durante a recuperação após uma falha, uma transação precisa ser refeita se e somente se tanto  $\langle Ti \text{ start} \rangle$  quanto  $\langle Ti \text{ commit} \rangle$  existirem no log.
- Refazer uma transação  $Ti$  (redo  $Ti$ ) define o valor de todos os itens de dados atualizado pela transação como os novos valores.
- Falhas podem ocorrer enquanto
  - a transação estiver executando as atualizações originais, ou
  - enquanto a ação de recuperação estiver sendo tomada
- transações de exemplo T0 e T1 (T0 executa antes de T1):

T0: read (A)

A: - A - 50

Write (A)

read (B)

B:- B + 50

write (B)

T1 : read (C)

C:- C- 100

write (C)

# Modificação de banco de dados adiada (cont.)

- A seguir mostramos o log conforme aparece em três instâncias de tempo.

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$
$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$	
$\langle T_1, C, 600 \rangle$	$\langle T_1, C, 600 \rangle$	
		$\langle T_1 \text{ commit} \rangle$
(a)	(b)	(c)

- Se o log no armazenamento estável no momento da falha for como neste caso:
- (a) Nenhuma ação de rede precisa ser tomada
  - (b) redo( $T_0$ ) precisa ser realizado, pois  $\langle T_0 \text{ commit} \rangle$  está presente
  - (c) redo( $T_0$ ) precisa ser realizado seguido por redo( $T_1$ ), pois  $\langle T_0 \text{ commit} \rangle$  e  $\langle T_1 \text{ commit} \rangle$  estão presentes

# Modificação de banco de dados imediata

- O esquema de modificação de banco de dados imediata permite que atualizações de banco de dados de uma transação não confirmada sejam feitas enquanto as escritas são emitidas
  - como pode ser preciso desfazer, os logs de atualização precisam ter valor antigo e valor novo
- O registro de log de atualização precisa ser escrito antes que o item do banco de dados seja escrito
  - Consideramos que o registro de log é enviado diretamente ao armazenamento estável
  - Pode ser estendido para adiar a saída do registro de log, desde que, antes da execução de uma operação output(B) para um bloco de dados B, todos os registros de log correspondentes aos itens B sejam esvaziados para o armazenamento estável
- A saída dos blocos atualizados pode ocorrer a qualquer momento antes ou depois do commit da transação
- A ordem em que os blocos são enviados pode ser diferente da ordem em que são escritos.

# Exemplo de modificação de banco de dados imediata

Log	Write	Output
-----	-------	--------

<T0 start>

<T0, A, 1000, 950>

To, B, 2000, 2050

A = 950

B = 2050

<T0 commit>

<T1 start>

<T1, C, 700, 600>

C = 600

BB, BC

<T1 commit>

BA

Nota: BX indica bloco contendo X.

# Modificação de banco de dados imediata (cont.)

- O procedimento de recuperação possui duas operações em vez de uma:
  - undo(Ti) restaura o valor de todos os itens de dados atualizados por Ti aos seus valores antigos, indo para trás a partir do último registro para Ti
  - redo(Ti) define o valor de todos os itens de dados atualizados por Ti aos novos valores, indo para frente a partir do primeiro registro para Ti
- As duas operações precisam ser idempotentes
  - Ou seja, mesmo que a operação seja executada várias vezes, o efeito é o mesmo que se fosse executada uma vez
    - Necessário porque as operações podem ser novamente executadas durante a recuperação
- Ao recuperar-se após a falha:
  - A transação Ti precisa ser desfeita (undo) se o log tiver o registro <Ti start>, mas não contém o registro <Ti commit>.
  - A transação Ti precisa ser refeita (redo) se o log tiver o registro <Ti start> e o registro <Ti commit>.
- Operações de undo são realizadas primeiro, depois as operações de redo.

# Exemplo de recuperação de modificação de BD imediata

- A seguir mostramos o log conforme aparece em três instâncias de tempo.

