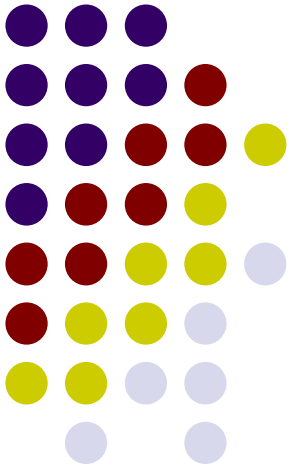
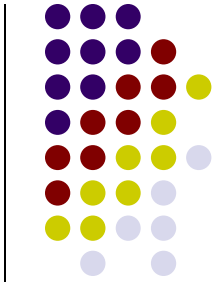
Transações em Bancos de Dados Relacionais e NO-SQL

Prof. Antonio Guardado

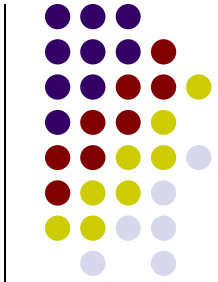


Agenda



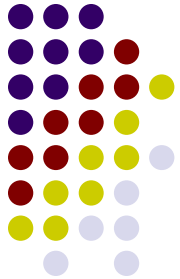
- 1- Gerenciamento de Transações e Propriedades Transacionais nos BDs Relacionais – ACID
- 2 - Serialização
- 3- Propriedades Transacionais nos BDs NO-SQL
- 4 – Propriedades BASE
- 5 – Teorema CAP
- 6 – Sistemas CAP
- 7 – Visão Geral CAP

Objetivos



- Entender os problemas das transações distribuídas em BDs Relacionais x Escalabilidade
- Compreender as propriedades transacionais para os BDs NO-SQL
- Estabelecer as diferenças destas propriedades para BDs Relacionais e BDs NO-SQL
- Compreender o Teorema CAP
- Compreender as combinações CAP para os sistemas NO-SQL e estabelecer suas diferenças e aplicações

1- Introdução a Transações



- SGBD
  - sistema de processamento de operações de acesso ao BD
- SGBDs são em geral multi-usuários
  - processam simultaneamente operações disparadas por vários usuários
    - deseja-se alta disponibilidade e tempo de resposta pequeno
  - execução intercalada de conjuntos de operações
    - exemplo: enquanto um processo *i* faz I/O, outro processo *j* é selecionado para execução
- Operações são chamadas transações

1.1 – Conceito de Transação

- Unidade lógica de processamento em um SGBD
- Composta de uma ou mais operações
  - seus limites podem ser determinados em SQL
- De forma abstrata e simplificada, uma transação pode ser encarada como um conjunto de operações de leitura e escrita de dados

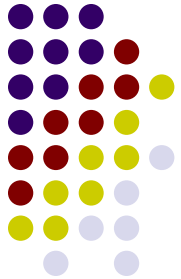
T_x

```
read(x)
x = x – i
read(y)
y = y * x
write(x)
write(y)
```

read(x) → lê o dado X do BD e o armazena na variável X

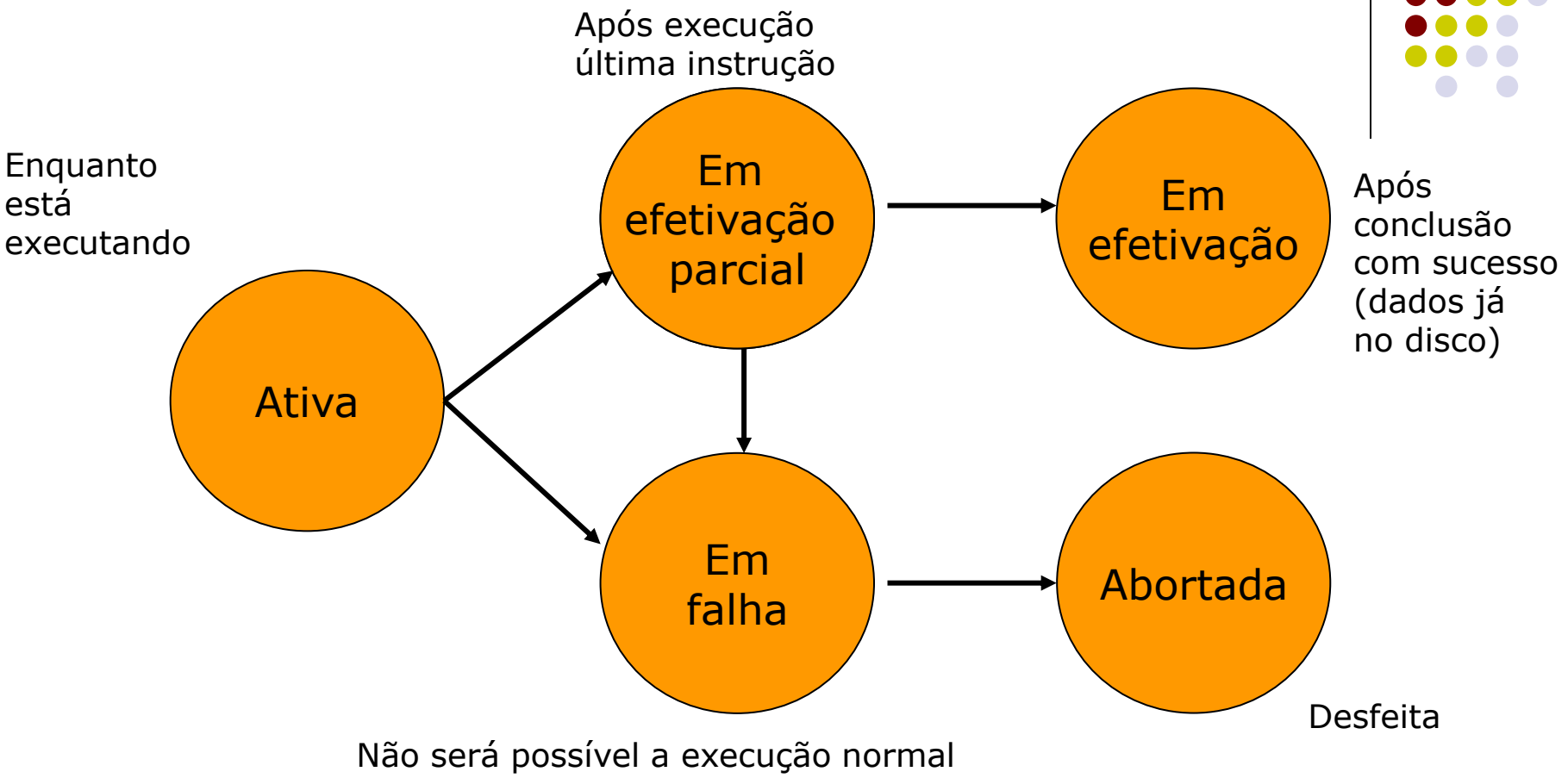
write(y) → grava no dado Y do BD o valor da variável Y

1.2 - Estados de uma Transação



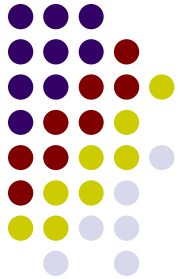
- Uma transação é sempre monitorada pelo SGBD quanto ao seu estado
  - que operações já fez? concluiu suas operações? deve abortar?
- Estados de uma transação
  - Ativa, Em processo de efetivação, Efetivada, Em falha, Em Efetivação(Concluída)
  - Respeita um Grafo de Transição de Estados

1.2.1 - Grafo de estados



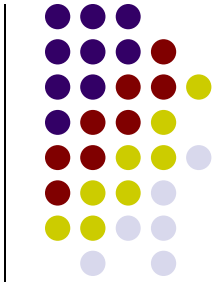
- A transação está concluída se estiver em efetivação ou abortada
- Em efetivação parcial: ainda na memória
- Supõe-se, por enquanto, que falhas não resultam em perdas no disco

1.3 - Propriedades de uma Transação nos SGBDs Relacionais



- Requisitos que sempre devem ser atendidos por uma transação
- Chamadas de **propriedades ACID**
  - **Atomicidade**
  - **Consistência**
  - **Isolamento**
  - **Durabilidade ou Persistência**

1.4 - Atomicidade



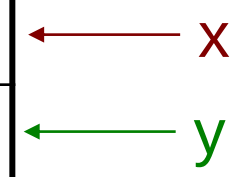
- Princípio do *"Tudo ou Nada"*
  - ou todas as operações da transação são efetivadas **(commit)** com sucesso no BD ou nenhuma delas se efetiva **(rollback)**
    - preservar a integridade do BD
- Responsabilidade do subsistema de recuperação contra falhas (subsistema de *recovery*) do SGBD
  - **desfazer (rollback)** as ações de transações parcialmente executadas

1.4 – Atomicidade (2)

- Deve ser garantida, pois uma transação pode manter o BD em um estado inconsistente durante a sua execução

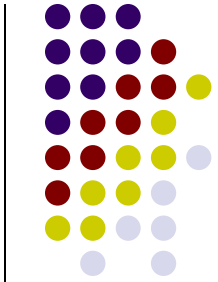
Contas

número	saldo
100	500.00
200	200.00
. . .	



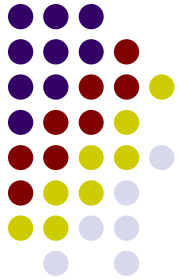
T_x (transferência bancária)

1.5 - Consistência



- Uma transação sempre conduz o BD de um estado consistente para outro estado também consistente
- Responsabilidade conjunta do
  - DBA
    - definir todas as RIs para garantir estados e transições de estado válidos para os dados
      - exemplos: salário > 0; salário novo > salário antigo
  - subsistema de *recovery*
    - desfazer as ações da transação que violou a integridade

1.6 - Isolamento



- No contexto de um conjunto de transações concorrentes, a execução de uma transação T_x deve funcionar como se T_x executasse de forma isolada
  - T_x não deve sofrer interferências de outras transações executando concorrentemente
- Responsabilidade do subsistema de controle de concorrência (*scheduler*) do SGBD
  - garantir escalonamentos sem interferências

1.6 – Isolamento (2)

T_1	T_2
read(A)	
A = A – 50	
write(A)	
	read(A)
	A = A+A*0.1
	write(A)
read(B)	
B = B + 50	
write(B)	
	read(B)
	B = B - A
	write(B)

escalonamento válido

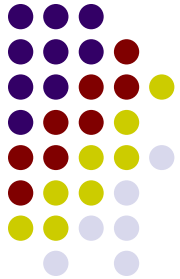
T_1	T_2
read(A)	
A = A – 50	
	read(A)
	A = A+A*0.1
	write(A)
	read(B)
write(A)	
read(B)	
B = B + 50	
write(B)	
	B = B - A
	write(B)

T_1 interfere em T_2

T_2 interfere em T_1

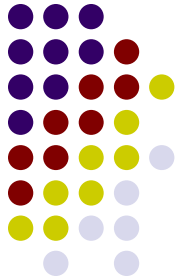
escalonamento inválido

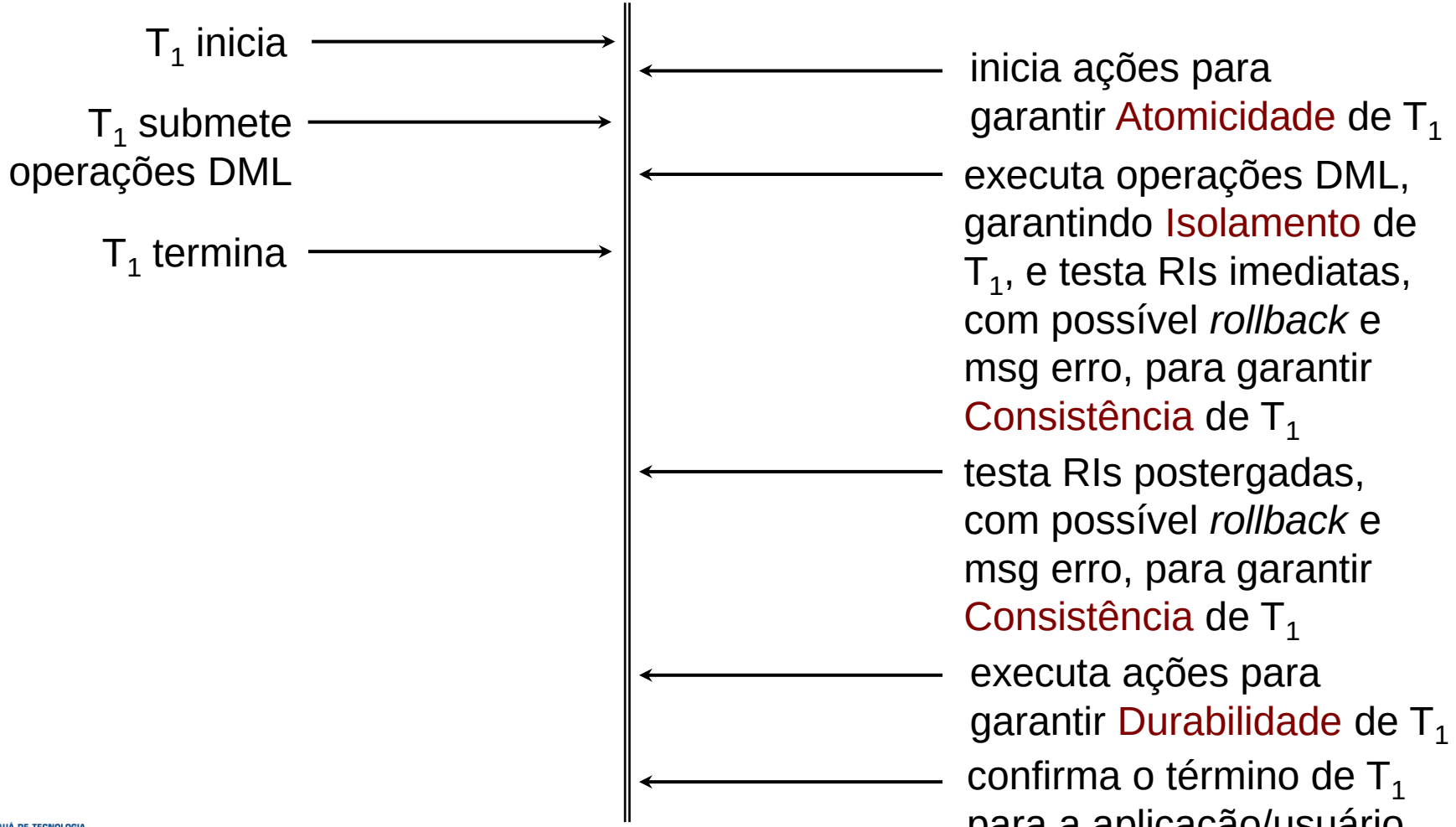
1.7 - Durabilidade



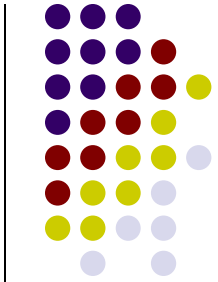
- Deve-se garantir que as modificações realizadas por uma transação que concluiu com sucesso persistam no BD
  - nenhuma falha posterior ocorrida no BD deve perder essas modificações
- Responsabilidade do subsistema de *recovery*
  - refazer transações que executaram com sucesso em caso de falha no BD

1.8- Gerência Básica de Transações



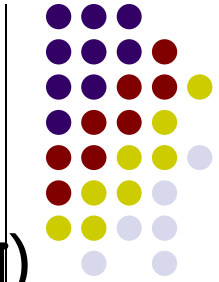


1.9 - Transações em SQL



- Por *default*, todo comando individual é considerado uma transação
  - exemplo: `DELETE FROM Pacientes`
    - exclui todas ou não exclui nenhuma tupla de pacientes, deve manter o BD consistente, etc
- SQL Padrão (SQL-92)
  - `SET TRANSACTION`
    - inicia e configura características de uma transação
  - `COMMIT [WORK]`
    - encerra a transação (solicita efetivação das suas ações)
  - `ROLLBACK [WORK]`
    - solicita que as ações da transação sejam desfeitas

