Banco de Dados Não Relacionais e Big Data

Aulão - P1 - Monitoria

O Que é NoSQL?

NoSQL = ***Not Only SQL***, ou seja, bancos de dados que não se limitam ao modelo relacional tradicional.

Relembrando: bancos como MySQL e PostgreSQL são exemplos do modelo relacional que vocês já viram.

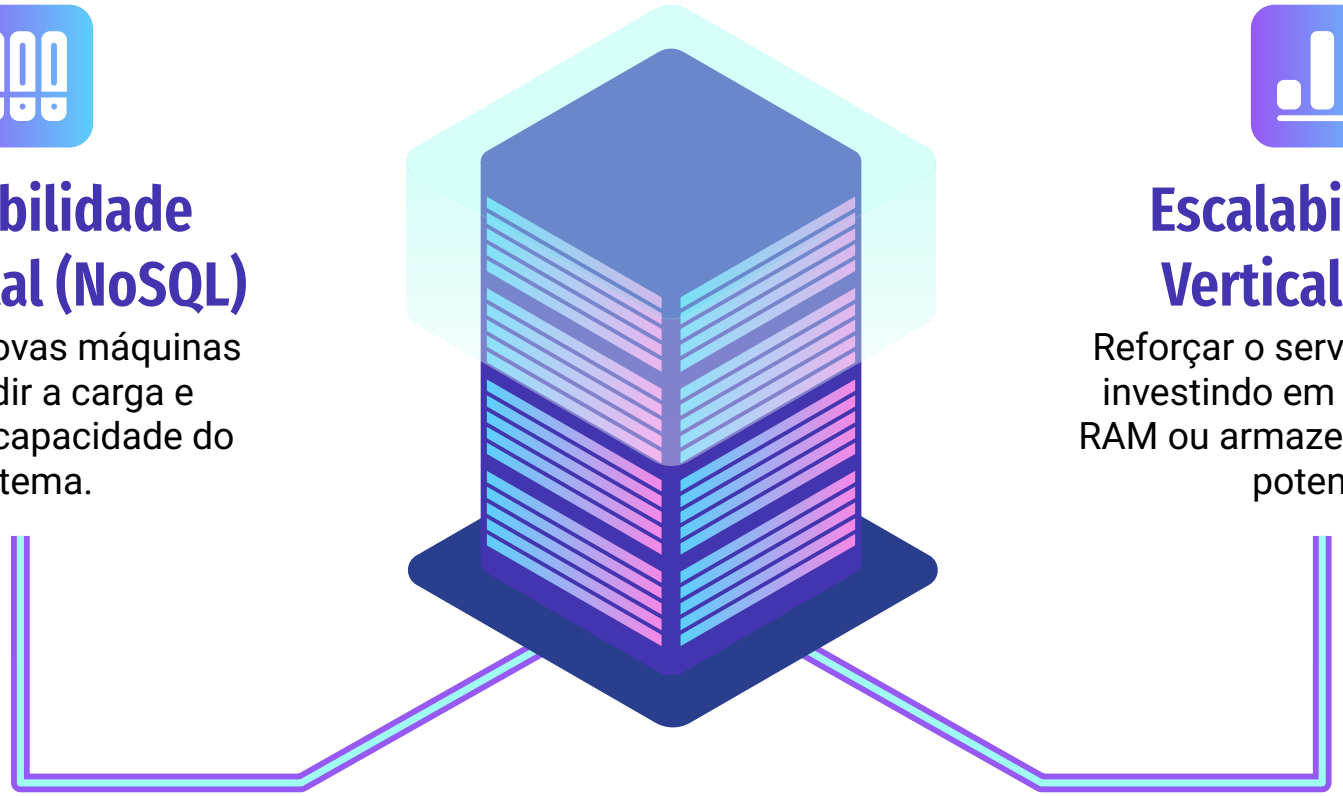
O **NoSQL** surge para lidar com grandes volumes de dados e estruturas variadas, atendendo aplicações web modernas e cenários de Big Data.

Empresas como Google, Amazon, LinkedIn (e até a Mauá) adotam NoSQL para suportar milhões de usuários e petabytes de informação.

Escalabilidade

Escalabilidade Horizontal (NoSQL)

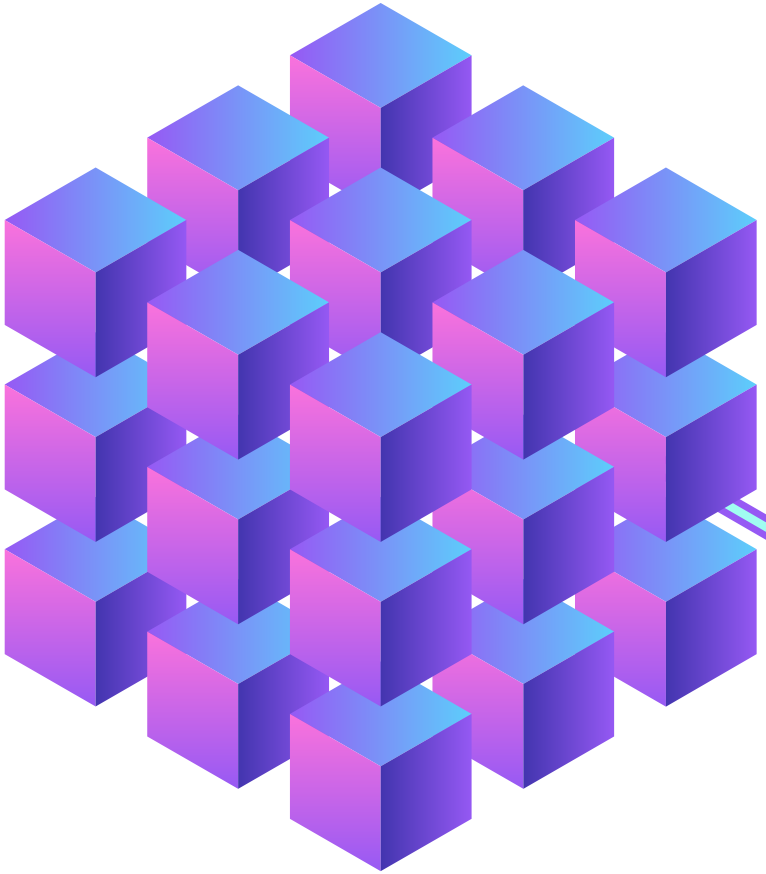
Adicionar novas máquinas para dividir a carga e expandir a capacidade do sistema.



Escalabilidade Vertical (SQL)

Reforçar o servidor existente, investindo em processador, RAM ou armazenamento mais potentes.

Ausência de Esquema (Schema-Free)



Diferente dos bancos relacionais, o **NoSQL não exige um esquema fixo** para armazenar dados.

Maior **flexibilidade** para lidar com dados heterogêneos.

Facilita **escalabilidade** e **alta disponibilidade**.

