Planos (para lembrar):

Plano restaurável: Exige um processo de restauração complexo (cascata). Necessário informação suficiente no log. Mas, em teoria, nenhuma transação efetivada jamais deveria precisar ser revertida.

Plano livre de cascata: As transações só lêem itens que foram gravados por transações efetivadas.

Plano restrito: As transações não podem nem ler nem gravar um item X até que a última transação que grave X tenha sido efetivada (ou abortada). Restauração mais simples. (**Toda transação só pode ler ou gravar itens que já foram confirmados por transações anteriores**).

Restauração mais simples.

Serialização

Planos seriais:

- São executados consecutivamente, sem intercalação.
- Todo plano serial é correto

Plano não-serial serializável:

- Um plano S não-serial é serializável é correto porque ele é **equivalente a um plano serial** que é considerado correto.

OBS: Um plano é equivalente ao outro se eles produzem resultados iguais.

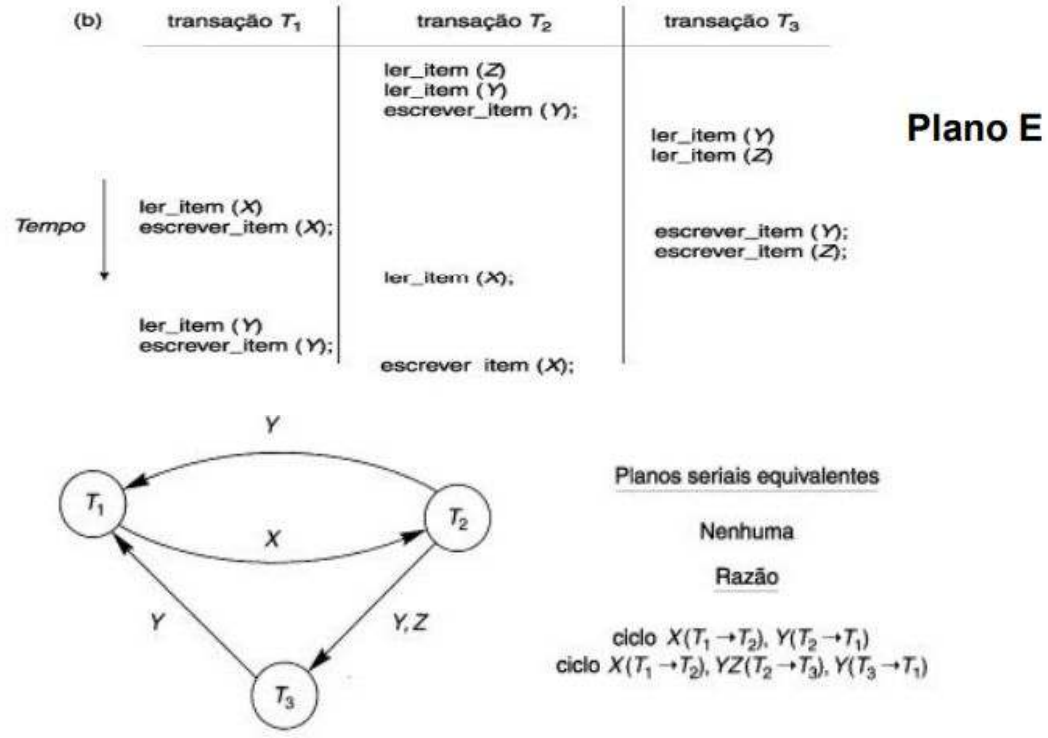
Planos conflito-serializáveis e conflito-equivalentes:

- Dois planos são conflito equivalentes se a ordem de quaisquer duas operações conflitantes for a mesma em ambos os planos
- Operações conflitantes: pertencem a diferentes transações usam o mesmo item do BD e pelo menos uma das duas é de gravação do item
    - Exemplo se em **S1** ocorrer r1(X); w2(X);
        - e em **S2** ocorrer w2(X); r1(X);
- Um plano S é **conflito serializável** se ele for (conflito) equivalente a um plano serial S'

Algoritmo para teste de conflito:

1) Criar um nó para cada Transação Ta do plano S
2) Quando **Tb ler_item(X)** depois que **Ta escrever_item(X)**, cria seta **[Ta -> Tb]**
3) Quando **Tb escrever_item(X)** depois que **Ta ler_item(X)**, cria seta **[Ta -> Tb]**
4) Quando **Tb escrever_item(X)** depois que **Ta escrever_item(X)**, cria seta **[Ta -> Tb]**
5) O plano será serializável se e apenas se o grafo não contiver ciclos.