1.9 - Transações em SQL (2)



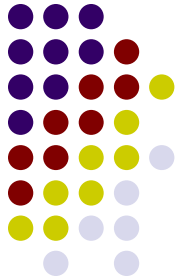
- Principais configurações (**`SET TRANSACTION`**)
  - modo de acesso
    - **`READ`** (somente leitura), **`WRITE`** (somente atualização) ou **`READ WRITE`** (ambos - *default*)
  - nível de isolamento
    - indicado pela cláusula **`ISOLATION LEVEL`** *nível*
    - *nível* para uma transação T_i pode assumir
      - **`SERIALIZABLE`** (T_i executa com completo isolamento - *default*)
      - **`REPEATABLE READ`** (T_i só lê dados efetivados e outras transações não podem escrever em dados lidos por T_i) – pode ocorrer que T_i só consiga ler alguns dados que deseja
      - **`READ COMMITTED`** (T_i só lê dados efetivados, mas outras transações podem escrever em dados lidos por T_i)
      - **`READ UNCOMMITTED`** (T_i pode ler dados que ainda não sofreram efetivação)

1.9 - Transações em SQL (3)

- Exemplo

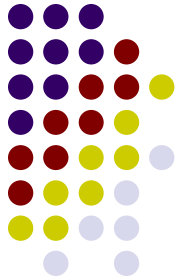
```
EXEC SQL SET TRANSACTION
         WRITE
         ISOLATION LEVEL SERIALIZABLE;
...
for(;;)
{...
EXEC SQL INSERT INTO Empregados
           VALUES (:ID,:nome,:salario)
...
EXEC SQL UPDATE Empregados
         SET salário = salário + 100.00
         WHERE ID = :cod_emp
if (SQLCA.SQLCODE <= 0) EXEC SQL ROLLBACK;
...
}
EXEC SQL COMMIT;
...
```

1.10 - Execução Concorrente



- Permitir que múltiplas transações concorram na atualização de dados traz diversas complicações em relação à consistência desses dados.
- Razões para permitir a concorrência:
  - Transação consiste em diversos passos :
    - atividades de I/O
    - atividades de CPU
    - Podem ser executadas em paralelo -> aumentar o rendimento
  - Mistura de transações em execução simultânea : algumas curtas e outras longas. Se a execução das transações for seqüencial, uma transação curta pode ser obrigada a esperar até que uma transação longa precedente se complete. Assim reduz-se o tempo médio de resposta: o tempo médio para uma transação ser completada após ser submetida.

1.10- Execução concorrente (2)



T1: transfere fundos de A para B	T2: transfere 10% de A para B
read(A);	read(A);
A := A – 50;	temp := A * 0,1;
write(A);	A := A – temp;
read(B);	write(A);
B := B + 50;	read(B);
write(B);	B := B + temp;
	write(B);

tempo

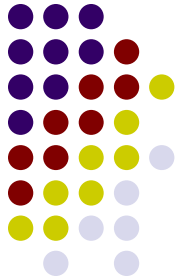
1.10- Execução concorrente (3)

Escalas de execução (*schedule*) em seqüência: observe que o estado do BD é sempre consistente.

T1	T2
read(A); **A := A – 50;** **write(A);** **read(B);** **B := B + 50;** **write(B);**	
	read(A); **temp := A * 0,1;** **A := A – temp;** **write(A);** **read(B);** **B := B + temp;** **write(B);**

T1	T2
	read(A); **temp := A * 0,1;** **A := A – temp;** **write(A);** **read(B);** **B := B + temp;** **write(B);**
read(A); **A := A – 50;** **write(A);** **read(B);** **B := B + 50;** **write(B);**	

1.10 - Execução concorrente (4)



Correta

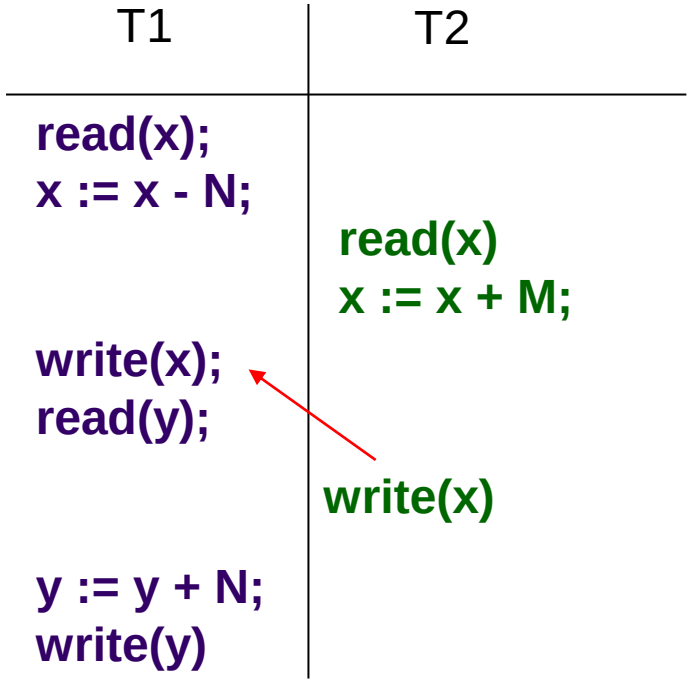
T1	T2
read(A);	
A := A – 50;	
write(A);	
	read(A);
	temp := A * 0,1;
	A := A – temp;
	write(A);
read(B);	
B := B + 50;	
write(B);	
	read(B);
	B := B + temp;
	write(B);

Incorreta

T1	T2
read(A);	
A := A – 50;	
	read(A);
	temp := A * 0,1;
	A := A – temp;
	write(A);
write(A);	
read(B);	
B := B + 50;	
write(B);	
	read(B);
	B := B + temp;
	write(B);

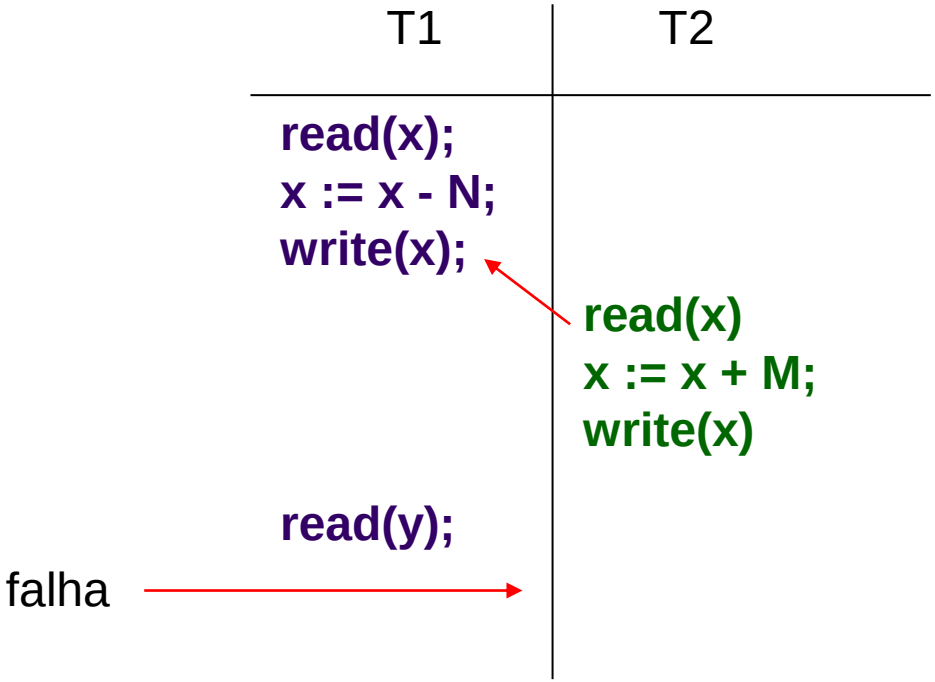
1.11 - Problema das atualizações perdidas

- Ocorre quando duas transações que acessam o mesmo item de dados possuem suas operações intercaladas de tal forma que o valor de algum item de dados fique incorreto.



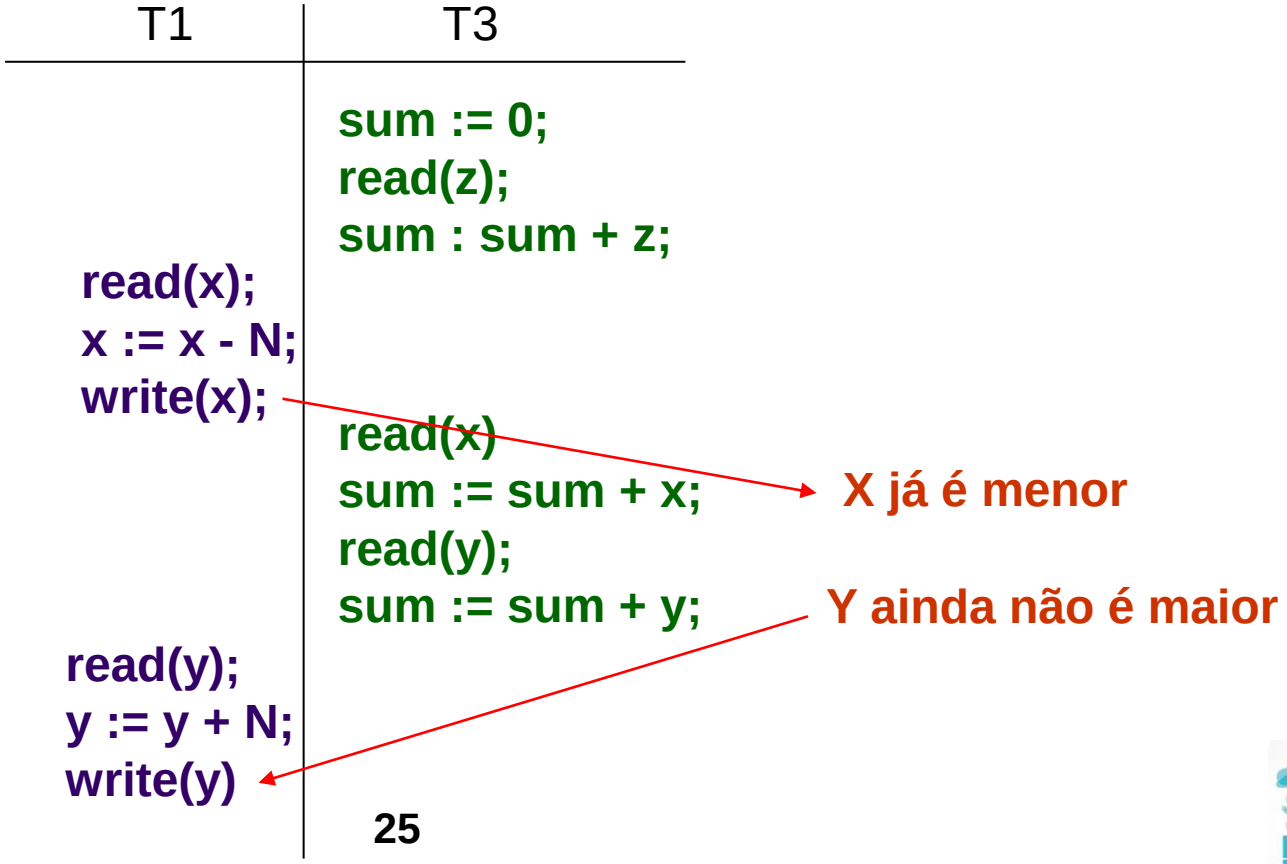
1.12 - Problema da dependência sem Commit (leitura suja)

- Ocorre quando uma transação altera um item de dados e depois ela falha por alguma razão. O item de dado é acessado por outra transação antes que o valor original seja confirmado.



1.13 - Problema do resumo incorreto (análise inconsistente)

- Se uma transação está calculando uma função agregada com um conjunto de registros e outras transações estão alterando alguns destes registros a função agregada pode calcular alguns valores antes deles serem alterados e outros depois de serem alterados.



2 - Serialização

- O sistema gerenciador de banco de dados deve controlar a execução concorrente de transações para assegurar que o estado do banco de dados permaneça consistente.
- A consistência do banco de dados, sob execução concorrente, pode ser assegurada garantindo-se que qualquer escala executada concorrentemente tenha o mesmo efeito de outra que tivesse sido executada sem qualquer concorrência.
- Isto é, uma escala de execução deve, de alguma forma, ser **equivalente a uma escala seqüencial (transações sem intercalação, executadas de forma serial)** .
- Formas de equivalência entre escalas de execução podem ser verificadas sob duas visões:
  - Por conflito
  - Por visão

2.1 - Serialização por Conflito (1)

- Considere uma escala de execução S com duas instruções sucessivas, I_i e I_j, das transações T_i e T_j (i <> j), respectivamente.

T_i	T_j
Ii(P)	Ij(Q)

- Se I_i e I_j referem-se a itens de dados diferentes, então é permitido alternar I_i e I_j sem afetar os resultados de qualquer instrução da escala.

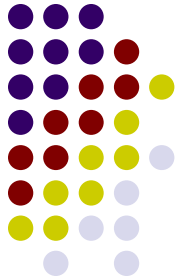
T_i	T_j
Ii(P)	Ij(Q)

- Se I_i e I_j referem-se ao mesmo item de dados Q, então a ordem dos dois passos pode importar.

T_i	T_j
Ii(Q)	Ij(Q)

?

2.1 - Serialização por Conflito (2)



1- I_i = **read**(Q) e I_j = **read**(Q)

A seqüência de execução de I_i e I_j não importa, já que o mesmo valor de Q é lido por T_i e T_j, independentemente da ordem destas operações.

2- I_i = **read**(Q) e I_j = **write**(Q)

Se I_i vier antes de I_j, então T_i não lê o valor de Q que é escrito por T_j na instrução I_j.

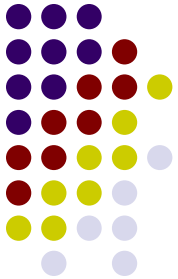
Se I_j vier antes de I_i, então T_i lê o valor de Q que é escrito por T_j.

Assim, a ordem de I_i e I_j importa.

3. I_i = **write**(Q) e I_j = **read**(Q)

A ordem de I_i e I_j importa por razões semelhantes às do caso anterior.

2.1 - Serialização por Conflito (3)



4. I_i = **write**(Q) e I_j = **write**(Q)

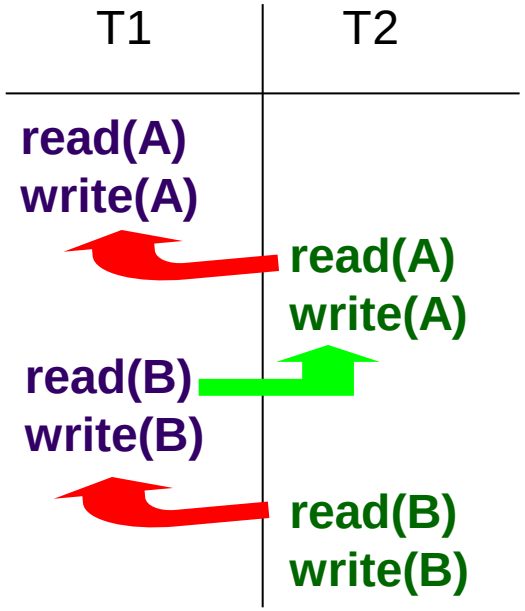
Como ambas as instruções são operações de escrita, a ordem dessas instruções não afeta T_i ou T_j.

Entretanto, o valor obtido pela próxima instrução **read(Q)** em S é afetado, já que somente o resultado da última das duas instruções **write(Q)** é preservado no banco de dados.

Se não houver nenhuma outra instrução de **write(Q)** depois de I_i e I_j em S, então a ordem de I_i e I_j afeta diretamente o valor final de Q no que se refere ao estado do banco de dados após a execução da escala S.