< $T_0$  start>

< $T_0$ , A, 1000, 950>

< $T_0$ , B, 2000, 2050>

< $T_0$  start>

< $T_0$ , A, 1000, 950>

< $T_0$ , B, 2000, 2050>

< $T_0$  start>

< $T_0$ , A, 1000, 950>

< $T_0$ , B, 2000, 2050>

(a)

(b)

(c)

< $T_0$  commit>

< $T_1$  start>

< $T_1$ , C, 700, 600>

< $T_0$  commit>

< $T_1$  start>

< $T_1$ , C, 700, 600>

< $T_1$  commit>

- As ações de recuperação em cada um destes são:

- (a) undo ( $T_0$ ): B é restaurado para 2000 e A para 1000.
- (b) undo ( $T_1$ ) e redo ( $T_0$ ): C é restaurado para 700, e depois A e B são definidos para 950 e 2050, respectivamente.
- (c) redo ( $T_0$ ) e redo ( $T_1$ ): A e B são definidos para 950 e 2050 respectivamente. Depois, C é definido para 600

# Pontos de verificação

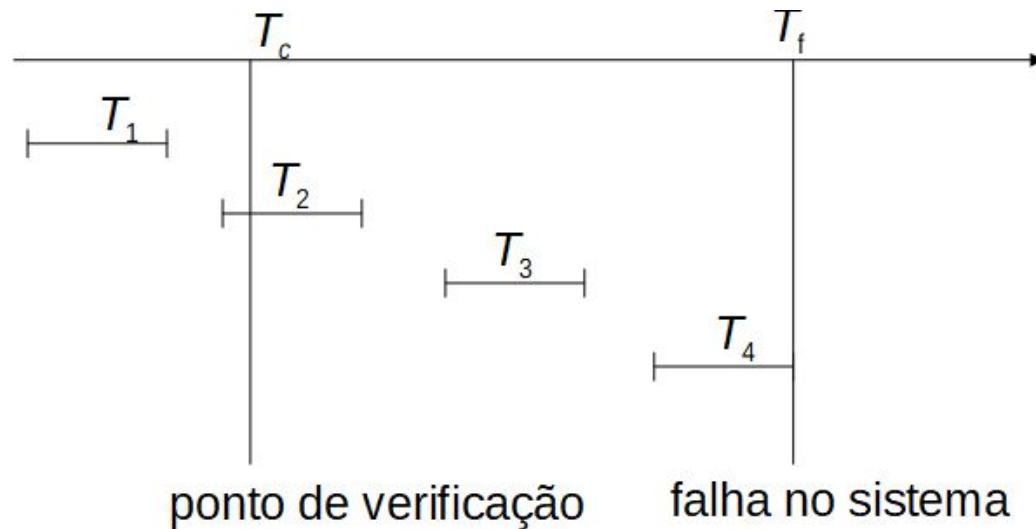
- Problemas no procedimento de recuperação, conforme discutimos:
  - pesquisar o log inteiro é demorado
  - poderíamos desnecessariamente refazer transações que já emitiram sua saída no banco de dados.
- Facilite o procedimento de recuperação realizando periodicamente o ponto de verificação
  - Envie todos os registros de log atualmente residindo na memória principal para o armazenamento estável.
  - Envie todos os blocos de buffer modificados para o disco.
  - Escreva um registro de log <checkpoint> no armazenamento estável.

## Pontos de verificação (cont.)

- Durante a recuperação, temos que considerar apenas a transação mais recente  $T_i$  que foi iniciada antes do ponto de verificação, e as transações que começaram após  $T_i$ .
  - Varra para trás a partir do final do log para encontrar o registro <checkpoint> mais recente.
  - Continue varrendo para trás até um registro < $T_i$  start> ser encontrado.
  - Só precisa considerar a parte do log vindo após o registro start. A parte inicial do log pode ser ignorada durante a recuperação, e pode ser apagada sempre que for desejado.
  - Para todas as transações (começando de  $T_i$  ou mais) sem < $T_i$  commit>, execute undo( $T_i$ ). (Feito apenas no caso de modificação imediata.)
  - Varrendo para frente no log, para todas as transações começando a partir de  $T_i$  ou depois com um < $T_i$  commit>, execute redo( $T_i$ ).