Ex:



Recuperação de Falhas:

Falhas Catastróficas: Quando há uma falha catastrófica, é necessário que um backup do Banco de Dados (última versão consistente) seja carregada de volta ao disco, refazendo operações das transações já confirmadas a partir do arquivo de log

Falha não catastrófica: reverte as alterações que causaram a inconsistência desfazendo operações de transações não confirmadas ou refazendo operações de transações confirmadas.

Registro Adiantado em Log, Roubado/Não-Roubado e Forçado/Não-Forçado

Para facilitar o processo de recuperação são mantidas listas relacionadas às transações:

- Transações ativas (iniciadas mas não efetivadas)
- Transações efetivadas/abortadas desde o último CHECKPOINT.

Um **checkpoint** é escrito periodicamente dentro do log, no ponto que o sistema grava no disco todos os dados em cache que tivessem sido modificados.

Todas as transações que tiverem um “commit” no log, antes de um “checkpoint”, não necessitarão ter suas operações WRITE refeitas no caso de falha.

- **REDO** inclui o novo valor (AFIM [After Image]) do item gravado.
  [REDO É IDEMPOTENTE]

- **UNDO** inclui o valor antigo (BFIM [Before Image]).
- No algoritmo **UNDO/REDO**, ambos os tipos de entradas devem ser combinadas.

UNDO/REDO → Quando uma transação falha antes de ser efetivada;

UNDO/NO-REDO → Quando as transações já foram feitas antes de gravar no log;

Duas técnicas principais de recuperação: **atualização adiada** e **atualização imediata**.

Atualização Adiada (Não-roubado):

Só atualiza em disco depois da efetivação da transação. Antes da efetivação, os registros são atualizados no buffer (cache).

Durante a efetivação, as atualizações são registradas no LOG e então gravadas no BD.

Não é necessário UNDO mas talvez precise do REDO (NO-UNDO/REDO).

Atualização Imediata (Roubado):

O disco pode ser atualizado por algumas operações antes do seu ponto de efetivação. Estas operações serão registradas no log (gravação forçada) antes de serem aplicadas em disco (para fins de recuperação).

Se uma transação falhar antes da efetivação, suas mudanças devem ser desfeitas.

Geralmente, as operações UNDO e REDO são necessárias (UNDO/REDO).

Variação: todas as atualizações são registradas em disco antes que uma transação seja efeitvada, isto requer apenas UNDO (UNDO/NO-REDO).

Método ARIES:

Usa uma abordagem **roubada/não-forçada** para gravação e está baseado em três conceitos:

1. Registro adiantado em log (*Write-ahead Log*);

2. Repetição do histórico durante o refazer;

3. Mudanças de log durante o Sistemas de Banco de Dados desfazer.

O que é *Write-ahead log* (*WAL*)?

Todas as modificações são escritas no LOG antes de serem aplicadas. Ambas as informações do UNDO e REDO são armazenadas no LOG.

Repetição histórico: o sistema relê todas as ações tomadas pelo SGBD antes da queda, para reconstruir o estado no momento da falha.

Usando o log durante UNDO, evitará que torne a desfazer operações já desfeitas, falha durante recuperação e reinício da recuperação.

Bloqueios

Binários:

Simples mas muito restritivos, não usados em prática

Um bloqueio binário pode ter dois estados ou valores: bloqueios e desbloqueios (ou 1 e 0, para simplificar). Um *bloqueio distinto* é associado a cada item X do banco de dados.

Se o valor do bloqueio em **X for 1**, o item X não pode ser acessado por uma operação de banco de dados que solicite o item.

Se o valor do bloqueio em **X for 0**, o item pode ser acessado quando solicitado.

Referimo-nos ao valor corrente (ou estado) do bloqueio associado a um item X como LOCK(X).

1. Uma transação T deve garantir a operação *lock_item(X)* antes que qualquer operação *ler_item(X)* ou *escrever_item(X)* seja executada em T.

2. Uma transação T deve garantir a operação *unlock_item(X)* depois que todas as operações *ler_item(X)* e *escrever_item(X)* sejam completadas em T.

3. Uma transação T não resultará em uma operação *lock_item(X)* se ela já tiver o bloqueio no item X.

4. Uma transação T não resultará em uma operação *unlock_item(X)*, a menos que ela já tenha o bloqueio no item X.

Compartilhados/exclusivos:

Fornecem mais capacidades gerais de bloqueio e são usados em esquemas práticos

Deveríamos permitir a diversas transações acessarem o mesmo item X, se todas elas acessassem X apenas com propósito de leitura.