2.1 - Serialização por Conflito (4)

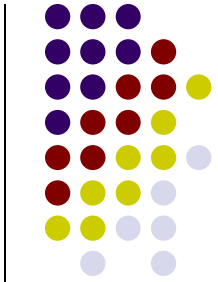
- Diz-se que duas instruções entram em **conflito** se elas são operações pertencentes a **transações diferentes, agindo no mesmo item de dados**, e pelo menos uma dessas instruções é uma operação de escrita (**write**).



A instrução write(A) de T1 entra em conflito com a instrução read(A) de T2.

Porém, a instrução write(A) de T2 não está em conflito com a instrução read(B) de T1.

2.2 - Serialização por Conflito (escalas equivalentes)

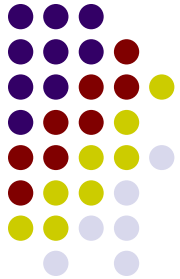


- Sejam I_i e I_j instruções consecutivas de uma escala de execução S.

- Se I_i e I_j são instruções de transações diferentes e não entram em conflito, então podemos trocar a ordem de I_i e I_j para produzir uma nova escala de execução S'.

- Diz-se que S e S' são equivalentes já que todas as instruções aparecem na mesma ordem em ambas as escalas de execução com exceção de I_i e I_j, cuja ordem não importa.

2.2 - Serialização por Conflito (escalas equivalentes (2))

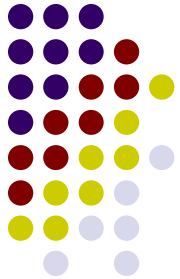


- Voltando à escala S anterior, a instrução write(A) de T2 não entra em conflito com a instrução read(B) de T1. Então é permitido trocar essas instruções para gerar uma escala de execução equivalente.

T1	T2
read(A)	
write(A)	
	read(A)
read(B) ↑	
	write(A) ↓
write(B)	
	read(B)
	write(B)

Em relação a um mesmo estado inicial do sistema, ambas as escalas (S e esta) Produzem o mesmo estado final no sistema.

2.2 - Serialização por Conflito (escalas equivalentes (3))



- Se as seguintes trocas de instruções não-conflitantes forem feitas ...
  - read(B) de T1 por read(A) de T2;
  - write(B) de T1 por write(A) de T2;
  - Write(B) de T1 por read(A) de T2

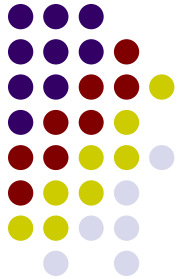
- ... uma escala com execuções seriais será obtida.

S'

T1	T2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

- Assim mostrou-se que a escala S é equivalente, no conflito, a uma escala seqüêncial.
-
- Essa equivalência garante que para um mesmo estado inicial do sistema, a escala S produzirá o mesmo estado final produzido por essa escala sequêncial.

2.2 - Serialização por Conflito (escalas equivalentes (4))



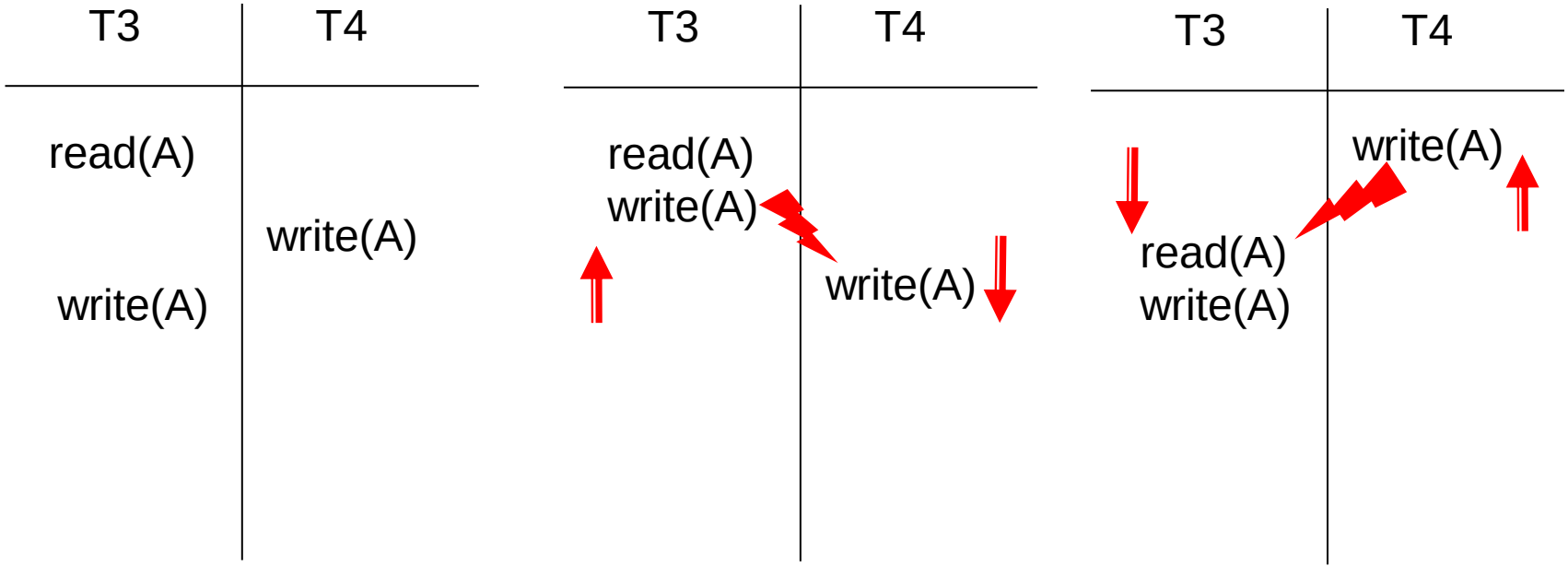
- Se uma escala de execução S puder ser transformada em outra, S', por uma série de trocas de instruções não conflitantes, dizemos que S e S' são **equivalentes em conflito**.

- O conceito de equivalência em conflito leva ao conceito de serialização por conflito.

- Uma escala de execução S é serializável por conflito se ela é equivalente em conflito a uma escala de execução seqüêncial.

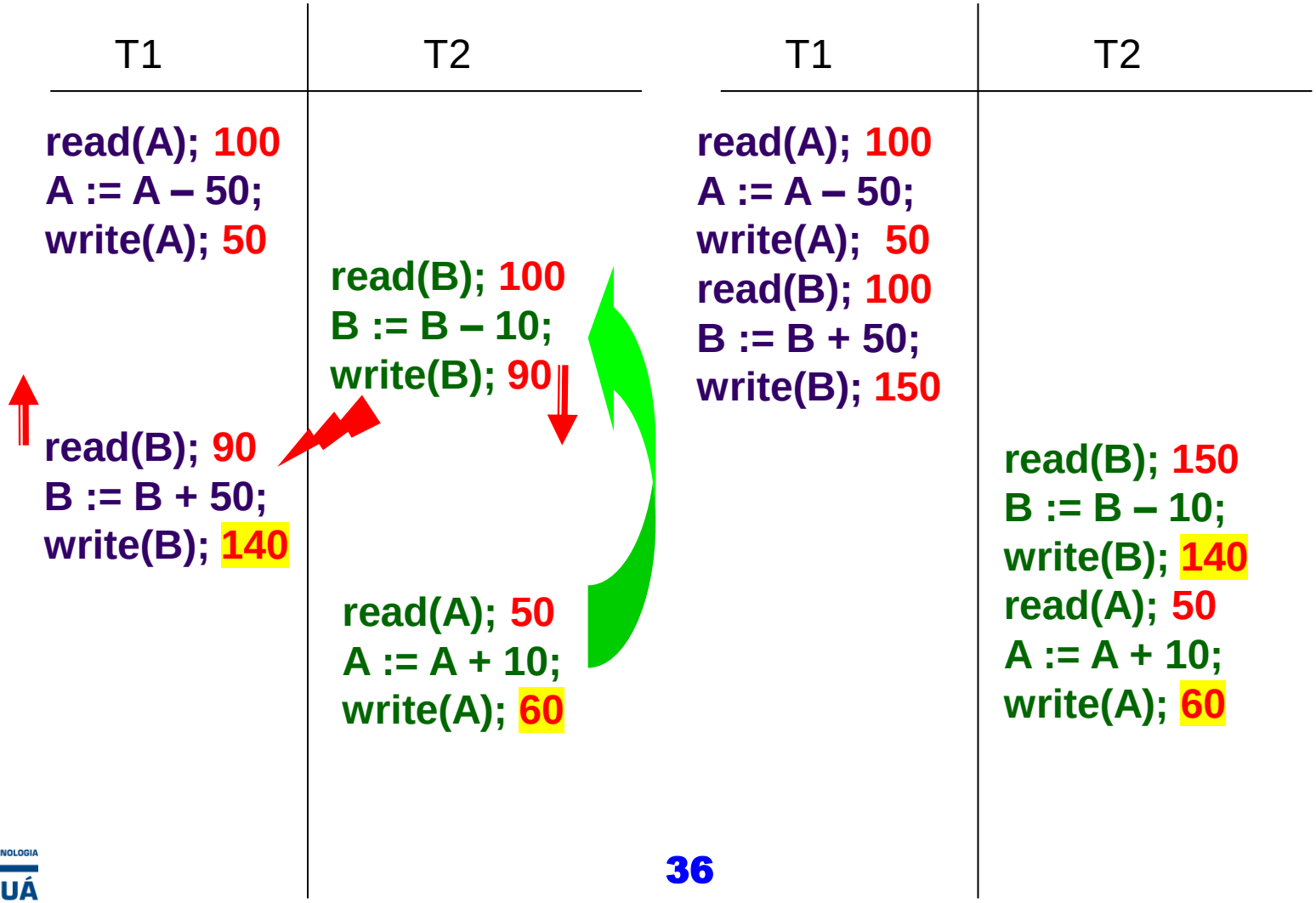
2.2.1 - Serialização por conflito - Exemplo

- A escala abaixo não é serializável por conflito pois não é equivalente em conflito nem à escala seqüencial <T3,T4> nem <T4,T3>.

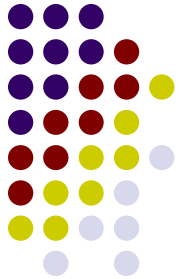


2.2.1 - Serialização por conflito – Exemplo

- A escala abaixo não é serializável por conflito, mas produz o mesmo resultado que uma escala seqüencial. Confira.



2.3 - Teste de serialização por conflito



- Seja S uma escala. Para saber se ela é serializável em relação às operações conflitantes é necessário criar um **grafo de precedência** para S.

- G = (V,E) em que V é um conjunto de vértices e E é um conjunto de arestas.
  - O conjunto de vértices é composto por todas as transações que participam da escala.
  - O conjunto de arestas consiste em todas as arestas $T_i \rightarrow T_j$ para as quais uma das seguintes condições é verdadeira:

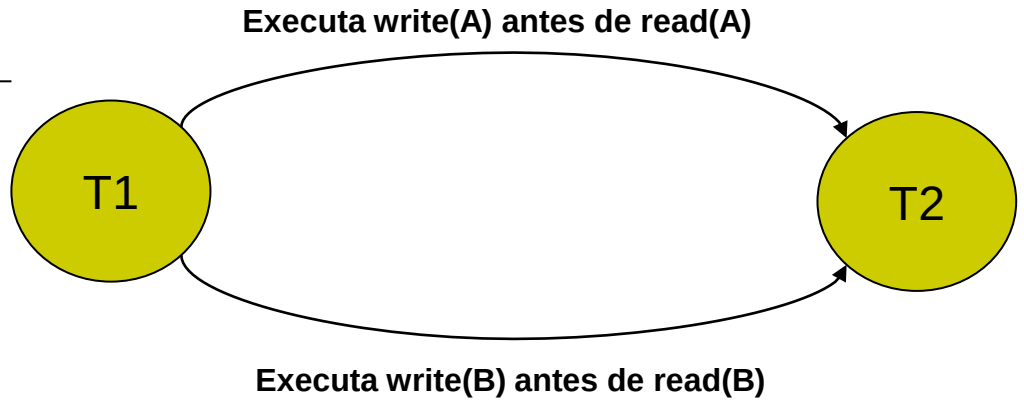
    - T_i executa **write**(Q) antes de T_j executar **read**(Q);
    - T_i executa **read**(Q) antes de T_j executar **write**(Q);
    - T_i executa **write**(Q) antes de T_j executar **write**(Q);

2.3 - Teste de serialização por conflito – Exemplo 1

- Se existir uma aresta $T_i \rightarrow T_j$ no grafo de precedência, então, em qualquer escala seqüencial **S'** equivalente a **S**, T_i deve aparecer antes de T_j.

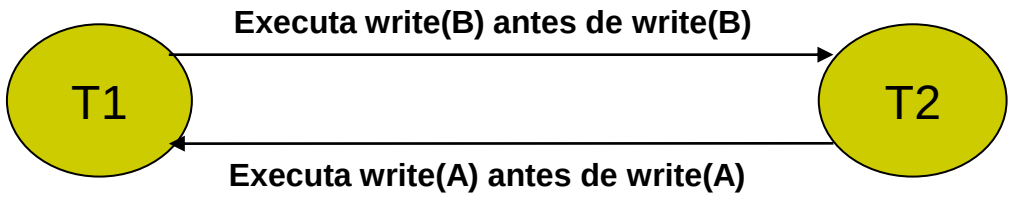
T1	T2

read(A);
A := A – 50;
write(A);
read(B);
B := B + 50;
write(B);
	read(A);
	temp := A * 0,1;
	A := A – temp;
	write(A);
	read(B);
	B := B + temp;
	write(B);



2.3 - Teste de serialização por conflito – Exemplo 2

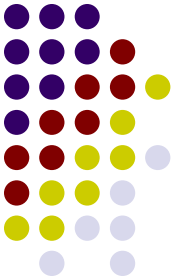
T1	T2
read(A)	
A := A - 50	
	read(A)
	temp := A * 0,1
	A := A – temp
	write(A)
	read(B)
write(A)	
read(B)	
B := B + 50	
write(B)	
	B := B + temp;
	write(B)



Ciclo no grafo: se o grafo de precedência possui ciclo, então a escala S não é serializável por conflito.

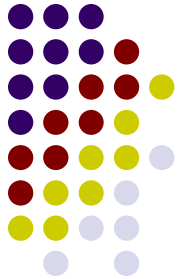
A ordem de serialização pode ser obtida por meio da classificação topológica, que estabelece uma ordem linear para a escala consistente com a ordem parcial do grafo de precedência.

3 – Propriedades Transacionais em BDs NO-SQL



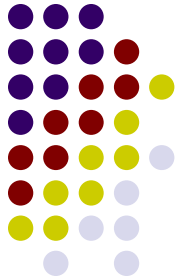
- **Bancos de dados relacionais tem problemas**
  - **Escalabilidade vertical**: adicionar recursos de hardware a uma mesma máquina (+ memória, + armazenamento, + processadores, + velocidade de processamento) ➡ limite físico computacional e alto custo
  - **Escalabilidade horizontal:** particionar os dados em várias máquinas, **torna a manutenção das tabelas bem difícil e complicado**; depende da fragmentação se é horizontal (linhas) ou vertical (colunas)

3 – Propriedades Transacionais em BDs NO-SQL



- **Bancos de dados relacionais tem problemas**
  - **alto volume de dados com baixa velocidade de resposta** - a medida que os dados crescem na base, fica cada vez mais difícil a disponibilização rápida das informações.
  - **os dados mudam constantemente** - alterar a entidade no mundo relacional em algumas situações não é fácil, dependendo do tipo do dado alguns bancos simplesmente bloqueiam o acesso a tabela para realizar uma simples alteração de coluna.