# Exemplo de pontos de verificação

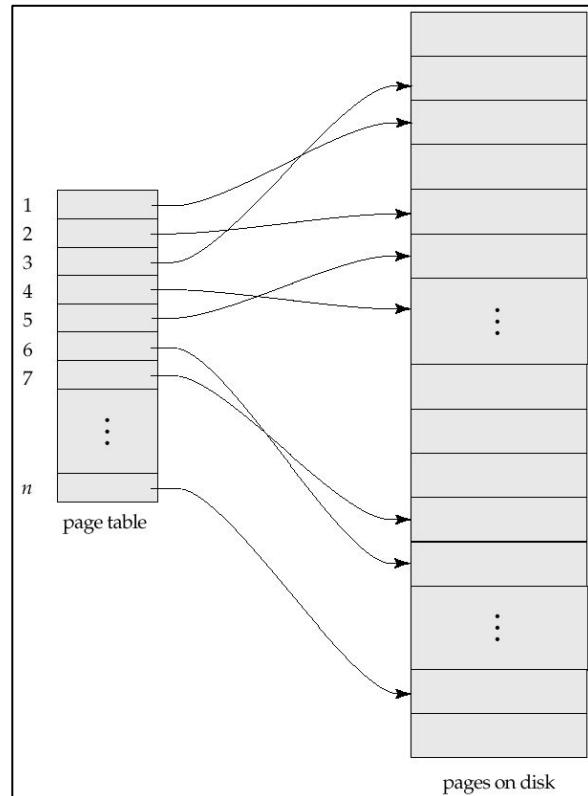
- T1 pode ser ignorada (atualizações já enviadas ao disco devido ao ponto de verificação)
  - T2 e T3 refeitos
  - T4 refeito



# Paginação de sombra

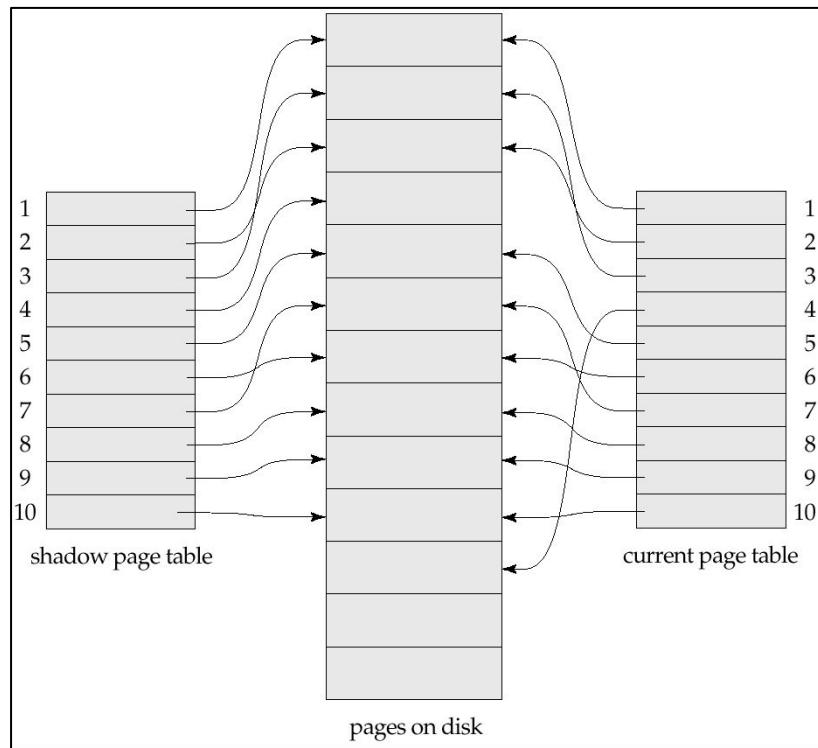
- A paginação de sombra é uma alternativa à recuperação baseada em log; esse esquema é útil se as transações forem executadas em série
- Idéia: manter duas tabelas de página durante o tempo de vida de uma transação - a tabela de página atual, e a tabela de página de sombra
- Armazene a tabela de página de sombra no armazenamento não volátil, de modo que o estado do banco de dados antes da execução da transação possa ser recuperado.
  - A tabela de página de sombra nunca é modificada durante a execução
- Para começar, as duas tabelas de página são idênticas. Somente a tabela de página atual é usada para os acessos ao item de dados durante a execução da transação.
- Sempre que qualquer página estiver para ser escrita pela primeira vez
  - Uma cópia dessa página é feita em uma página não usada.
  - A tabela de página atual aponta para a cópia
  - A atualização é realizada na cópia

