Banco de Dados II

Profª MSc. Tanisi Carvalho
Profª MSc. Simone Vicari

Notas de Aula

Sumário

1 Gerência de Transações

Um conceito bastante importante no contexto de sistemas gerenciadores de banco de dados (SGBDs) é o conceito da *transação*:

∗ A transação é uma unidade de trabalho do usuário (da aplicação) que é atômica do ponto de vista da aplicação [DAT 81]

∗ A transação é uma unidade de execução de programa que acessa e, possivelmente, atualiza vários itens de dados. Geralmente, é o resultado da execução de um programa escrito em uma linguagem de manipulação de dados de alto nível ou em uma linguagem de programação (por exemplo, SQL, COBOL, C ou Pascal) [SIL 99]

A partir dos dois conceitos acima é importante definir que:

- a transação sempre leva o banco de dados de um estado inicial consistente a um outro final, também consistente;
- a transação é atômica, ou seja, a transação deve ser tratada pelo SGBD como uma unidade única, ou todas as alterações devem estar no BD ou nada deve acontecer.

Normalmente, as transações definidas em linguagens de manipulação apresentam comandos para marcar o início e o fim da transação (*begin transaction/end transaction*). A Figura 1.1-1 apresenta um exemplo de uma transação que retira R$ 100,00 de uma Conta Bancária. A Figura 1.1-2 apresenta o mesmo exemplo em SQL.

```
Begin transaction
  Read(Cta=x, saldo)
  Saldo = saldo - 100 (valor informado)
  Write(cta=x, saldo)
end transaction
```

Figura 1.1-1: Exemplo de transação

O padrão SQL92 define que uma transação inicia a partir do primeiro comando SQL, todos os comandos subseqüentes são considerados da mesma transação até que seja encontrado um comando de fim de transação (COMMIT ou ROLLBACK).

```
UPDATE CONTA
  SET saldo = saldo - 100
  WHERE nro_conta = x;
COMMIT;
```

Figura 1.1-2: Exemplo de transação em SQL

Uma transação é encerrada com os comandos de commit(confirma os comandos executados na transação) e rollback (desfaz todas as alterações nos dados realizadas pela transação). No Oracle todo o comando DDL (create, drop) e DCL (grant e revoke) encerram implicitamente uma transação.

Se a ferramenta de acesso ao banco de dados estiver configurada com auto commit todo comando tem um commit implícito.

É possível definir marcas na transação denominadas savepoints com isso é possível fazer o rollback de partes das transações.

```
SQL> insert into cargo values(1,'Gerente');
SQL> savepoint ponto1;
SQL> insert into cargo values(2,'Operador');
SQL> select * from cargo;
SQL> id_cargo nome
        1       Gerente
        2       Operador
SQL> rollback to ponto1;
SQL> select * from cargo;
SQL> id_cargo nome
        1       Gerente
```

1.1 Propriedades da Transação

Para garantir a integridade do banco de dados é necessário que o sistema de banco de dados mantenha as seguintes propriedades da transação:

- **A**tomicidade: o programa que representa a transação deve, ou executar por completo e com sucesso sobre o banco de dados, ou deve parecer nunca ter executado (Lei do Tudo-Ou-Nada). Essa propriedade é, normalmente, garantida pelo SGBD, através de seu componente de recuperação após falhas;
- **C**onsistência: a transação só pode realizar operações corretas, do ponto e vista da aplicação, sobre o banco de dados. Isto é, a transação não pode transgredir qualquer restrição de integridade (RI) declarada ao SGBD (restrições de integridade de chave primária e restrições de integridade de chave estrangeira.
- **I**solamento: em um ambiente multiusuário (multiprogramado), a execução de uma transação não deve ser influenciada pela execução de outras. Essa propriedade é, normalmente, garantida pelo SGBD, através do módulo de Controle de Concorrência (scheduler);
- **D**urabilidade: os resultados de uma transação que termina com sucesso devem permanecer inalterados, no banco de dados, até que outra transação os altere e também termine com sucesso. Isto é, os resultados de transações que terminam com sucesso devem sobreviver a falhas (de transação, de sistema ou de meio de armazenamento. A Durabilidade também é garantida pelo componente de recuperação após falhas.

As transações que seguem estas quatro propriedades são também conhecidas como transações ACID.

1.2 Estados da Transação

Uma transação pode concluir sua execução com sucesso ou não. Quando uma transação não conclui com sucesso, a propriedade da atomicidade deve ser garantida, isso significa que todas as modificações feitas sobre o banco de dados, por essa transação,

devem ser desfeitas (*rollback*). Se, no entanto, a transação conclui com sucesso seus efeitos estão materializados no banco de dados e ela é dita *committed*.

Podemos dizer que uma transação, ao longo da sua execução, passa por vários estados:

- *ativa*: é o estado inicial, uma transação fica nesse estado enquanto estiver executando;
- *em efetivação*: após a execução da última declaração;
- *em falha*: após a descoberta de que a execução normal não poderá se realizar;
- *abort*: depois que a transação foi desfeita e o banco de dados foi restabelecido ao estado anterior ao início da execução da transação;
- *commit*: após a conclusão com sucesso da transação.

A Figura 1.3 apresenta o diagrama de estados correspondente a uma transação.