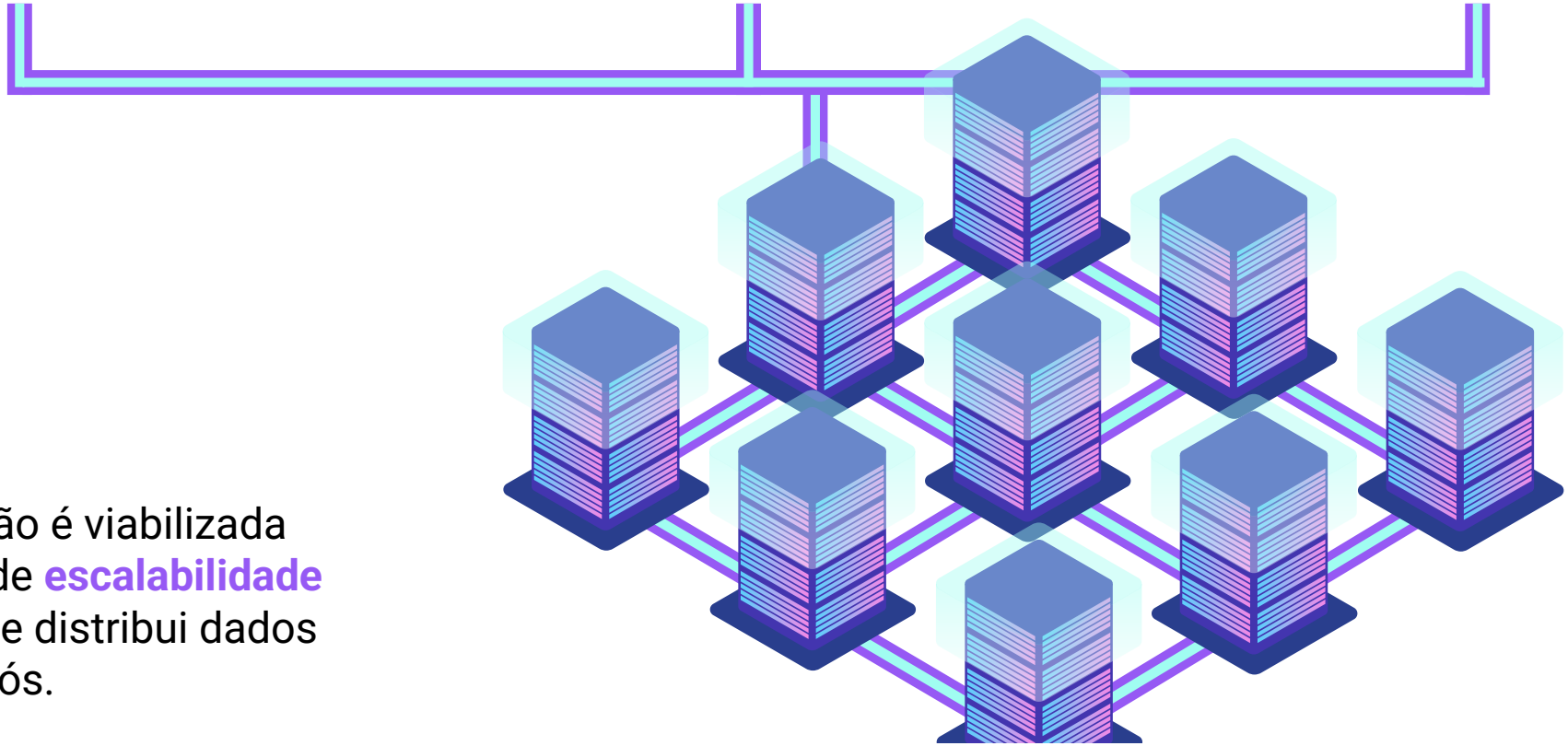
Pode gerar **inconsistência** e **falta de integridade** se não houver controle adequado.

Suporte a Replicação

Sistemas NoSQL oferecem **replicação nativa** dos dados.

Alta disponibilidade: se um servidor cair, outro assume.

Recuperação rápida: reduz o tempo de restauração em caso de falhas.



Essa replicação é viabilizada pelo modelo de **escalabilidade horizontal**, que distribui dados entre vários nós.

Limitações do NoSQL

Nem todos os servidores refletem imediatamente a última atualização.
O dado “propaga” com o tempo.

Consistência eventual

Falta de garantias de integridade

A flexibilidade de não ter esquema rígido pode comprometer **ACID** e dificultar controles de **chaves estrangeiras** e **restrições**.

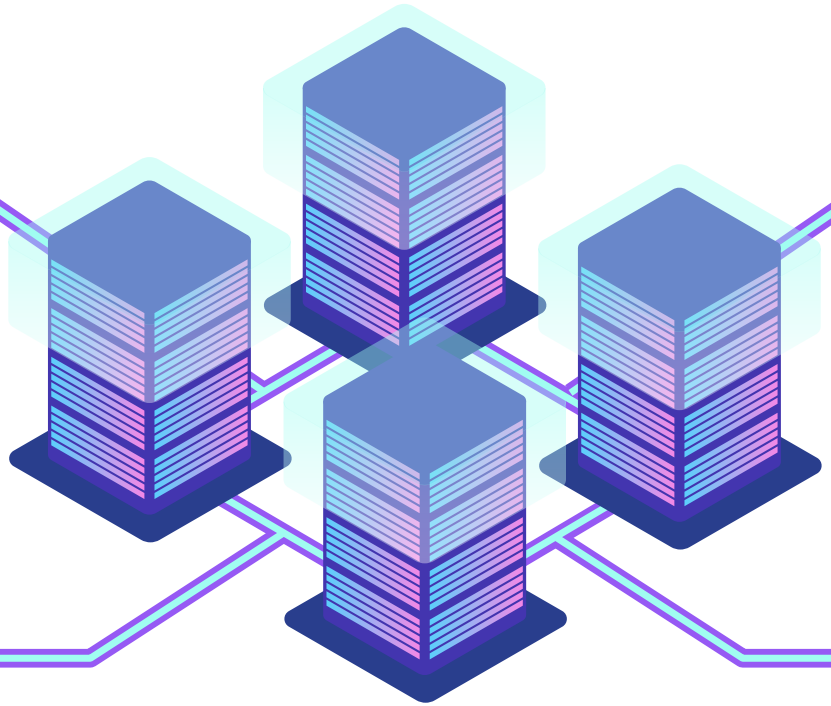
Modelos de Dados NoSQL

Chave-Valor

Redis, **Amazon DynamoDB**.

Documento

MongoDB, **CouchDB**.



Colunar

Apache Cassandra, **HBase**.

Grafos

Neo4j, **JanusGraph**.

Modelos de Dados NoSQL

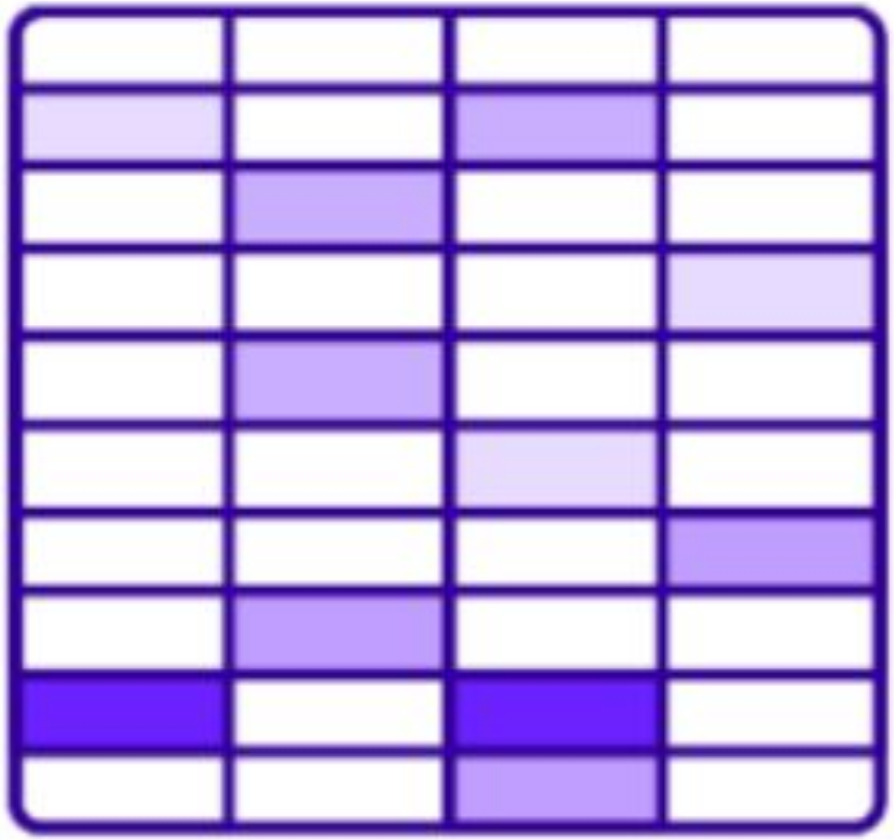
Chave-Valor

- ★ Estrutura mais simples: um **par (chave, valor)**.
- ★ A **chave** é única e identifica o registro.
- ★ O **valor** pode ser de qualquer tipo (texto, JSON, binário, objeto).
- ★ Muito eficiente para consultas rápidas quando se conhece a chave.