# Exemplo de tabela de página



# Exemplo de paginação de sombra

- Tabelas de sombra e página atual após escrita na página 4



# Paginação de sombra (cont.)

- Para confirmar uma transação:
  - 1. Esvazie todas as páginas modificadas na memória principal para o disco
  - 2. Envie a tabela de página atual para o disco
  - 3. Torne a tabela de página atual a nova tabela de página de sombra, da seguinte maneira:
    - mantenha um ponteiro para a tabela de página de sombra em um local fixo (conhecido) no disco.
    - para tornar a tabela de página atual a nova tabela de página de sombra, basta atualizar o ponteiro para apontar para a tabela de página atual no disco
- Quando o ponteiro para a tabela de página de sombra tiver sido escrito, a transação está confirmada.
- Nenhuma recuperação é necessária após uma falha - novas transações podem começar imediatamente, usando a tabela de página de sombra.
- As páginas não apontada de/para a tabela de página atual/sombra devem ser liberadas (com coleta de lixo).

# Paginação de sombra (cont.)

- Vantagens da página de sombra em relação aos esquemas baseados em log
  - a recuperação é trivial
  - nenhuma sobrecarga de escrita de registros de log
- Desvantagens:
  - Copiar a tabela de página inteira é muito dispendioso
    - Pode ser reduzido usando uma tabela de página estruturada como uma árvore B+
      - Não é preciso copiar árvore inteira, somente os caminhos na árvore que levam a nós de folha atualizados
  - A sobrecarga do commit é alta mesmo com a extensão acima
    - Precisa esvaziar cada página atualizada, e tabela de página
  - Os dados são fragmentados (páginas relacionadas ficam separadas no disco)
  - Após o término de cada transação, as páginas do banco de dados contendo versões antigas de dados modificados precisam passar pela coleta de lixo
  - Difícil de estender o algoritmo para permitir que transações sejam executadas simultaneamente
    - Mais fácil para estender esquemas baseados em log

# Recuperação com transações concorrentes

- Modificamos os esquemas de recuperação baseados em log para permitir que várias transações sejam executadas simultaneamente.
  - Todas as transações compartilham um único buffer de disco e um único log
  - Um bloco de buffer pode ter itens de dados atualizados por uma ou mais transações
- Consideramos o controle de concorrência usando um bloqueio estrito em duas fases;
  - ou seja, as atualizações de transações não confirmadas não devem ser visíveis a outras transações
    - Caso contrário, como realizar o undo se T1 atualiza A, depois T2 atualiza A e confirma, e finalmente T1 precisa abortar?
- O logging é feito conforme descrevemos anteriormente.
  - Os registros de log de diferentes transações podem ser intercalados no log.
- A técnica de ponto de verificação e as ações tomadas na recuperação precisam ser alteradas
  - pois várias transações podem estar ativas quando um ponto de verificação é realizado.