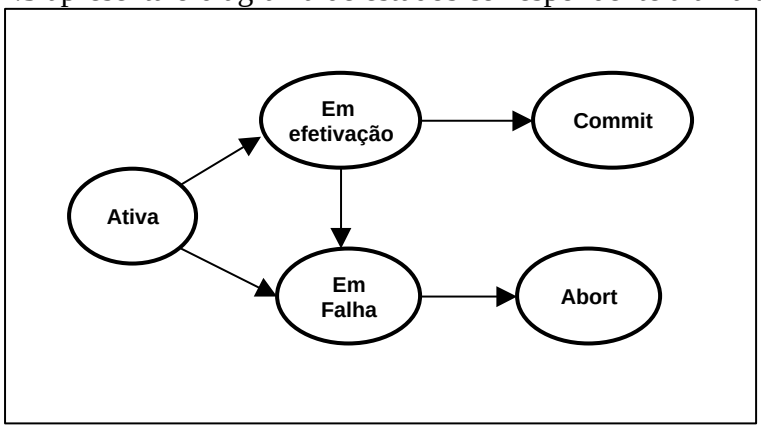


Figura 1.3: Diagrama de Estados da Transação

Enquanto uma transação estiver executando ela estará no estado *ativa*. No momento em que a transação concluir a execução da última declaração ela entra no estado de *efetivação*. Uma transação não vai direto para o estado de *commit*, porque o SGBD precisa concluir uma série de operações como, por exemplo, atualizar arquivos de log, atualizar o banco de dados, para garantir as propriedades de atomicidade e durabilidade. O estado de efetivação é uma preparação para o *commit*. Pode ocorrer que durante esse estado ocorra alguma falha e a transação não possa ser concluída normalmente. Nessa situação, a transação passa de um estado de *efetivação* para um estado *em falha* onde todos os efeitos da transação deverão ser desfeitos. Uma transação passará para o estado de *commit* quando todas as operações necessárias estiverem concluídas. A partir desse momento a aplicação teria conhecimento de que a transação realmente concluiu. O estado *abort* se dará quando uma vez detectada a falha o estado do banco de dados, anterior à falha, tenha sido estabelecido.

A Figura 1.4 apresenta um esquema mostrando a relação existente entre o programa de aplicação e a execução de uma transação pelo SGBD (Gerente de Transações).

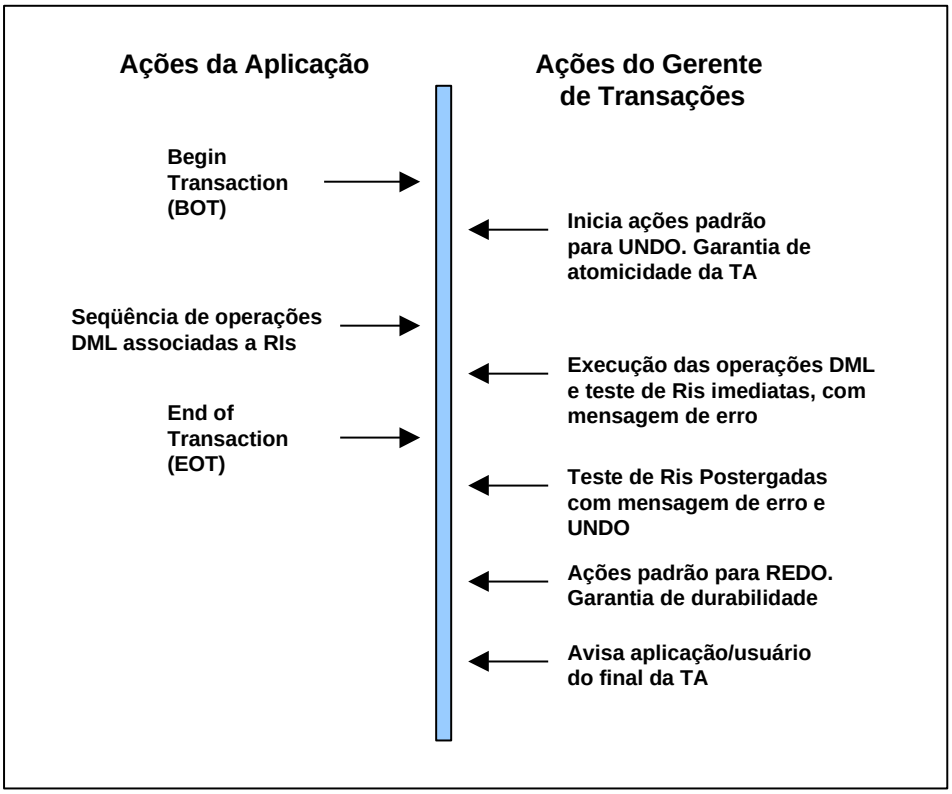


Figura 1.4: Ações da Aplicação x Ações do Gerente de Transações

Uma transação começa a executar quando uma instrução Begin Transaction é encontrada. O gerente de transações deve então executar um conjunto de ações para garantir a atomicidade de uma transação no caso de uma falha. A aplicação continua executando as suas operações. estas operações são processadas pelo SGBD, as restrições de integridade associadas são testadas. A aplicação envia a declaração de fim de Transação. A partir desse momento a transação está em estado de *efetivação*, o SGBD testa as restrições de integridade postergadas, que só podem ser testadas no fim, executada as ações de REDO para garantir a durabilidade e, por fim, envia um aviso a aplicação de que a transação foi concluída com sucesso. Nesse ponto, a transação passa a ser *commit* e seus efeitos estão materializados no banco de dados. No caso de uma falha, o sistema precisa executar um conjunto de operações para garantir a atomicidade e a transação é abortada.

Bibliografia utilizada no capítulo:

[BER 87] BERNSTEIN, Philip A; HADZILACOS, Vassos; GOODMAN, Nathan. **Concurrency Control and Recovery in Database Systems**. Reading:Addison-Wesley, 1987.

[DAT 81] DATE, Chris. J. **An Introduction to Database Systems**. Reading:Addison-Wesley, 1983, p. 112-123.

[IOC 95] IOCHPE, Cirano. **Notas de Aula CMP97 – Sistemas de Banco de Dados.** Curso de Pós-Graduação em Ciência da Computação, 1995.

[SIL 99] SILBERSCHATZ, Abraham.; KORTH, Henry F.; SUDARSHAN, S. Sistema de Banco de Dados. 3ª edição., São Paulo: Makron Books, 1999.

2 Controle de Concorrência

O controle de concorrência é a atividade de coordenar a ação de processos que operam em paralelo, acessam dados compartilhados e, portanto, interferem uns com os outros.

Quando duas ou mais transações executam concorrentemente, suas operações podem ser processadas, pelo sistema de computação, de forma intercalada (*interleaving*). A Figura 2-1 apresenta um exemplo da execução serial das transações T_1, T_2 e T_3 e da execução *interleaving* das mesmas transações.