3 – Propriedades Transacionais em BDs NO-SQL

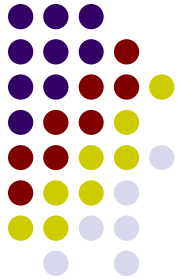


- **Bancos de dados relacionais tem problemas**
  - Protocolos de bloqueio de efetivação em duas fases distribuído – 2PC – para preservar as propriedades ACID, em especial a consistência, aumentam consideravelmente o tempo de resposta do sistema

latência muito alta implica em **baixa disponibilidade**

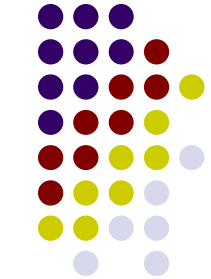
- Bancos de dados NO-SQL precisam de alta disponibilidade e redundância (mesmo dado em muitos nós) → Não é possível usar propriedades ACID

4- Propriedades BASE - NoSQL



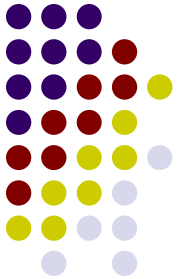
- **B**asically **A**vailable – Basicamente Disponível : disponibilidade é prioridade, o sistema deve estar em funcionamento na maior parte do tempo
- **S**oft-State – Estado Leve : não precisa ser consistente o tempo todo
- **E**ventually Consistent – Eventualmente Consistente : consistente em algum momento não determinado, a consistência nem sempre é mantida para todos os nós, nós podem não ter a mesma versão dos dados.

4.1- Comparativo ACID x BASE



ACID	BASE
Consistência forte	Fraca consistência
Isolamento	Disponibilidade em primeiro lugar
Concentra-se em "commit"	Melhor esforço em disponibilidade para partições
Transações aninhadas	Respostas aproximadas
Conservador (pessimista, bloqueia todos os registros para evitar conflitos)	Agressivo (otimista, detectam os conflitos e depois faz o tratamento)
Evolução difícil (por exemplo, esquema)	Evolução mais fácil

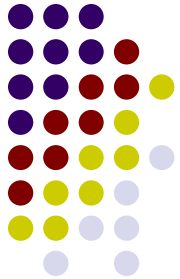
5 – Teorema CAP



- Consistência – **C**onsistency.
- Disponibilidade – **A**vailability.
- Tolerância ao Particionamento - **P**artition tolerance.
- Teorema CAP : é impossível garantir essas três propriedades ao mesmo tempo
- é possível garantir quaisquer duas dessas propriedades ao mesmo tempo

Gilbert, S.; Lynch, N. A. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, 33(2):51–59, 2002.

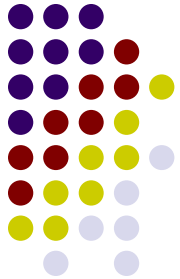
5.1 - Consistency - Consistência



O sistema garante a leitura do dado mais atualizado, quando ele foi escrito.

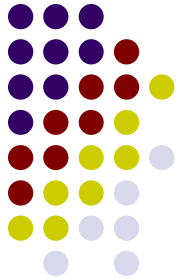
O cliente pode ler o dado no mesmo nó que este dado foi escrito ou de um nó diferente, o mesmo dado será retornado para a aplicação cliente. Mesmo que alguém tenha mudado o estado do domínio da aplicação com novas informações, o comportamento de consistência, garantirá que o cliente não verá dados velhos, apenas os novos dados serão visualizados.

5.2- Availability - Disponibilidade



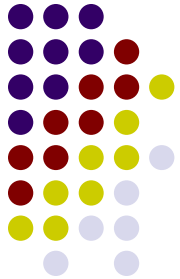
- Quando o cliente lê ou escreve um dado em um dos nós, o nó que está sendo utilizado pelo cliente, pode estar indisponível.
- Isso não quer dizer que um nó não pode falhar, o que esse comportamento quer dizer é que, um nó pode sim ficar offline/cair/falhar, mas o sistema/aplicação continuaria disponível

5.2- Availability - Disponibilidade



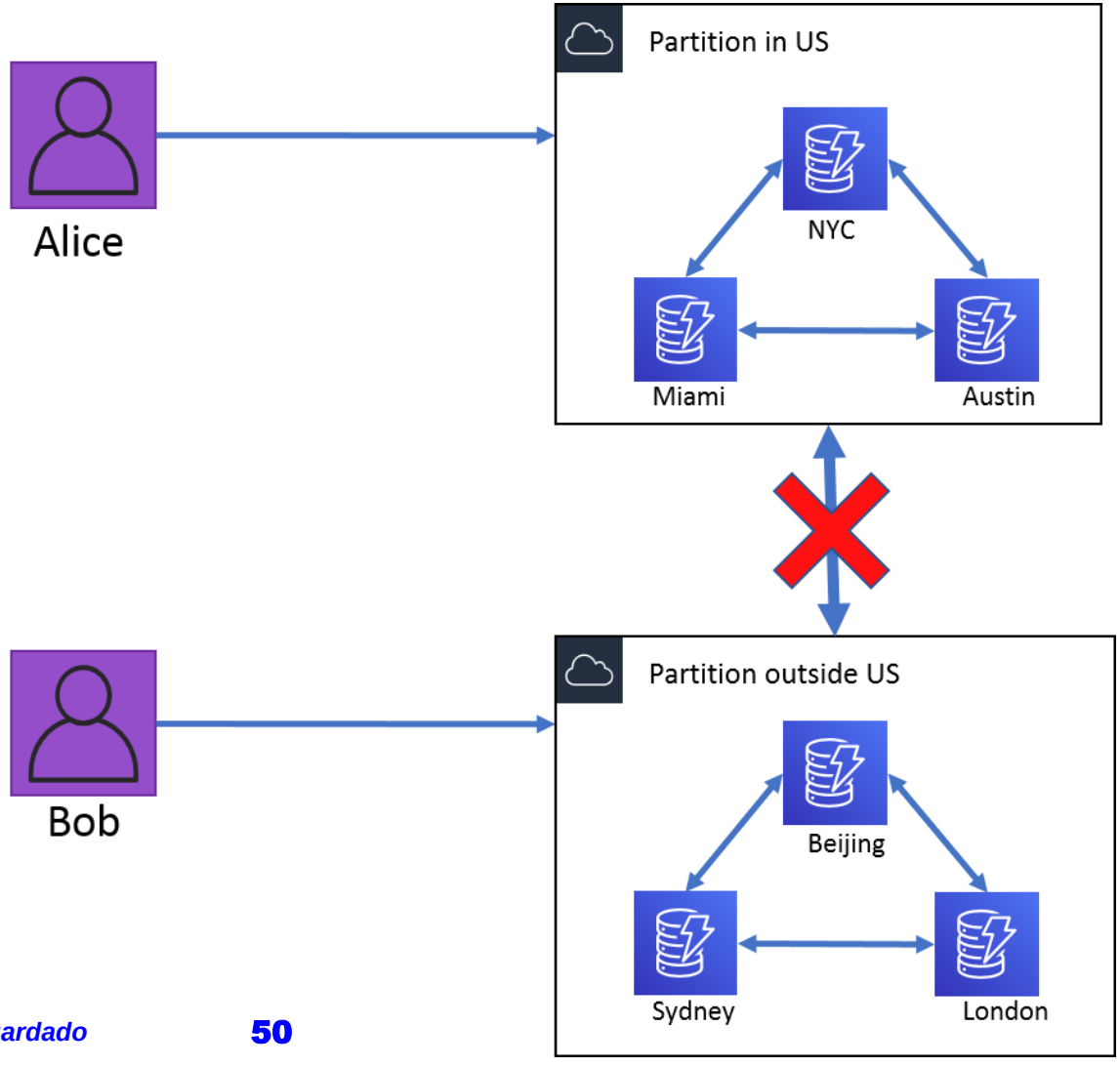
- Se um sistema é capaz de obter acesso de leitura/escrita em um nó que não possui falhas e este responde em um tempo razoável, temos aqui a garantia de disponibilidade.
- **Mas pode trazer um dado não atual**

5.3- Partition Tolerance – Tolerância a falhas de partição

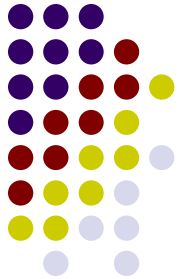


É a garantia de que o sistema continue operante mesmo no caso da ocorrência de uma falha que isole os nós em grupos, onde os nós de um grupo não consigam se comunicar com os dos demais grupos. O sistema **continua operando, mesmo que aconteça alguma falha na rede** (falha de conectividade).

5.3- Partition Tolerance – Tolerância a falhas de partição



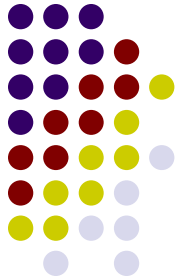
5.3- Partition Tolerance – Tolerância a falhas de partição



Os sistemas distribuídos são, por natureza, não confiáveis. Eles são suscetíveis a diferentes tipos de problemas: falhas na rede, perda de mensagens, quebra de máquinas, ataques maliciosos, etc. Quanto mais nós no sistema, maior o risco de problemas.

A ocorrência de problemas como esses podem impedir (ou dificultar) a comunicação entre os nós, causando uma **partição do sistema**: ou seja, os nós podem ficar "isolados" em grupos que conseguem se comunicar internamente, mas que não conseguem se comunicar com outros grupos.

5.3- Partition Tolerance – Tolerância a falhas de partição

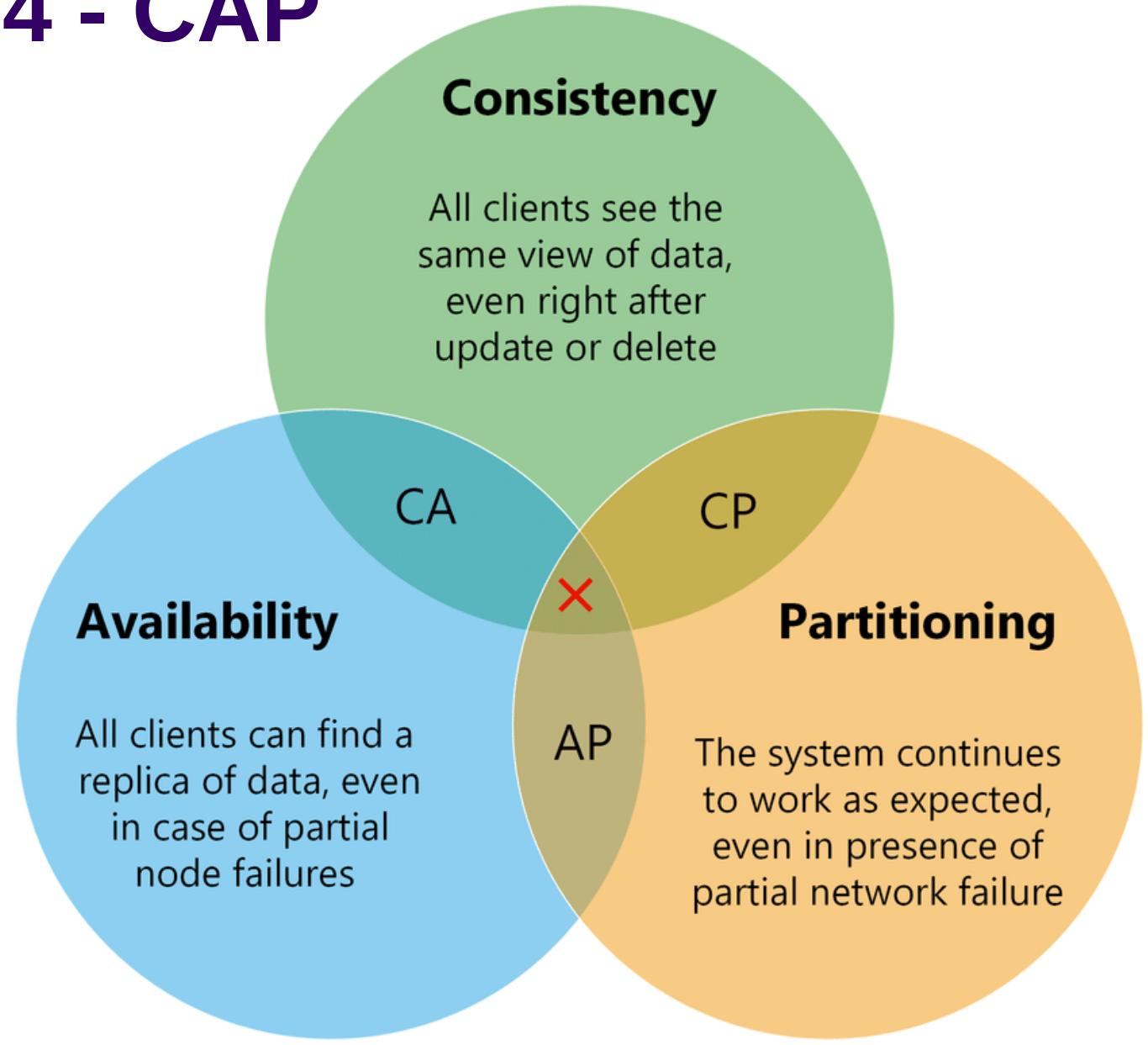


Para garantir a consistência em caso de partição do sistema, duas estratégias são possíveis:

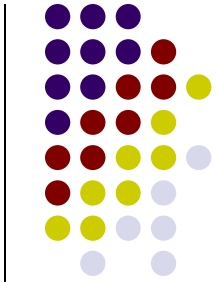
1) fazer com que seus nós deixem de receber requisições dos clientes enquanto o problema da partição persistir;

2) continuar recebendo as requisições dos clientes, mas elas só serão atendidas quando o problema da partição for resolvido.

5.4 - CAP

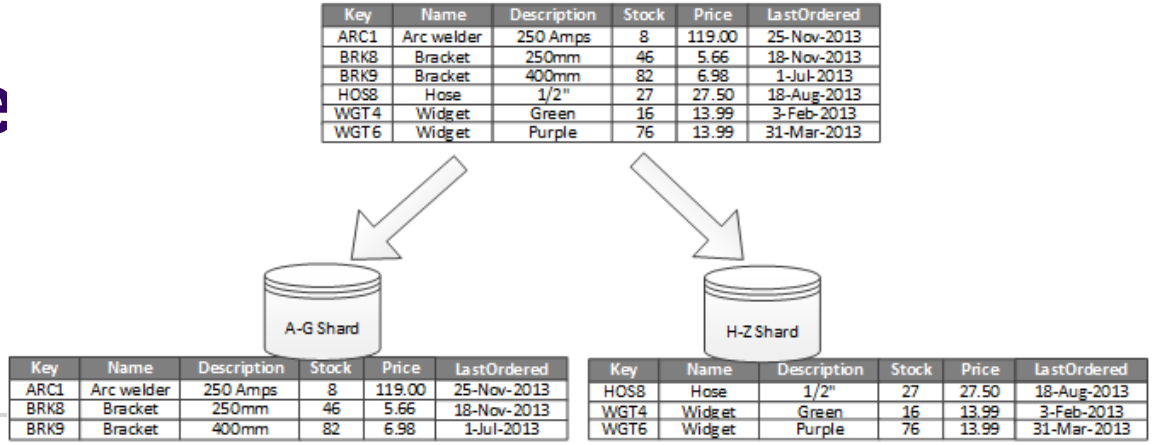

Consistency
All clients see the same view of data, even right after update or delete
CA
CP
Availability
All clients can find a replica of data, even in case of partial node failures
AP
Partitioning
The system continues to work as expected, even in presence of partial network failure