Entretanto, se uma transação for alterar um item X, ela deve ter acesso exclusivo a X. Para esse propósito, temos o chamado bloqueio de múltiplo-modo.

Nesse esquema — chamado *bloqueios compartilhados/exclusivos* ou de *leitura/escrita* — há três operações de bloqueio:

read_lock(X) **write_lock(X)** **unlock(X)**

Bloqueio Compartilhado: quando uma transação recebe este tipo de bloqueio e a instrução é de leitura, então mais de uma transação poderá acessar o mesmo dado.

Se a instrução for de gravação, então ela não poderá participar de um bloqueio compartilhado, ou seja, **é permitido que várias transações acessem um**

mesmo item "A" se todas elas acessarem este item "A" apenas para fins de leitura.

Bloqueio Exclusivo: quando uma transação recebe este tipo de bloqueio, ela fica exclusivamente reservada para a instrução que compõe a transação, não permitindo que outra transação faça uso do dado que está sendo utilizado, logo, **um item bloqueado para gravação é chamado de bloqueado exclusivo, pois uma única transação mantém o bloqueio no item.**

Um bloqueio associado a um item X, *LOCK(X)*, tem, agora, **três** possíveis estados: *'read_locked'*, *'write_locked'* ou *'unlocked'.*

Um item read-locked também é chamado de bloqueado-compartilhado (*share-locked*), porque **permite** que outras transações **leiam** o item.

Enquanto um item *write-locked* é chamado de bloqueado-exclusivo (*exclusive-locked*), porque uma transação **única** controla **exclusivamente** o bloqueio no item.

1. Uma transação T deve garantir a operação read_lock(X) ou write_lock(X) antes de qualquer operação ler_item(X) ser executada em T.

2. Uma transação T deve garantir a operação write_lock(X) antes de qualquer operação escrever_item(X) ser executada em T.

3. Uma transação T deve garantir a operação unlock(X) depois que todas as operações ler_item(X) e escrever_item(X) são completadas em T.

4. Uma transação T não vai gerar uma operação read_lock(X) se ela já controlar um bloqueio de leitura (compartilhado) ou um bloqueio de escrita (exclusivo) no item X.

5. Uma transação T não resultará uma operação write_lock(X) se ela já controlar um bloqueio de leitura (compartilhado) ou um bloqueio de escrita (exclusivo) no item X.

6. Uma transação T não resultará uma operação unlock(X) a menos que ela já controle um bloqueio de leitura (compartilhado) ou um bloqueio de escrita (exclusivo) em um item X.

Conversão de Bloqueios: Às vezes é desejável **relaxar as condições 4 e 5** da lista, de forma a permitir **conversão de bloqueio**, isto é, a uma transação que já controla um bloqueio no item X é permitido, sob certas condições, converter o bloqueio de um estado bloqueado para um outro.

Por exemplo, é possível para uma transação T resultar em uma operação read_lock(X) e depois, com a promoção do bloqueio, gerar uma operação write_lock(X).

Conversão de Bloqueios: Quando um bloqueio de leitura se torna um bloqueio de escrita, ou seja, *aumenta* seu grau de bloqueio.

Bloqueio em duas fases para garantir serialização:

*[Com a serialização, transações são executadas **sem a influência de outras transações**]*
[Em um plano serial jamais acontecerá de duas ou mais transações estarem sendo executadas (intercalação)]