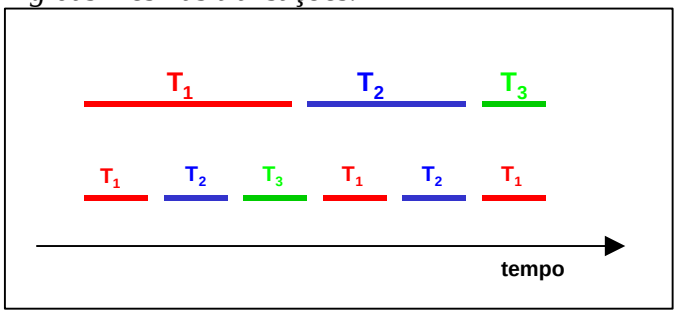


Figura 2-1: Exemplos de execuções das transações T_1, T_2 e T_3

A execução concorrente das transações permite um melhor aproveitamento dos recursos do sistema. A execução de uma transação geralmente envolve operações de E/S liberando o uso da CPU, tornando-a ociosa. Esse tempo de CPU pode ser destinado à execução de outra transação. A idéia central é manter em paralelo o processamento da E/S e da CPU melhorando, assim, o *throughput* do sistema (número de transações executadas por unidade de tempo).

Um outro aspecto é que em um sistema podem existir transações longas e transações curtas. Se a execução for seqüencial, então pode ocorrer da transação curta ter que aguardar pela transação longa acarretando em um tempo de resposta maior (tempo que uma transação leva para começar a responder à solicitação realizada). A Figura 2-1 apresenta um exemplo dessa situação, a transação T3 tem que aguardar a execução de T1 e T2; já na execução *interleaving*, o tempo de T3 é melhorado.

Na Figura 2-2 temos a arquitetura do SGBD do ponto de vista de gerência de transações:

- gerente de transações: executa as operações do banco de dados e das transações, passando-as para o scheduler;
- scheduler: controla a ordem da execução das operações *read*, *write*, *commit* e *abort* de diferentes transações que são executadas, garantindo o isolamento. As ações do scheduler podem ser: executar, rejeitar ou postergar a operação que chega para o scheduler;
- gerente de recuperação: responsável pela finalização das transações (garante a atomicidade e a durabilidade);

- gerente de buffers: armazena parte do banco de dados em memória durante a execução das transações. É responsável pela tansferência dos dados entre o disco e memória principal.

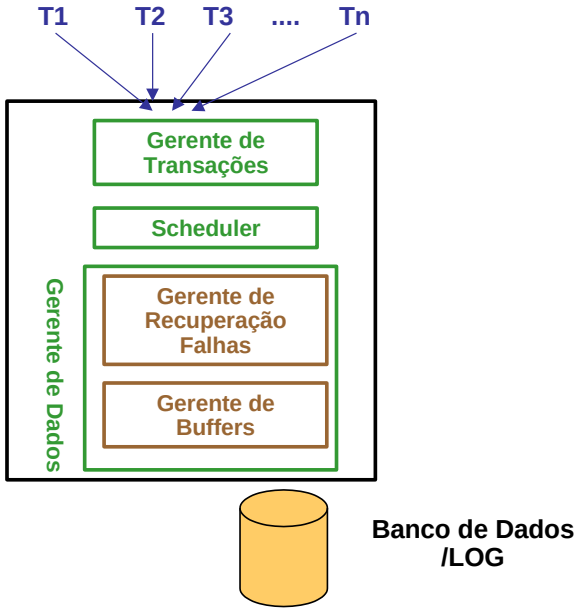


Figura 2-2 Gerência de Transações

2.1 Problemas associados à execução concorrente de TAs

Quando se tem transações executando concorrentemente pode-se levar o banco de dados a um estado inconsistente.

Problemas que podem existir sem um mecanismo de controle de concorrência:

Atualização perdida: ocorre quando uma atualização de uma transação é sobreescrita pela atualização de outra transação, fazendo com que uma delas seja perdida;

a) execução serial

T1	T2
read(x) x=x-10 write(x) read(y) Y=y+10 write(y) commit	read(x) x=x+100 write(x) commit

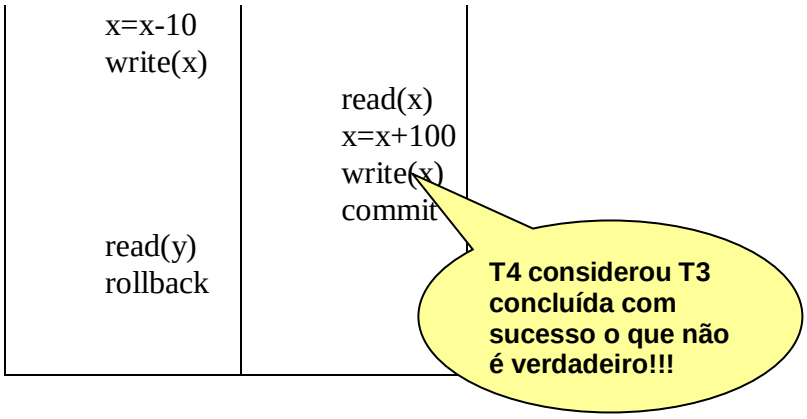
b) execução não serial

T1	T2
read(x) x=x-10 write(x) read(y) y=y+n write(y) commit	 read(x) x=x+100 write(x) Commit

TAs não recuperáveis:

Uma transação não pode se basear em dados de uma transação que ainda não foi efetivada (princípio da recuperabilidade).

T3	T4
read(x)	



2.2 Mecanismos para Controle de Concorrência

Os mecanismos de controle de concorrência têm por objetivo permitir a concorrência garantido o princípio da serializabilidade. Serão estudados três implementações de mecanismos de controle de concorrência:

- marcas de tempo (*timestamp*);
- versões.
- bloqueios (*locking);*

2.2.1 Bloqueio simples

Consiste em estabelecer bloqueios sobre o dado a ser acessado pela transação, de modo a restringir o acesso a este dado.

A unidade de bloqueio, geralmente, é a de registro. Quanto menor for a granularidade de bloqueio, maior o grau de concorrência.

São definidos dois tipos de bloqueios:

- Exclusivo (E): se uma transação obteve um bloqueio E em um dado, então a transação pode ler e escrever neste dado.
- Compartilhado (C): se uma transação obteve um bloqueio C em um dado, esta transação e as demais podem somente ler este dado

Matriz de Compatibilidade de Bloqueios:

	C	E
C	✓	X
E	X	X

Quando uma transação Ti deseja acessar um dado, ela deve primeiro bloqueá-lo com um dos dois tipos de bloqueio, dependendo da operação desejada. Se o dado já está bloqueado com tipo incompatível, Ti precisa esperar até que todos os bloqueios incompatíveis sejam liberados.