6 – Combinações CAP

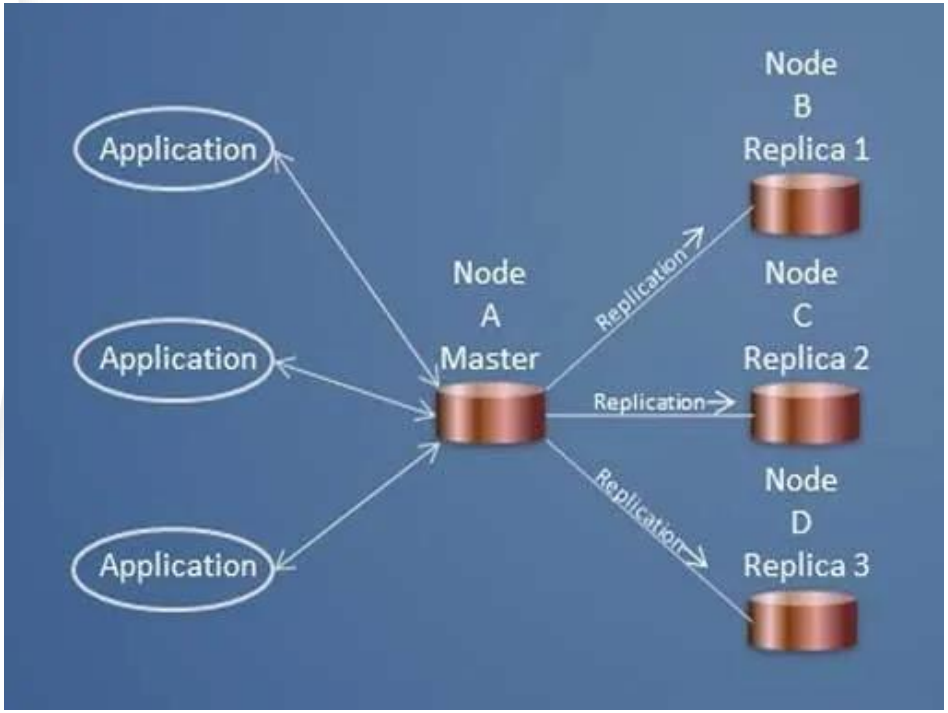


- Dependendo da aplicação escolher entre consistência forte, alta disponibilidade e tolerância ao particionamento
- **Sistemas CA (Consistência e Disponibilidade)**
- **Sistemas CP (Consistência e Particionamento)**
- **Sistemas AP (Disponibilidade e Particionamento)**

6.1 – Modelos de Distribuição



- Particionamento (*Sharding*)
  - *colocar diferentes partes dos dados em diferentes máquinas*
  - *Permite paralelismo nos acessos*
  - Escala escritas e leituras
- Replicação
  - *Copiar os mesmos dados em diferentes locais*
  - Escala somente leituras
  - Dois modelos possiveis:
    - Mestre-Escravo e Par-a-Par
- Particionamento + Replicação

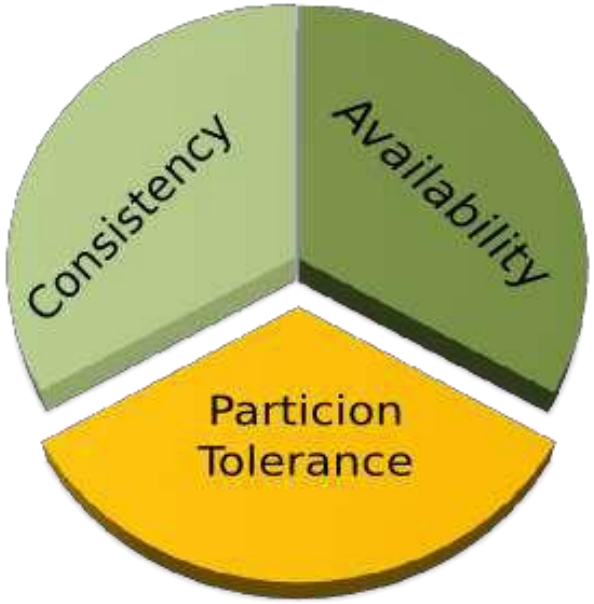


6.2 - Sistemas CA

Os sistemas com consistência forte e alta disponibilidade não sabem lidar com a possível falha de uma partição.

Caso ocorra, sistema inteiro pode ficar indisponível até o membro do cluster voltar.

Situação em que o sistema garante apenas alta disponibilidade (**em um único nó**) Uma máquina não pode ser particionada. -> BDs Relacionais Centralizados



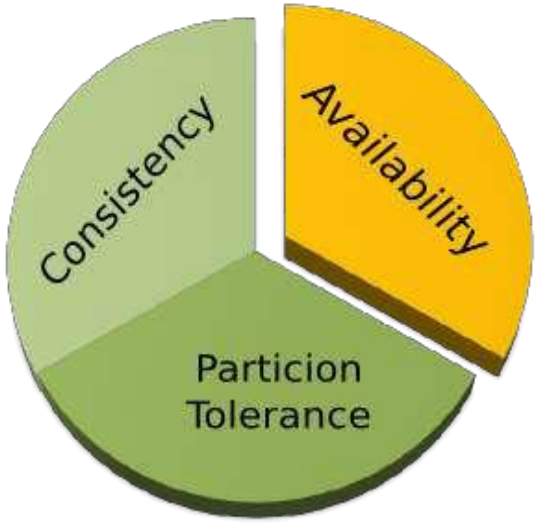
6.2 - Sistemas CA

6.3 - Sistemas CP

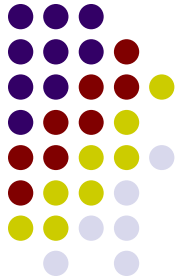
Para sistemas que precisam da **consistência forte e tolerância a particionamento** é necessário abrir a mão da disponibilidade (um pouco).

O sistema fica operante no caso de particionamento, mas pode demorar bastante tempo para conseguir responder às requisições que recebe ou, até mesmo, nunca respondê-las.

Exemplos são Google BigTable, HBase ou MongoDB entre vários outros.

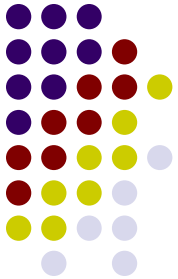


6.3.1- Consistência por meio de Quóruns



- Na replicação de dados, quanto mais nós envolvidos no tratamento de leituras ou escritas, maior a chance de evitar inconsistências
- Quórum de uma operação = quantidade de votos (confirmações) que a operação precisa receber para poder ser aplicada sobre um item de dados em um nó
- Cada nó com uma réplica do item tem direito a um voto

6.3.1- Quórum de Escrita

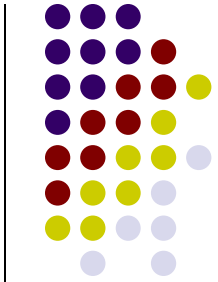


- W = quórum de escrita
  - número de nós que confirmam uma escrita
- N = fator de replicação dos dados
  - número de réplicas
- Para garantir escritas com consistência forte, a seguinte fórmula precisa ser respeitada:

  $$W > N / 2$$

  - O número de nós que confirmam uma escrita precisa ser maior que a metade do número de réplicas do item a ser escrito, ou seja, a maioria
  - Dessa forma, se houver escritas conflitantes, somente uma será confirmada pela maioria dos nós

6.3.1- Quórum de Leitura

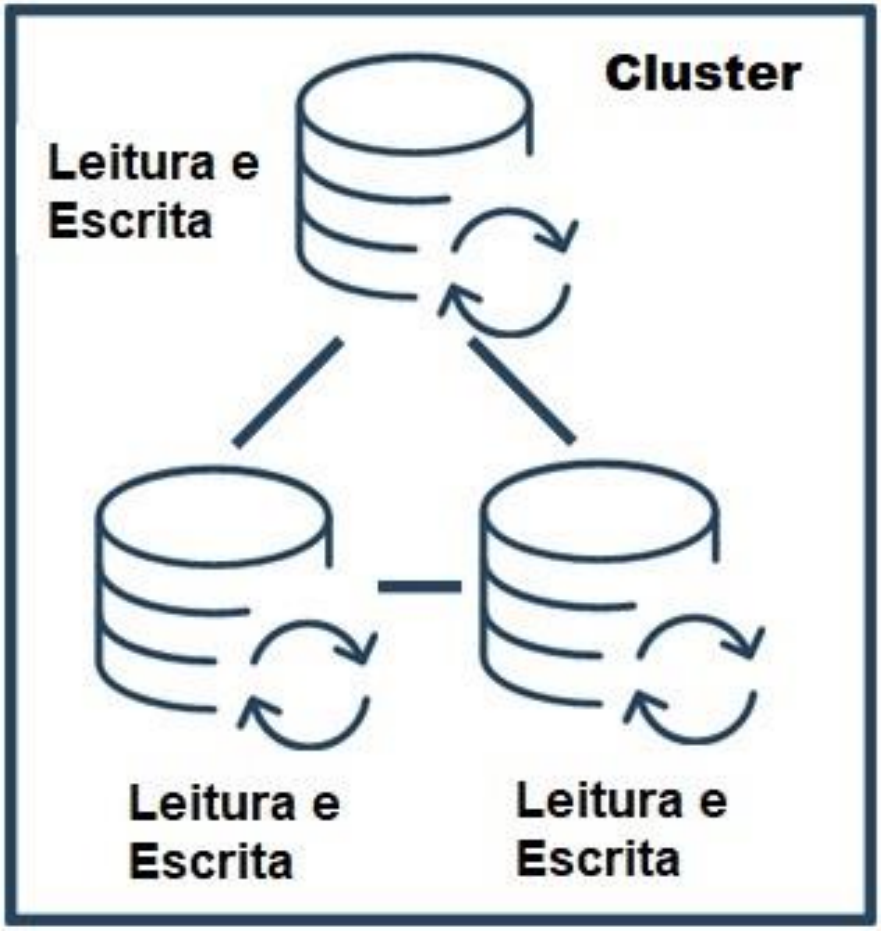


- Quantos nós é preciso contatar para se ter a garantia de que o valor mais recente para um item de dado foi lido?
  - A resposta depende do quórum de escrita W
- R = quórum de leitura
  - número de nós contatados para a leitura
- Para garantir leituras altamente consistentes, a seguinte fórmula precisa ser respeitada:

$$R + W > N$$

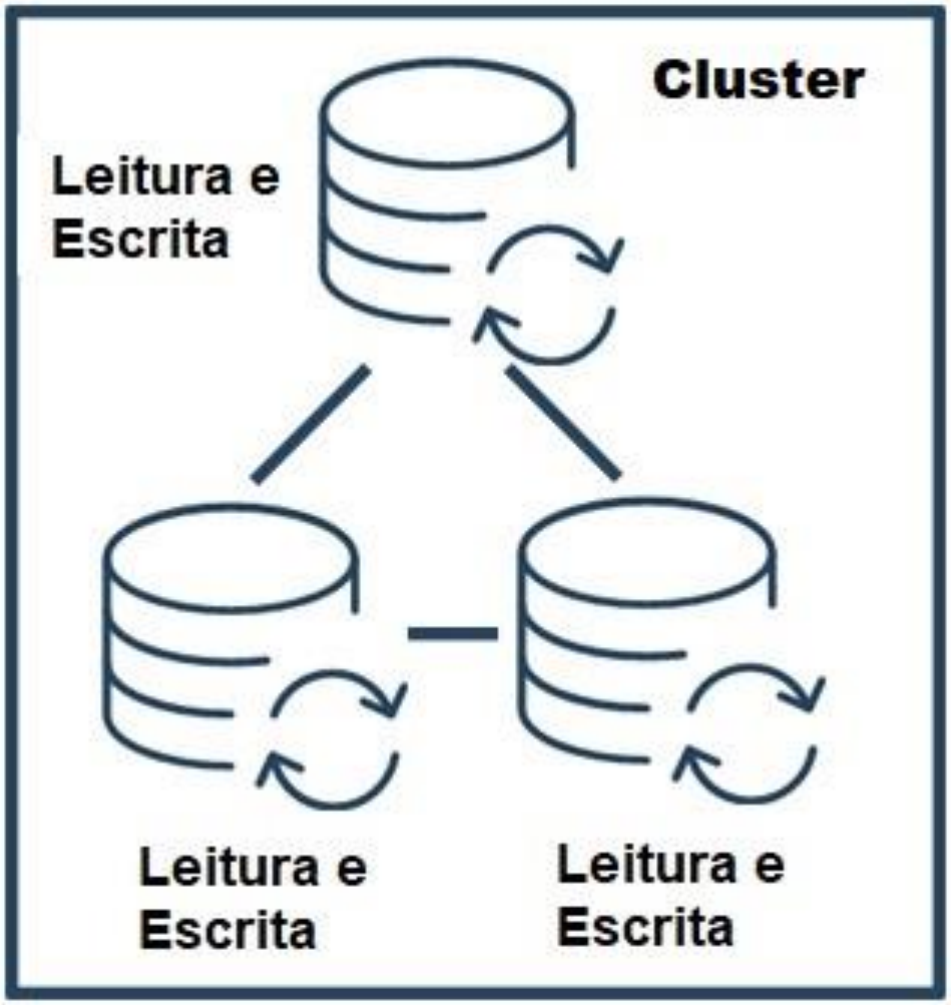
  - A fórmula garante que, entre os R nós de réplicas contatados, haverá pelos menos 1 que tem a versão mais nova do item de dado

6.3.2 - Sistemas CP - Consenso



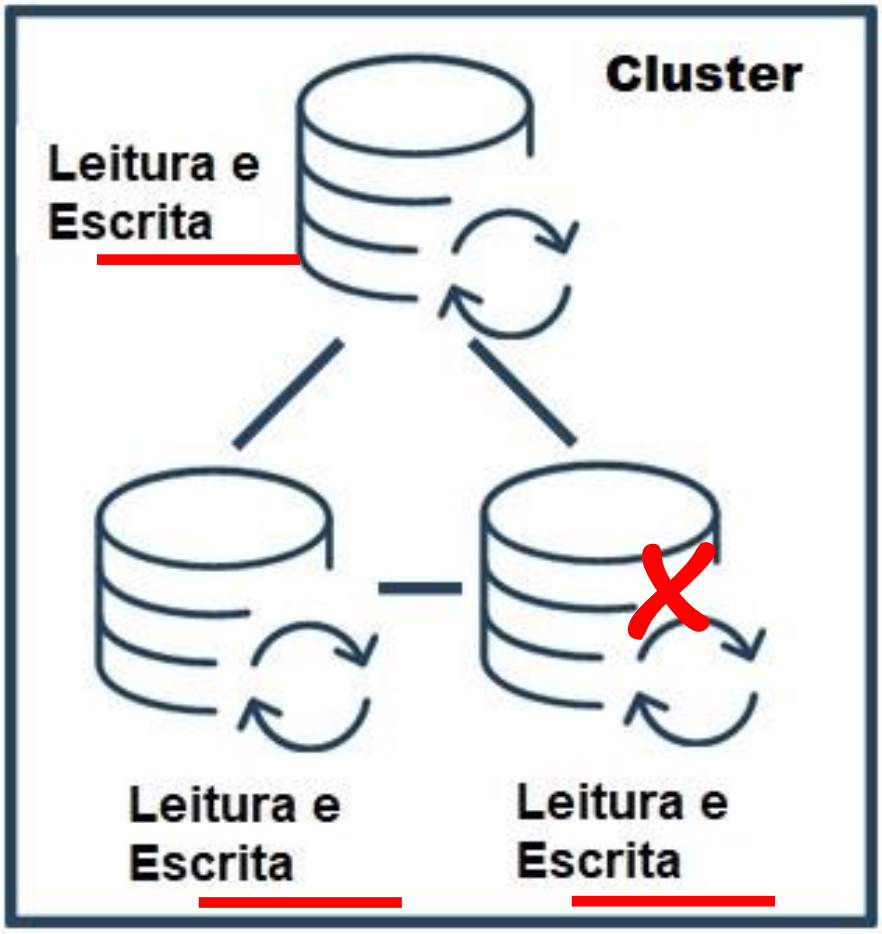
Se precisar de leituras rápidas e fortemente consistentes, pode-se exigir que as escritas sejam reconhecidas por todos os nós, permitindo assim que as leituras entrem em contato com apenas um. Isso significaria que as escritas são lentas, pois elas precisam entrar em contato com todos os nós, e o sistema não seria capaz de tolerar a perda de um nó.