Modelos de Dados NoSQL

Modelo Orientado a Colunas

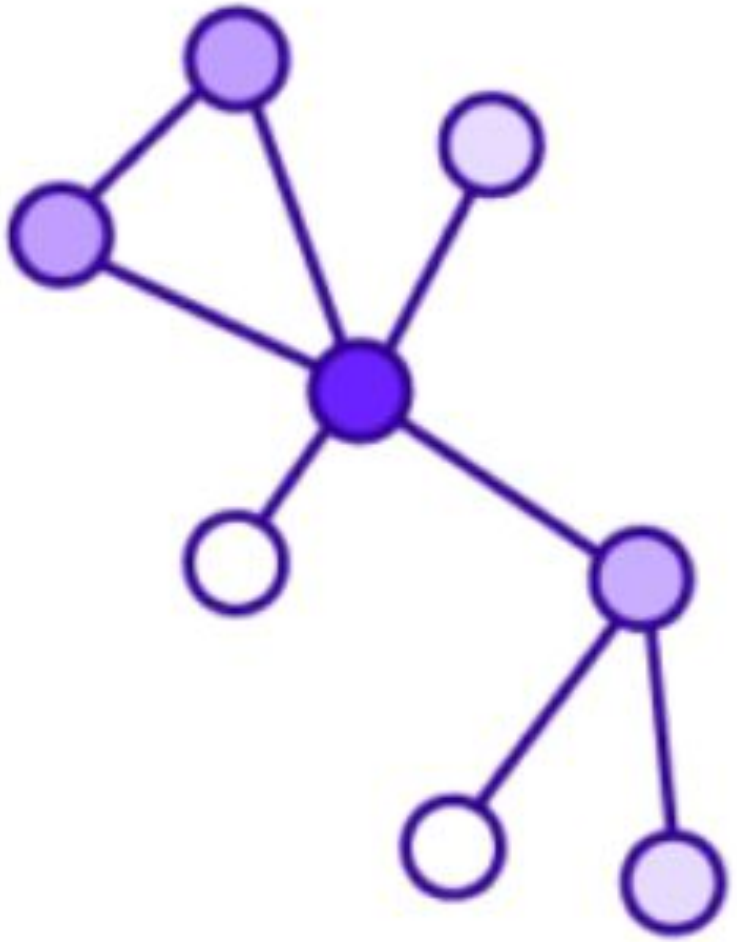
- ★ Dados organizados por **colunas**, não por linhas.
- ★ Permite armazenar informações em **famílias de colunas**.
- ★ Vantagens:
  - Excelente para consultas **analíticas** e **big data**.
  - Reduz custo de I/O ao buscar apenas os atributos necessários.
- ★ Exemplo: **Apache Cassandra**, **HBase**.



Modelos de Dados NoSQL

Modelo de Grafos

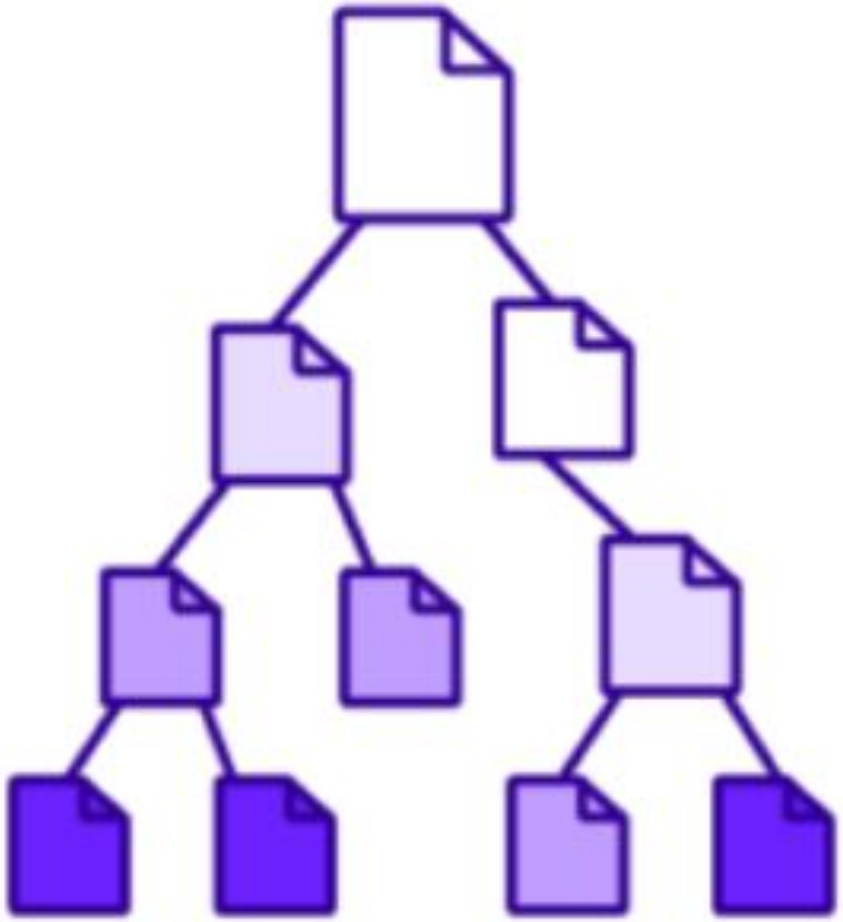
- ★ Estrutura baseada em **vértices (ou nós)** conectados por **arestas**.
- ★ Tanto os vértices quanto as arestas podem armazenar informações.
- ★ Excelente para cenários de **relacionamentos complexos**:
  - Redes sociais
  - Sistemas de recomendação
  - Roteamento e logística



Modelos de Dados NoSQL

Modelo de Documentos

- ★ Armazena dados em **documentos** (geralmente JSON, BSON ou XML).
- ★ Cada documento é **auto-descritivo** e pode ter **atributos diferentes**.
- ★ Muito flexível para dados **semiestruturados**.
- ★ Exemplo: **MongoDB, Couchbase**.



Replicação

Replicação – Master-Slave

Um servidor é definido como **Master** (responsável por leituras e escritas).

Outros servidores atuam como **Slaves** (somente leitura).

Implicações

Leituras inconsistentes podem ocorrer durante atualizações.

Se o **Master falhar**, novas escritas ficam impossibilitadas.

Falhas na replicação geram **dados divergentes** entre os servidores.

Replicação – Masterless (Peer to Peer)

Qualquer nó pode executar **leituras e escritas**.

Os servidores precisam se **sincronizar constantemente** para evitar perda de dados.

Vantagens

Alta disponibilidade: mesmo que um nó caia, o sistema continua funcionando.

Escalabilidade **horizontal**: permite adicionar quantos nós forem necessários.

Desvantagem

Pode haver **consistência eventual**, já que nem todos os nós são atualizados ao mesmo tempo.

Replicação – Masterless (Peer to Peer)

Qualquer nó pode executar **leituras e escritas**.

Os servidores precisam se **sincronizar constantemente** para evitar perda de dados.

Vantagens

Alta disponibilidade: mesmo que um nó caia, o sistema continua funcionando.

Escalabilidade **horizontal**: permite adicionar quantos nós forem necessários.

Desvantagem

Pode haver **consistência eventual**, já que nem todos os nós são atualizados ao mesmo tempo.

Comparação dos Modelos de Replicação

Criterio	Master-Slave	Masterless
Escrita	Apenas no Master	Em qualquer nó
Leitura	Feita nos Slaves	Em qualquer nó
Escalabilidade	Vertical	Horizontal
Disponibilidade	Baixa (depende do Master)	Alta (tolerância a falhas)
Consistência	Mais forte	Eventual

Transações – ACID vs BASE

ACID

Atomicidade: a transação é executada por completo ou não acontece.

Consistência: o banco passa sempre de um estado válido para outro.

Isolamento: operações concorrentes não interferem entre si.

Durabilidade: uma vez concluída, a transação persiste mesmo em caso de

O modelo **ACID** é ideal para sistemas que exigem **consistência forte**, como bancos financeiros ou aplicações críticas, mas sacrifica **escalabilidade**.

BASE

Basically Available (Basicamente Disponível): o sistema prioriza a disponibilidade dos dados.

Soft-State (Estado Flexível): os dados podem permanecer em estado intermediário por um tempo.

