



Universidade Federal
de Campina Grande



Banco de Dados I

Unidade 10: Controle de Concorrência

Prof. Cláudio de Souza Baptista, Ph.D.
Laboratório de Sistemas de Informação – LSI
UFCG

Introdução

- Até então assumimos que nossas aplicações executam 'sozinhas' no SGBD.
- Porém, na realidade precisamos permitir múltiplos acessos num mesmo instante aos dados do BD, preservando a integridade dos dados.
Ex.: Sistema de reserva de passagens aéreas.
- Para garantir a integridade do BD é necessário usarmos o conceito de Transações 'serializáveis'

Transação

- é uma unidade lógica de trabalho;
- é um conjunto de operações que devem ser processadas como uma unidade;
- Uma transação é uma visão abstrata que o SGBD tem de um programa de usuário: uma sequência de leituras (reads) e escritas (writes).
- A execução concorrente de programas usuário é essencial para o bom desempenho de um SGBD.
 - Multi-processadores
 - Multithreading
 - I/O assíncrono (discos são lentos)

Concorrência em SGBD

- Usuários submetem transações e podem pensar que cada transação é executada sozinha.

e A concorrência é aplicada pelo SGBD, que intercala ações (leituras e escritas de objetos do BD) de várias transações.

Cada transação deve deixar a base de dados em um estado consistente se o BD está consistente quando a transação é iniciada.

O SGBD garantirá algumas restrições de integridade.

ele Além disso, o SGBD não compreende a semântica dos dados (i.e, não compreende como o saldo de uma conta bancária é calculado).

- Problemas:
 - Efeito de intercalar transações;
 - Quedas no sistema.