6.3.2 - Sistemas CP - Consenso



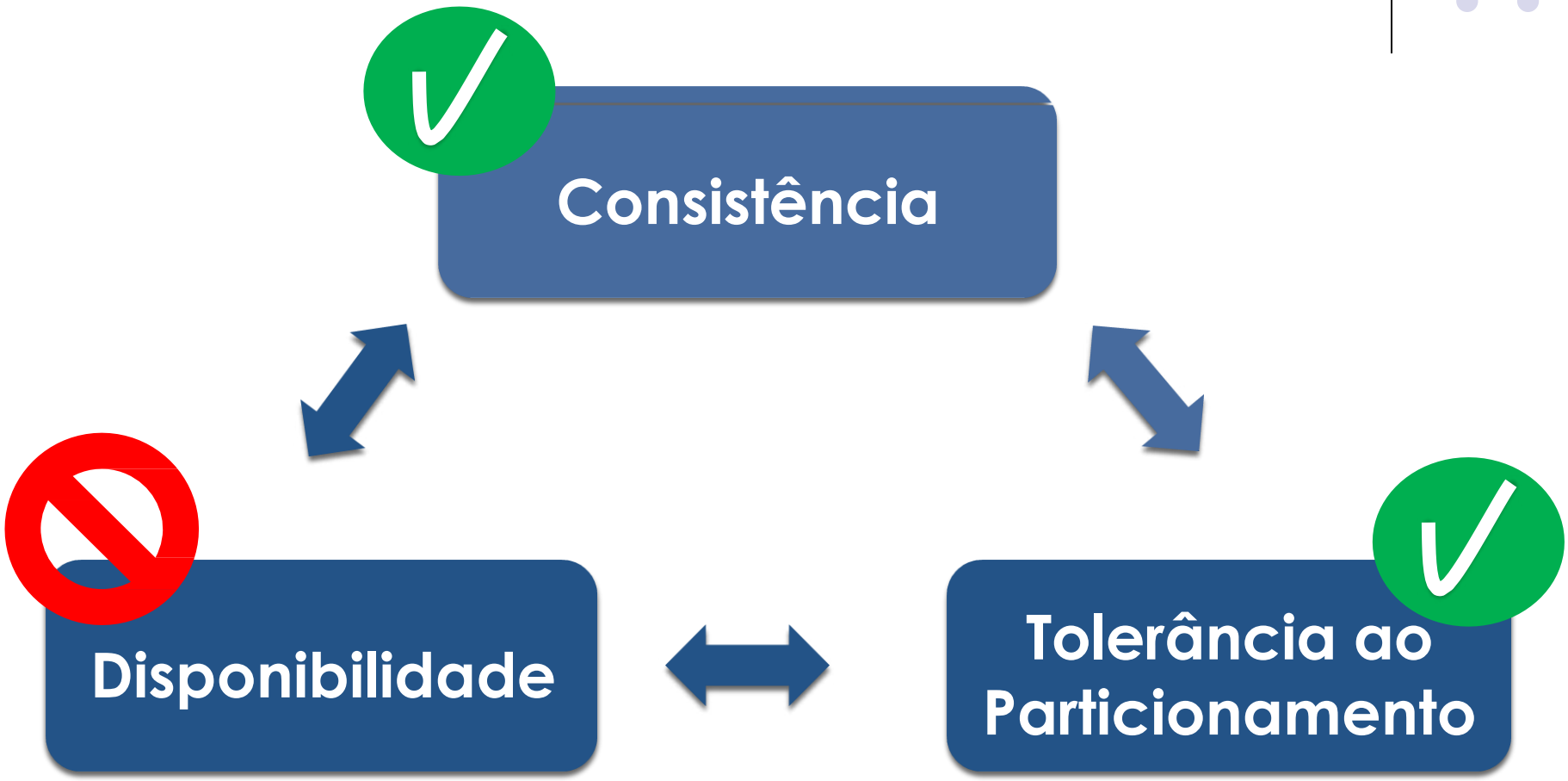
Consenso : Fator de Replicação **N** igual a 3
Consideremos **R** = 1 e **W** = 3
Portanto, para escrever no banco é necessária a confirmação de escrita de três nós (quórum)
W > N/2 e R+W >N

6.3.2 - Sistemas CP - Consenso



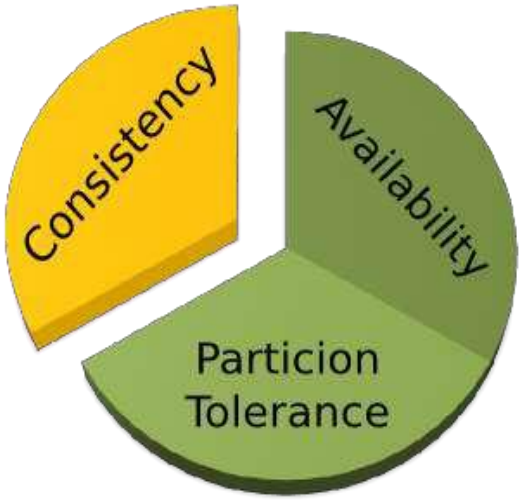
Consenso : Com a falha de um nó a escrita é comprometida (não há quórum, precisa W = 3) Perde-se a disponibilidade (A) para escrita (sistema para de escrever), mas a leitura ainda pode ocorrer em qualquer um dos demais nós (R = 1). Se falhar mais um nó a leitura ainda pode ser realizada com consistência.

6.3.2 - Sistemas CP



6.4 - Sistemas AP

- Há sistemas que jamais podem ficar offline, portanto não desejam sacrificar a disponibilidade. Para ter alta disponibilidade mesmo com uma tolerância a particionamento é preciso comprometer a consistência

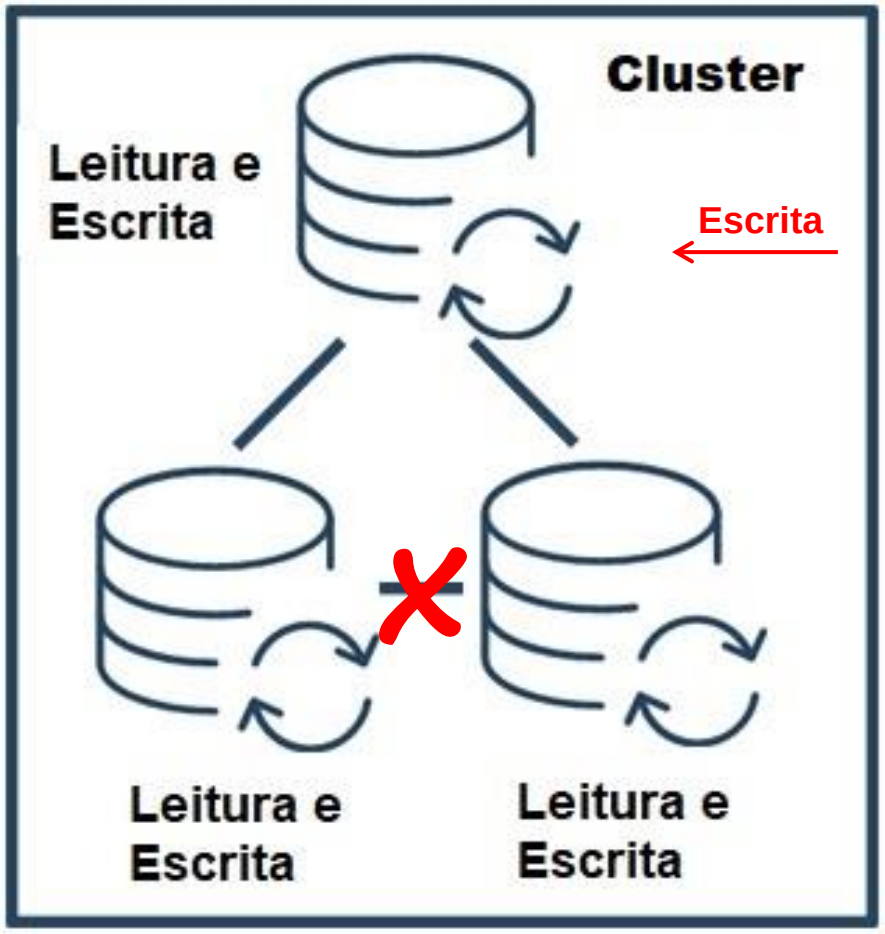


- Exemplos de Bancos são: Cassandra, MongoDB, Voldemort

6.4 - Sistemas AP

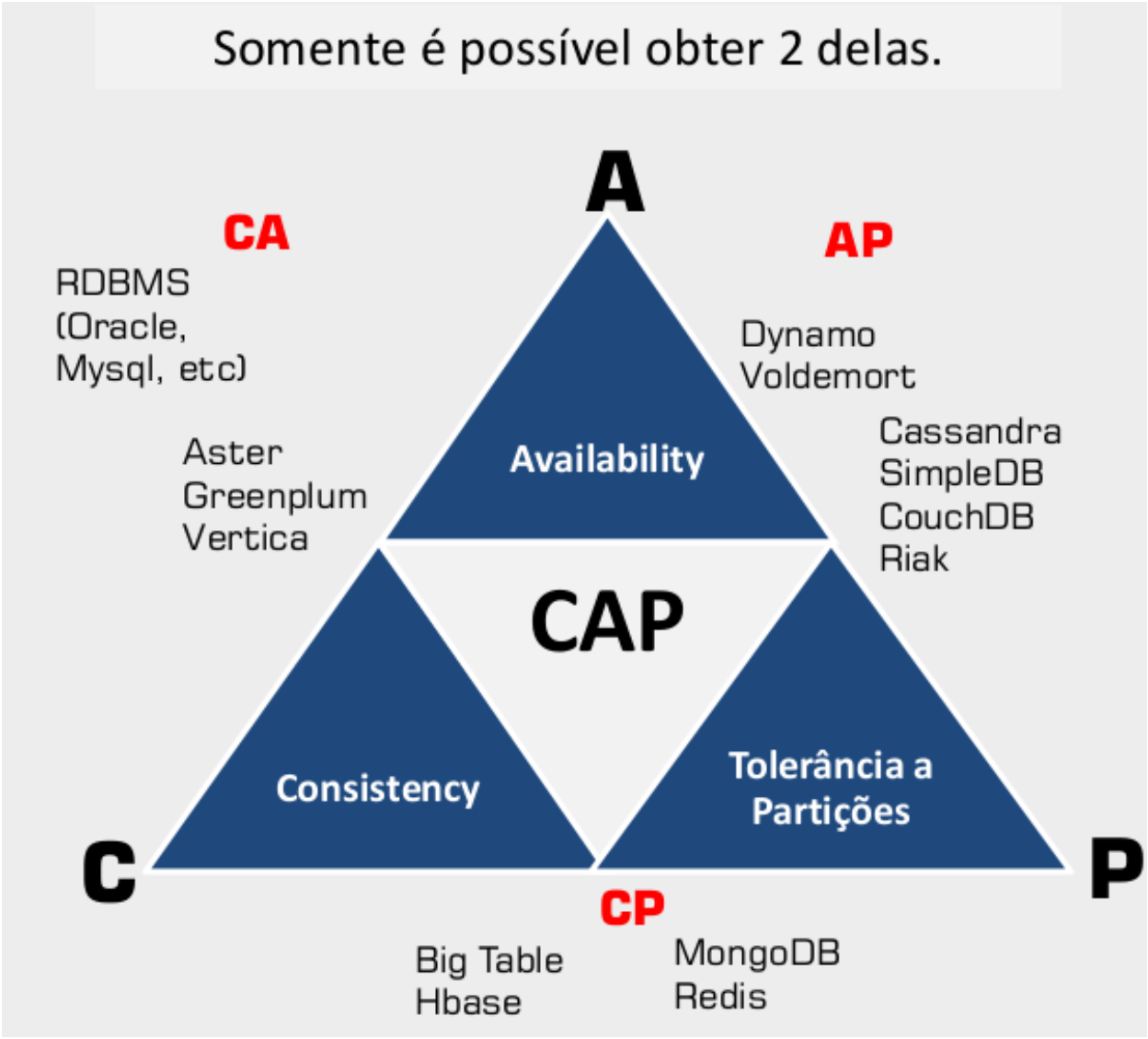
Consistência Eventual : sempre disponível para escrita e depois sincroniza os dados com os demais nós.

Se em uma escrita a comunicação com algum nó falhar, os dados são replicados apenas aos servidores disponíveis sem que o serviço fique indisponível (escrita ainda é possível). Após o reestabelecimento da comunicação, os dados são sincronizados com os demais servidores.



6.4 - Sistemas AP

7 – Visão Geral CAP



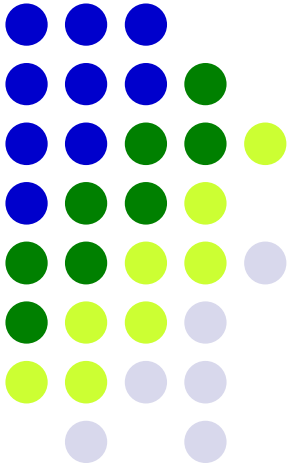
Referências

- Sadalage, Pramod J.; Fowler, Martin. NOSQL Essencial , Editora Novatec, 2013
- Gilbert, S.; Lynch, N. A. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, 33(2):51–59, 2002.

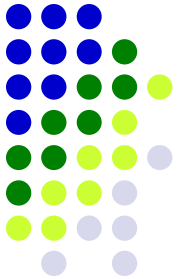
Controle de Concorrência

Sistemas de Banco de Dados

Prof. Antonio Guardado

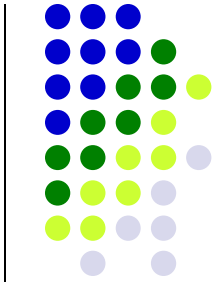


1. Problema



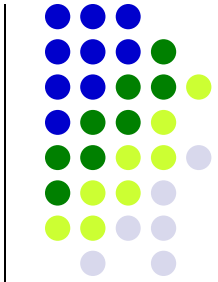
- Escalas não serializáveis podem violar a consistência do BD.
- Forçar um comportamento sequencial fazendo uma transação esperar que a outra execute determinadas operações.
  - Como ? Bloqueando o acesso aos dados que uma transação utiliza para as demais transações concorrentes.
  - A transação solicitará ao Subsistema de Controle de Concorrência o bloqueio dos dados.

1.1 - Protocolos com base em bloqueios (lock)



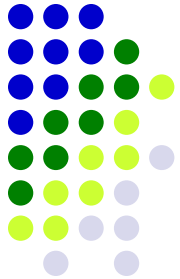
- Obrigar que o acesso a dados seja feito de maneira mutuamente exclusiva preservando o ISOLAMENTO
- Enquanto uma transação acessa um item de dados, nenhuma outra pode modificá-lo.
- As transações devem solicitar o bloqueio de modo apropriado, dependendo do tipo de operação realizada.
- O gerenciador de controle de concorrência concede (*grants*) o bloqueio para a transação (ela pode ter que esperar).