Eventual Consistency (Consistência Eventual): a consistência não é garantida de imediato, mas tende a ser atingida.

Já o **BASE** prioriza **disponibilidade e tolerância a falhas**, permitindo sistemas distribuídos em larga escala, ao custo de aceitar **consistência eventual**.

Teorema CAP

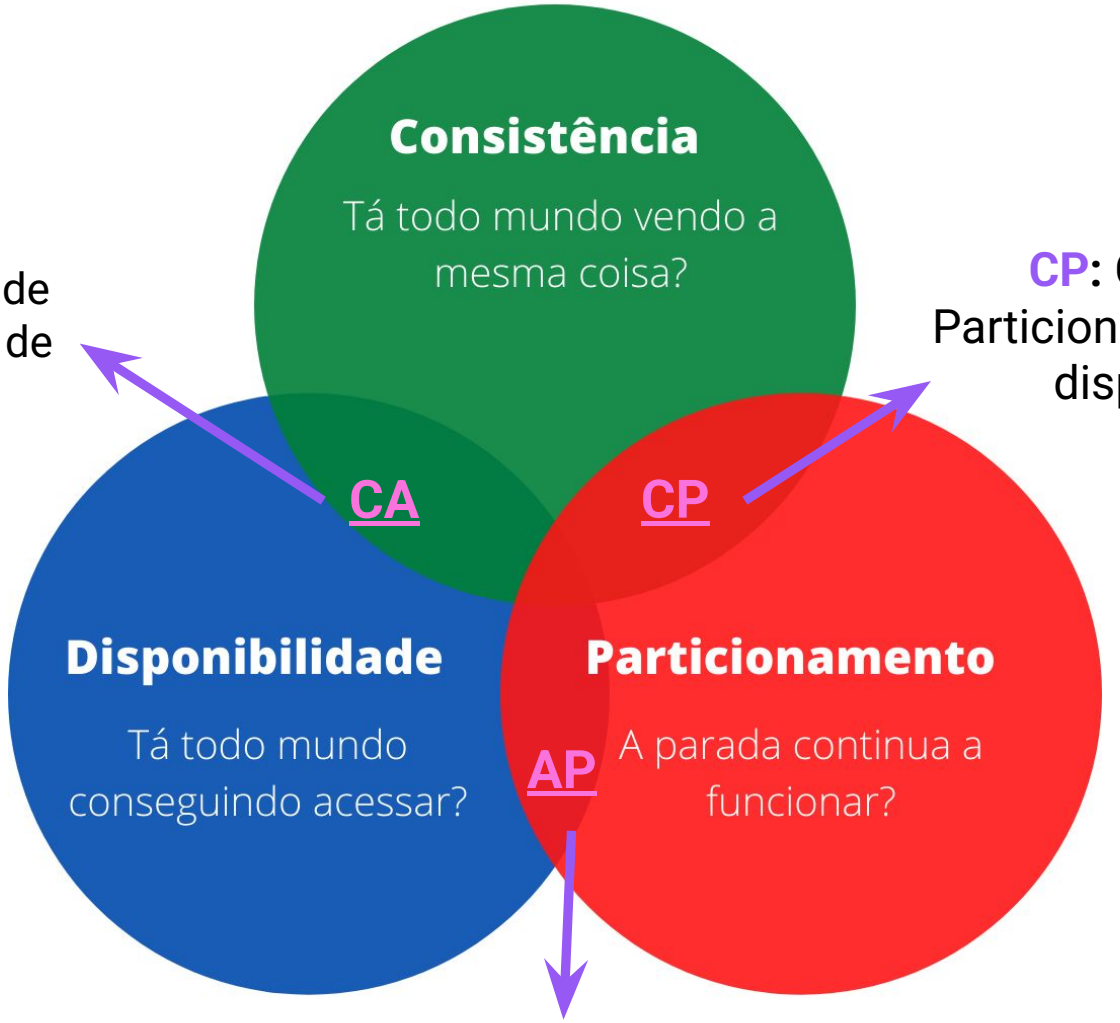
C (Consistência): todos os nós exibem os mesmos dados ao mesmo tempo.

A (Disponibilidade): o sistema continua respondendo requisições, mesmo com falhas.

P (Tolerância a Partições): o sistema suporta falhas de comunicação entre servidores.

Teorema CAP

CA: Consistência + Disponibilidade (abre mão da tolerância a falhas de rede).



CP: Consistência + Particionamento (sacrifica a disponibilidade).

AP: Disponibilidade + Particionamento (sacrifica a consistência imediata).

Particionamento + Replicação

Quando falamos de bancos de dados **distribuídos**, precisamos lidar com dois conceitos fundamentais

Particionamento

dividir os dados em pedaços menores para distribuir entre servidores

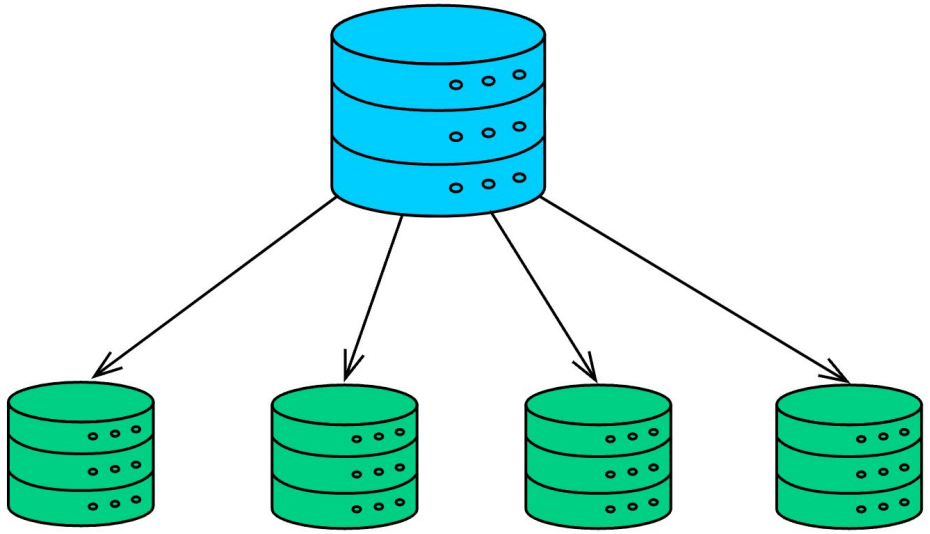
Replicação

manter cópias dos dados em múltiplos servidores para garantir **disponibilidade** e **tolerância a falhas**.

Essas duas técnicas, combinadas, permitem que bancos NoSQL lidem com **enormes volumes de dados** e **milhões de acessos simultâneos**.

Particionamento (Sharding)

- O **sharding** é o particionamento **horizontal** do banco de dados.
- Cada **shard** contém apenas uma fração do conjunto total de dados.
- Os shards são distribuídos em diferentes servidores (ou nós).



Em resumo:

- Particionar dados significa dividi-los entre diferentes servidores, chamados de shards.
- Cada shard contém uma parte do conjunto de dados total, facilitando o paralelismo e a escalabilidade.

O que é Quórum em Leitura e Escrita?

Nos bancos distribuídos, quando temos **replicação** de dados em vários servidores, surge a pergunta:
"Quantos nós precisam responder para que uma operação seja considerada válida?"

Quórum de Escrita (W)

número mínimo de réplicas que devem confirmar uma atualização para que ela seja considerada **persistida** no sistema.

Evita que o cliente leia dados desatualizados (inconsistentes).

Quórum de Leitura (R)

número mínimo de réplicas consultadas para que o sistema devolva um valor **confiável**.

Garante que os dados gravados não se percam, mesmo que algum servidor falhe logo em seguida.

Fórmulas do Quórum

Variaveis

R = Quórum de Leitura
W = Quórum de Escrita
N = Número total de réplicas (servers)

Assim, o Quórum é um mecanismo que busca o equilíbrio entre **disponibilidade** e **consistência** nos sistemas distribuídos.

Escrita com Consistência Forte

- Para que uma escrita seja considerada segura:
  - **W > N/2**
- Significa que mais da metade dos servidores precisam confirmar a operação.