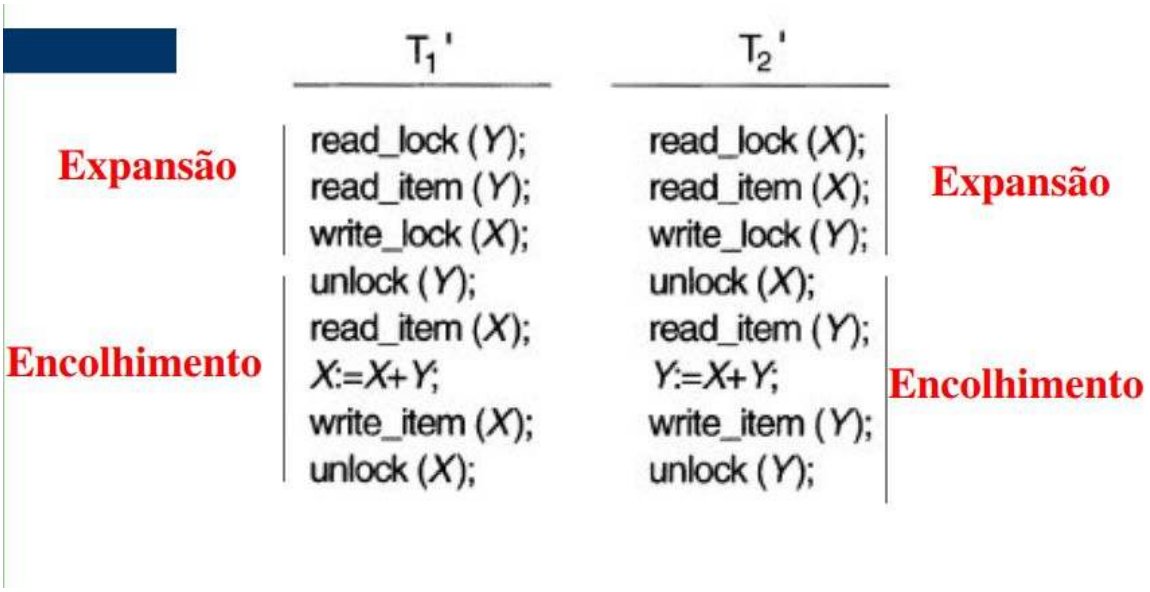
Se toda transação em um plano seguir o protocolo de bloqueio em duas fases, é garantido que o plano seja serializável. Não há necessidade de testes.

Diz-se que uma transação segue o protocolo de bloqueio em duas fases se todas as operações (read_lock, write_lock) precedem a primeira operação de desbloqueio na transação.

Tal transação pode ser dividida em **duas fases**:

- Fase de expansão ou crescimento (primeira), durante a qual novos bloqueios nos itens podem ser adquiridos, mas não podem ser liberados.

- Fase de encolhimento (segunda), durante a qual os bloqueios existentes podem ser liberados, mas novos bloqueios não podem ser adquiridos.



Bloqueio em Duas Fases Básico, Conservador, Estrito e Rigoroso:

- Vários tipos de bloqueios em duas fases (2PL = 2 Phases Lock), o mencionado **acima** é o **básico**.

- Conservador ou estático: a transação deve bloquear todos os itens que ela acessa antes de iniciar a sua execução. Difícil de usar na prática devido à necessidade da pré-declaração de read-set e write-set que não é possível na maioria das situações.

- Estrito: garante planos restritos. T não libera nenhum de seus bloqueios exclusivos até que ela efetive ou aborte.

Lidando com Deadlock e Starvation:

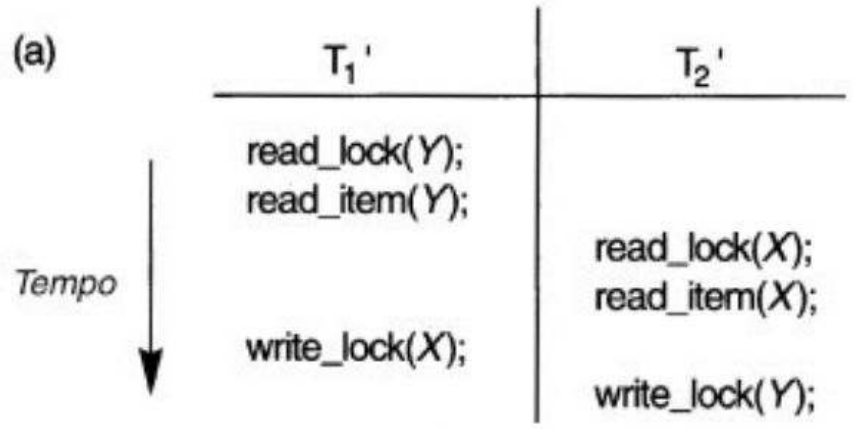
Temos alguns possíveis problemas quando lidamos com locks:

- ***Deadlock***: Quando cada transação em um conjunto de duas ou mais transações espera por algum item que esteja bloqueado por alguma outra transação T´ no conjunto.

Possíveis soluções:

Em estado de deadlock, algumas transações devem ser abortadas.

Outro esquema: *timeouts*. Método prático: se uma transação esperar por um período maior que um tempo definido, se assume que está em deadlock e deve abortar.



Exemplo de solução:

Ti tenta bloquear um item X mas X está bloqueado por Tj com um bloqueio conflitante.

As alternativas são:

Esperar-morrer: Se TS(Ti) < TS(Tj), ou seja, Ti mas antiga do que Tj, então Ti é autorizada a esperar.