# Recuperação com transações concorrentes (cont.)

- Os pontos de verificação são realizados como antes, exceto que o registro de log do ponto de verificação agora tem a forma <checkpoint L> onde L é a lista de transações ativas no momento do ponto de verificação
  - Consideramos que nenhuma atualização está em andamento enquanto o ponto de verificação é executado (isso será aliviado mais tarde)
- Quando o sistema se recupera de uma falha, ele primeiro faz o seguinte:
  - Inicializa a lista de undo e lista de redo para vazio
  - Varre o log para trás a partir do fim, parando quando o primeiro registro <checkpoint L> for encontrado. Para cada registro encontrado durante a varredura:
    - se o registro for <Ti commit>, acrescenta Ti à lista de redo
    - se o registro for <Ti start>, então se Ti não está na lista de redo, acrescenta Ti à lista de undo
  - Para cada Ti em L, se Ti não estiver na lista de redo, acrescenta Ti à lista de undo

# Recuperação com transações concorrentes (cont.)

- Neste ponto, lista de undo consiste em transações incompletas, que precisam ser desfeitas, e lista de redo consiste em transações acabadas, que precisam ser refeitas.
- A recuperação agora continua da seguinte forma:
  - Varra o log para trás a partir do registro mais recente, parando quando registros  $\langle Ti \text{ start} \rangle$  tiverem sido encontrados para cada  $T_i$  na lista de undo.
    - Durante a varredura, realize undo para cada registro de log que pertence a uma transação na lista de undo.
  - Localize o registro  $\langle \text{checkpoint L} \rangle$  mais recente.
  - Varra o log para frente a partir do registro  $\langle \text{checkpoint L} \rangle$  até o final do log.
    - Durante a varredura, realize redo para cada registro de log que pertence a uma transação na lista de redo.

# Exemplo de recuperação

- Percorra as etapas do algoritmo de recuperação no log a seguir:

```
>><T0 start>
>><T0, A, 0, 10>
>><T0 commit>
>><T1 start>
>><T1, B, 0, 10>
>><T2 start>          /* Varredura na etapa 4 pára aqui */
>><T2, C, 0, 10>
>><T2, C, 10, 20>
>><checkpoint {T1, T2}>
>><T3 start>
>><T3, A, 10, 20>
>><T3, D, 0, 10>
>><T3 commit>
```

# Buffering de registro de log

- Buffering de registro de log: os registros de log são mantidos na memória principal, em vez de serem enviados diretamente para o armazenamento estável.
  - Registros de log são enviados ao armazenamento estável quando um bloco de registros de log no buffer estiver cheio, ou uma operação de log forçado for executada.
- O log forçado é realizado para confirmar uma transação forçando todos os seus registros de log (incluindo o registro de commit) para o armazenamento estável.
- Vários registros de log, portanto, podem ser enviados por meio de uma única operação de saída, reduzindo o custo da E/S.

# Buffering de registro de log (cont.)

- As regras a seguir precisam ser seguidas se os registros de log forem colocados em buffer:
  - Os registros de log são enviados para o armazenamento estável na ordem em que são criados.
  - A transação  $T_i$  só entra no estado de commit quando o registro de log  $\langle T_i \text{ commit} \rangle$  tiver sido enviado ao armazenamento estável.
  - Antes que um bloco de dados na memória principal seja enviado ao banco de dados, todos os registros de log pertencentes aos dados nesse bloco precisam ter sido enviados ao armazenamento estável.
    - Essa regra é chamada logging de escrita antecipada ou regra WAL (Write Ahead Logging)
      - Estritamente falando, WAL só requer que informações de undo sejam enviadas