Tão logo o dado deixe de ser utilizado o bloqueio deve ser liberado.

T1	T2
lock_E(x) read(x)	
	lock_E(x) espera

x=x-1	...
write(x)	...
unlock(x)	...
	lock_E(x)
	x=x+100
	write(x)
	unlock(x)

Um dos problemas associados a este protocolo é o fato dos bloqueios serem liberados muito cedo. Observe o exemplo abaixo:

T3	T4
	sum=0
	lock_C(x)
	read(x)
	unlock(x)
	sum=sum+x
	lock_C(y)
lock_E(x)	
x=x-10	
write(x)	
unlock(x)	
commit	
	read(y)
	unlock(y)
	sum-sum+y

O protocolo de bloqueio simples não garante o isolamento por completo. No exemplo acima, caso T4 seja abortada T3 foi efetivada com valores de ***x*** que não existiram no banco de dados (atomicidade para T4).

2.2.2 Bloqueio de Duas Fases (2PL)

O mecanismo de bloqueio de duas fases (*Two-Phase Lock*), garante a serializabilidade tratando os bloqueios em duas fases:

- *fase de aquisição (growing phase)*: nesta fase a transação apenas adquire os bloqueios;
- *fase de liberação (shrinking phase)*: nesta fase a transação somente libera bloqueios. A partir do momento que o primeiro bloqueio é liberado nenhum bloqueio pode ser adquirido pela transação.

O Strict 2PL garente execuções concorrentes serializáveis, recuperáveis, evita o cascating abort e liberando os bloqueios apenas no commit ou rollback.

O mecanismo funciona de forma análoga ao mecanismo de bloqueios simples, diferenciando-se apenas pela forma como são tratadas a aquisição e a liberação de bloqueios.

T1	T2
read(x)	
x=x-10	
	read(x)
	x=x+100
	write(x)
	commit
write(x)	
read(y)	
y=y+10	
write(y)	

T1	T2
Lock_E(x)	
x=x-10	
	lock_E(x)
	espera
	...
	...
write(x)	...
lock_E(y)	...
read(y)	...
y=y+10	...

commit	

write(y) commit unlock(x,y)	... ... continua ...

A implementação do mecanismo 2PL tem algumas variações conforme mostra a Figura 2-3. As variações caracterizam-se pela forma como são implementadas as fases de aquisição e liberação de bloqueios.

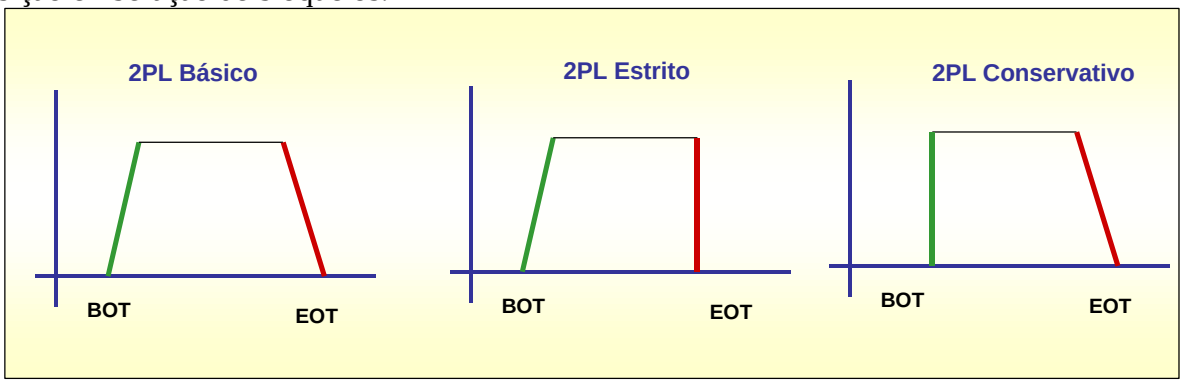


Figura 2-3: Variações do mecanismo 2PL

Na versão 2PL Básico, os bloqueios são liberados à medida que a transação deixa de utilizá-los. Isso pode acarretar num problema conhecido como aborto em cascata (*cascading abort*). Uma vez que o bloqueio foi liberado, este dado pode ser utilizado por qualquer outra transação, assim se a transação não concluir com sucesso, outras transações terão se baseado em um dado intermediário levando a uma inconsistência do banco de dados.

A implementação estrita resolve esta situação liberando os bloqueios apenas no final e em um momento único, garantindo com isso que o problema de aborto em cascata não ocorra. Esta implementação pode ter ainda uma otimização, permitindo que os bloqueios compartilhados sejam liberados deixando apenas os bloqueios exclusivos. A implementação estrita é a mais utilizada comercialmente.

Uma característica dos protocolos baseados em bloqueios são as situações de impasse (*deadlocks*). Uma situação de impasse ocorre quando uma transação está a espera de um dado que está bloqueado por uma segunda transação e esta está a espera de um dado segurado pela primeira. As situações de *deadlock* são indesejáveis, pois degradam o desempenho. Este assunto será detalhado na próxima sessão.

A implementação conservativo evita que o *deadlock* ocorra, pois solicita todos os dados de antemão. Esta abordagem, no entanto, tende a diminuir o throughput do sistema, visto que uma transação não vai utilizar todos os dados ao mesmo. Além disso, é difícil saber quais dados serão utilizados por uma transação. Esta implementação pode levar a outro problema conhecido como postergação definida, situação em que uma transação fica a espera de um evento que nunca venha a ocorrer.

Uma outra característica dos protocolos de bloqueios é que alguns sistemas permitem iniciar o bloqueio como compartilhado e depois passar este bloqueio para o modo exclusivo (*lock upgrade/downgrade*). A vantagem disso é que pode-se retardar a aquisição de bloqueios no modo exclusivo, permitindo assim um maior grau de concorrência. Tanto para o *upgrade* ou *downgrade* a matriz de compatibilidade de bloqueios deve ser obedecida. Isso pode acarretar em situações de *deadlock*.