Se TS(Ti) > TS(Tj), ou seja, Ti mais nova do que Tj, então Ti é abortada e reiniciada posteriormente com o mesmo valor de registro de timestamp

As **transações mais velhas são as que esperam**, nenhum ciclo é criado.

Ferir-esperar: Se TS(Ti) < TS(Tj), ou seja, Ti mais antiga, aborta Tj (**Ti fere Tj**) e reinicia mais tarde com o mesmo timestamp;

Senão (Ti mais nova) Ti espera.

As **transações somente esperam as mais velhas**, nenhum ciclo é criado.

Ambos os esquemas finalizam abortando a transação mais jovem, que pode estar envolvida em um deadlock. Problema de aborto desnecessário.

- ***Starvation***: Ocorre quando uma transação não pode continuar por um período indefinido de tempo, enquanto que outras sim. Ou seja: uma transação espera por um recurso enquanto outras “pegam” o recurso antes dela repetidamente.

Possíveis soluções:

Solução um esquema de espera imparcial => uma fila.

Outro esquema é o de prioridade por tempo de espera ou abortadas muitas vezes. Os esquemas “esperar-morrer” e “ferir-esperar” evitam inanição.

Controle de concorrência baseado em ordenação de Timestamp

Timestamp: identificador único criado pelo SGBD para identificar uma transação -> se dá pelo momento de início da transação: TS(T).

Não são usados bloqueios, portanto, não ocorrem deadlocks.

Sobre o algoritmo de ordenação por timestamp:

Se uma transação velha Ti tem um timestamp TS(Ta), uma transação nova Tb é atribuída um time-stamp TS(Tb) tal que TS(Ta) < TS(Tb).

O algoritmo gerencia a execução concorrente de tal forma que os timestamps determinam a ordem de serializabilidade.

Para fazer isso, o algoritmo associa a cada item X do BD dois valores de timestamp:

- **Read_TS(X)**: Maior timestamp entre todos os timestamps de transações que tenham lido o item X com sucesso.

Read_TS(X) = TS(T), onde T é a mais nova que tenha lido X com sucesso.

- **Write_TS(X)**: Maior timestamp entre todos os timestamps de transações que tenham escrito o item X com sucesso.

Write_TS(X) = TS(T), onde T é a mais nova que tenha escrito X com sucesso.

Suponhamos que uma transação Ti tenta executar um **escrever_item(X)**.

- Se **read_TS(X)** > TS(Ti) ou se write_TS(X) > TS(Ti), então Ti aborta e reverte, e a operação é rejeitada. Uma operação mais nova que Ti já leu ou escreveu o valor do item X antes de Ti, violando a ordenação por timestamp.
- Senão, Ti executa a operação **escrever_item(X)** e grava write_TS(X) com TS(Ti).

Suponhamos que uma transação Ti tenta executar um **ler_item(X)**.

- Se **write_TS(X)** > TS(Ti), então Ti aborta, reverte e rejeita a operação. Isso ocorre porque alguma transação mais nova, já teria escrito o valor no item X antes que Ti tivesse a chance de ler X.

- Se **write_TS(X)** ≤ TS(Ti), então executa a operação de ler_item(X) de Ti, , e atualiza read_TS(X) com max(TS(Ti,), read_TS(X)).

Bloqueio em multiversão:

As técnicas de controle de concorrência multiversão **armazenam os valores antigos** de um item de dado quando ele é atualizado.

Quando uma transação escreve um item, uma nova versão do item é escrita e a versão antiga do item é retida.

Algumas operações de leitura, que seriam rejeitadas em outras técnicas, passam a ser aceitas por meio da leitura de uma versão anterior do item para manter a serialização.

Desvantagem: necessidade de um volume maior de armazenamento para manter as versões dos itens do banco. • As técnicas multiversão são ideais para banco de dados temporal.

2 métodos mais usados:

- Bloqueios de certificação
- Ordenação por timestamp (TO)

Bloqueios de Certificação:

Utiliza 3 modos de bloqueio: **ler**, **gravar** e **certificar**.

Assim tem-se as seguintes operações:

- **rl(X)**, **wl(X)** e **ul(X)** -> já vistas
- **Certify_locked(X)** – **cl(X)** -> bloqueio de certificação

Ordenação por timestamp (TO):

Já mostrado lá em cima, mesmo princípio.

Granularidade

Um item de BD pode ser:

- Um registro.
- Um valor de um campo de um registro.
- Um bloco de disco.
- Um arquivo.
- Um banco de dados inteiro.

A **granularidade** pode afetar a execução do controle de **concorrência** e **recuperação**.