Leitura Altamente Consistente

- Para garantir que a leitura sempre traga os dados mais recentes:
  - **R + W > N**
- Isso assegura que pelo menos uma réplica consultada contenha a última versão do dado.

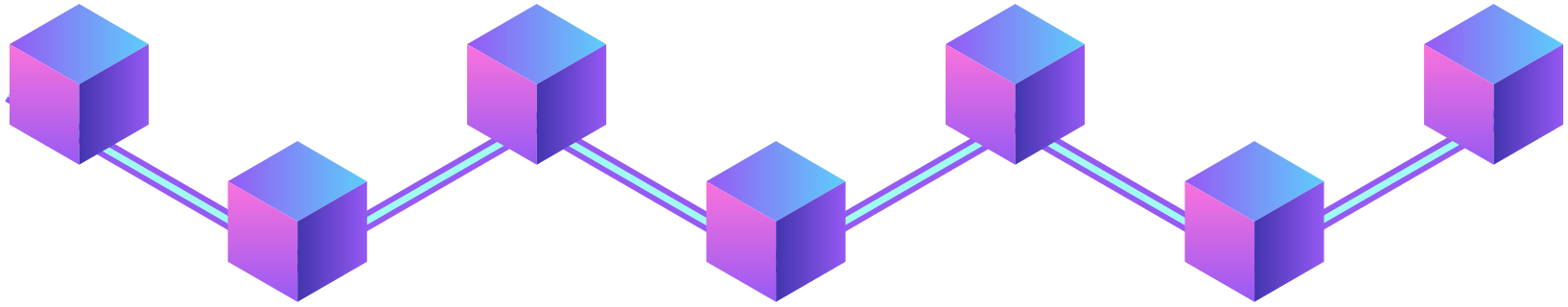
Serialização de Bancos de Dados NoSQL

Serialização garante que múltiplas **transações concorrentes** (***read(x)*** e ***write(y)***) produzam **o mesmo resultado** que se fossem executadas **em série**.

Em bancos não relacionais, onde há alta concorrência e escalabilidade distribuída, esse controle evita anomalias como:

Leitura suja → ler dados que ainda não foram confirmados.
Leitura não repetível → ler valores diferentes na mesma transação.
Escrita perdida → uma atualização sobrescreve a outra.

O objetivo é preservar a consistência, mesmo em ambientes com grande volume de leitura e escrita simultânea.



Exemplo de Problema de Serialização por Concorrência

T1: transfere fundos de A para B	T2: transfere 10% de A para B

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);

tempo

Problema:

- Ambas as transações acessam **os mesmos dados (A e B)**.
- Se executadas em paralelo, podem causar **atualizações perdidas** e resultados incorretos.

Necessário aplicar **serialização** para evitar inconsistência.

Exemplo de Problema de Serialização por Concorrência

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);

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);

Aplica **serialização**:

- T1 executa **antes** de T2, ou
- T2 executa **antes** de T1.

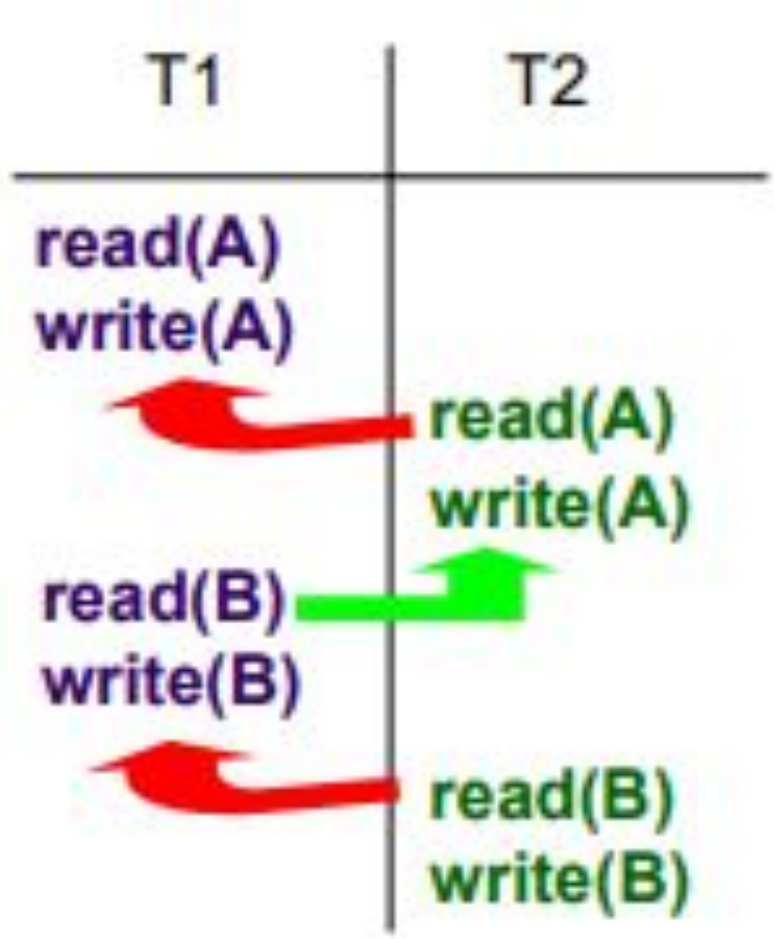
Resultado Final:

- Mesmo efeito que se as transações fossem executadas **em série**
- **Consistência preservada**, sem anomalias de concorrência.

Serialização por Conflito

Quando duas transações trabalham em **dados diferentes**, a ordem não importa.

Mas, se compartilham o **mesmo dado**, a ordem de leitura/escrita pode alterar o resultado.



Situações típicas:

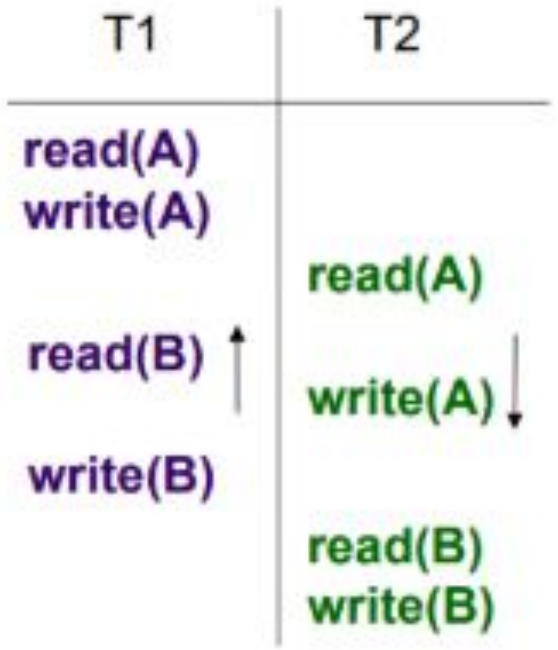
1. **Ii = read(Q) e Ij = read(Q)**
   Dois *reads* no mesmo item → não há conflito.
2. **Ii = read(Q) e Ij = write(Q)**
   Read antes de *write* → ordem influencia no valor lido.
3. **Ii = write(Q) e Ij = read(Q)**
   Write antes de *read* → ordem também importa.
4. **Ii = write(Q) e Ij = write(Q)**
   Dois *writes* → só o último prevalece, logo a ordem muda o valor final.

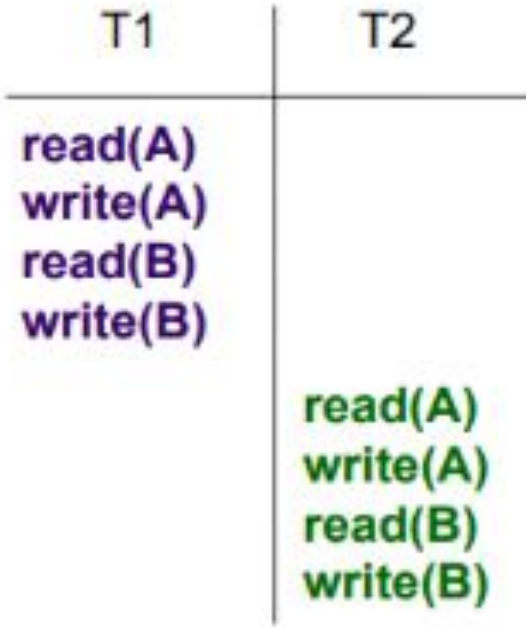
Escalas Equivalentes

Se duas transações não entram em conflito, elas podem ser reordenadas sem alterar o resultado final.