# Buffering de banco de dados

- O banco de dados mantém um buffer na memória dos blocos de dados
  - Quando um novo bloco é necessário, se o buffer estiver cheio, um bloco existente precisa ser removido do buffer
  - Se o bloco escolhido para remoção tiver sido atualizado, ele terá que ser enviado para o disco
- Como resultado da regra de logging de escrita antecipada, se um bloco com atualizações não confirmadas for enviado ao disco, os registros de log com informações de undo para as atualizações são enviados ao log no armazenamento estável primeiro.
- Nenhuma atualização deve estar em progresso em um bloco quando for enviada ao disco. Pode ser garantido da seguinte forma.
  - Antes de escrever um item de dados, a transação adquire bloqueio exclusivo sobre o bloco contendo o item de dados
  - O bloqueio pode ser liberado quando a escrita terminar.
    - Tais bloqueios mantidos por uma curta duração software chamados de latches.
  - Antes que um bloco seja enviado ao disco, o sistema adquire um latch exclusivo sobre o bloco
    - Garante que nenhuma atualização pode estar em andamento no bloco

# Gerenciamento de buffer (cont.)

- O buffer de banco de dados pode ser implementado
  - em uma área de memória principal real reservada para o banco de dados, ou
  - na memória virtual
- Implementar o buffer na memória principal reservada tem desvantagens:
- A memória é particionada de antemão entre o buffer de banco de dados e as aplicações, limitando a flexibilidade.
  - As necessidades podem mudar, e embora o sistema operacional saiba melhor como a memória deve ser dividida a qualquer momento, ele não pode mudar o particionamento da memória.

# Gerenciamento de buffer (cont.)

- Os buffers de banco de dados geralmente são implementados na memória virtual apesar de algumas desvantagens:
  - Quando o sistema operacional precisa expulsar uma página que foi modificada, para criar espaço para outra página, a página é escrita no espaço de swap no disco.
  - Quando o banco de dados decide escrever a página de buffer no disco, a página de buffer pode estar no espaço de swap e pode ter que ser lida do espaço de swap no disco e enviada ao banco de dados no disco, resultando em E/S extra!
    - Conhecido como problema de paginação dual.
  - O ideal é que, quando houver um swap de uma página de buffer do banco de dados, o sistema operacional passe o controle ao banco de dados, que por sua vez envia a página ao banco de dados, ao invés do espaço de swap (cuidado para enviar primeiro os registros de log)
    - Assim, a paginação dual pode ser evitada, mas os sistemas operacionais comuns não admitem tal funcionalidade.

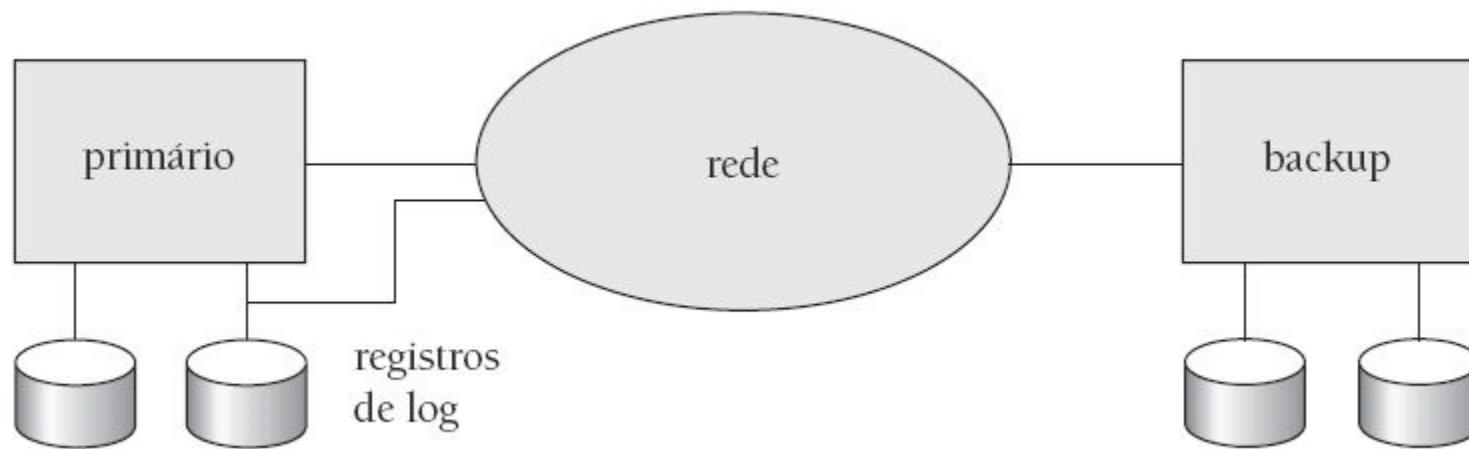
# Falha com perda de armazenamento não volátil