Considerações sobre Níveis de Granularidade para Bloqueio:

- O tamanho dos itens de igual granularidade.
- Quanto **maior o tamanho do item**, **menor** é o grau de **concorrência** permitido.
- Porém, quanto **menor o tamanho** do item, **maior** o **número de itens**, ou seja: maior sobrecarga do sistema.
- Portanto, qual é o melhor tamanho do item?
  Depende dos tipos de transações envolvidas.
  Se uma transação típica acessa **muitos registros** em um mesmo arquivo -> **granularidade maior**.

Bloqueio de Nível de Granularidade Múltipla:

Já que o melhor tamanho de granularidade depende da transação dada, teremos múltiplos níveis de granularidades.

Para tornar prático este protocolo, temos bloqueios de intenção.

A idéia é que a **transação indique**, ao longo do caminho da raíz ao nó desejado, qual **tipo de bloqueio** ele irá solicitar.

Três tipos de bloqueio:

- ***Intenção-compartilhada (IS)***: indica que bloqueios compartilhados serão solicitados em alguns nós descendentes;

- ***Intenção-exclusiva (IX)***: indica que bloqueios exclusivos serão solicitados em alguns nós descendentes;

- ***Intenção-compartilhada-exclusiva (SIX)***: indica que o nó corrente está bloqueado compartilhado, mas bloqueios exclusivos serão solicitados em alguns nós descendentes;

O protocolo de bloqueio de granularidade múltipla (MGL) consiste das seguintes regras:

1. Cumprir a compatibilidade definida na tabela apresentada a seguir;

2. A raíz da árvore deve ser bloqueada primeiro (qualquer modo);

3. Um nó pode ser bloqueado no modo **S** ou **IS** por uma transação T, só se o pai estiver bloqueado pela transação T no modo **IS** ou **IX**;

4. Um nó pode ser bloqueado no modo **X**, **IX** ou **SIX** por uma transação T, só se o pai estiver bloqueado pela transação T no modo **IX** ou **SIX**;

5. Uma transação pode bloquear um nó apenas se ela não tiver nenhum nó desbloqueado (2PL = 2 Phases Lock);

6. Uma transação T pode bloquear um nó apenas se nenhum dos seus nós filhos estiverem correntemente bloqueados por T.

	IS	IX	S	SIX	X
IS	sim	sim	sim	sim	não
IX	sim	sim	não	não	não
S	sim	não	sim	não	não
SIX	sim	não	não	não	não
X	não	não	não	não	não

Níveis de Isolamento

Read Uncommitted: Este é o nível menos isolado e o como o próprio nome já sugere, ele permite a leitura antes da confirmação.

Neste nível de isolamento todos os problemas (os 3 da tabela) podem ocorrer sem restrição.

Read Committed: Neste nível de isolação não podem ocorrer Dirty Reads mas são permitidos Nonrepeatable reads e Phantom Reads. Este é o nível padrão do PostgresSQL.

Repeatable Readed: Aqui apenas ocorrem Phantom Reads. O SGBD bloqueia o conjunto de dados lidos de uma transação, não permitindo leitura de dados alterados ou deletados mesmo que committados pela transação concorrente, porém ele permite a leitura de novos registros committados por outras transações.

Serializable: Este é o nível mais isolado que não permite nenhum tipo de problema (Dirty Read, Nonrepeatable read e Phantom Read).

Nível de Isolamento	Dirty Read	Nonrepeatable read	Phantom Read
Leitura Não-Efetivada	Sim	Sim	Sim
Leitura Efetivada	Não	Sim	Sim
Leitura Repetível/Repetida	Não	Não	Sim
Serializável	Não	Não	Não

Projeto Físico e Tuning:
Índices

Para a escolha de índices, devemos levar em consideração:

- As consultas mais importantes e a sua freqüência;
- As atualizações mais importantes e a sua freqüência;
- O desempenho desejado para essas consultas e atualizações.

Antes de ser definido um novo índice, deve ser avaliado o impacto sobre as atualizações:

- Consultas mais rápidas;
- Atualizações mais lentas;
- Utiliza mais espaço.

Atributos mencionados na cláusula WHERE são candidatos a índices.

Índice clusterizado (agrupado) por relação:

No **máximo um** índice pode ser **clusterizado por relação** e eles podem melhorar bastante o desempenho.

Consultas de faixas são bastante beneficiadas por estes índices.

Se várias consultas de faixas são feitas numa relação:

Se for uma estratégia de avaliação baseada unicamente em índices, não é necessário clusterizar.