Nesse caso, diferentes execuções são consideradas **equivalentes em conflito**.

Isso permite flexibilizar a ordem de execução, desde que a consistência seja preservada.

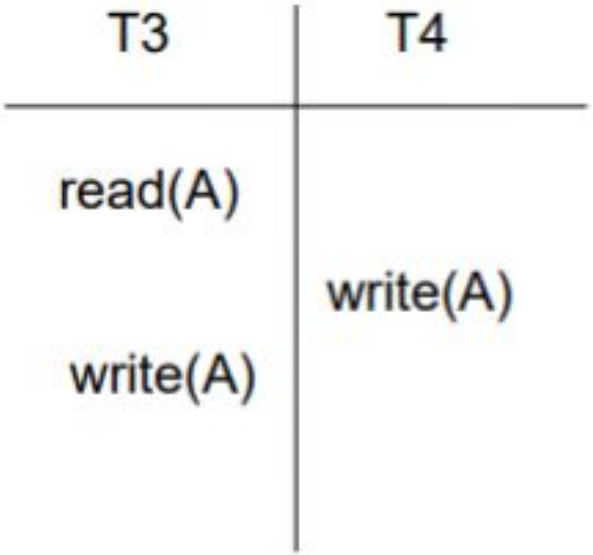


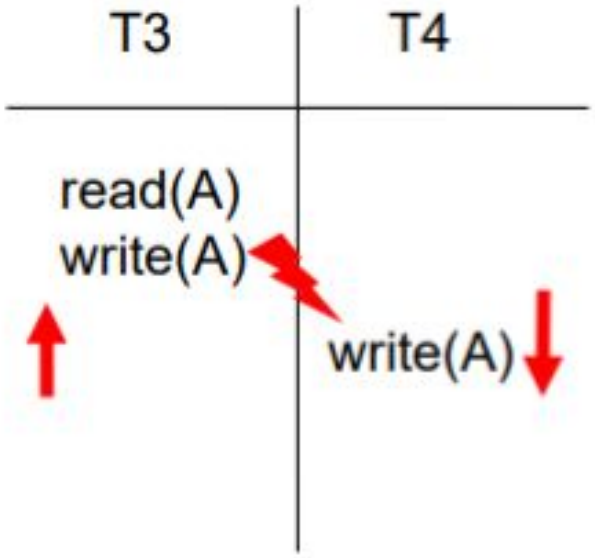


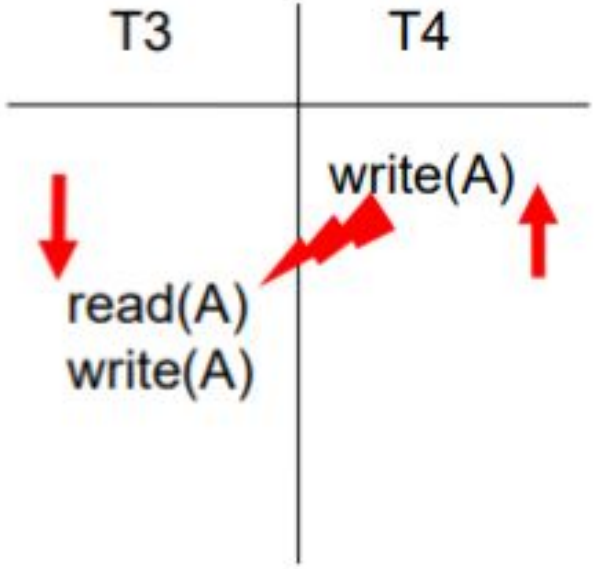
Exemplo de Escala Não Serializável

Existem execuções onde nenhuma ordem sequencial consegue reproduzir o mesmo resultado final.

Essas execuções não são serializáveis e podem gerar inconsistência.





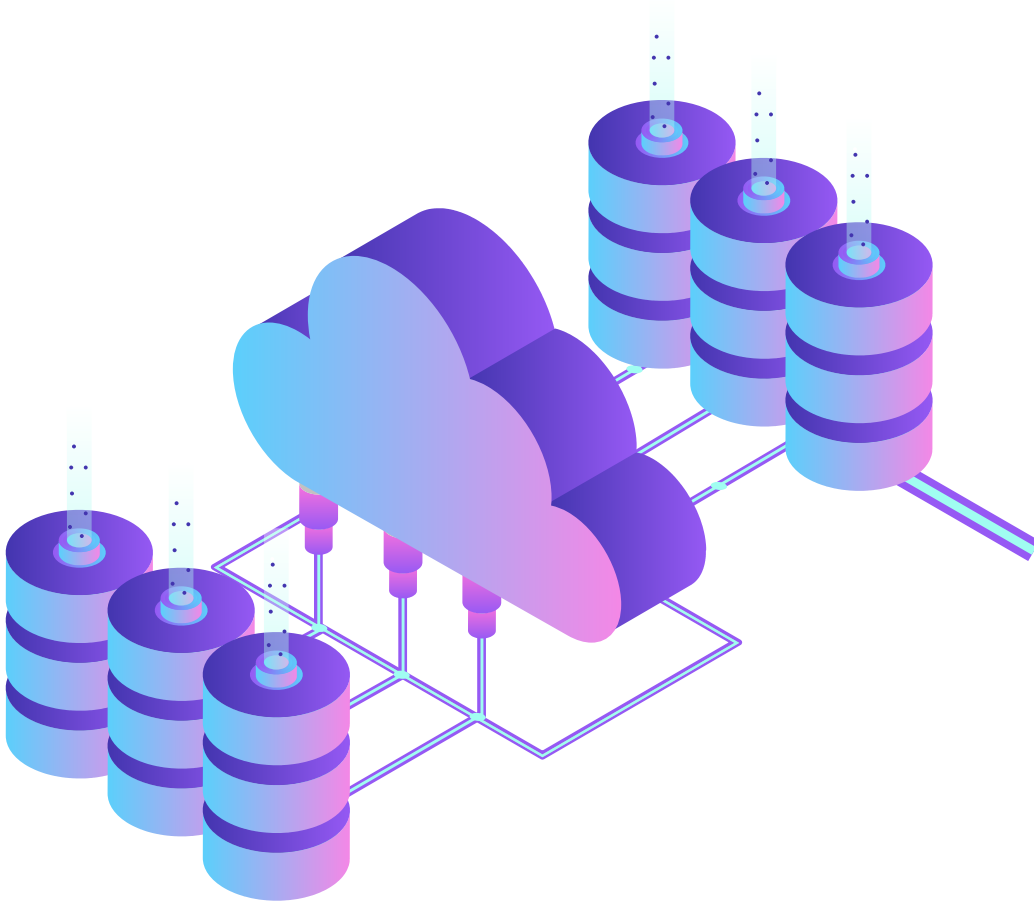


Controle de Concorrência

Algumas execuções de transações **não são serializáveis,** como vimos.

Essas execuções incorretas podem comprometer a **consistência dos dados**.

O **controle de concorrência** é o conjunto de técnicas que garantem que os dados permaneçam **consistentes**.



Protocolos de Bloqueio (Lock)

Um **lock** isola o item de dado durante a transação. Evita problemas como **leitura suja** (ler valor não confirmado) ou **escrita perdida**.

- Se um dado está **bloqueado**, outras transações não podem modificá-lo.
- O tipo de bloqueio depende da operação.

Tipos de Lock

Compartilhado (S – shared):

- Várias transações podem **ler** o mesmo item ao mesmo tempo.
- Mas nenhuma pode **escrever** enquanto o lock existir.

Exclusivo (X – exclusive):

- Apenas uma transação pode **ler e escrever** o item.
- Bloqueia totalmente o acesso de outras transações.