2.3 *Deadlock*

Uma situação que pode ocorrer em sistemas concorrentes é conhecida por impasse ou *deadlock* (abraço mortal). O *deadlock* está associado a utilização de recursos compartilhados que só podem ser utilizados de forma exclusiva, no caso de banco de dados, os dados utilizados pelas transações.

Um sistema está em *deadlock* sempre que uma transação Ti está esperando por um item de dado que está bloqueado por uma transação Tj e Tj está esperando por um item de dado que está bloqueado por Ti.

Há dois métodos para resolver um *deadlock*:

- utilizar alguma técnica que evite a sua ocorrência;
- possuir mecanismos para detecção e recuperação de *deadlock*.

2.3.1 Prevenção de *Deadlock*

Há duas abordagens para a prevenção de *deadlock*. Uma garante que nenhum ciclo de espera poderá ocorrer pela ordenação de solicitações de bloqueio, ou pela aquisição de todos os bloqueios juntos. A outra faz com que a transação seja refeita, em vez de esperar por um bloqueio, sempre que a espera possa potencialmente decorrer em *deadlock*.

Pela primeira abordagem, cada transação é obrigada a bloquear todos os itens de dados antes de sua execução. Além disso, ou todos os dados são bloqueados de uma só vez ou nenhum será. Há duas desvantagens nesse protocolo: a dificuldade de prever, antes da transação começar, quais itens de dados deverão ser bloqueados. Segundo, a utilização do item de dado pode ser bastante reduzida já que os dados podem ser bloqueados e não ser usados por um longo período de tempo.

Pela segunda, são utilizados *timeouts* para decidir se uma transação deve ficar em wait ou deve ser refeita (REDO). Se o tempo de espera ultrapassar um valor *x*, a transação é abortada independente de ter ocorrido o deadlock ou não.

2.3.2 Detecção e Recuperação de Deadlock

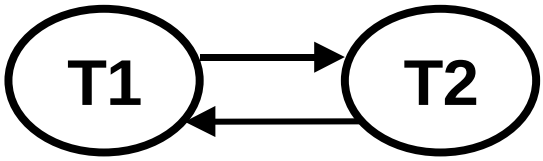
As técnicas de detecção e recuperação são utilizadas quando o subsistema de controle de concorrência não possui nenhum mecanismo de prevenção de *deadlock*. Normalmente a detecção é feita pela geração de um grafo de espera, onde a ocorrência de ciclos indica a presença de *deadlock*.

Para recuperar do *deadlock* é necessário que uma ou mais transações sejam selecionadas como vítimas para serem abortadas. Para a seleção das vítimas podem ser utilizados critérios como, por exemplo, a transação mais recente, o quanto falta para uma transação terminar, quantas atualizações a transação realizou ou até mesmo nenhum critério (por facilidades de implementação).

Um problema é decidir quando o algoritmo de detecção deve ser acionado. Se os intervalos forem curtos, muito *overhead*. Se forem intervalos longos, pode-se levar muito tempo para verificar que um deadlock ocorreu.

Grafo de espera:

- é criado um nó para cada transação do sistema
- se Ti está esperando por um dado utilizado por Tj, é criada uma borda Ti $\Leftarrow$ Tj
- quando o dado é liberado, a borda é removida
- ocorre um dealock quando o grafo contiver um ciclo



2.4 Granularidade Múltipla

Os exemplos de bloqueios apresentados até agora se referiam a bloquear um registro por vez. Existe, porém, situações em que é vantajoso bloquear diversos itens de dados, tratando-os como uma unidade de acesso do que bloquear um registro. Por exemplo:

- atualização de todos os dados de um arquivo;
- leitura de todos os dados do banco de dados;

Nessas situações é mais interessante bloquear todo o arquivo, pois em uma única solicitação todo o arquivo estará bloqueado e o tempo necessário para realizar os bloqueios (de cada registro) é evitado. Em contrapartida, se a transação necessita de apenas um registro, bloquear todo o arquivo é desnecessário e, também, elimina a concorrência.

É importante, então, que o sistema permita definir *múltiplos níveis de granularidade* de bloqueio. A granularidade de bloqueio é, então, a porção de itens de dados que pode ser bloqueada por uma transação. As granularidades mais usuais são:

- registro (tupla);
- página;
- arquivo (tabela).

Isso pode ser implementado através de uma *árvore de granularidade*, onde cada nodo representa a porção de dados (grão) que está sendo bloqueada e existe uma hierarquia entre os nodos. Observe a Figura 2-4:

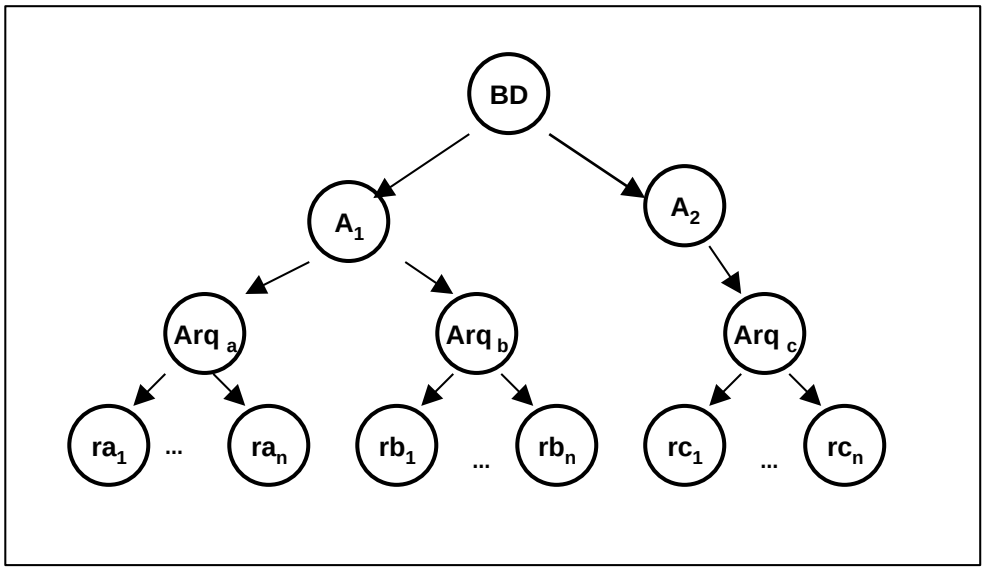


Figura 2-4: Árvore de Granularidades

Um banco de dados (BD) pode ser dividido em áreas (A). Cada área pode ser dividida em um ou mais arquivos e cada arquivo possui um ou mais registros.