- Até aqui, não assumimos perda de armazenamento não volátil
- Técnica semelhante ao ponto de verificação usada para lidar com perda de armazenamento não volátil
  - Periodicamente faz o dump do conteúdo inteiro do banco de dados para armazenamento estável
  - Nenhuma transação pode estar ativa durante o procedimento de dump; um procedimento semelhante ao ponto de verificação precisa ocorrer
    - Envie todos os registros de log atualmente residindo na memória principal para o armazenamento estável.
    - Envie todos os blocos de buffer para o disco.
    - Copie o conteúdo do banco de dados para o armazenamento estável.
    - Envie um registro <dump> para log no armazenamento estável.
  - Para se recuperar da falha de disco
    - restaurar o banco de dados do dump mais recente
    - Consulte o log e refaça todas as transações que foram confirmadas após o dump
- Pode ser estendido para permitir que as transações estejam ativas durante o dump; conhecido como dump difuso ou dump online
  - Estudaremos o ponto de verificação difuso mais adiante

# Sistemas de backup remoto

# Sistemas de backup remoto

- Os sistemas de backup remoto oferecem alta disponibilidade, permitindo que o processamento de transação continue mesmo que o site primário seja destruído.



# Sistemas de backup remoto (cont.)

- Detecção de falha: o site de backup precisa detectar quando o site primário falhou
  - para distinguir a falha do site primário da falha do enlace, mantenha vários links de comunicação entre o site primário e o backup remoto.
- Transferência de controle:
  - Para assumir o controle, o site de backup primeiro realiza a recuperação usando sua cópia do banco de dados e todos os registros de log que recebeu do site primário.
    - Assim, transações completadas são refeitas e transações incompletas são descartadas.
  - Quando um site de backup assume o processamento, ele se torna o novo site primário
  - Para transferir o controle de volta ao antigo primário quando se recuperar, o antigo primário precisa receber logs de redo do backup antigo e aplicar todas as atualizações localmente.

# Sistemas de backup remoto (cont.)

- Tempo para recuperação: Para reduzir o atraso na retomada, o site de backup periodicamente processa os registros de log de redo (com efeito, realizando a recuperação do estado anterior do banco de dados), realiza um ponto de verificação e pode então excluir partes mais antigas do log.
- A configuração hot-spare permite a retomada muito rápida:
  - O backup processa continuamente o registro de log de rede enquanto chega, aplicando as atualizações localmente.
  - Quando a falha do site primário é detectada, o backup reverte transações incompletas e está pronto para processar novas transações.
- Alternativa para o backup remoto: banco de dados distribuído com dados replicados
  - O backup remoto é mais rápido e mais barato, porém menos tolerante a falhas

# Sistemas de backup remoto (cont.)

- Garante a durabilidade das atualizações adiando o commit da transação até que a atualização seja registrada no backup; evite esse atraso permitindo menores graus de durabilidade.
- One-safe: confirme assim que o registro de log de commit da transação for escrito no site primário
  - Problema: as atualizações podem não chegar no backup antes que ele assuma.
- Two-very-safe: confirme quando o registro de log de commit da transação for escrito no site primário e no backup
  - Reduz a disponibilidade, pois as transações não podem confirmar se um site falhar.
- Two-safe: prossegue como no two-very-safe se o site primário e de backup estiverem ativos. Se apenas o primário estiver ativo, a transação confirma assim que seu registro de log de commit for escrito no primário.
  - Melhor disponibilidade do que two-very-safe; evita problema de transações perdidas no one-safe.