// Reserva de assento em um avião

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
    int voo;
```

```
    char data[10];
```

```
    char cadeira[3];
```

```
    int ocupado;
```

```
EXEC SQL END DECLARE SECTION;
```

```
void escolhaAssento() {
```

```
    printf("Digite vôo, data e cadeira desejados\n");
```

```
    scanf ("%d\n%s\n%s", &voo, data, cadeira);
```

```
    EXEC SQL select estaOcupado into :ocupado
```

```
        from Voos
```

```
        where numVoo = :voo and dataVoo = :data and
```

```
            cadeiraVoo = :cadeira;
```

```
    if (!ocupado) {
```

```
        EXEC SQL update voos
```

```
            set estaOcupado = 1
```

```
            where numVoo = :voo and dataVoo = :data and
```

```
                cadeiraVoo = :cadeira;
```

```
    }
```

```
    else
```

```
        printf("Cadeira não disponível\n");
```

```
}
```

Transação

- Lembre-se de que a função `escolhaAssento()` pode ser chamada por vários clientes.
- Suponha que dois agentes de viagem estão tentando reservar o mesmo assento no mesmo instante!

Suponha a seguinte ordem de execução:

1) Agente 1 encontra cadeira livre

2) Agente 2 encontra cadeira livre

3) Agente 1 coloca ocupado na cadeira

4) Agente2 coloca ocupado na cadeira

Quem realmente vai ficar com a cadeira?!?!?

Transação

- Transação deve obedecer às propriedades ACID:
 - **Atomicidade:** ou todo o trabalho é feito, ou nada é feito
 - **Consistência:** as regras de integridade são asseguradas
 - **Isolação:** tudo se parece como se ela executasse sozinha
 - **Durabilidade:** seus efeitos são permanentes mesmo em presença de falha

Se não usarmos o conceito de transação, poderemos ter problemas de consistência, que podem ou não ser tolerados dependendo da aplicação.

Problema 1: Leituras Sujas (Dirty reads)

- Suponha um BD de locadora de vídeos. Suponha que existam duas transações:
 - **T1** venda de um vídeo e **T2** inventário de estoque(ambas modificam a tabela vídeo)
- Suponha a seguinte execução

Tempo

- 1.T1.** Vende um dvd do filme ‘Titanic’ (havia 3 dvds então é modificado para 2)
- 2.T2** verifica o estoque do DVD ‘Titanic’ e lê o valor 2 DVDs
- 3.T1** O cliente não tem crédito (“estorou o cartão :-(“)
e desiste da compra => Volta a existir 3 DVDs
- 4.T2** Imprime um relatório ERRADO com a informação de que existem 2 DVDs de ‘Titanic’

Problema 1: Leituras Sujas (Dirty reads)

- Este nível de isolação é o mais flexível, portanto, ao usar este nível, devemos saber que poderemos ter informações erradas!
- O problema advém do fato de T2 ter lido um valor de T1 que ainda não havia sido confirmado (commit)

Problema 2: Leituras não repetíveis

- Quando uma transação lê mais de uma vez um mesmo dado e obtém valores diferentes

Suponha duas transações :

T1: lê duas vezes o número de DVDs do filme 'Titanic'
(Ex. Cliente pergunta quantas cópias tem, espera um pouco e depois pede 2 cópias (que necessitará uma re-leitura))

T2: Compra todos os DVDs do filme 'Titanic'

1. **T1** Consulta o número de cópias de Titanic e obtém 3
2. **T2** Compra todas as cópias de Titanic => cópias = 0