1.2 -Tipos de Bloqueios



- Compartilhado: Se uma transação Ti obteve um bloqueio compartilhado (denotado por S de *share*) sobre o item Q, então T*i* pode ler, mas não escrever Q
- Exclusivo: Se uma transação T*i* obteve um bloqueio exclusivo (denotado por X de *eXclusive*) sobre o item Q, então T*i* pode tanto ler como escrever Q

1.3 -Função de compatibilidade

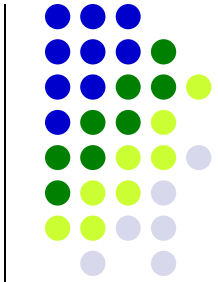


- Um elemento comp(A,B) da matriz possui valor verdadeiro se, e somente se, o modo A é compatível com o modo B

	S	*X*
S	verdade	falso
X	falso	falso

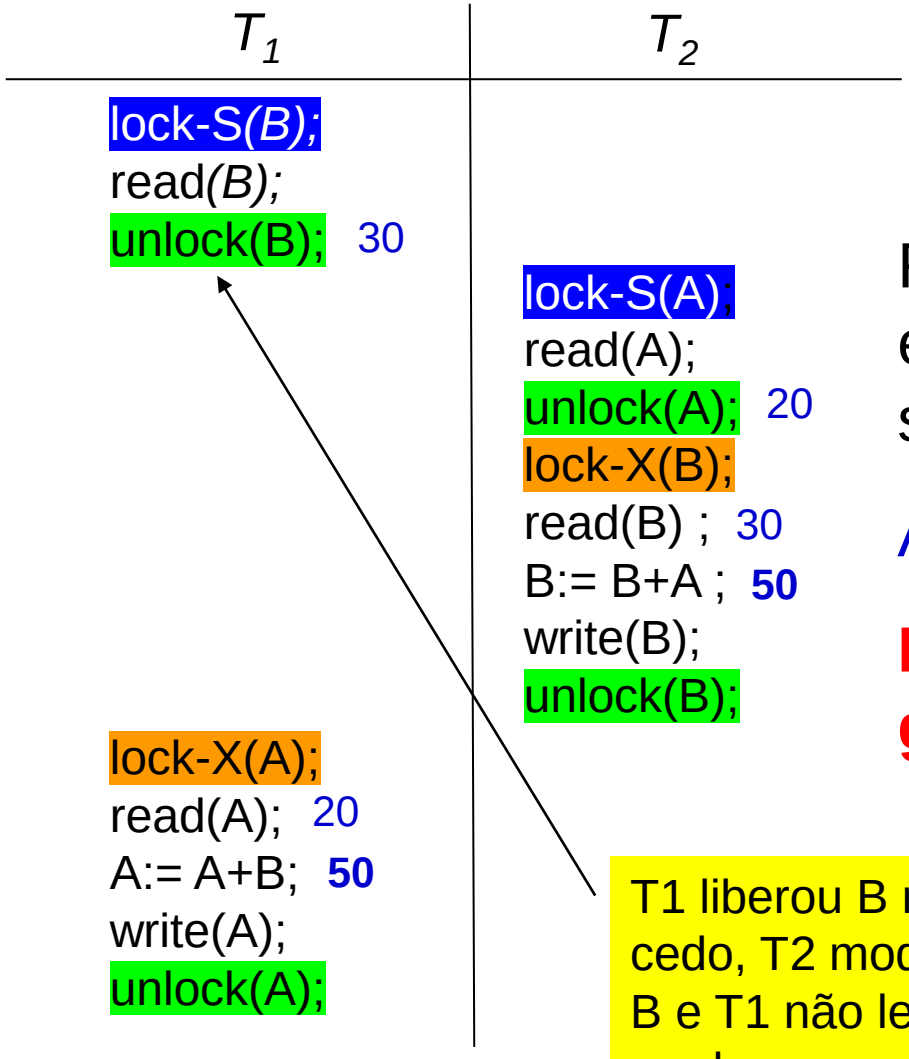
S = Share
X - EXclusive

1.4 -Exemplos de locks



- Instruções:
  - lock-S(Q) : bloqueio no modo compartilhado
  - lock-X(Q) : bloqueio no modo exclusivo
  - unlock(Q) : remove todos os bloqueios
- Para manter o acesso a um item de dado, a transação precisa primeiro bloqueá-lo. Se já estiver incompativelmente bloqueado, o gerenciador não concederá o bloqueio até que todos os bloqueios incompatíveis sejam desfeitos

1.4 - Exemplos

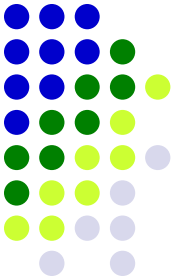


Resultado deste escalonamento não-serializável

A=50 e B=50

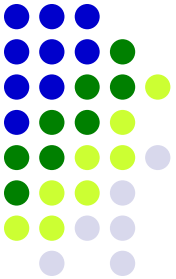
Bloqueios por si só não garantem a serialização

2 - Protocolo de bloqueio



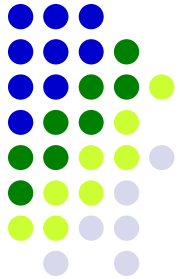
- Conjunto de regras que determina quando uma transação pode ou não bloquear ou desbloquear um item de dados
- Restringe o número de escalas possíveis (serializadas)

2.1 - Protocolo de bloqueio em duas fases (Two-Phase Locking ou 2PL)



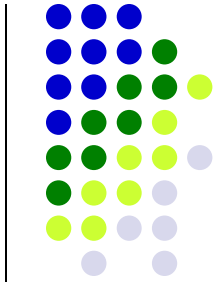
- Garante a serialização
- Exige que cada transação emita suas solicitações de bloqueio e desbloqueio em duas fases:
  1. Fase de expansão
  2. Fase de encolhimento

2.1.1 - Fases do 2PL



- Fase de expansão ou crescimento: uma transação pode obter bloqueios, mas não pode liberar nenhum
- Fase de encolhimento ou retrocesso: uma transação pode liberar bloqueios, mas não consegue obter nenhum bloqueio novo
- Não garante a ausência de deadlock
- Pode ocorrer o *rollback* em cascata

2.1.2 - Exemplo do 2PL



$T_{1'}$	$T_{2'}$
lock-S(*B*);	lock-S(*A*);
read(*B*);	read(*A*);
lock-X(*A*);	lock-X(*B*);
unlock(*B*);	unlock(*A*);
read(*A*);	read(*B*) ;
A:= A+B;	B:= B+A ;
write(*A*);	write(*B*);
unlock(*A*);	unlock(*B*);

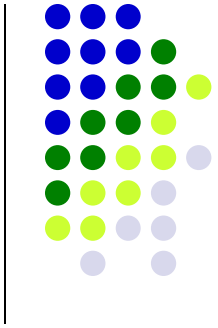
Expansão

Obtenção dos bloqueios

Encolhimento

Liberação dos bloqueios

2.1.2- Exemplo do 2PL



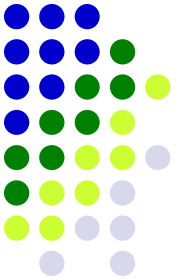
$T_{1'}$	$T_{2'}$
lock-S(*B*);	
read(*B*); 30	
lock-X(*A*);	
unlock(*B*);	
	lock-S(*A*);
	read(*A*); 50
	lock-X(*B*);
	unlock(*A*);
	read(*B*) ; 30
	B:= B+A ; 80
	write(*B*);
	unlock(*B*);
read(*A*); 20	
A:= A+B; 50	
write(*A*);	
unlock(*A*);	

2PL garante a serialização, pois T2' tem que esperar a liberação de A em T1' para executar

Resultado deste escalonamento serializável

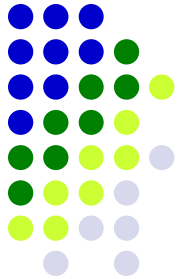
A=50 e B=80

2.1.3 – Variações do 2PC



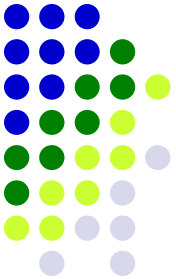
- Bloqueio em duas fases **severo**: exige que todos os bloqueios de modo exclusivo tomados por uma transação sejam mantidos até que a transação seja efetivada (commit)
  - Não permite rollback em cascata
- Bloqueio em duas fases **rigoroso**: exige que todos os bloqueios tomados por uma transação sejam mantidos até que a transação seja efetivada (commit)
  - As transações podem ser serializadas na ordem de efetivação

2.2 - Protocolo de *commit* em duas fases Distribuído



- *Two-phase commit protocol:* **2PC**
- A ação de *commit* deve ser “instantânea” e indivisível.
- Pode ser necessária a cooperação de muitos processos, em máquinas distintas, cada qual com um conjunto de objetos envolvidos na transação.
- Um dos processos é designado como **coordenador** (normalmente o próprio cliente que inicia a transação).
- Os demais processos são designados como **participantes.**
- Toda ação é registrada em *log,* armazenado em *memória estável*, para o caso de falha durante o protocolo.

2.2.1 - Fases do 2PC Distribuído



- **Fase 1: Votação**
  - O coordenador envia mensagem VOTE_REQUEST para todos os participantes e aguarda as respostas.
  - Cada participante responde VOTE_COMMIT ou VOTE_ABORT para o coordenador.
- **Fase 2: Decisão**
  - Se todos os participarem tiverem respondido VOTE_COMMIT, o coordenador envia para todos os participantes um GLOBAL_COMMIT senão envia um GLOBAL_ABORT.
  - Cada participante confirma ou aborta a sua transação local, conforme receba GLOBAL_COMMIT ou GLOBAL_ABORT, respectivamente.

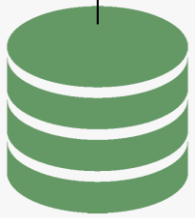
Fase 1 - Votação

Vote_Commit

Brasília

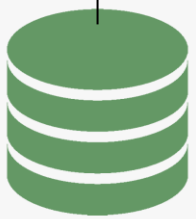
Empregados de Brasília,
Empregados de São Paulo

Projetos de Brasília



Vote_Commit

São Paulo



Empregados de São Paulo,
Empregados de Brasília

Projetos de São Paulo,
Projetos de Brasília

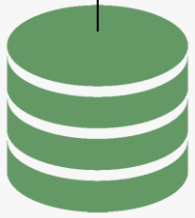
Projetos de Porto Alegre

Rede de Comunicação

Porto Alegre

Empregados de Porto Alegre

Projetos de Porto Alegre,
Projetos do Rio de Janeiro



Coordenador

Rio de Janeiro

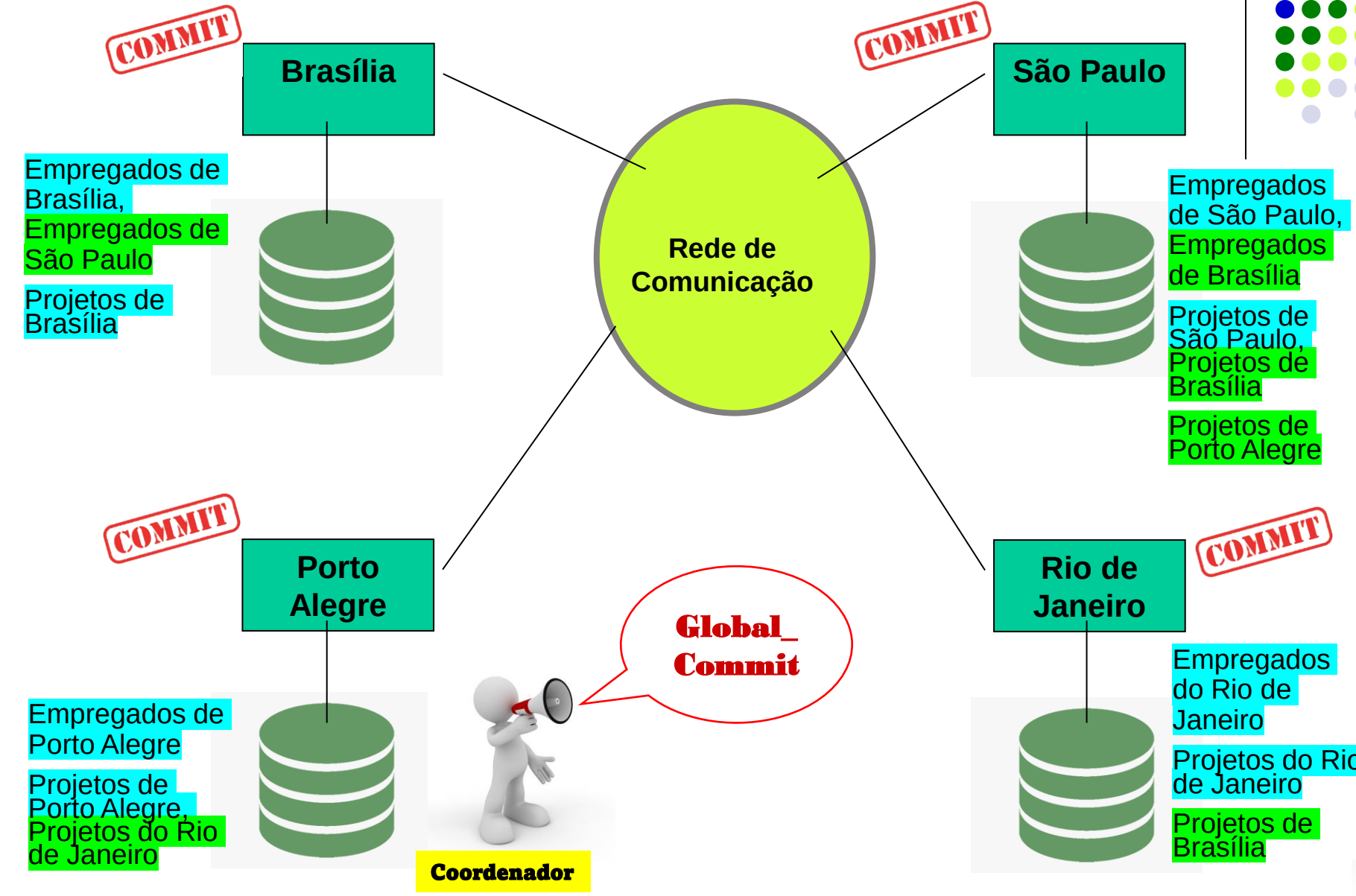
Vote_Commit

Empregados do Rio de Janeiro

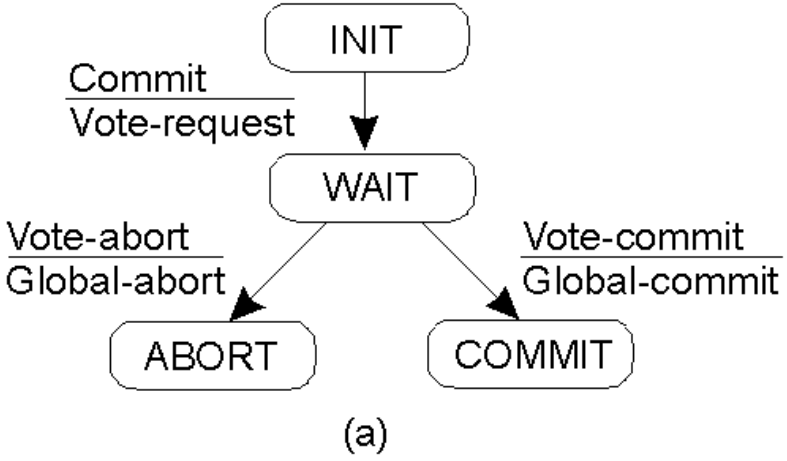
Projetos do Rio de Janeiro

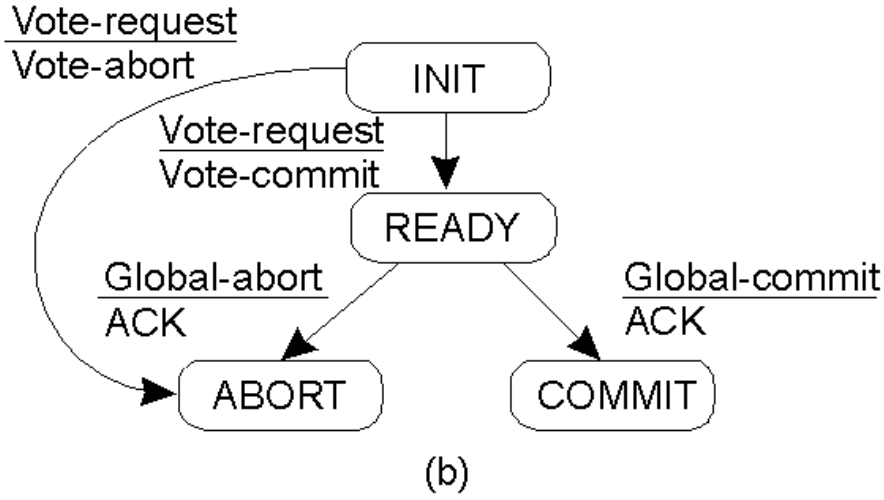
Projetos de Brasília

Fase 2 - Decisão



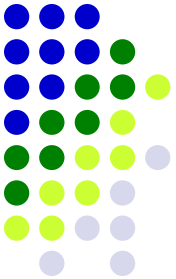
2.2.1 - Fases do 2PC Distribuído





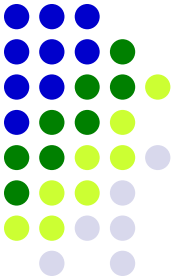
a) Máquina de estados do coordenador
b) Máquina de estados de cada participante