Índices por Árvore B+:

Consultas por faixas de dados, por exemplo:

```
SELECT * FROM Pessoa
WHERE idade > 30
```

Índices HASH:

Consultas EXATAS, por exemplo:

```
SELECT * FROM Pessoa
WHERE idade = 35
```

Clustering beneficia consultas por faixas e consultas exatas **se** vários dados contêm o mesmo valor da chave.

Hash vs. Árvores

Um índice em **árvore B+ é preferível** por suportar consulta de faixas e consultas de igualdade.

Mas um índice hash é preferível:

- Suporte de junções de laços aninhados indexados: a relação indexada é a interna, e a chave de busca inclui as colunas de junção.
- Uma consulta de igualdade muito importante e nenhuma por faixa.

Exemplos:

SELECT E.dno FROM Emp E
WHERE E.age > 40

Índice em **Árvore B+** sobre **E.age** pode ser usado para obter as tuplas.

SELECT E.dno, COUNT(*)
FROM Employee E
WHERE E.age > 30
GROUP BY E.dno

Considerando o GROUP BY, vale a pena fazer um índice de agrupamento para **E.dno**, pois se várias tuplas têm **E.age > 30**, o uso do índice **E.age** e ordenação das tuplas recuperadas pode ser custosa.

SELECT E.dno
FROM Emp E
WHERE E.hobby = 'Stamps'

Consultas de igualdade com duplicatas:

Agrupamento sobre **E.hobby** vale a pena.

Atributos Compostos

Quando há busca por uma combinação de campos.

Índices compostos são maiores e atualizados com mais frequência.

Consulta igualdade:

Todo campo é igual a uma constante.

Índice <*sal, age*> para **age = 20 e sal = 10**

Consulta faixa:

Algum campo não é uma constante.

Índice <*sal, age*> para **age = 20 e sal > 10**

ou para **age = 20**

Para recuperar registros **Emp** com **age=30 e sal=4000**, um **índice HASH** <*age, sal*> seria melhor que um índice sobre age ou um índice sobre sal.

Se a condição for **20<age<30** e **3000<sal<5000** (faixas), então um **índice em árvore agrupado** sobre <*age, sal*> ou <*sal, age*> é melhor.

Se a condição for **age=30** e **3000<sal<5000**, então **índice agrupado** por <*age, sal*> é melhor que <*sal, age*>.

```
SELECT E.eid
FROM Employee E
WHERE E.age BETWEEN 20 AND 30
AND E.sal BETWEEN 3000 AND 5000
```

Um **índice B+ composto agrupado** sobre <*age, sal*> (nesse caso pode ser por <sal,age> também, pois ambos estão em faixas) pode ajudar.

```
SELECT E.eid
FROM Employee E
WHERE E.age = 25
    AND E.sal BETWEEN 3000 AND 5000
```

Um **índice B+ composto agrupado** sobre <*age, sal*> dará um bom desempenho. Por outro lado, se fosse sobre <*sal, age*>, seria RUIM, pois nesse caso estamos procurando por faixas de salário. Se usarmos o índice <*age,sal*> teremos "buckets" dos valores de salário para cada idade, o que é muito vantajoso, pois basta ir no bucket de idade 25 e puxar o salário que condiz com a faixa.

Já se fizemos um índice <*sal,age*>, teremos buckets com as diferentes idades para cada salário, que nesse caso é inútil.

```
SELECT E.dno, COUNT(*)
FROM Employee E
WHERE E.sal = 10.000
GROUP BY E.dno
```

Um índice sobre dno não permite avaliação só no índice.
Mas um **índice composto**, sim. Com um **índice B+** por <*sal, dno*> é possível e mais eficiente, mas se a condição passa a ser **E.sal > 10000**, já não é possível.

```
SELECT E.ename, D.dname
FROM Emp E, Dept D
WHERE E.sal BETWEEN 10000 AND 20000
AND E.hobby = ‘Stamps'
AND E.dno=D.dno
```

Claramente, **Emp** deveria ser a relação externa.
Seria bom um **índice hash** sobre **D.dno**.

Qual índice deveríamos construir para **Emp**?

Um **árvore B+ sobre E.sal** poderia ser usado, ou um **índice sobre E.hobby** poderia ser usado.

A escolha entre qual usar depende da seletividade das condições (em geral, as seleções de igualdades são mais seletivas que as seleções por faixas).

Agrupamento e Indexação

Agrupamento é especialmente importante quando acessamos **tuplas internas numa junção de laço aninhado indexada**.