3. **T1** Consulta o número de cópias de Titanic e obtém 0!

Ou seja uma leitura diferente da anterior dentro da mesma transação

Problema 3: Atualizações Perdidas

Suponha duas transações que atualizam uma mesma conta corrente:

T3

1. Select saldo into sldtmp
from conta
Where num = 1500
3. SldTmp += 100
5. Update conta
set saldo = sldtmp
where num = 1500

T4

2. Select saldo into sldtmp
from conta
where num = 1500
4. SldTmp += 200
6. Update conta
set saldo = sldtmp
where num = 1500

A atualização de T3 foi perdida!!!

Concorrência: Escalonamento

- Considere duas transações bancárias:
T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END
- Intuitivamente, a primeira transação transfere \$100 da conta B para a conta A. A segunda acrescenta 6% nas duas contas. Suponha que as duas contas tenham R\$1.000 cada.
- Se as duas são submetidas juntas, não há garantias de que a transação T1 executará antes da transação T2, ou vice-versa. Entretanto, o efeito final deve ser equivalente ao efeito da execução serial, em alguma ordem, das duas transações.
- Um **escalonamento** (scheduling) é uma lista de ações de um conjunto de transações que sofreram commit. Representa uma sequência de execução que deve conservar a mesma ordem de execução das operações das transações presentes nele.

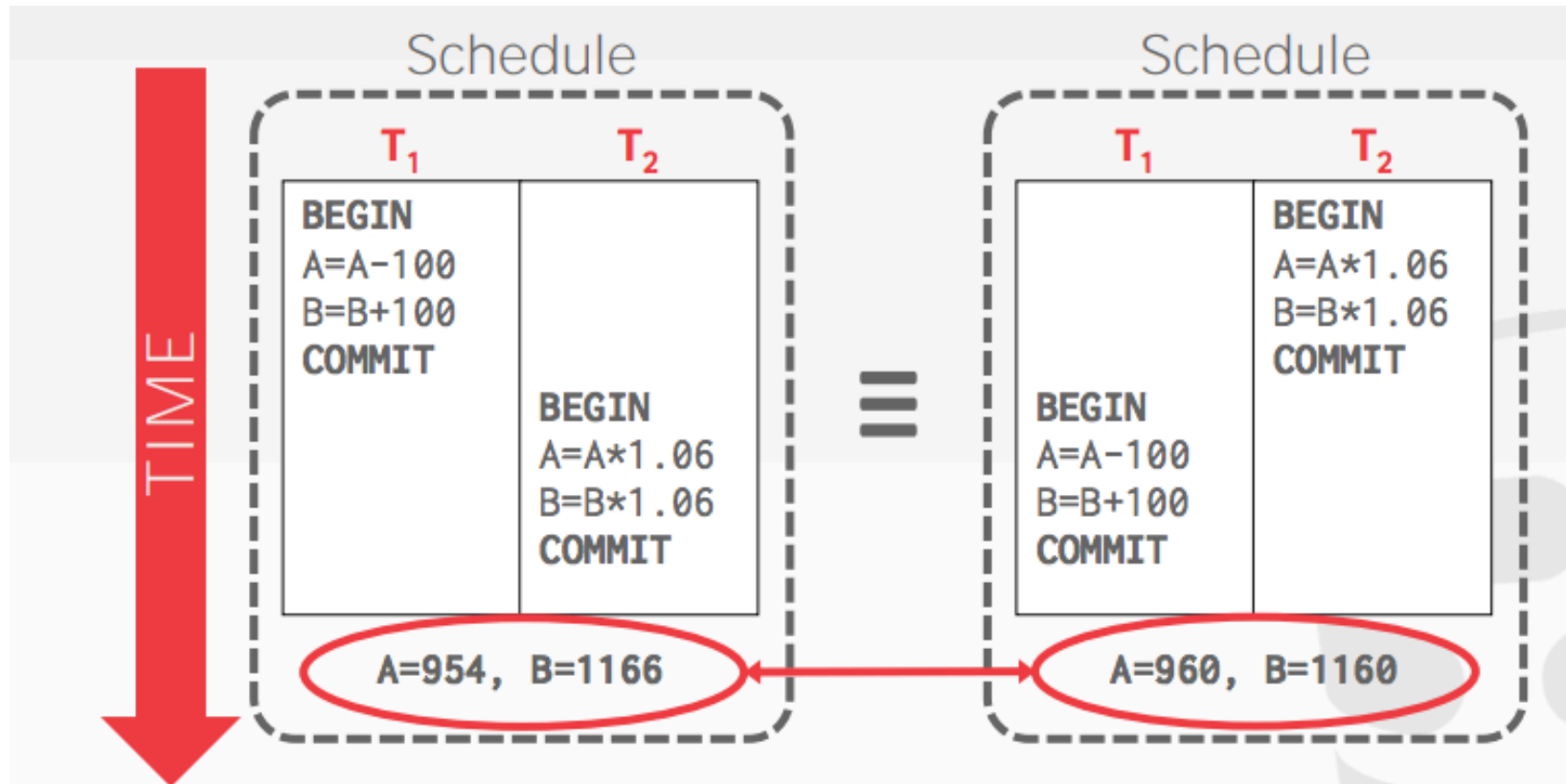


Escalonamento

- Serial: é aquele no qual as transações concorrentes são executadas uma após outra, ou seja, não há intercalação de operações de diferentes transações

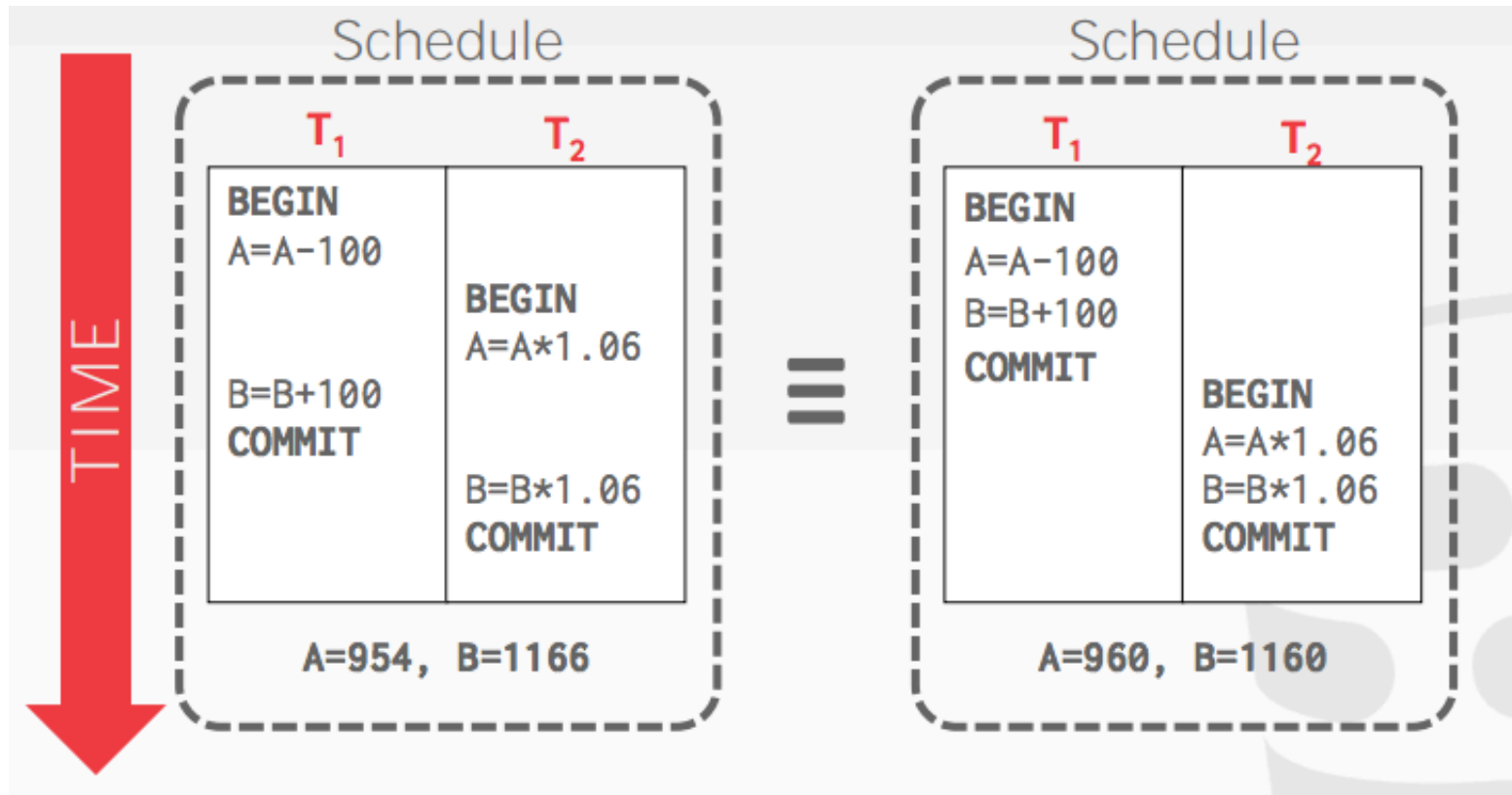
Ex.: Escalonamento Serial

A = 1000 e B = 1000



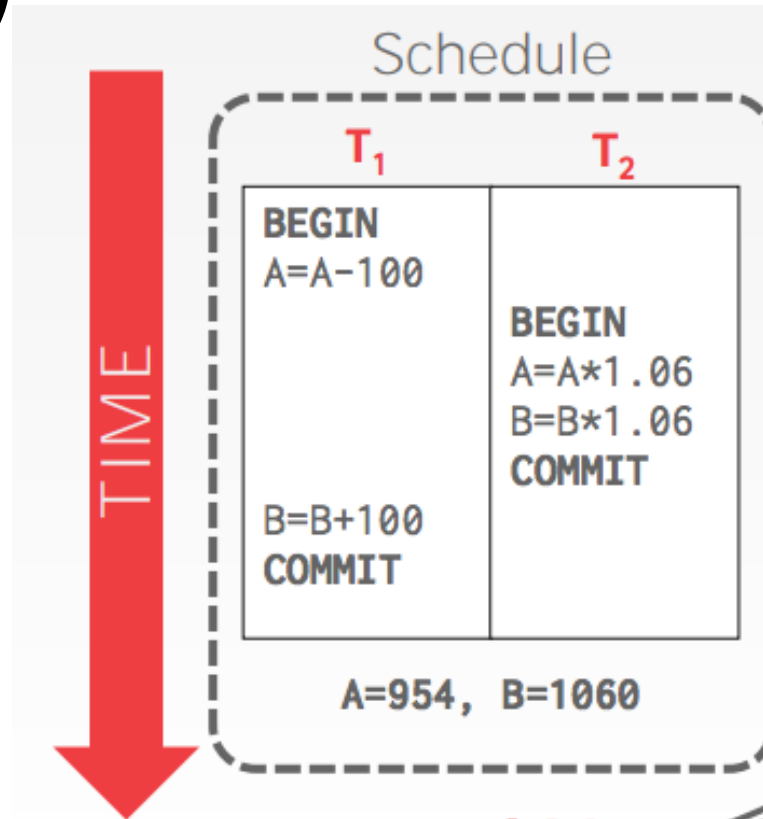
A+B = 2120

Ex.: Escalonamento Não Serial (correto)



$$A+B = \text{R\$}2120$$

Ex.: Escalonamento Não Serial (incorreto)



$A + B = R\$2014$
(desapareceram \$106!!!)



Escalonamento Serializável

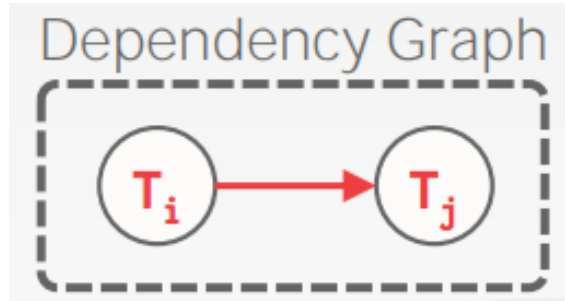
- Serializável: um conceito que garante que um conjunto de transações, quando executadas concorrentemente, produzirão um resultado equivalente ao resultado produzido se estas transações fossem executadas uma após outra (serial).

Escalonamento Serializável

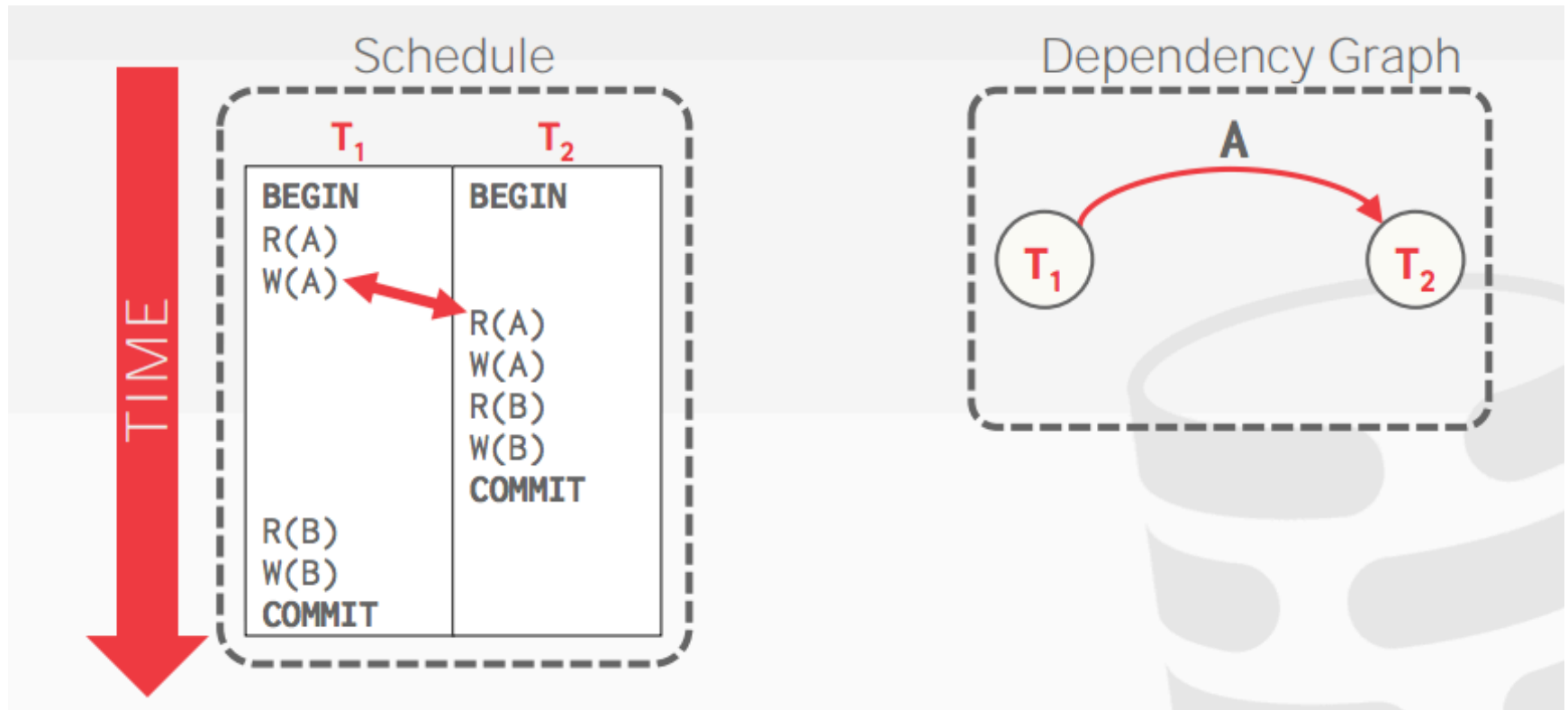
- Operações conflitantes entre transações:
 - operam sobre o mesmo objeto e
 - pelo menos uma dela é um Write (W)
- Detecção de Escalonamento Serializável: constroi-se um grafo direcionado, chamado **grafo de dependência**, cujos vértices são as transações e as arestas são conflitos em operações de transações. Caso o grafo **NÃO** tenha ciclos, o escalonamento é dito serializável.

Escalonamento Serializável

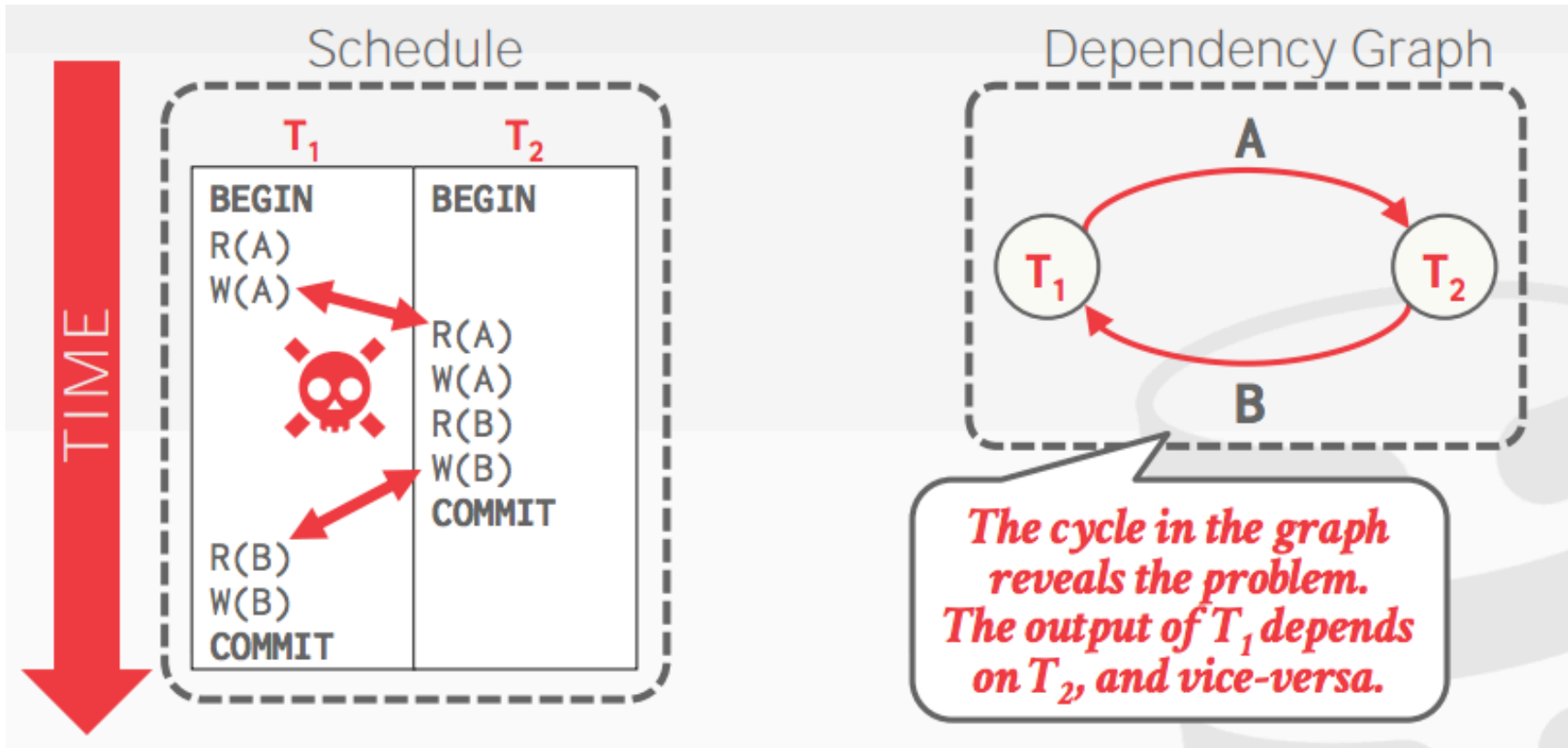
- Grafo de Dependência:
 - Um nó por transação
 - Um arco de T_i para T_j se:
 - Uma operação Op_i de T_i conflita com uma operação Op_j de T_j e
 - Op_i aparece antes no escalonamento do que Op_j
- O escalonamento será serializável se seu grafo de dependência é **Acíclico**.



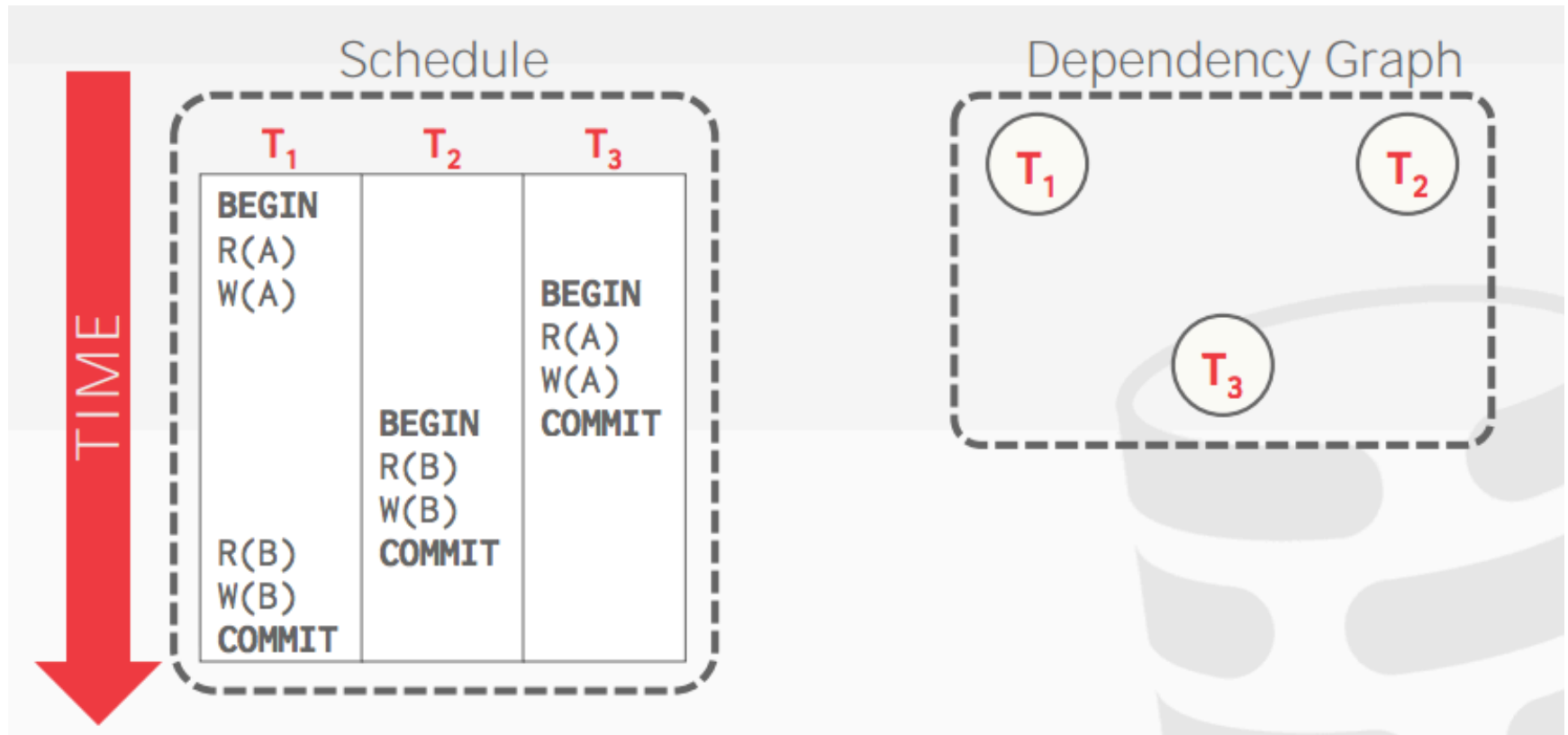
Ex.1 Escalonamento Serializável



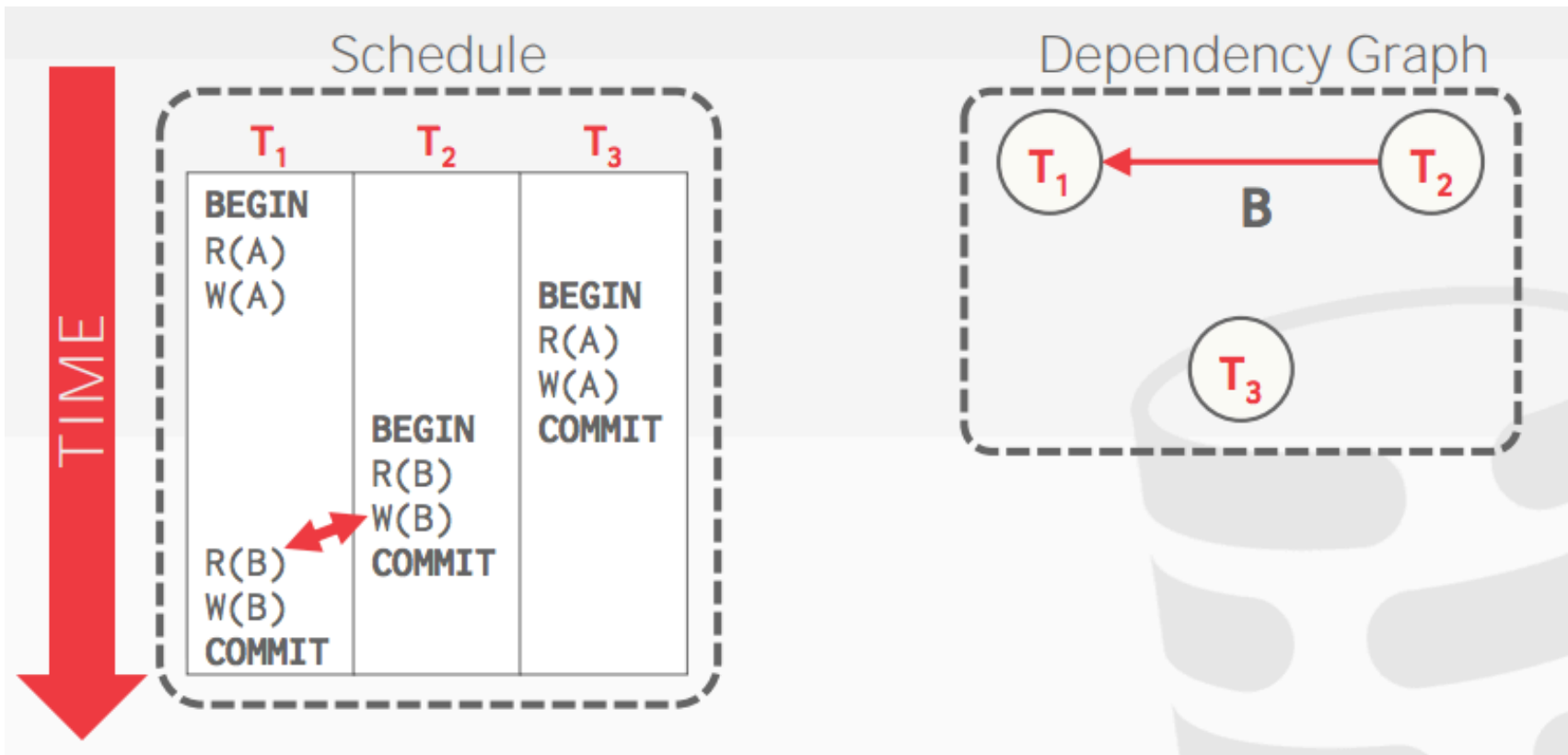
Ex.1 Escalonamento Não Serializável