2.2.2 - Falhas durante o 2PC Distribuído



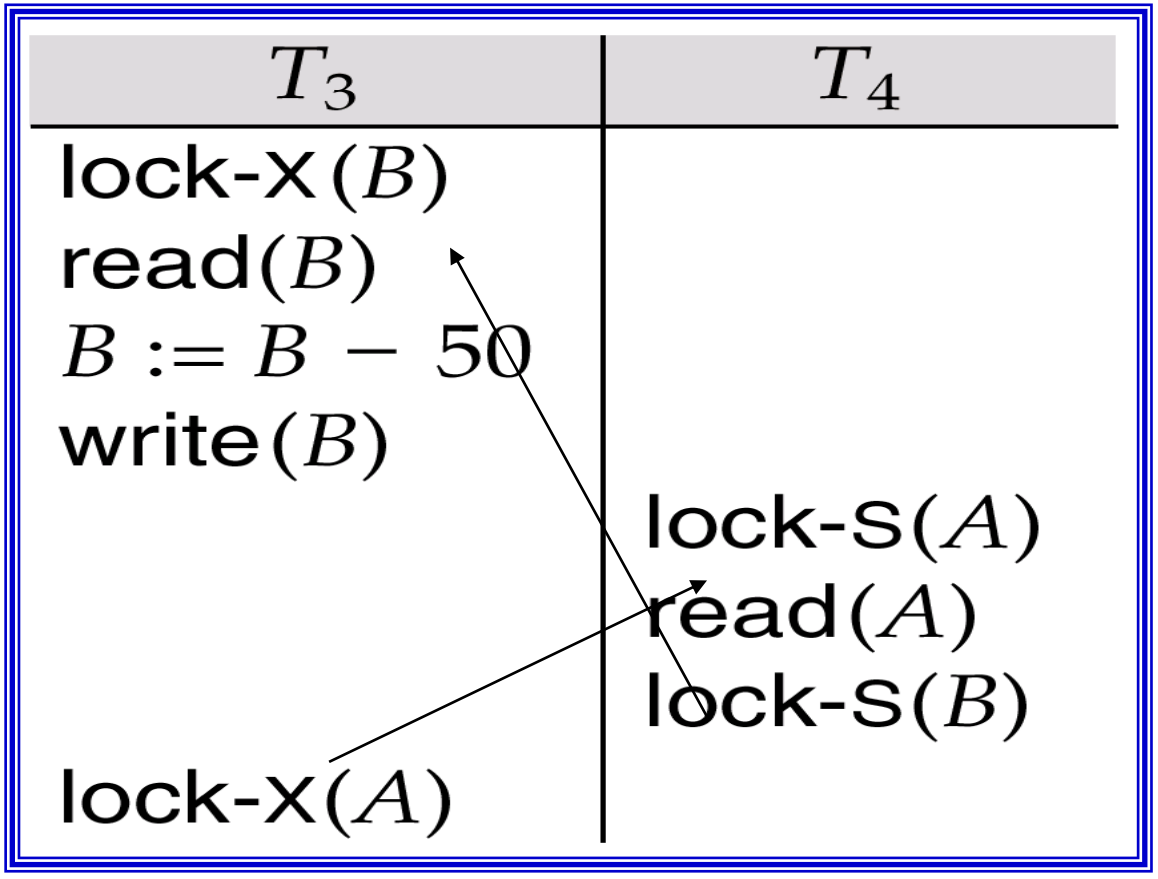
- Coordenador pode ficar bloqueado em WAIT, aguardando os votos dos participantes
  - Decide por *abort* se todos os votos não chegarem dentro de um certo tempo.
- Participante pode ficar bloqueado em INIT, aguardando um VOTE_REQUEST do coordenador
  - Vota por *abort* se o VOTE_REQUEST não chegar dentro de um certo tempo.

2.2.2 - Falhas durante o 2PC Distribuído



- Participante pode ficar bloqueado em READY, aguardando a decisão do coordenador
  - Se a decisão não chegar dentro de um certo tempo, consulta outros participantes para descobrir qual foi a decisão.
  - Caso um participante consultado esteja bloqueado em INIT, a decisão é por *abort*.
  - Caso todos os participantes estejam bloqueados em READY, o protocolo bloqueia até que o coordenador se recupere.

3 - Deadlock (impasse)



T4 espera T3 desbloquear B

T3 espera T4 desbloquear A para prosseguir

Uma das transações tem que ser desfeita (rollback)

Lista de Exercícios – Transações – Serialização por conflito Controle de Concorrência – Protocolos de Bloqueio

1 – Verificar se as escalas abaixo são conflito serializáveis.

a) $r_1(x)$, $r_2(y)$, $r_1(z)$, $w_2(y)$, $r_3(z)$, $r_2(x)$, $r_1(y)$, $w_1(x)$
b) $r_1(x)$, $w_2(y)$, $r_1(z)$, $r_3(z)$, $w_2(x)$, $r_1(y)$, $w_3(y)$, $w_1(x)$
c) $r_1(x)$, $w_2(y)$, $r_1(z)$, $r_3(z)$, $w_1(x)$, $r_2(y)$, $w_3(y)$, $w_3(x)$

2- Para as escalas acima proceder a verificação através do grafo de precedência.

3 – Para as escalas de 1 que são conflito serializáveis construir a escala equivalente.

4 – Para as escalas não serializáveis por conflito, construir a esquala equivalente aplicando o protocolo 2PL.

Transações Distribuídas – Relacional e BD NO-SQL
Teorema CAP

Atividade 02 : Responda às questões conceituais abaixo:

1- Quais as vantagens e desvantagens de se permitir acesso concorrente ao banco de dados?

Vantagens: i) mais tarefas executadas simultaneamente

ii) melhor aproveitamento dos recursos intercalando uso da CPU com leituras/escritas no disco

iii) maior interatividade com o usuário.

Desvantagens: i) sobrecarga (overhead) criado pelo controle de acesso

ii) possibilidade de inconsistências

iii) possibilidade de deadlocks

2- Para cada propriedade ACID (atomicidade, consistência, isolamento, durabilidade), descreva um problema que pode acontecer caso o SGBD não a garanta.

Atomicidade: armazenamento de valores parciais de uma transação que não se completou.

Consistência: execução de uma transação deixa o banco em um estado inconsistente, por exemplo, uma chave estrangeira que referencia uma chave primária inexistente.

Isolamento: uma transação afeta inadvertidamente o resultado de outra transação.

Durabilidade: o resultado de uma transação que recebeu *commit* é perdido por falha no banco.

3 – A Consistências em ACID tem o mesmo significado que no teorema CAP ? Explique.

Não. A consistência em ACID significa que uma transação vai levar o BD a um novo estado consistente após ser executada completamente. Isso significa que os valores irão respeitar regras de consistência e integridade. Após uma transação o BD não pode ficar em um estado inconsistente, embora nos estágios intermediários de execução de uma transação isso possa ocorrer.

Transações Distribuídas – Relacional e BD NO-SQL
Teorema CAP

A consistência no CAP significa que todos os clientes veem os mesmos dados ao mesmo tempo, ou seja, é a garantia de que todo nó do sistema devolverá a resposta correta a cada requisição de dados que receber. Os dados devolvidos precisam ser os mais novos disponíveis e todas as réplicas têm o mesmo valor.

4- Para cada requisito abaixo indique qual estrutura/família de BD NOSQL (chave-valor, documentos, colunas ou grafos) é mais indicada:

Requisito/Característica	Tipo de Estrutura
a. Utilização vantajosa de índices complexos (multichave, geoespacial, busca completa de texto, etc.)	Documento
b. Consultas complexas para determinar as relações entre os pontos de dados	Grafo
c. Esquema Simples	Chave-Valor
d. Sem padrões de consulta ad-hoc, índices complexos ou alto nível de agregações	Coluna
e. Alto volume de dados	Coluna
f. Alta velocidade de leitura/escrita com atualização pouco frequentes	Chave-Valor
g. Alto desempenho e relação Leitura/Escrita balanceada	Documento/Coluna
h. Capacidade de armazenar propriedades de cada ponto de dados, bem como a relação entre eles	Grafo
i. Alto desempenho e escalabilidade	Chave-Valor
j. Sem consultas complexas que envolvam várias chaves ou junções	Chave-Valor
k. Esquema flexível com consultas complexas	Documento
l. Extrações de dados por colunas usando chave da linha	Coluna
m. Aplicativos que requerem travessia entre pontos de dados	Grafo
n. Formatos de dados JSON/BSON ou XML	Documento

o. Velocidades extremas de gravação com leituras de velocidade relativamente menor	Coluna
p. Necessidade para detectar padrões entre os pontos de dados	Grafo

5 – Explique o consenso em sistemas CP. Suponha um sistema com 12 servidores, fator de replicação = 8, R=4 e W=6.

i) A partir de quantos servidores falhos a disponibilidade para escrita fica sem quórum? O que acontece a partir de então?
ii) A partir de quantos servidores a leitura consistente fica comprometida?

O consenso é uma característica em sistemas distribuídos em que se necessita forte consistência, ou seja, leitura dos dados sempre atualizada. O consenso estabelece que para garantir leituras consistentes, a soma do número de nós disponíveis para leitura e escrita deve ser maior que o número de nós com réplicas dos dados:

$$R + W > N$$

Onde R = número de nós contatados para uma leitura ou quórum de leitura

W = número de nós contatados para confirmar a escrita ou quórum de escrita

N = fator de replicação dos dados ou número de réplicas

A fórmula garante que, entre os R nós de réplicas contatados, haverá pelos menos 1 que tem a versão mais nova do item de dado. Lembrando que todos os nós fazem leitura e escrita.

Na configuração acima temos 4(R) + 6(W) > 8(N) ✓

Para ter uma leitura consistente precisamos R > 8(N) -6(W) > 2, como temos 4 está OK. Nesta configuração precisamos de 3 pois R > 2.

A partir da falha de 1 servidor de escrita (W=5), ou 3 servidores com replicação dentre os 8 (N=8) a confirmação para escrita (quórum) não é mais satisfeita, a escrita fica comprometida:

W > N/2 → W > 4 (8/2), como W = 6 inicialmente está satisfeito

Se falhar 2 servidores destes 8, ainda temos 6 réplicas para confirmar uma escrita, porém se mais um falhar, ficarmos com 5 réplicas, o quórum de escrita não existe mais, precisamos de W=6.
A partir de então, o sistema para de escrever, mas fica disponível para leitura consistente, somente quando um nó for restaurado o quórum é restabelecido para escrita.

6 – Considere um sistema baseado em NO-SQL em que temos um fator de replicação igual a 3 e precisamos de consenso, ou seja R+W > N. Para cada situação abaixo comente o que pode acontecer, vantagens ou desvantagens:

i) R = 3 e W= 1

Isso melhora o desempenho das escritas em detrimento das leituras (só escreve em 1 nó então demora menos para ter consenso de escrita), o que provavelmente é uma má ideia, pois geralmente as leituras são mais comuns do que as escritas. Além disso, essa escolha de quórum é ruim porque uma escrita pode acontecer em uma única réplica que falha. Se essa réplica perder seu estado, o resultado da escrita será perdido. Portanto, geralmente gostaríamos de ter W> 1.

ii) R = 1 e W= 3

Isso funciona muito bem para leituras, o que geralmente é bom, pois leituras são comuns. Mas não é desejável para escritas porque se uma das réplicas estiver inativa ou inacessível, uma escrita não poderá ser concluída até que a réplica seja recuperada (precisa de quórum de 3 para escrita) -> unanimidade para escrita

iii) R = 2 e W= 2

Essa escolha é um bom compromisso em comparação com as opções R = 1 e W = 3, pois aumenta o custo das leituras em troca de fornecer disponibilidade razoável para escritas.

Transações Distribuídas – Relacional e BD NO-SQL
Teorema CAP

7 – Para cada situação apresentada abaixo indique o tipo de sistema CAP (CA, CP ou AP) mais adequado e justifique:

I - A Amazon descobriu que cada 100 milissegundos de latência custam a eles 1% nas vendas. Os usuários do aplicativo, clientes e visitantes do site fazem um julgamento instantâneo sobre o aplicativo e o negócio. Se a aplicação for rápida, uma primeira impressão forte é feita. É uma vitória para a experiência do usuário.
É um aspecto psicológico considerar mais confiáveis os aplicativos mais rápidos. Relacionamos velocidade com eficiência, confiança e segurança. Por outro lado, um aplicativo ou site lento nos faz pensar que é inseguro e não confiável. É difícil reverter essa primeira impressão negativa.

Alta disponibilidade e tolerância a partição – Sistema precisa estar disponível o tempo todo e oferecer desempenho, a consistência fica comprometida
AP – Availability-Partition Tolerance

II - Considere uma plataforma socioprofissional como o LinkedIn, que mostra a contagem de conexões para cada usuário (minha rede de conexões). A contagem de conexões é exibida no próprio perfil do usuário e em outras sugestões de rede como conexões mútuas, embora as contagens sejam diferentes para a conexão mútua e real. Considere que o banco de dados de conexões é replicado nos Estados Unidos, Europa e Ásia. Quando um usuário na Índia confirma 10 novas conexões, essa alteração leva alguns minutos para se propagar para as réplicas dos Estados Unidos e da Europa. Isso pode ser suficiente para tal sistema porque uma contagem precisa de conexões e conexões mútuas nem sempre é essencial. Se um usuário nos Estados Unidos e outro na Europa estivessem falando ao telefone enquanto um estava expandindo as conexões, o outro usuário veria a atualização segundos depois e estaria tudo bem. Se a atualização levasse minutos devido ao congestionamento da rede ou horas devido a uma queda da rede, o atraso ainda não seria terrível.
Consistência não é o mais importante, pode ser eventual.
Mais importante é estar disponível o tempo todo sacrificando a consistência.
AP – Alta disponibilidade e tolerância a partição

III - Agora imagine um aplicativo bancário. Uma pessoa nos Estados Unidos e outra na Índia poderiam coordenar suas ações para sacar dinheiro da mesma conta ao mesmo tempo. O caixa eletrônico que cada pessoa usa consultaria sua réplica de banco de dados mais próxima, que atestaria que o dinheiro está disponível (tem saldo) e pode

ser retirado. Se as atualizações se propagassem lentamente, os dois teriam o dinheiro antes que o banco percebesse que o dinheiro já havia sumido.

CA – Consistency-Availability

Forte Consistência é o mais importante sacrificando a tolerância a partição. O sistema bancário trabalha com leituras e escritas atômicas, garantindo as propriedades ACID. Disponibilidade é importante para as aplicações locais.

IV- Considere um usuário no LinkedIn tentando acessar um recurso como postagem ou vídeo compartilhado durante o horário de pico. Agora, devido à sobrecarga, o LinkedIn responderá às solicitações com um código de erro "tente novamente mais tarde". Ouvir isso imediatamente é mais favorável do que esperar minutos ou horas antes de desistir.

CP – Consistency-Partition Tolerance

A disponibilidade é sacrificada quando uma partição acontece entre quaisquer dois nós, o sistema "desliga" o nó não-consistente, deixando indisponível, até o problema da partição ser resolvido. Quando a comunicação é restabelecida as réplicas são sincronizadas.