Princípio do bloqueio de grãos:

- quando uma transação bloqueia um determinado grão (ou nodo), ela bloqueia também todos os nodos filhos deste grão no mesmo modo de bloqueio;
- quando uma transação bloqueia um grão, todos os seus ancestrais devem ser bloqueados intencionalmente no mesmo modo de bloqueio.

Modos de bloqueio:

- compartilhado (C): o nodo e toda a sua subárvore estão bloqueados explicitamente no modo compartilhado;
- exclusivo (E): o nodo e toda a sua subárvore estão bloqueados explicitamente no modo exclusivo;
- compartilhado intencional (CI): um nodo com este bloqueio significa que algum bloqueio compartilhado explícito está sendo mantido na sua subárvore;
- exclusivo intencional (EI): um nodo com este bloqueio significa que algum bloqueio exclusivo explícito está sendo mantido na sua subárvore;
- compartilhado e exclusivo intencional (EI): um nodo está bloqueado explicitamente no modo compartilhado e algum nodo na sua subárvore está bloqueado explicitamente no modo exclusivo;

	CI	**EI**	**C**	**CEI**	**E**
CI	V	V	V	V	F
EI	V	V	F	F	F
C	V	F	V	F	F
CEI	V	F	F	F	F
E	F	F	F	F	F

Regras:

1) respeitar a matriz de compatibilidade dos modos de bloqueio
2) a raiz da árvore precisa ser bloqueada primeira e pode ser bloqueada em qualquer modo
3) um nodo n pode ser bloqueado por Ti no modo C ou CI apenas se os pais de n estão correntemente bloqueados por Ti no modo EI ou CI

4) Um nodo n pode ser bloqueado por Ti no modo E, CEI, ou EI apenas se os pais de n estão correntemente bloqueados por Ti no modo EI ou CEI
5) Ti pode bloquear um nodo apenas se ele não desbloqueou nenhum nodo antes (segue a técnica de bloqueio de duas fases)
6) Uma Ti pode desbloquear um nodo apenas se nenhum dos filhos estiver bloqueado por Ti.

O protocolo de granularidade múltipla exige que os bloqueios sejam feitos de cima para baixo (top-down – da raiz para as folhas), enquanto a liberação deve ser de baixo para cima (bottom-up – das folhas para a raiz).

Esse protocolo aumenta a concorrência e reduz o overhead por bloqueio. Isso é particularmente útil em aplicações que misturam:

- Transações curtas que mantêm acesso em poucos itens de dados;
- Transações longas que produzem relatórios a partir d eum arquivo ou de um conjunto de arquivos.

Os seguintes fenômenos podem ocorrer:

- **leitura suja (dirty reads)**: ocorre quando uma transação tem acesso aos dados modificados por uma transação que ainda não concluiu, ou seja, a transação está acessando dados intermediários.
- **Leituras não repetidas (Nonrepeatable Read)**: A TA faz a mesma consulta em vários momentos e encontra valores diferentes que foram modificados ou deletados.
- **Leitura fantasma (Phantom reads)**: A TA faz a mesma consulta com uma determinada condição que retorna um conjunto de valores, esta mesma consulta é executada novamente e retorna mais linhas, que foram adicionadas por TAs commited e que satisfazem a condição.

Bibliografia utilizada no capítulo:

[BER 87] BERNSTEIN, Philip A; HADZILACOS, Vassos; GOODMAN, Nathan. **Concurrency Control and Recovery in Database Systems**. Reading:Addison-Wesley, 1987.

[CON 98] CONNOLLY, Thomas; BEGG, Carolyn. **Database Systems – a practical approach to design, implementation and management.** Harlow:Addison-Wesley, 1998.

[IOC 95] IOCHPE, Cirano. **Notas de Aula CMP97 – Sistemas de Banco de Dados.** Curso de Pós-Graduação em Ciência da Computação, 1995.

[SIL 99] SILBERSCHATZ, Abraham.; KORTH, Henry F.; SUDARSHAN, S. **Sistema de Banco de Dados.** 3ª edição., São Paulo: Makron Books, 1999.

3 Recuperação Após Falhas

Diversas falhas podem ocorrer em um sistema de computador como, por exemplo, queda de energia, falha na unidade de armazenamento físico, falha no programa de aplicação. Tais falhas podem tornar os dados armazenados no banco de dados inconsistentes, fazendo com que testes e procedimentos sejam incorporados ao SGBD para tratamento dessas falhas. O módulo do SGBD responsável por essa tarefa é chamado de recuperação após falhas ou recovery.

O objetivo do subsistema de recovery é levar o banco de dados a um estado consistente após uma falha que o tenha deixado em um estado não consistente. Para que isso ocorra, todo o sistema de recovery está baseado na redundância de informações.

Essa redundância normalmente é obtida a partir das seguintes hierarquias de armazenamento:

- Arquivos de log: correspondem a arquivos que mantém um histórico das transações executadas. Contém as informações necessárias para reconstruir o estado mais recente do *database buffer* e, normalmente, estão armazenados em disco. Suporta falhas de sistema;
- Archive log: é uma cópia de um estado consistente do banco de dados. Suporta falhas de mídia (disco). É armazenado em discos ou fitas magnéticas.
- Arquivos de backup: correspondem às cópias de segurança. Podem estar armazenadas em disco ou fita magnética e armazenam o conteúdo de um banco de dados consistente em um dado período do tempo.

3.1 Projeto de um Subsistema de Recovery

O projeto de um subsistema de recovery deve levar em consideração:

- Meios de armazenamento de dados;
- Tipos de falhas e como estas falhas afetam a integridade dos dados armazenados;
- Procedimentos para recuperação destas falhas.

Os meios de armazenamento podem ser divididos em:

- Meio de armazenamento volátil: caracteriza-se pela perda dos conteúdos armazenados em caso de falta de energia. Por exemplo, a memória principal;
- Meio de armazenamento não volátil (físico): caracteriza-se por manter o conteúdo armazenado, mesmo em caso de falta de energia. Exemplo: memória secundária (discos e fitas magnéticas).