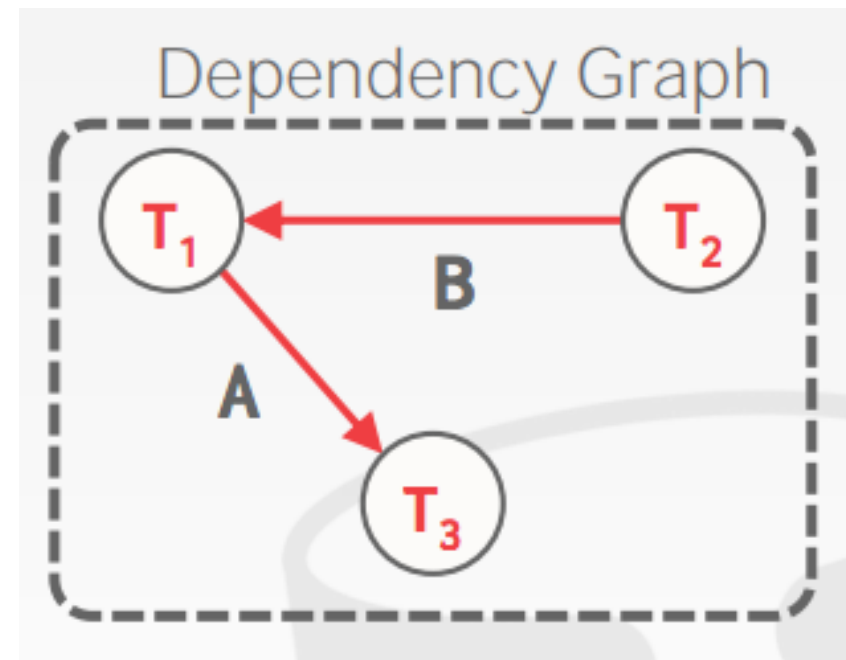
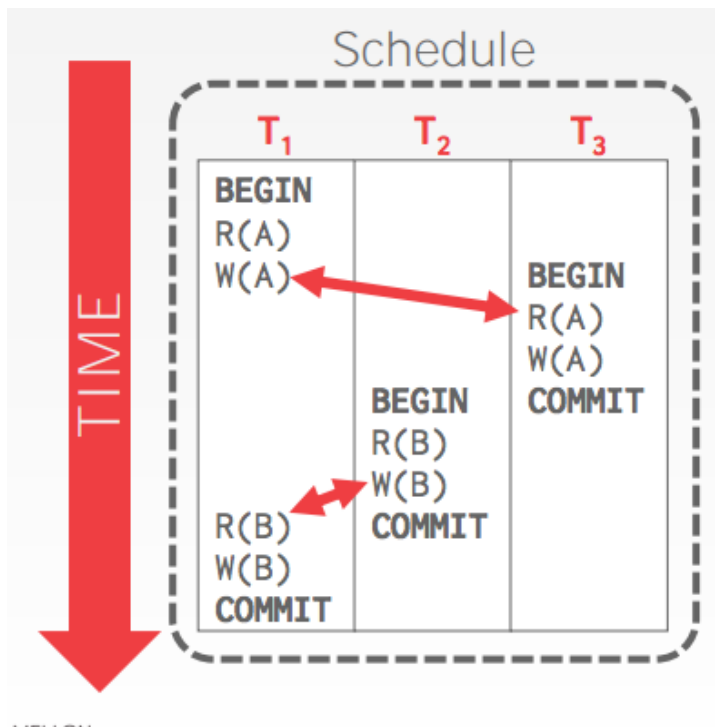
Ex.2 Escalonamento Serializável



Ex.2 Escalonamento Serializável



Ex.2 Escalonamento Serializável



Transação

- Uma transação pode começar com um comando **BEGIN TRANSACTION**
 - Uma transação termina com um dos comandos:
 - **COMMIT:** todo o trabalho da transação é refletido no BD; todas as mudanças antes de “commit” são invisíveis a outras transações (Ex. EXEC SQL COMMIT)
 - **ROLLBACK:** como se a transação nunca tivesse ocorrido (Ex. EXEC SQL ROLLBACK)
 - O Rollback pode ter um **SAVEPOINT** que permite desfazer até ele e não a transação toda (bom para transações longas)
- OBS.:** Em interfaces de programação, as transações começam quando o usuário se conecta, e terminam ou quando um comando **COMMIT** ou **ROLLBACK** é executado

Exemplo

Vende(bar, cerveja, preco)

- O bar Tricolor só vende Bud for 2,50 e Miller por 3,00
- Salete está querendo saber o maior e o menor preço de cerveja do bar Tricolor
 - (1) SELECT MAX(preco) FROM Vende
WHERE bar = 'Tricolor';
 - (2) SELECT MIN(preco) FROM Vende
WHERE bar = 'Tricolor';
- Ao mesmo tempo, o gerente do bar Tricolor decide substituir Miller e Bud por Heineken a 3,50
 - (3) DELETE FROM Vende
WHERE bar = 'Tricolor' AND
(cerveja = 'Miller' OR cerveja = 'Bud');
 - (4) INSERT INTO Vende
VALUES('Tricolor', 'Heineken', 3.50);
- Se a ordem de execução dos comandos for 1, 3, 4, 2, então aparece para Salete que o preço mínimo do bar Tricolor é maior que seu preço máximo

O comando SET TRANSACTION

- Usado para colocar o nível de isolamento e operações permitidas (R/W).
- Sintaxe:

SET TRANSACTION <acesso>, <isolação>

Onde: <acesso> : READ ONLY | READ WRITE

<isolação>: READ UNCOMMITTED
READ COMMITTED
REPEATABLE READ
SERIALIZABLE

O comando SET TRANSACTION

- Caso não usemos o comando **SET TRANSACTION**, o default será usado
- Default
 - SET TRANSACTION READ WRITE, ISOLATION LEVEL SERIALIZABLE

Obs.: Se você especificar que o nível de isolamento é **READ UNCOMMITTED**, então o modo de acesso tem que ser **READ ONLY!**

O comando SET TRANSACTION

Exemplos:

1) SET TRANSACTION READ ONLY,
ISOLATION LEVEL READ UNCOMMITTED

⇒ Indica que nenhum update será permitido por comandos SQL executados como parte da transação.

⇒ É o nível mais baixo de isolamento

2) SET TRANSACTION READ WRITE,
ISOLATION LEVEL SERIALIZABLE

⇒ Permite updates durante a transação como também consultas.

⇒ É o nível de isolamento mais alto

Protocolo de bloqueio em 2 fases

- **Bloqueio** (“Lock”) é um mecanismo usado para sincronizar acesso a dados compartilhados.
- **Tipos de bloqueios:**
 - **S compartilhado** (“Shared”, em inglês), para operação de leitura
 - **X exclusivo** (“eXclusive”, em inglês), para operação de modificação

Protocolo de bloqueio em 2 fases

- Matriz de compatibilidade de bloqueios:

	S	X
S	SIM	NÃO
X	NÃO	NÃO

Protocolo de bloqueio em 2 fases

- **Duas fases:**

- **Fase de aquisição:** fase em que se adquirem os bloqueios.
- **Fase de liberação:** quando se libera o primeiro bloqueio, a partir de então não se pode mais adquirir bloqueios apenas liberá-los