Protocolo de Bloqueio em Duas Fases (2PL)

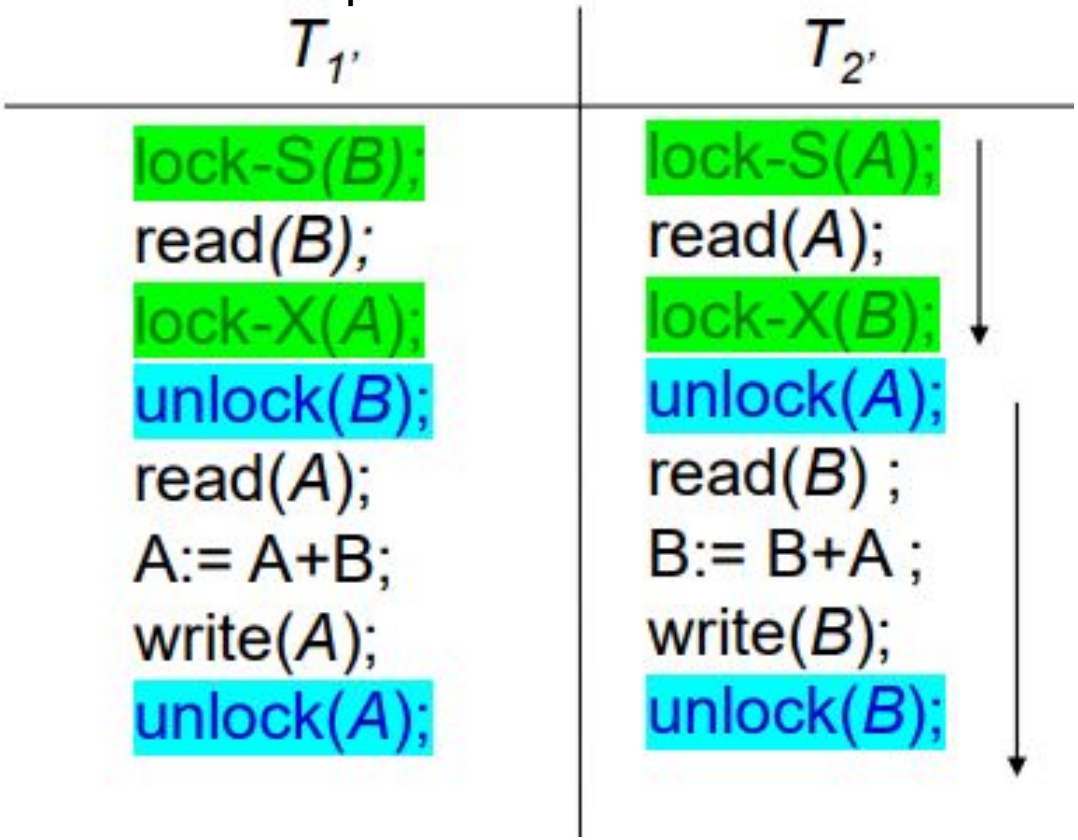
O **Two-Phase Locking (2PL)** é um protocolo de controle de concorrência que garante que as transações sejam **serializáveis**.

Ou seja: o resultado final das transações concorrentes será equivalente ao de uma execução em série

A execução de locks é dividida em **duas fases**:

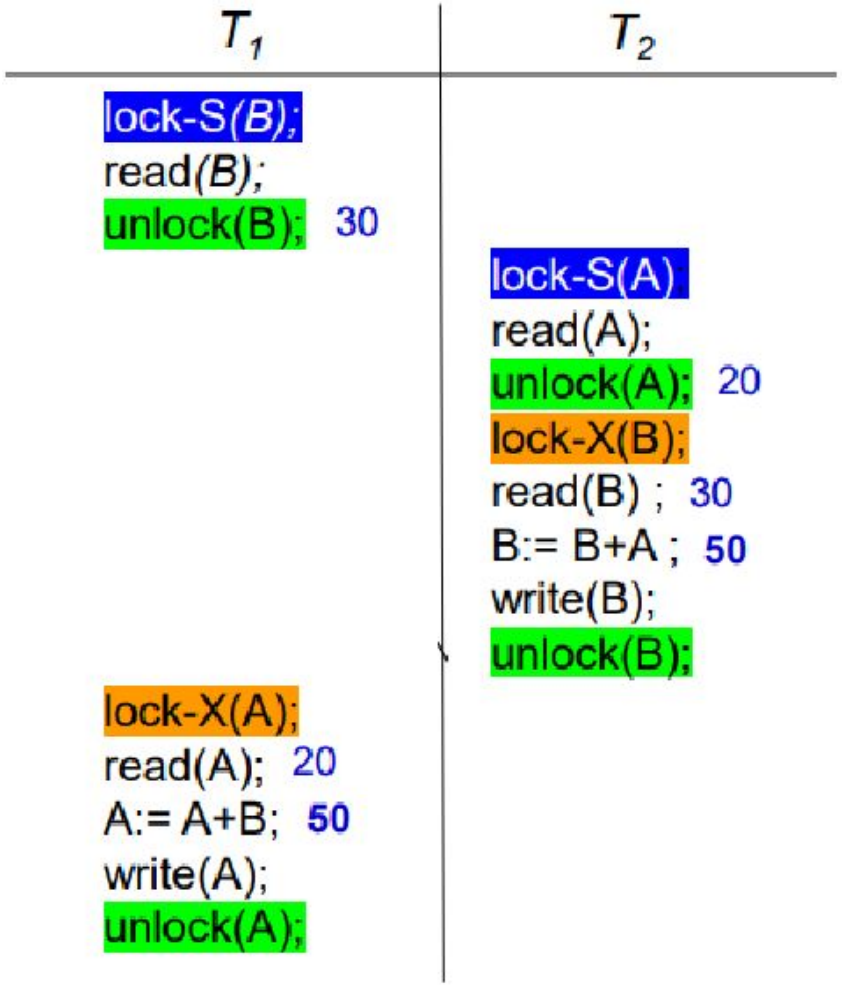
- **Fase de Expansão (ou crescimento)**: a transação pode **adquirir novos bloqueios**, mas não liberar.

- **Fase de Encolhimento (ou retração):** a transação pode **liberar bloqueios**, mas não adquirir novos.



Exemplo

- Qual o problema nessa transação?