Com relação às falhas podemos caracterizar três tipos básicos:

- Falha de transação: erro detectado pela própria transação. Exemplo: overflow, divisão por zero, violação de proteção de memória. Qual o meio de armazenamento comprometido? Nenhum! Qual a ação a ser executada? Desfazer a transação.
- Falha de sistema: interrupção do sistema. Exemplo: queda de luz, falha no sistema operacional. Meio de armazenamento comprometido: volátil. Ação: desfazer as transações em execução no momento da falha que não foram concluídas.

- Falha no meio físico: perda total ou parcial do meio físico. Exemplo: falha no cabeçote de gravação do disco, erro de hardware. Meio de armazenamento comprometido: físico. Ações: acionar a última cópia de segurança e refazer transações executadas com sucesso após a última cópia.

A partir dos tipos de falhas acima apresentados é possível identificar quatro ações a serem implementadas pelo mecanismo de recovery:

- *transaction UNDO*: desfaz os efeitos de uma única transação, garantindo a atomicidade (ou a transação é concluída com sucesso e seus efeitos são materializados no banco de dados, ou é como se a transação nunca tivesse existido); - Falha de Transação
- *global UNDO*: desfaz os efeitos de todas as transações atualmente sendo executadas, garantindo a atomicidade de um conjunto de transações; - Falha de sistema
- *partial REDO*: refaz os efeitos de um conjunto de transações que terminaram com sucesso (committed), os quais talvez não tenham sido salvos no BD; garante durabilidade - Falha de sistema
- *global REDO*: refaz os efeitos de um conjunto de transações que terminaram com sucesso (committed), independente do meio onde estes efeitos estejam armazenados. – Falha de disco

A implementação das operações acima depende, no entanto, da forma como alguns aspectos do SGBD são implementados. Quatro aspectos devem ser analisados:

- tipo de propagação dos dados: como os dados são transferidos do buffer de dados para o banco de dados;
- gerência dos buffers: como determinar que buffers de dados serão liberados para outros dados;
- tratamento de EOT (end-of-transaction): quando uma transação termina com sucesso que tratamento recebem os dados que ainda estão nos buffers de dados;
- checkpoints: técnica utilizada para reduzir o esforço de recovery. Alguns sistemas permitem sua implementação.

3.2 Procedimentos para Recuperação de Falhas Baseados em LOG

Periodicamente é realizada uma cópia do banco de dados. Cada mudança feita no banco de dados por uma transação fica registrada no arquivo de LOG (histórico).

A estrutura do LOG é composta por três registros:

<Início Ti>
<Ti, nome arquivo, id-registro, atributo, valor antigo, valor novo>
<Fim Ti>

No caso de uma falha, é executado o seguinte algoritmo:

1. Aloca duas listas: lista-UNDO e lista-REDO
2. Percorre o arquivo de LOG do fim para o início até chegar ao primeiro registro de checkpoint, examinando cada registro:
   2.1. se achou <Fim Ti>, adiciona Ti na lista-REDO
   2.2. se achou <Inicio Ti> e Ti não está na lista-REDO, adiciona Ti na lista-UNDO
3. Quando se chega ao registro checkpoint, para cada transação Ti na lista de transações desse registro:
   3.1. se Ti não estiver na lista-REDO, então adiciona Ti na lista-UNDO
4. Percorre o arquivo de LOG do fim para o início:
   4.1. realizando UNDO(Ti) para todas as transações Ti existentes na lista-UNDO
   4.2. marcando na lista-REDO as transações Ti cujos registros <Início Ti> estão sendo encontrados nessa varredura
5. caso todas as transações existentes na lista-UNDO tenham sido desfeitas e ficou alguma transação Ti não marcada na lista-REDO;
   5.1. continua percorrendo o arquivo de LOG para trás até que todos os registros <Inicio Ti> das transações não marcadas na lista-REDO tenham sido encontradas
6. Percorre o arquivo de LOG para a frente, realizando REDO(Ti) para todas as transações existentes na lista-REDO.

Esse algoritmo evita a varredura e a verificação de todas as transações existentes em todo o arquivo de LOG. A operação de REDO deve ser idempotente, ou seja, a execução do REDO(Ti) diversas vezes deve ser equivalente a executá-la apenas uma única vez!

Exemplo: Suponha um ambiente bancário multiusuário e um arquivo de LOG com as seguintes informações num dado instante:
<início T1>
<T1, Conta, c1, 1000, 500>
<início T2>
<início T3>
<T2, Conta, c2, 3000, 3500>
<fim T1>
<início T4>
<T3, Conta, c3, 1500, 1200>
<T2, Conta, c6, 500, 100>
<T4, Conta, c8, 200, 0>
<fim T3>
<início T5>
<T4, Conta, c9, 500, 600>
<Inicio T6>
<T5, Conta, c5, 100, 260>
<fim T5>
<T2, Conta, c7, 700, 850>
<T6, Conta, c8, 200, 550>

Supondo que a técnica de modificação imediata do banco de dados esteja sendo

utilizada:

a) que ações devem ser realizadas pelo subsistema de recovey caso ocorra:

1) uma falha da transação T4;

2) uma falha de sistema;

b) se houvesse um registro de checkpoint imediatamente após o registro <início T4>, que ações deveriam ser realizadas se ocorresse uma falha de sistema?

Antes da transação ser efetivada no banco de dados, ela deve estar gravada no LOG físico (WAL – Write Ahead Log).

Como as modificações realizadas pela transação estão armazenadas no LOG, pode-se retardar o momento em que as mesmas serão transferidas para o banco de dados como, por exemplo, no momento da seleção de uma vítima. Em caso de falha de sistema, é necessário fazer o *partial redo* para recuperar as transações *committed* no momento da falha e o *global undo* para desfazer o efeito das transações que não haviam sido concluídas.

No segundo caso, as modificações realizadas por uma transação são efetivadas no banco de dados no momento em que a transação está sendo executada (imediato). O procedimento WAL continua valendo, ou seja, primeiro a informação é armazenada no arquivo de log. A implementação da política *force* pode gerar problemas no processamento de transação, pois o *overhead* no sistema é maior.