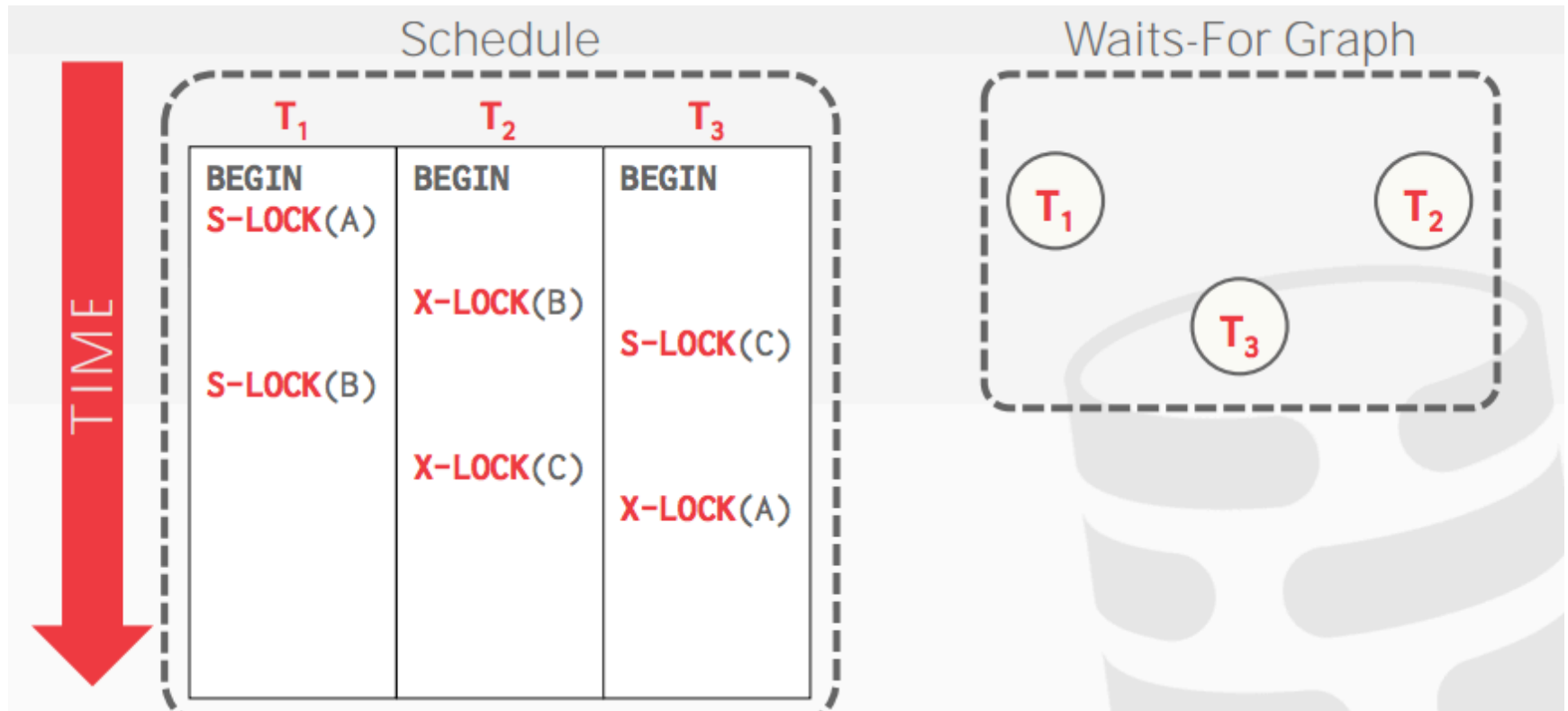
Protocolo de bloqueio em 2 fases

- **Problema:** Deadlock (interblocagem)
 - É quando é criado um impasse em que uma transação fica esperando pela liberação de um bloqueio de uma outra transação e vice-versa.
 - Há duas maneiras de lidar com deadlocks:
 - Abordagem 1: Detecção de Deadlock
 - Abordagem 2: Prevenção de Deadlock

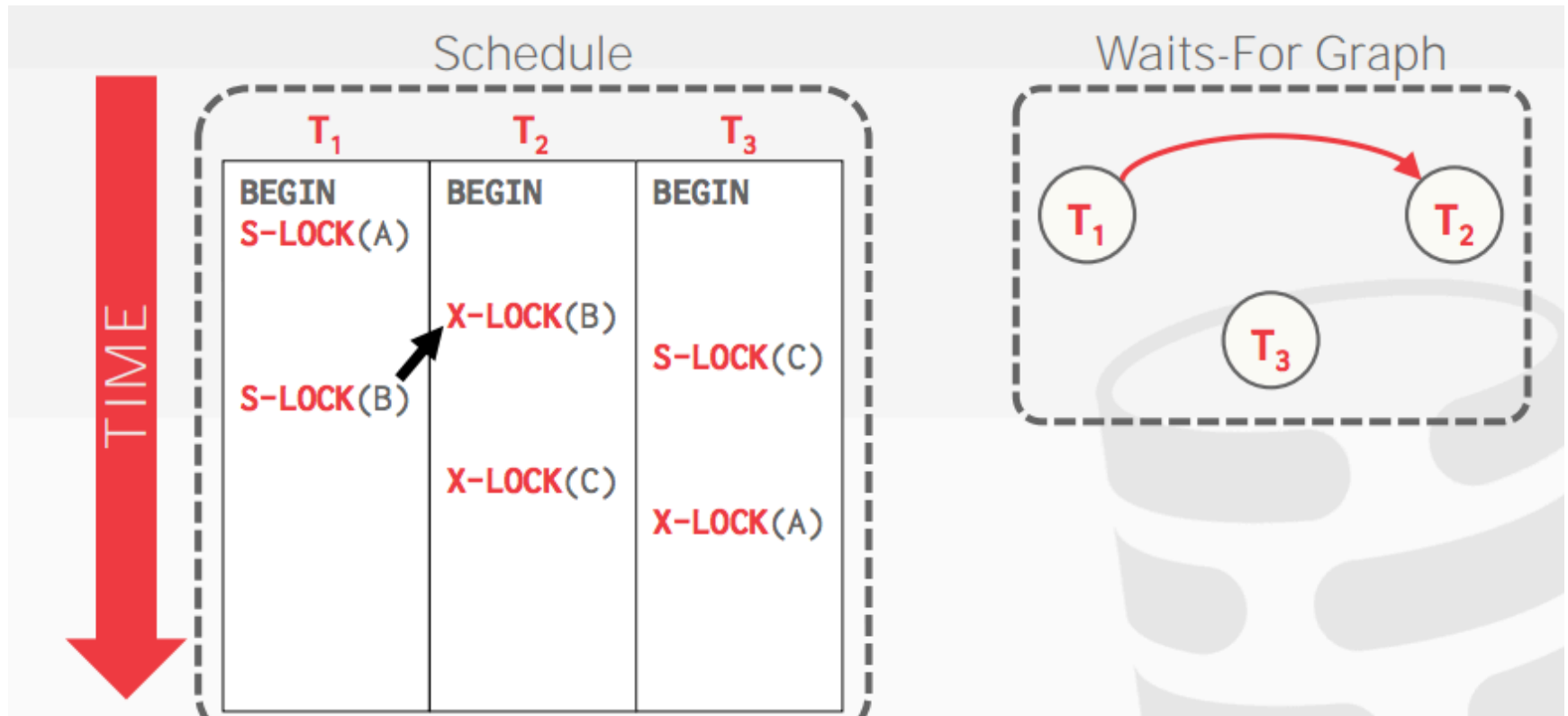
Detecção de Deadlocks

- O SGBD cria um grafo waits-for para manter o registro de quais locks cada transação está aguardando a aquisição:
 - Os nós são as transações
 - Um arco de T_i para T_j se T_i está esperando por T_j liberar um bloqueio
- O SGBD verifica periodicamente se há ciclos no grafo (deadlock) e toma uma decisão de como quebrar o ciclo

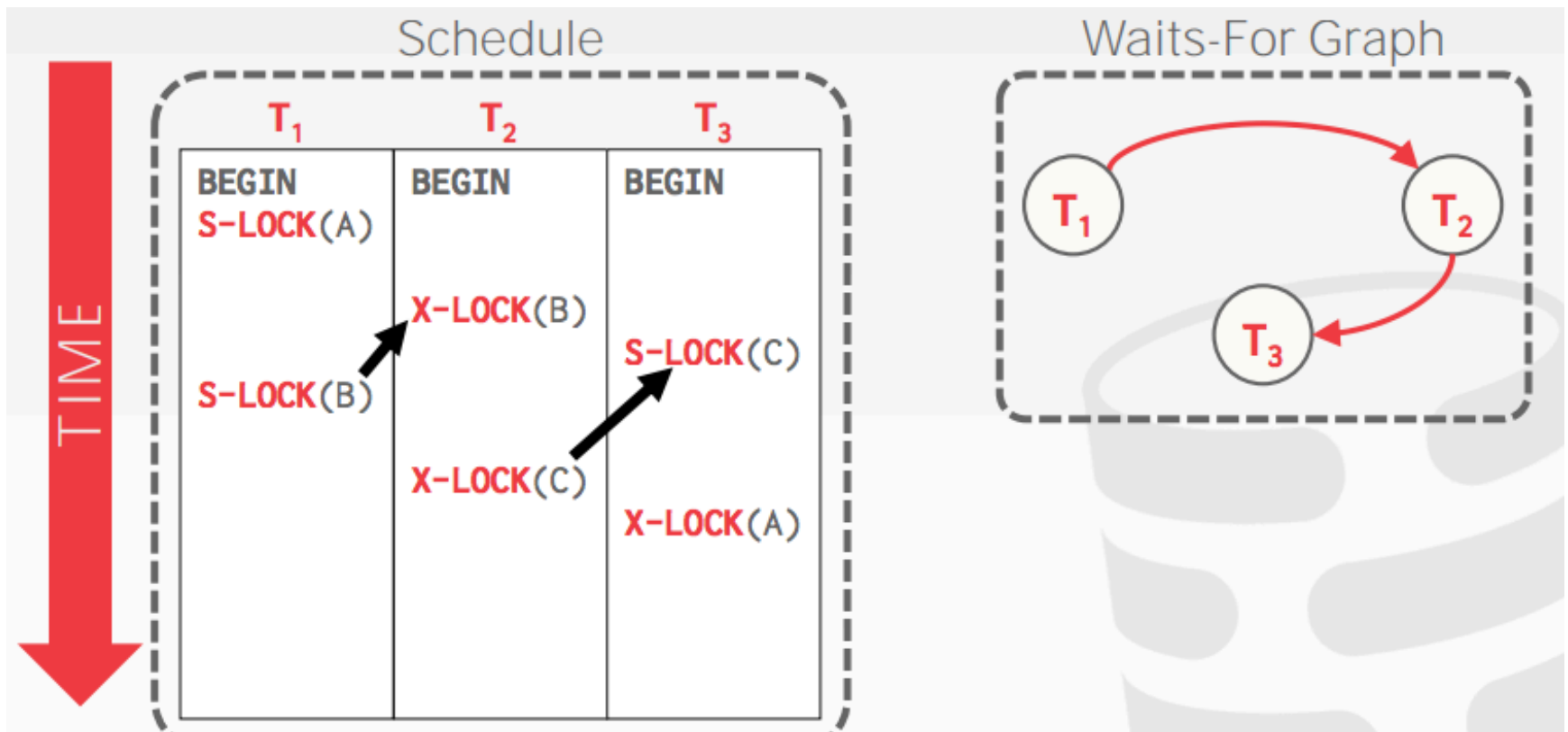
Detecção de Deadlocks



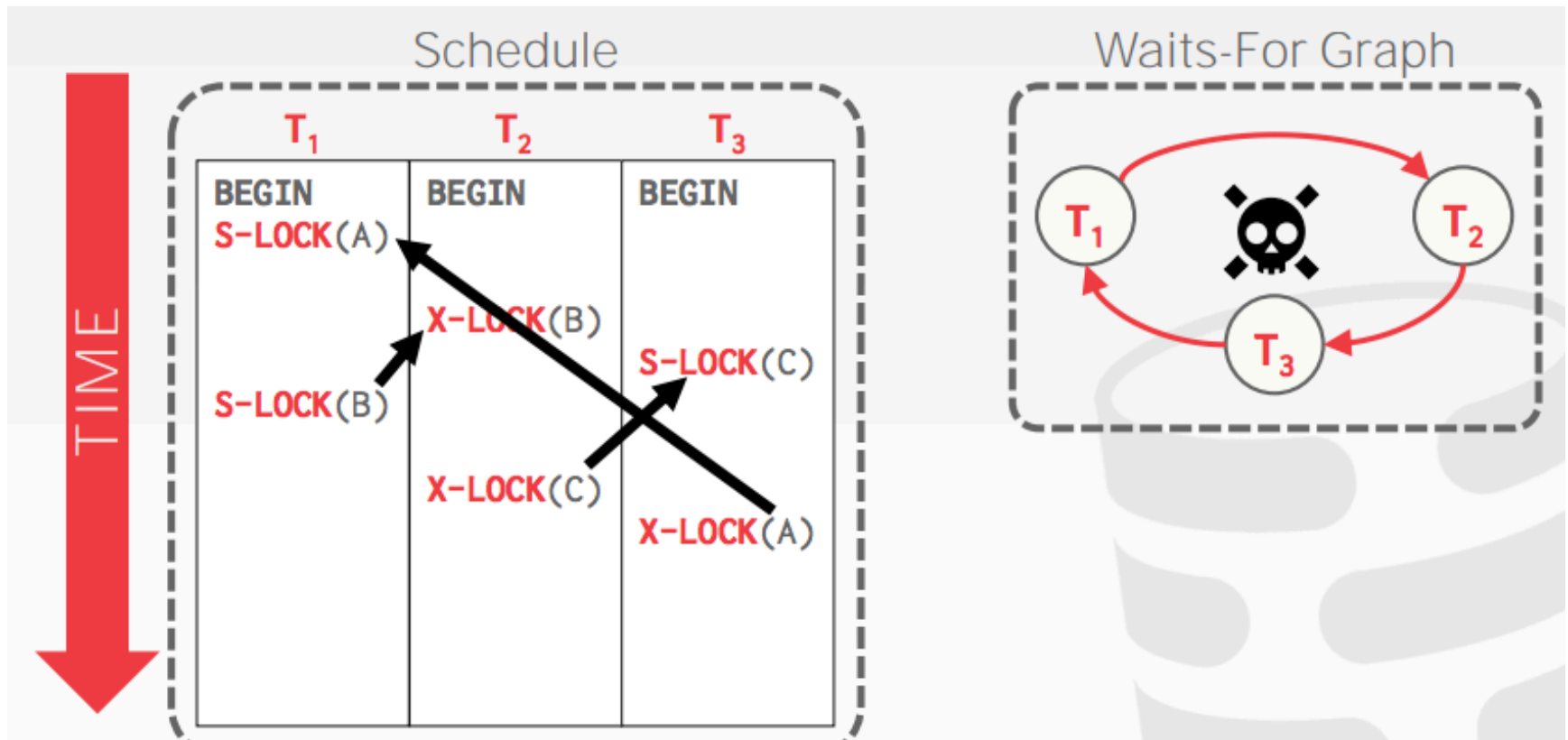
Detecção de Deadlocks




Detecção de Deadlocks



Detecção de Deadlocks





Manipulação de Deadlock

- Quando o SGBD detecta um deadlock, seleciona a vítima (transação) a ser abortada para quebrar o ciclo
- Como eleger a vítima (transação)?

Manipulação de Deadlock

- A seleção da vítima depende de um conjunto de variáveis:
 - Por idade (menor timestamp)
 - Por progresso (menos/mais queries executadas)
 - Pelo número de item já bloqueados
 - Pelo número de transações que precisam ser abortadas
 - OBS: Tem que ser considerado também o número de vezes que uma transação foi abortada para prevenir starvation

Prevenção de Deadlocks

- Quando uma transação tenta adquirir um lock que está assegurado a outra transação, o SGBD mata uma delas para prevenir um deadlock
- Não requer grafo wait-for para detectar deadlock

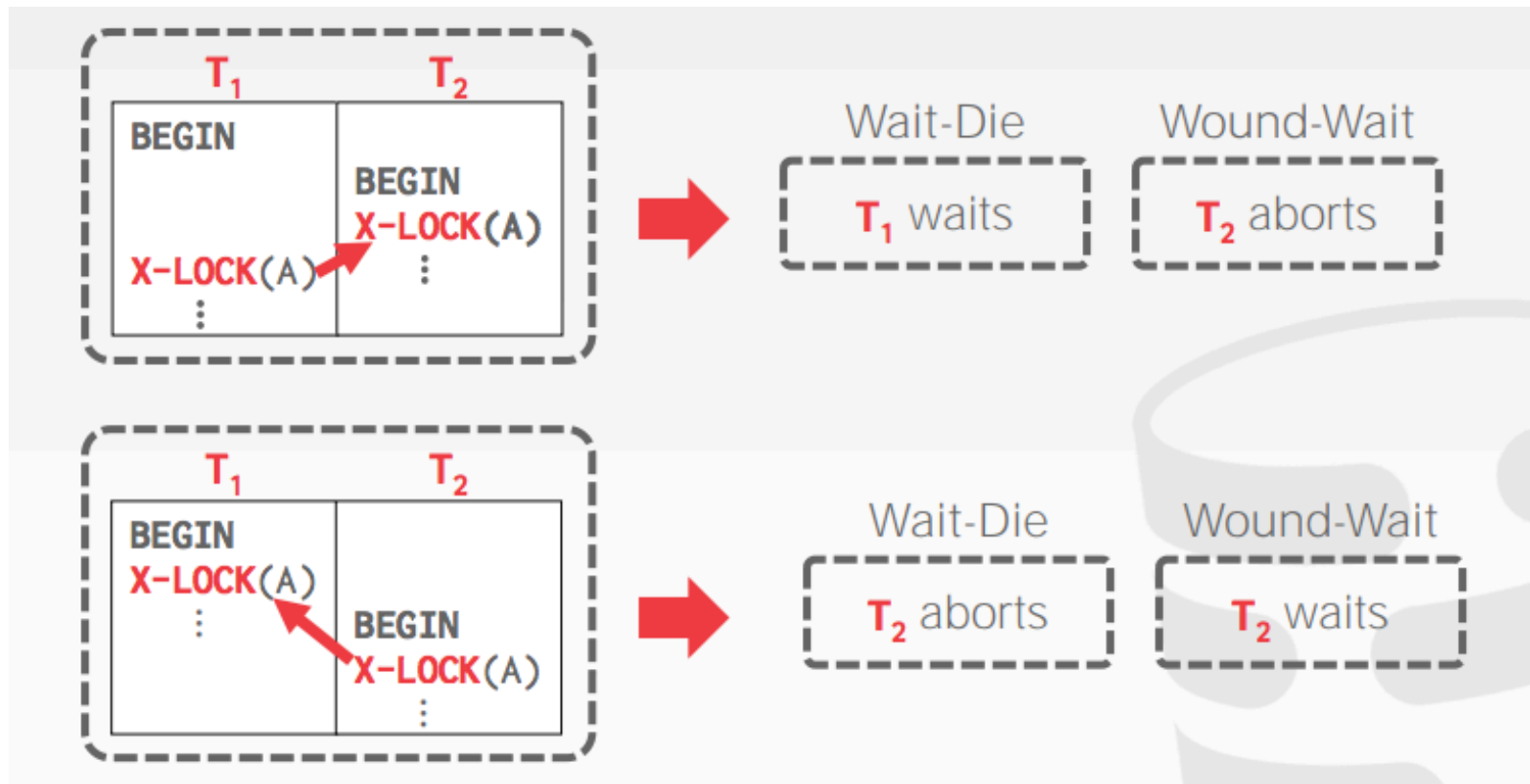
Prevenção de Deadlocks

- Atribua prioridades baseadas nos timestamps:
 - Older timestamp = Higher priority (ex. $T1 > T2$)
- Wait-Die (“Antiga espera pela Jovem”)
 - Se uma transação $T1$ requer um lock e tem maior prioridade que a transação $T2$ que já detém o lock, então $T1$ espera por $T2$
 - Do contrário, $T1$ aborta

Prevenção de Deadlocks

- Atribua prioridades baseadas nos timestamps:
 - Older timestamp = Higher priority (ex. $T1 > T2$)
- Wound-Wait (“Jovem espera por Antiga”)
 - Se a transação solicitante de bloqueio T1 tem maior prioridade do que a transação que já detém o bloqueio T2, então T2 aborta e libera o lock
 - Caso contrário, T1 espera

Prevenção de Deadlocks





Prevenção de Deadlocks

- Por que estas abordagens anteriores garantem que não há deadlocks?

Apenas um tipo de direção é permitido quando espera por um lock.

Prevenção de Deadlocks

- Quando a transação reinicia, qual a sua nova prioridade?
 - Seu timestamp original.

Transação em JDBC

- Quando uma conexão é criada ela está em **mode auto-commit**
 - ⇒ Cada comando SQL individual é tratado como uma transação e quando ele termina a execução será confirmado (commit)
- Se quisermos criar uma unidade de trabalho (transação), para agrupar vários comandos devemos desabilitar o auto-commit:
 - ⇒ `con.setAutoCommit(false);`