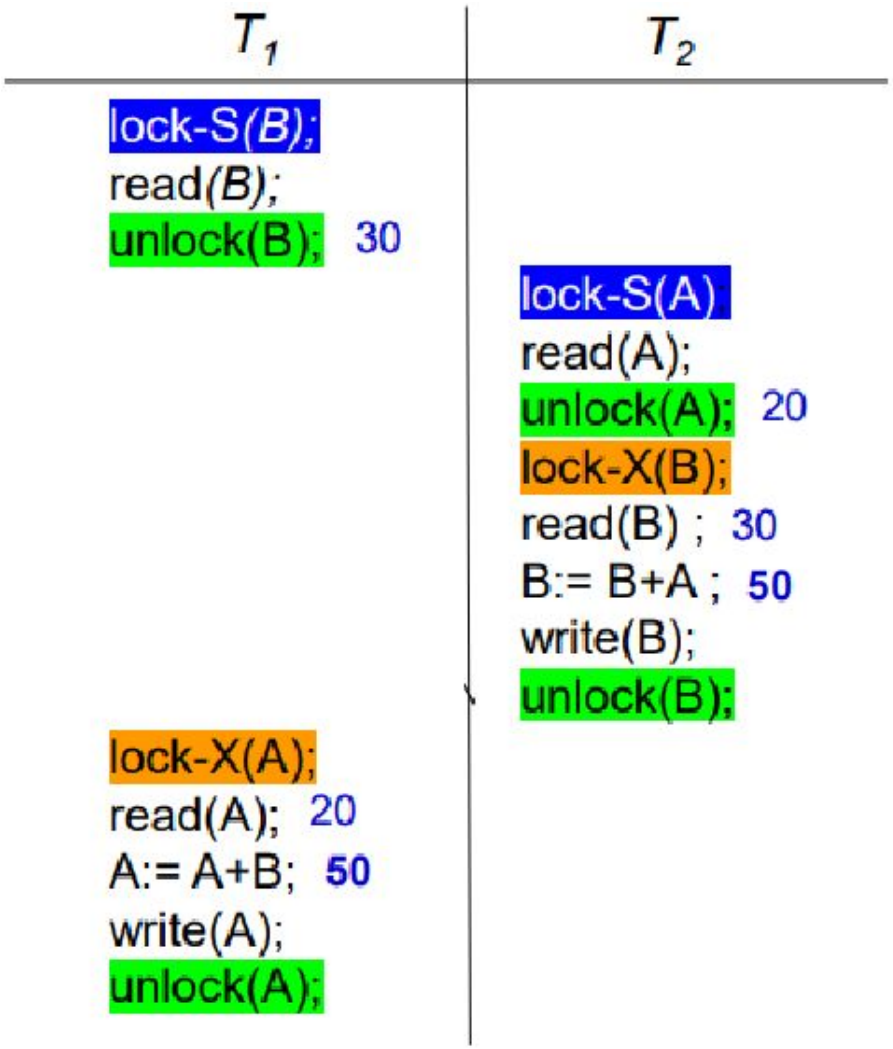


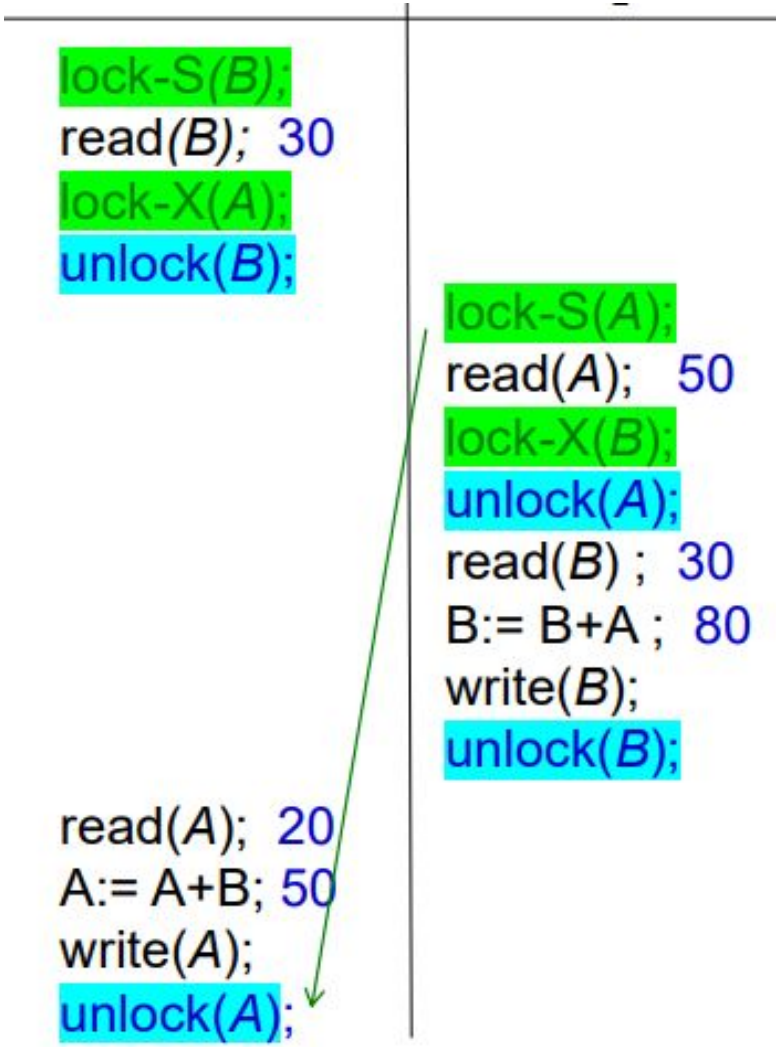
O **protocolo de bloqueio em duas fases (2PL)** está sendo **violado**.

- No **2PL**, uma transação **não pode liberar um bloqueio antes de terminar de adquirir todos os bloqueios necessários** (fase de crescimento —> fase de encolhimento).

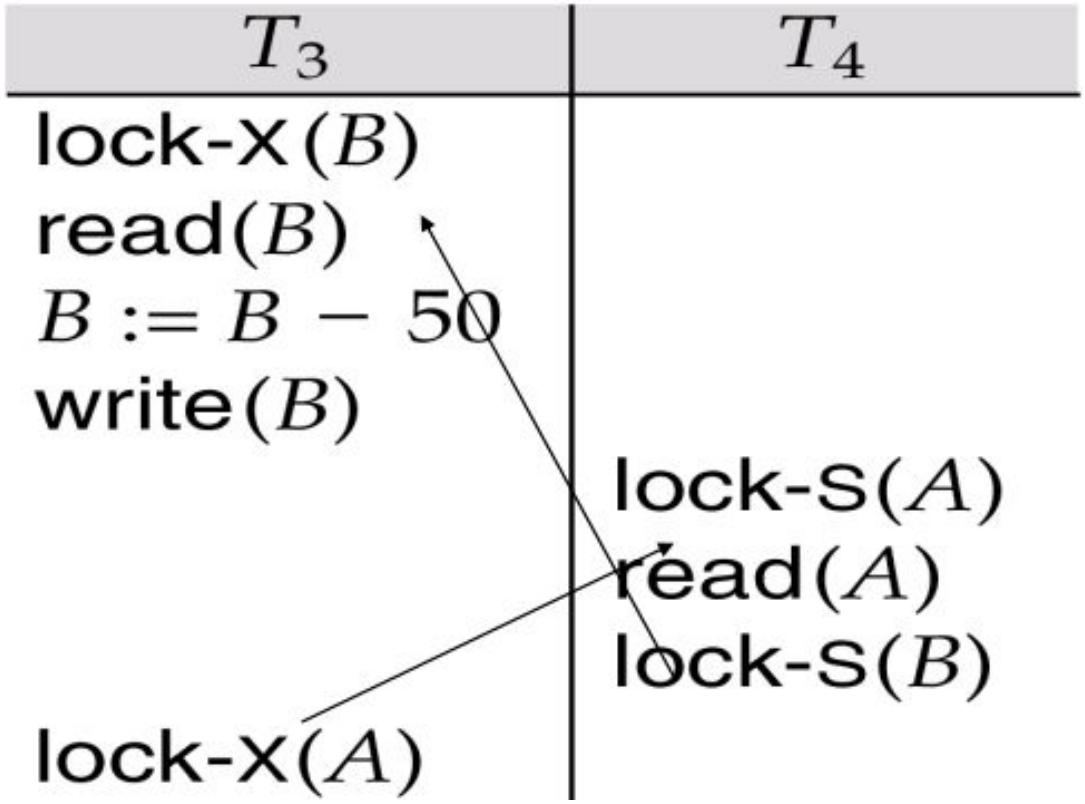
- **T2** libera lock-S(A) **antes** de pedir lock-X(B). Isso quebra o protocolo, pois T2 continua adquirindo bloqueios após já ter liberado um.

Então corrigindo





Exemplo de Deadlock



- Temos a transação T3 que bloqueia **B exclusivo**
- Quando chega a fase de bloqueio de de T4 ele bloqueia **A compartilhado**
- Quando o T4 solicita o bloqueio compartilhado de B temos o primeiro **impasse**, teriamos que dar um rollback dessa ação
- E quando acontece o bloqueio de A no T3, acontece **outro impasse**, e deve acontecer um **outro rollback**

Protocolo de Commit em Duas Fases (2PC)

Garante que o **commit seja atômico** (tudo ou nada).

Usado em **sistemas distribuídos**, onde vários processos/máquinas participam da mesma transação.

Um processo atua como **coordenador** (geralmente o cliente que iniciou a transação).

Os demais são **participantes**, responsáveis por aplicar ou abortar a transação local.

Funcionamento do 2PC

Fase 1 – Votação

- O coordenador envia VOTE_REQUEST para todos os participantes.
- Cada participante responde com VOTE_COMMIT (pronto para confirmar) ou VOTE_ABORT (precisa abortar).

Fase 2 – Decisão

- Se **todos** votarem COMMIT → coordenador envia GLOBAL_COMMIT.
- Caso **algum** vote ABORT → coordenador envia GLOBAL_ABORT.
- Participantes executam o commit ou rollback local conforme a decisão global.

Obrigado por participar!