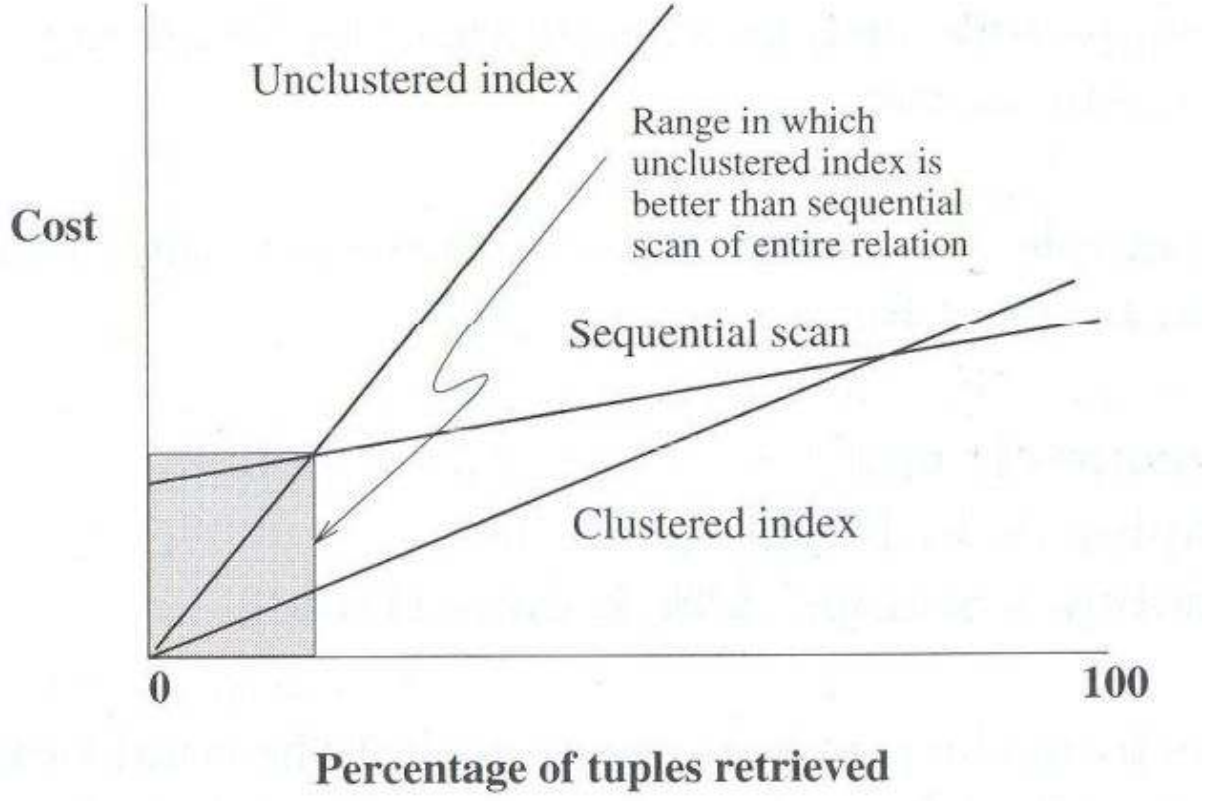
Dica: Agrupamento é útil sempre que várias tuplas têm que ser recuperadas.

Exemplo:

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE D.dname='Toy'
AND E.dno=D.dno
```

Deveríamos fazer um **índice agrupado** sobre **E.dno**.

E se a cláusula WHERE fosse **WHERE E.hobby=Stamps AND E.dno=D.dno**? Se muitos empregados colecionam selos, junção sort-merge pode ser válido considerar. Um **índice agrupado sobre D.dno** ajudaria.



Normalização

Recapitulando:

- 1NF: Não podem haver atributos multivalorados.

- 2NF: Nenhum atributo não-chave pode ser dependente de apenas parte da chave.

- 3NF: Nenhum atributo não-chave de R pode possuir dependência transitiva, para cada chave candidata de R.
  Ou seja: todos os atributos dessa tabela devem ser **independentes** uns dos outros, ao mesmo tempo que devem ser **dependentes exclusivamente** da chave primária da tabela.

- BCNF: Todo atributo não chave deve depender funcionalmente **diretamente da chave primária**, ou seja, não há dependências entre atributos não chave

Às vezes, deixar na BCNF (Boyce-Codd) pode acabar quebrando uma dependência funcional; assim, se a aplicação fizer uma consulta em 2 tabelas, a performance cai. Logo, vale mais a pena desnormalizar e manter na 3NF.