```
//Transação de depósito de 50 reais da conta X para Y, valores iniciais das contas X = 100 e Y = 100
try {
    ...
    con.setAutoCommit(false);
    PreparedStatement updateContaX = con.prepareStatement("Update ContaCorrente set saldo = ? where
número = ?");
    updateContaX.setInt(1, 50);
    updateContaX.setString(2,"87.234-2");
    updateContaX.executeUpdate();
    PreparedStatement updateTotalContaY = con.prepareStatement("Update ContaCorrente set saldo = ?
where número = ?");
    updateContaY.setInt(1, 150);
    updateContaY.setString(2,"32.234-2");
    updateContaY.executeUpdate();
    con.commit();
    con.setAutoCommit(true);
    ...
} catch (SQLException ex) {
    System.err.println("SQLException: ", + ex.getMessage());
    if (con != null) {
        try {
            System.err.println("Transação vai sofrer ROLLBACK");
            con.rollback();
        } catch (SQLException ex) {
            System.err.println("SQLException: ", + ex.getMessage());
        }
    }
}
```


Técnica de Timestamp

- É uma técnica de controle de concorrência que usa os timestamps (TS) das transações para determinar a ordem de serializabilidade (sem bloqueios).
- Se $TS(T_i) < TS(T_j)$, então o SGBD deve assegurar que a execução do escalonamento é equivalente a um escalonamento serial se T_i aparece sempre antes de T_j

Técnica de Timestamp

- A cada Transação T_i é atribuído um timestamp
- Não há bloqueios nos objetos!!!
- Cada objeto X é rotulado com o timestamp da última transação que o leu/gravou com sucesso
 - W-TS(X): Write timestamp em X
 - R-TS(X): Read timestamp em X

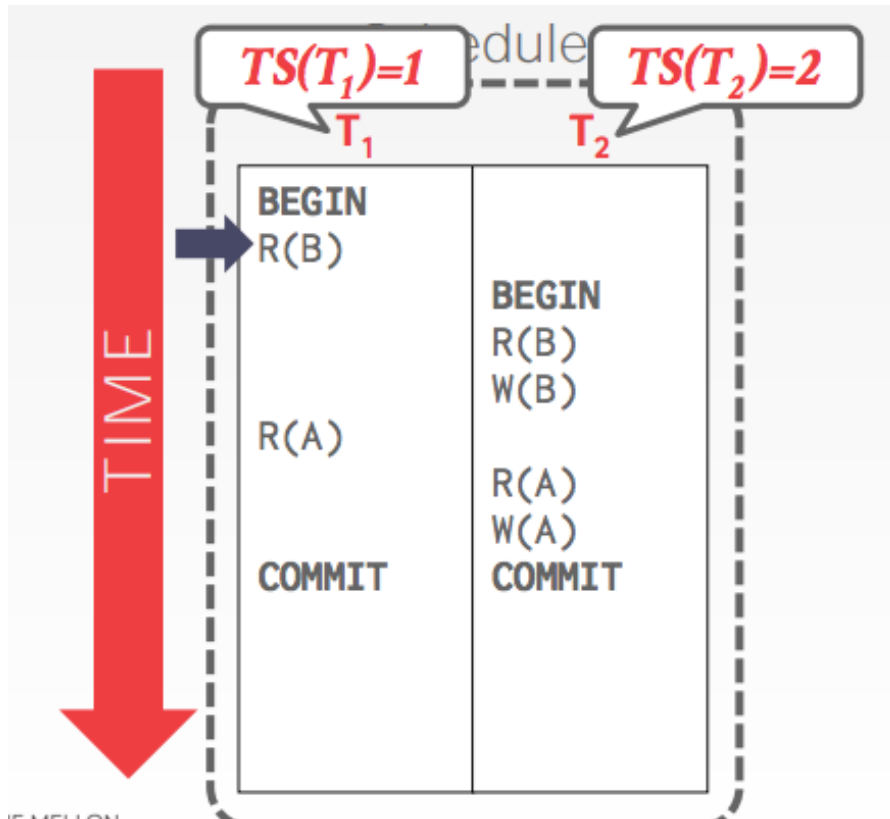
Técnica de Timestamp

- Suponha que T_i execute um **READ(X)**
- Se $TS(T_i) < W-TS(X)$, viola a ordem de timestamp de T_i com relação a quem escreveu X , então T_i é **Abortada**
- Senão
 - Permita T_i ler X
 - $R-TS(X) = \text{Max}(R-TS(X), TS(T_i))$

Técnica de Timestamp

- Suponha que T_i execute um **WRITE(X)**
- Se $TS(T_i) < R-TS(X)$ ou $TS(T_i) < W-TS(X)$ viola a ordem de timestamp de T_i com relação a quem leu ou escreveu X , então T_i é **Abortada**
- Senão
 - Permita T_i gravar X
 - $W-TS(X) = TS(T_i)$

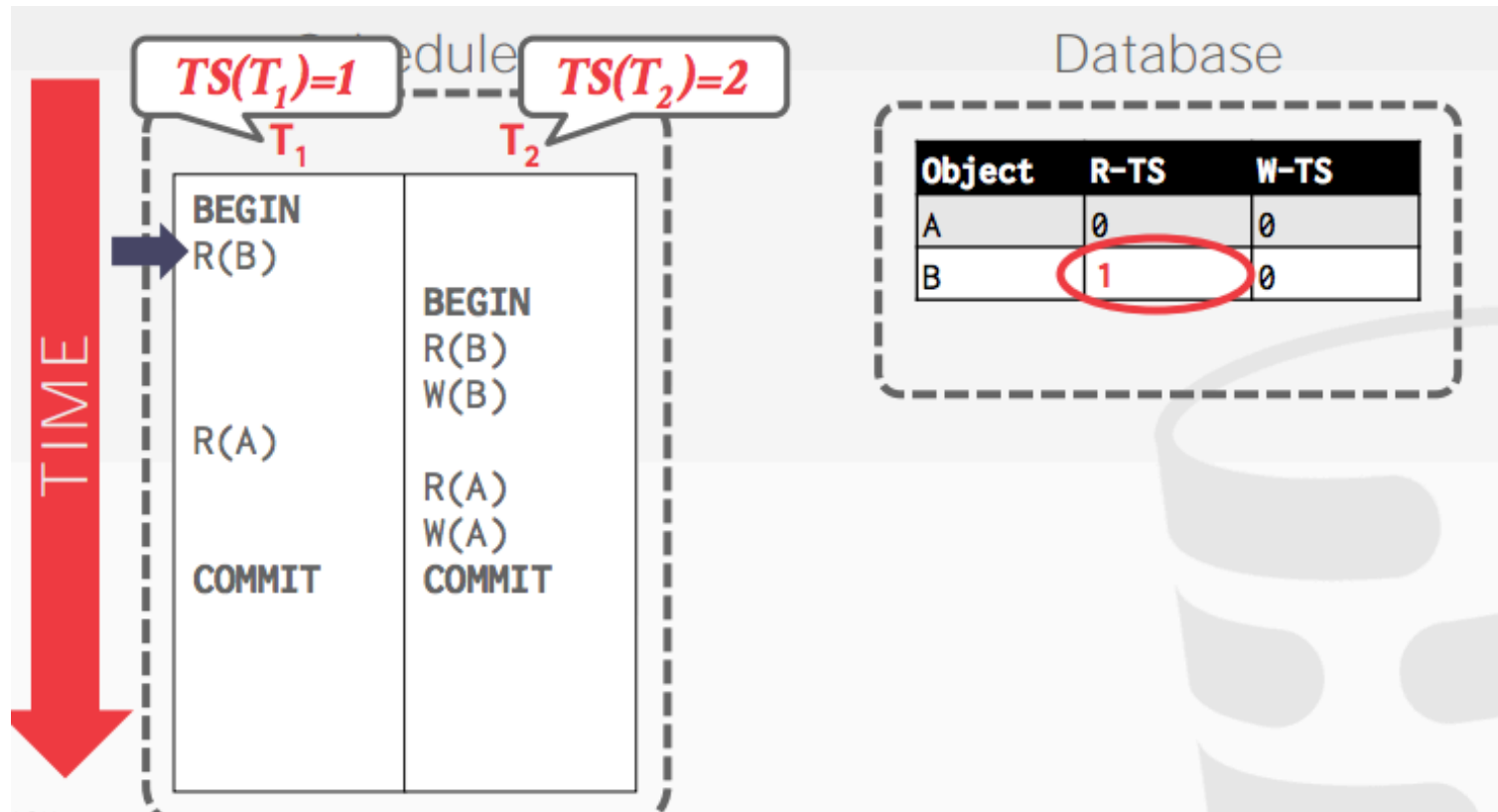
Ex1: Timestamp



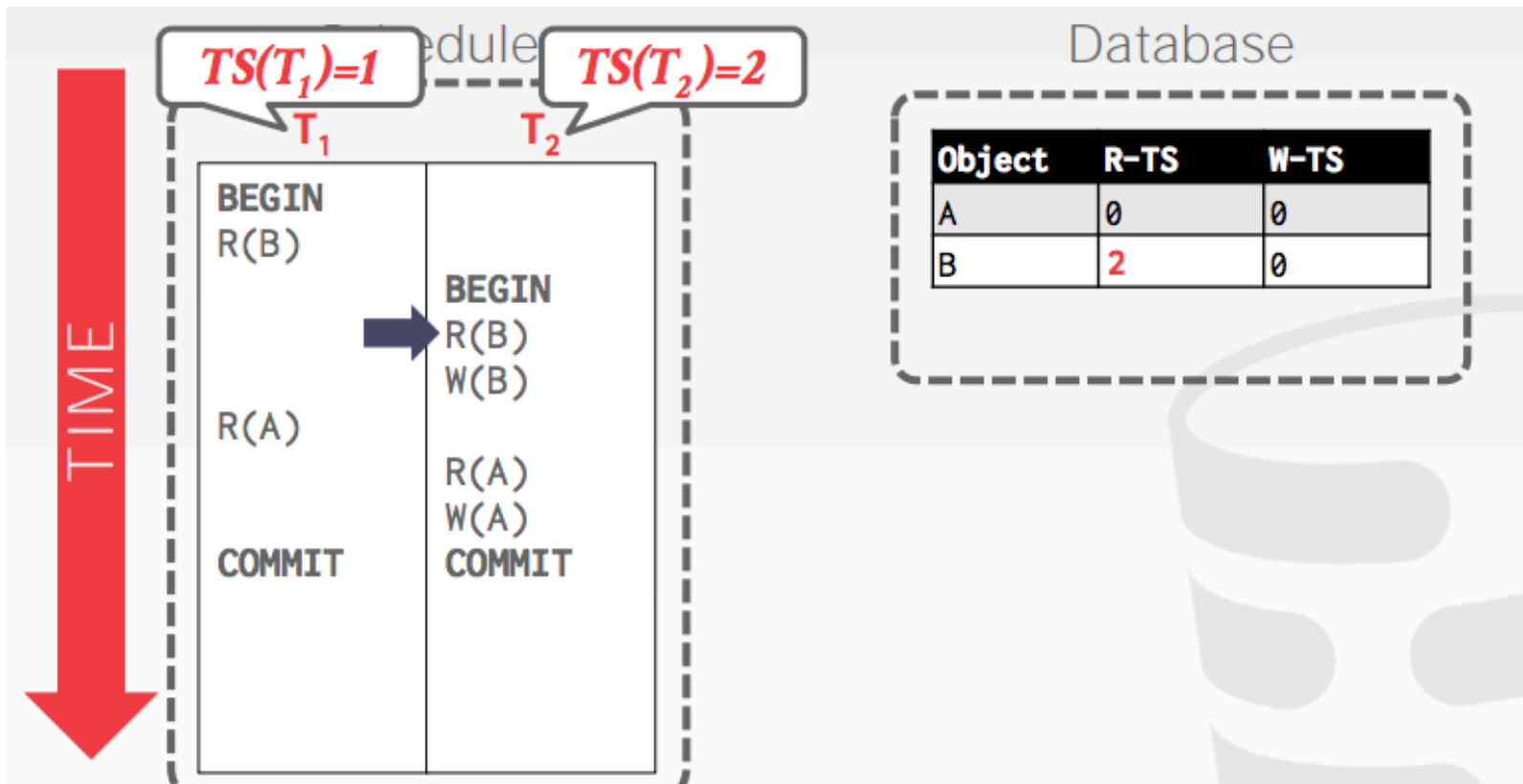
Database

Object	R-TS	W-TS
A	0	0
B	0	0

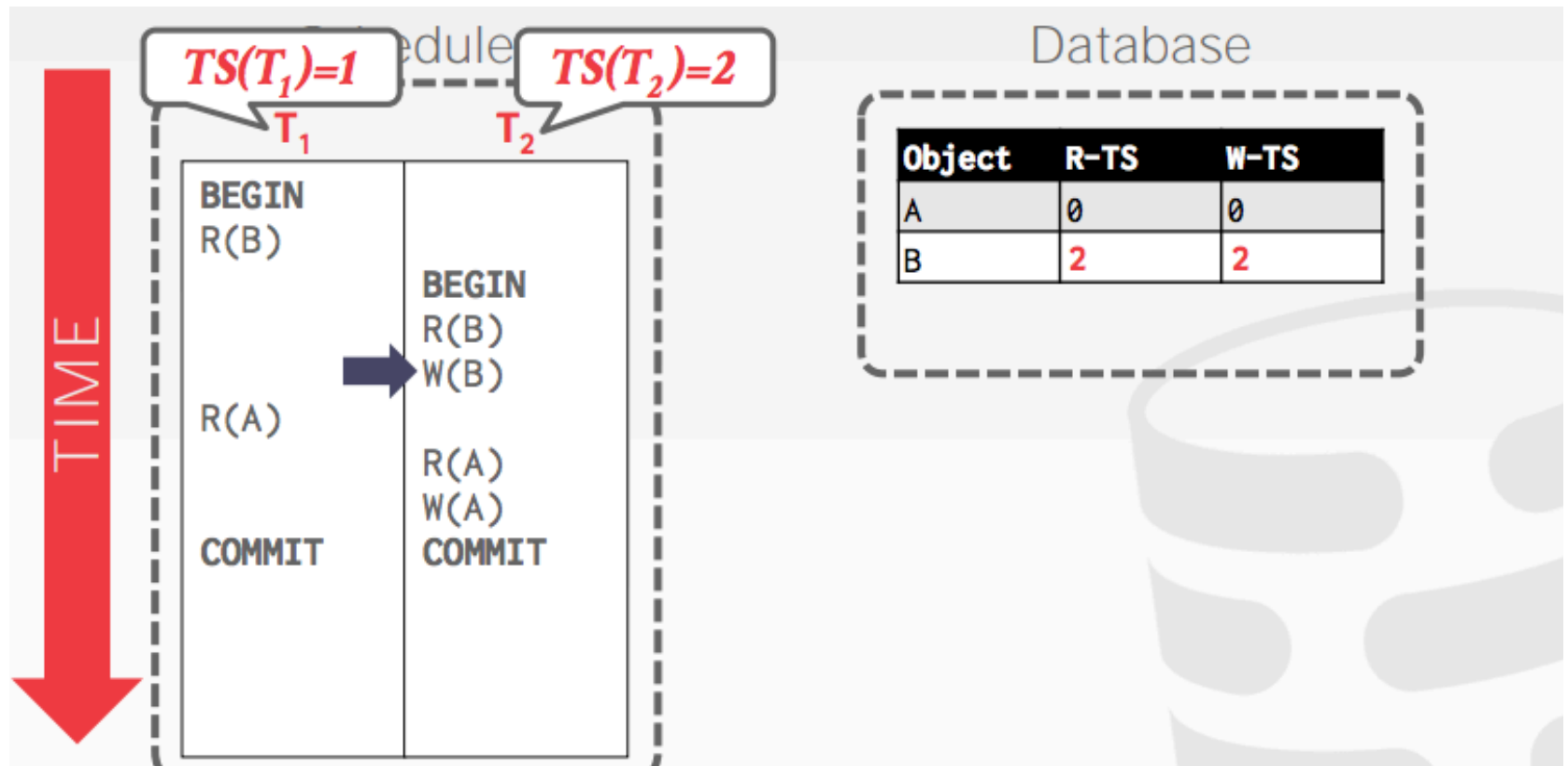
Ex1: Timestamp



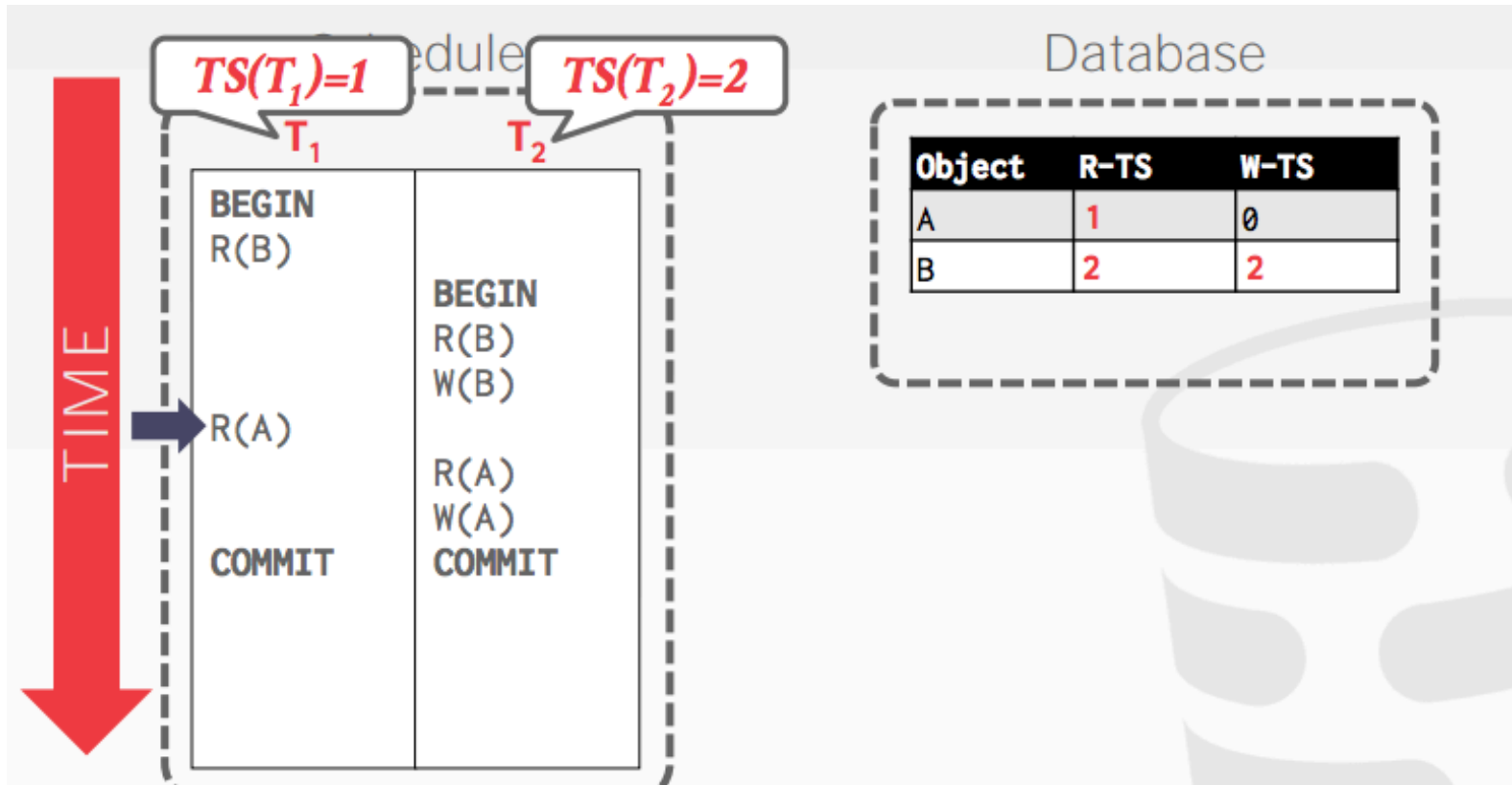
Ex1: Timestamp



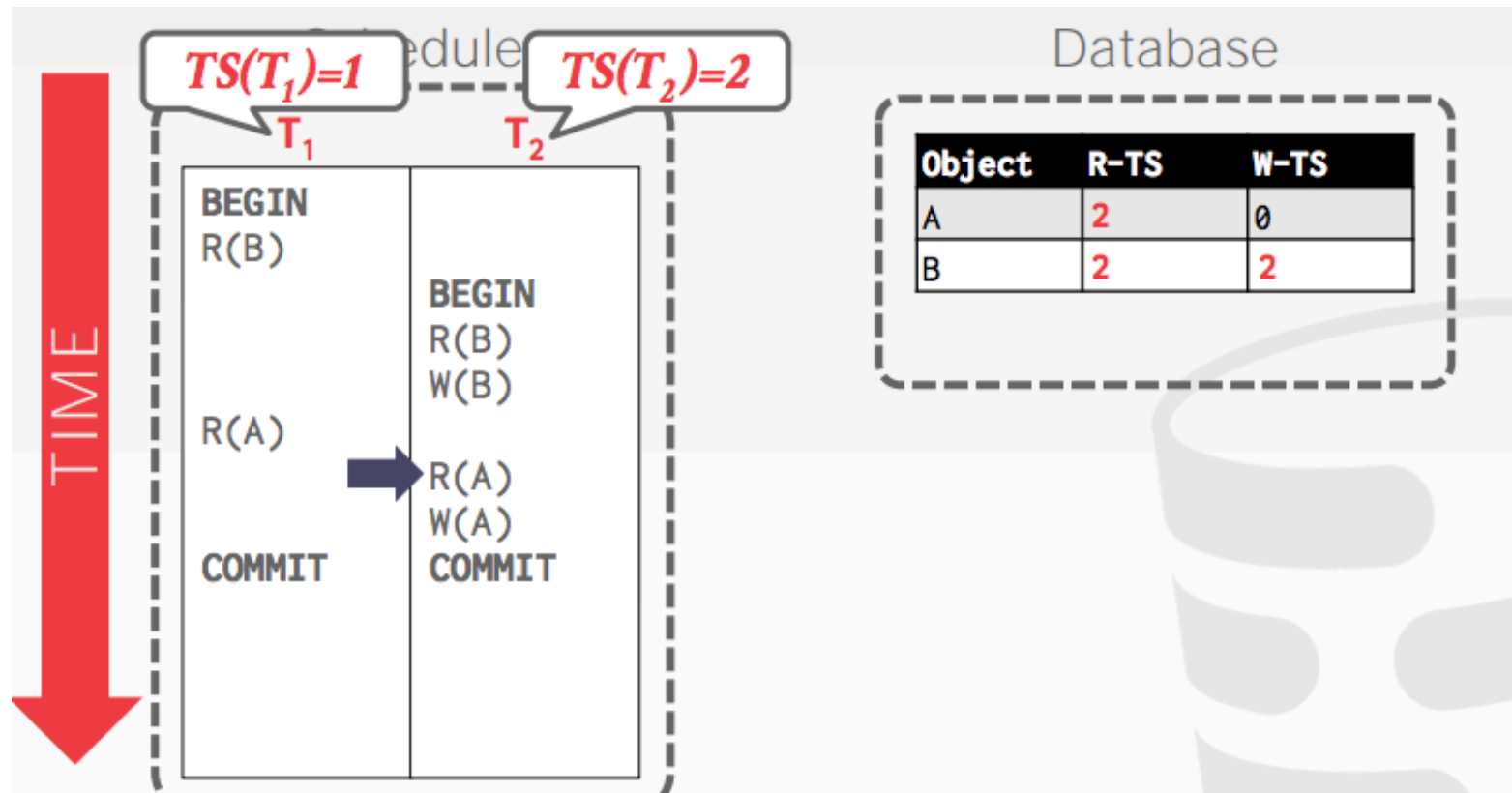
Ex1: Timestamp



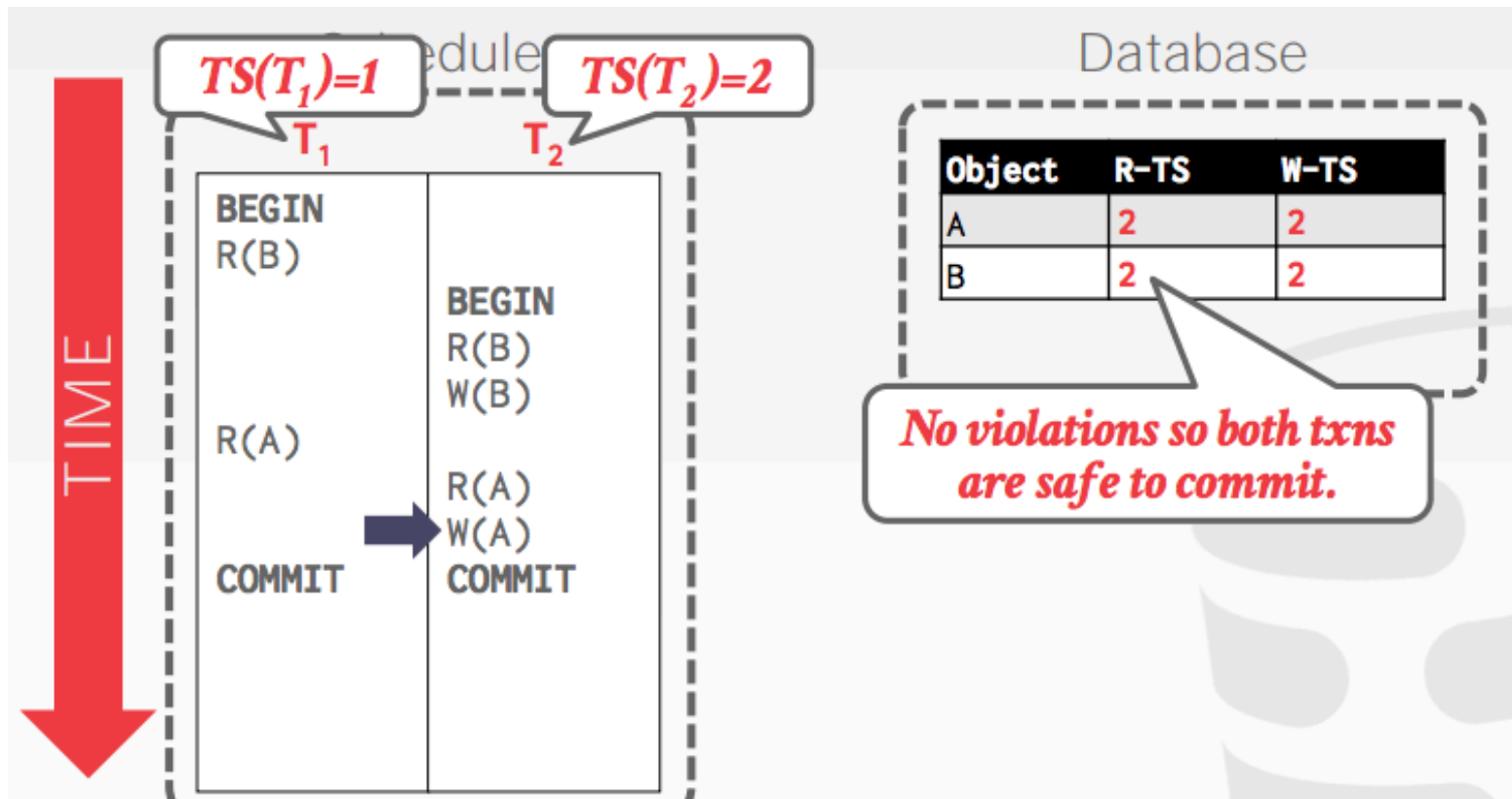
Ex1: Timestamp



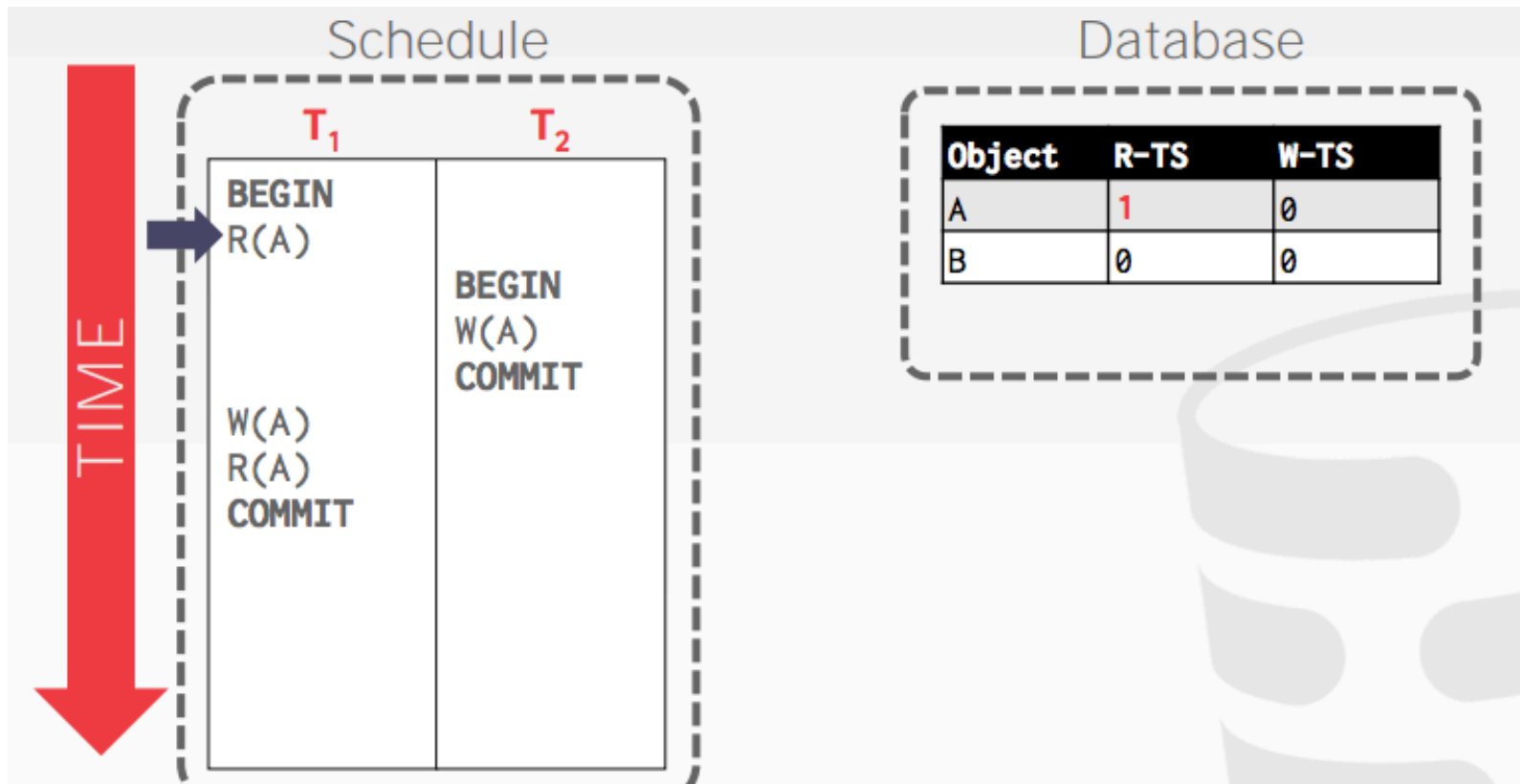
Ex1: Timestamp



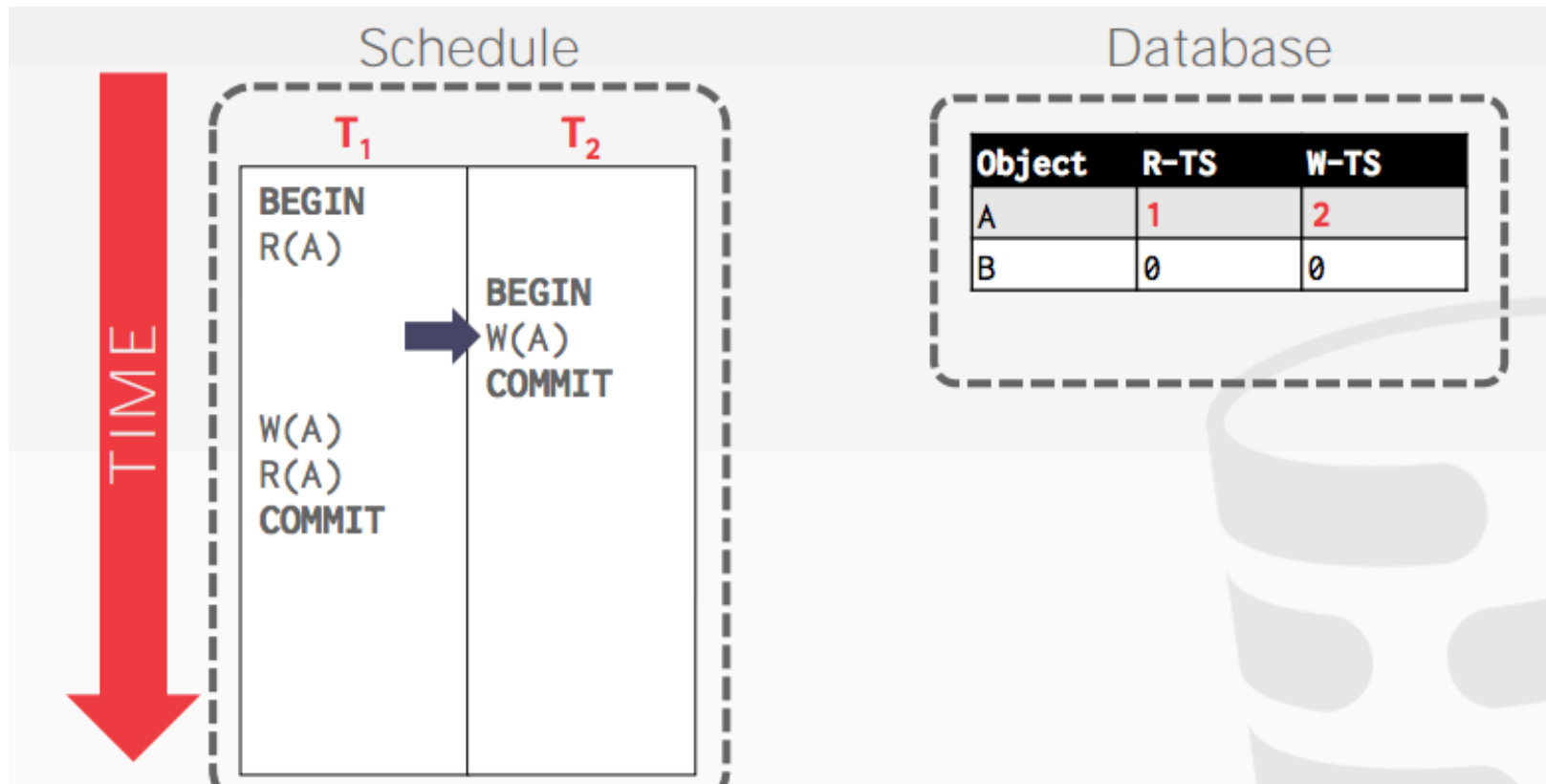
Ex1: Timestamp



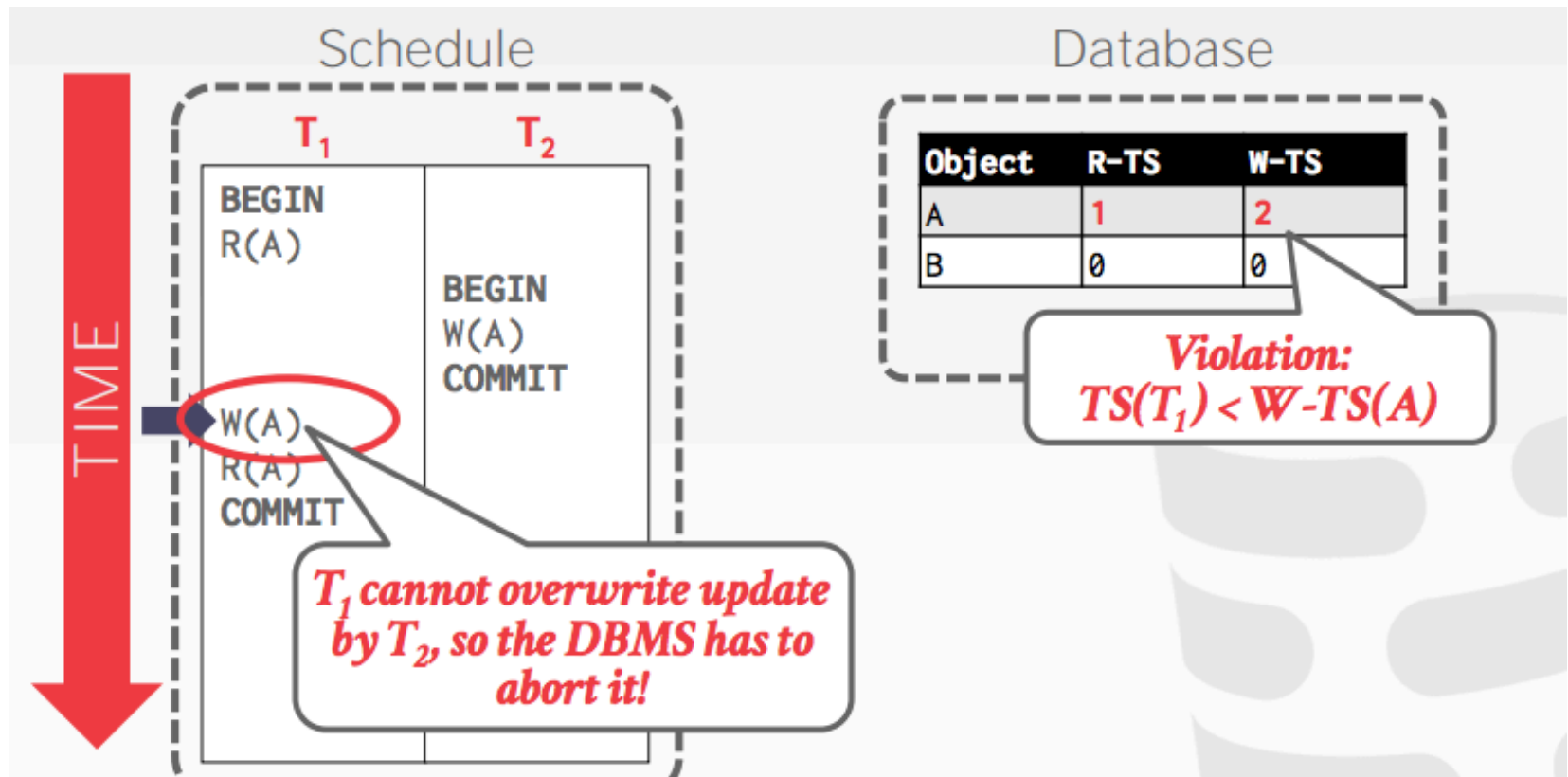
Ex2: Timestamp



Ex2: Timestamp



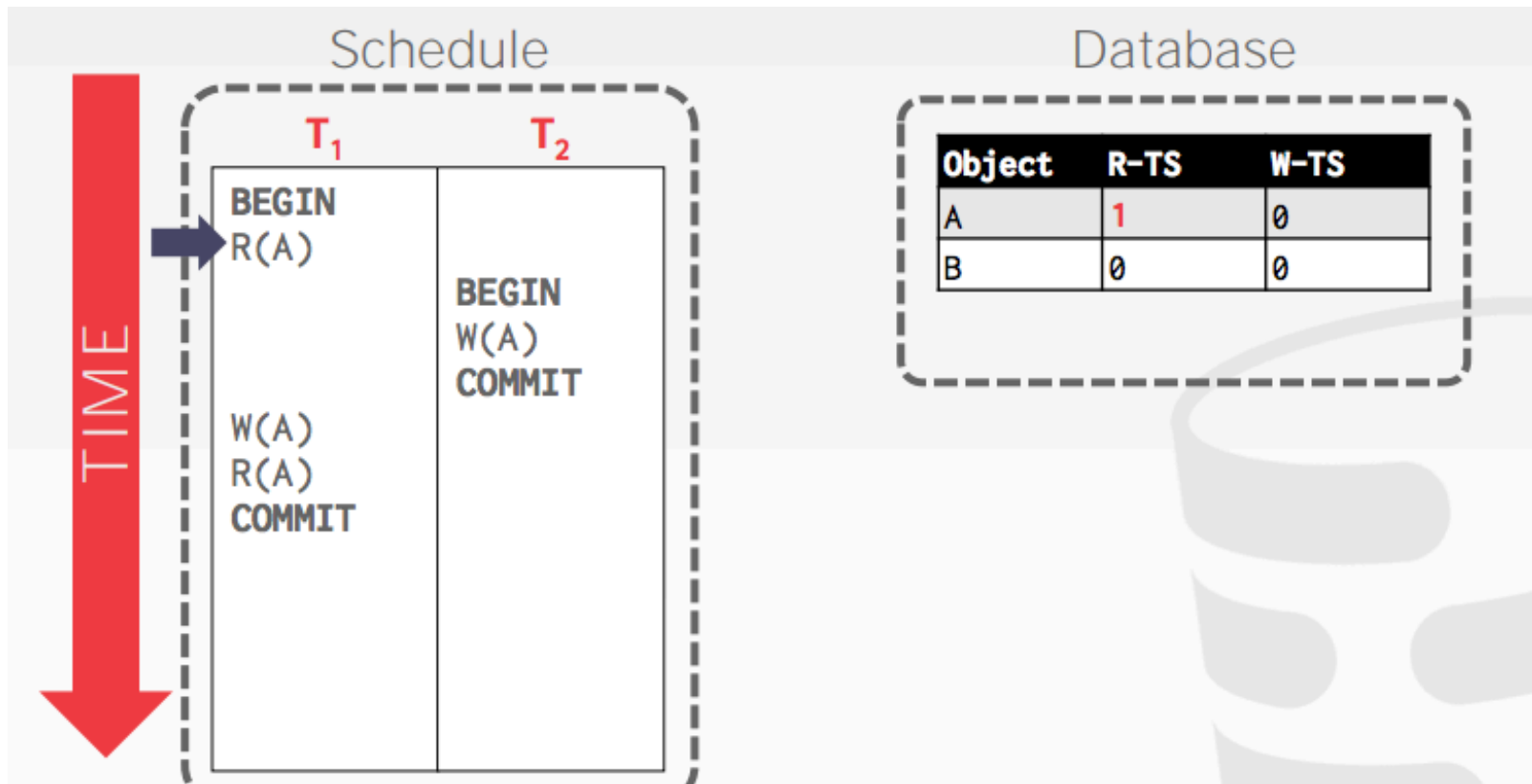
Ex2: Timestamp



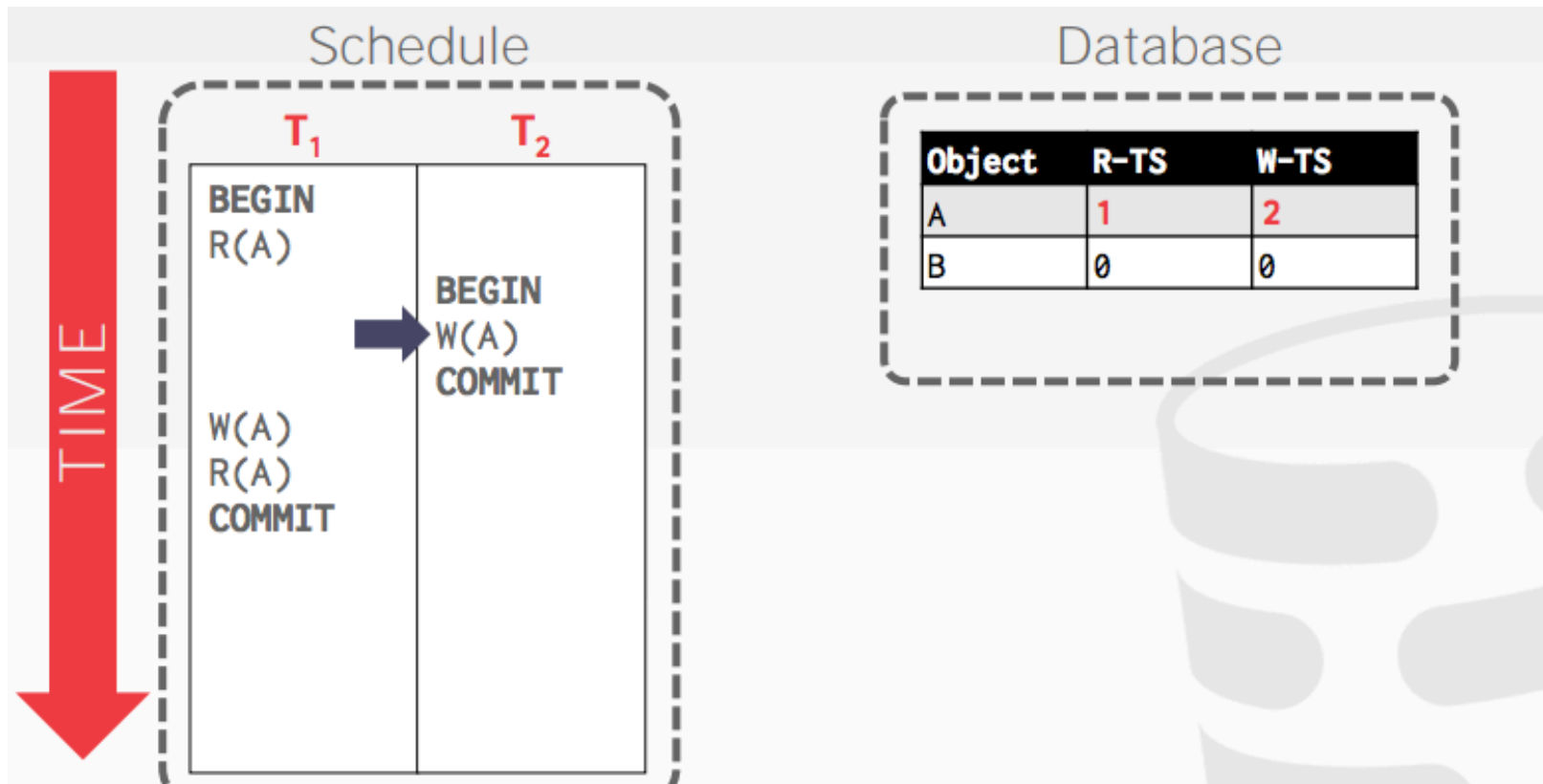
Timestamp: Thomas Write Rule

- If $TS(T_i) < R-TS(X)$:
Aborta e reinicia T_i
- If $TS(T_i) < W-TS(X)$:
Ignora o write e a Transação continua
- Else:
Permite T_i efetuar $Write(X)$
 $W-TS(X) = TS(T_i)$

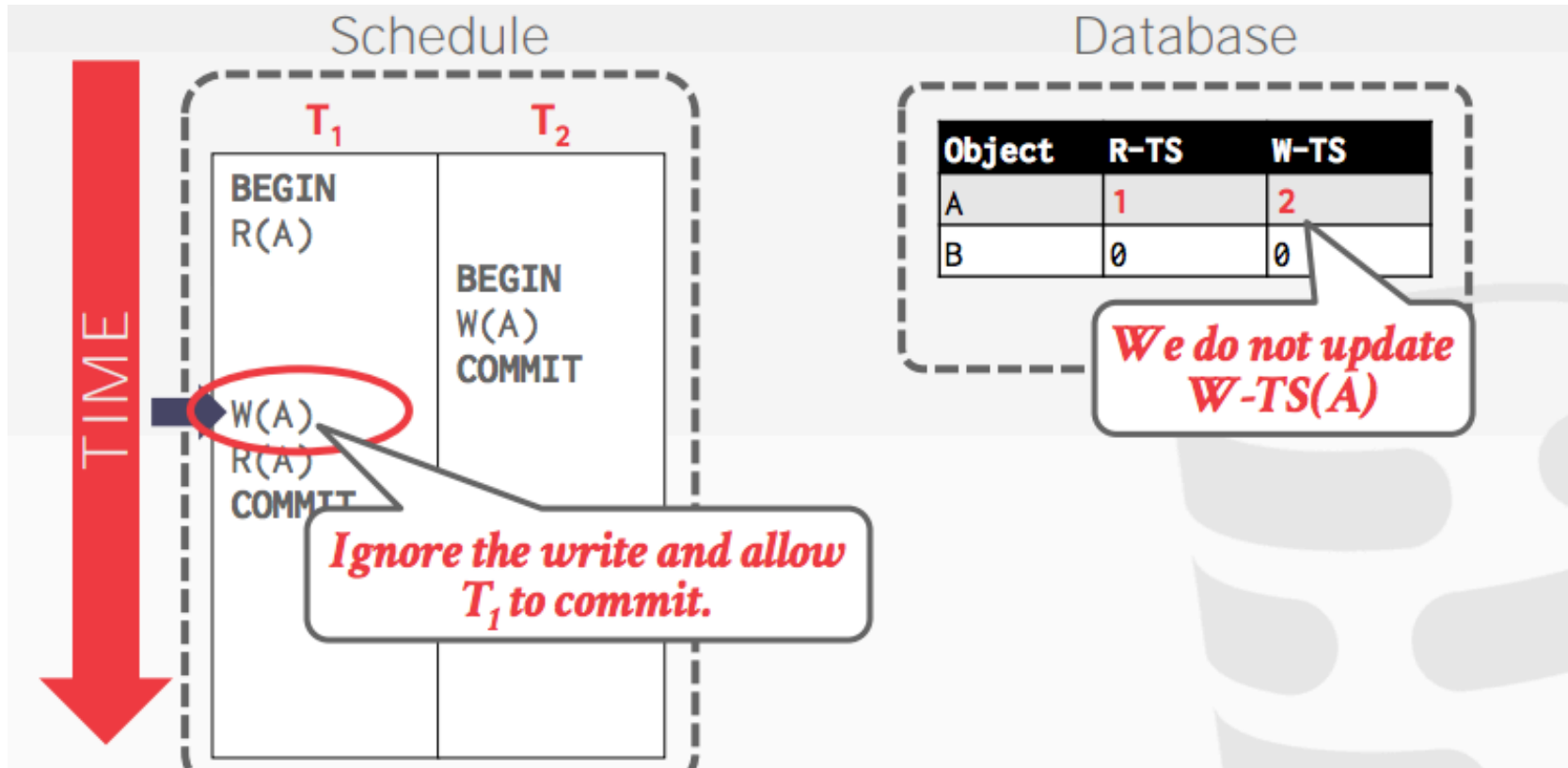
Ex 3. Timestamp



Ex 3. Timestamp



Ex 3. Timestamp



Concorrência Multi-versão

- Protocolo MVCC (Multi-Version Concurrency Control)
- O SGBD mantém múltiplas versões **físicas** de um único objeto **lógico** no BD
 - Quando a transação grava um objeto, o SGBD cria uma nova versão do objeto
 - Uma transação NUNCA espera para ler um objeto do BD

Concorrência Multi-versão

- Cada **versão** do objeto X do BD tem as variáveis:
 - $R_TS(X_i)$: read timestamp de X_i , que representa o maior de todos os timestamps de transações que leram X_i
 - $W_TS(X_i)$: write timestamp de X_i , timestamp da transação que gravou o valor da versão X_i .
- Uma Transação T_i lê a versão mais atual do dado, cujo timestamp precede $TS(T_i)$
- Se uma Transação T_i deseja gravar um objeto, deve-se assegurar que o objeto não foi lido por outra Transação T_j tal que $TS(T_i) < TS(T_j)$.

Concorrência Multi-versão

- T_i deseja **gravar** um objeto X , e a versão k de X tem o maior $W_TS(X_k)$ de todas as versões de X
 - Se $TS(T_i) < R_TS(X_k)$:
 - Então T_i é abortada e reiniciada com um novo TS
 - Senão
 - T_i cria uma nova versão para X (digamos X_{k+1})
 - $R_TS(X_{k+1}) = W_TS(X_{k+1}) = TS(T_i)$
- Onde: $RTS(X_k)$: Read Timestamp do objeto X_{k+1}
 $WTS(X_k)$: Write Timestamp do objeto X_{k+1}

Concorrência Multi-versão

- Ti deseja **ler** um objeto X
 - Encontre a versão k de X que tem o maior $W_TS(X_k)$ de todas as versões de X , e que é menor ou igual a $TS(T_i)$
 - Retorne o valor de X_k para T_i
 - $R_TS(X_k) = \text{MAX} (TS(T_i), R_TS(X_k))$