3.2.1 Checkpoints

Checkpoints são pontos de verificação que garantem que até aquele ponto os conteúdos dos buffers de LOG e do banco de dados foram descarregados nos respectivos meios físicos. Os checkpoints são executados periodicamente pelo sistema de recovery e tem por objetivo reduzir o esforço de recovery.

Os seguintes passos são executados quando da ocorrência de um checkpoint:

- o buffer de LOG é descarregado para o arquivo de LOG;
- o buffer de dados é descarregado para o banco de dados físico;
- um registro de checkpoint é gravado no arquivo de LOG: <checkpoint <t1, t2, ..., tn>>

Bibliografia utilizada no capítulo:

[BER 87] BERNSTEIN, Philip A; HADZILACOS, Vassos; GOODMAN, Nathan. **Concurrency Control and Recovery in Database Systems**. Reading:Addison-Wesley, 1987.

[SIL 99] SILBERSCHATZ, Abraham.; KORTH, Henry F.; SUDARSHAN, S. Sistema de Banco de Dados. 3ª edição., São Paulo: Makron Books, 1999.

4 Segurança em SGBDs

O termo *segurança*, em banco de dados, refere-se à proteção do banco de dados contra acessos intencionais ou não intencionais utilizando controles baseados em computador ou não [CON 98]

São considerados acessos intencionais aqueles realizados propositadamente, por exemplo, um usuário do sistema cede sua senha a pessoas não autorizadas. Acessos não intencionais são aqueles que, ao ocorrer, causam algum tipo de perda (por exemplo, perda da consistência do banco de dados ou perda de informação), mas não foram propositados. Exemplo, queda de energia que corrompe o banco de dados.

Segundo [CON 98], o contexto de segurança pode ser analisado segundo as seguintes situações:

- roubo e fraude de informação;
- perda de confiabilidade;
- perda de privacidade;
- perda de integridade;
- perda de disponibilidade.

Os controles que podem ser implementados podem ser baseados em computador ou não. Os controles não baseados em computador geralmente estão associados a políticas e planos de segurança estabelecidos pela organização. Exemplo: política para cessão de contas de usuários, os usuários devem trocar suas senhas mensalmente e não devem deixá-las registradas em locais de fácil acesso.

Os controles baseados em computador são agrupados nas seguintes categorias:

- *autorização*: refere-se a concessão de um direito ou de um privilégio a um usuário (ou a um programa) a acessar legitimamente o sistema ou um objeto do sistema;
- *visões*: é um mecanismo que permite estabelecer porções de dados que podem ser visualizados por um determinado usuário (ou programa);
- *backup e recuperação de falhas*: refere-se ao mecanismos necessários para garantir a disponibilidade do sistema em caso de falhas;
- integridade: refere-se aos controles que contribuem para manter a segurança do sistema evitando que dados inválidos sejam registrados no sistema (ver Capítulo 2);
- criptografia: refere-se à codificação dos dados a partir de um algoritmo especial que torna os dados impossibilitados de serem lidos sem que se tenha a chave de criptografia. Esse tópico não faz parte do escopo da disciplina;
- auditorias: tem por objetivo verificar os acessos que são realizados sobre o sistema e observar se os acessos realizados seguem as políticas de segurança propostas. Esse tópico não faz parte do escopo da disciplina.

4.1 Autorização de Acesso

Os mecanismos de autorização envolvem duas abordagens: autenticação e autorização de acesso. A primeira tem por objetivo verificar se o usuário que tenta acessar o sistema é quem realmente diz ser. Em SGBDs, a autenticação é realizada através de senhas embora este recurso não seja totalmente garantido. O mecanismo de autorização permite conceder privilégios para cada um dos objetos aos usuários. Desse modo, apenas os usuários que têm privilégio sobre o objeto podem acessá-lo.

Em SQL existe um conjunto de comandos DDL que permite criar usuários e conceder privilégios.

4.1.1 Criando usuários

Os usuários são criados pelo administrador do sistema (DBA) e, geralmente, é o administrador do sistema que concede os privilégios que esse usuário terá.

Exemplo:

```
CREATE USER ALUNO IDENTIFIED BY ALUNOSENHA;
```

A execução do comando acima criará um usuário chamado ALUNO, cuja senha é ALUNOSENHA.

4.1.2 Concedendo/Revogando privilégios de acesso

O fato de criar um usuário não significa que ele terá acesso aos objetos do sistema. Em SQL, para que um usuário tenha acesso aos objetos do esquema ele deve receber, explicitamente, esse privilégio. Caso ele deixe de ter acesso aos objetos do esquema, os direitos deverão ser revogados, também, explicitamente.

Quem pode conceder/revogar privilégios? A resposta é o proprietário do esquema já que, inicialmente, apenas ele tem conhecimento da existência do objeto.

Que privilégios podem ser concedidos/revogados? Em SQL, os privilégios que podem ser atribuídos a objetos são:

- SELECT: permite recuperar dados de uma tabela;
- INSERT: permite inserir novas tuplas em uma tabela;
- UPDATE: permite modificar tuplas de uma tabela;
- DELETE: permite remover tuplas de uma tabela;
- EXECUTE: permite executar uma stored procedure;
- REFERENCES: permite referenciar colunas de uma tabela já existente (de outro esquema) em restrições de integridade.

O comando SQL para conceder privilégios a um usuário é GRANT.

Sintaxe:

```
GRANT <privilégios> ON <nome tabela ou visão> TO <usuários>;
```

Exemplo: autorização de leitura e atualização dos atributos da tabela Funcionário para o usuário Pedro.

```
GRANT SELECT, UPDATE ON FUNCIONARIO TO PEDRO;
```

Para revogar privilégios:

```
REVOKE <privilégios> ON <nome tabela ou visão> FROM <usuários>;
```

Exemplo: retirar a autorização de atualização sobre a tabela Funcionário do usuário Pedro.

```
REVOKE UPDATE ON FUNCIONARIO FROM PEDRO;
```

4.1.3 Roles

Como você pode observar, para cada objeto do esquema deve ser definido quais usuários terão direito sobre ele e que tipo de direito terão. Alguns SGBDs oferecem alguns recursos para facilitar a administração do sistema, as *roles*. Uma *role* (papel) consiste de um conjunto de privilégios que são definidos sobre uma tabela e que podem ser atribuídas a um usuário, facilitando a tarefa de concessão de privilégios.

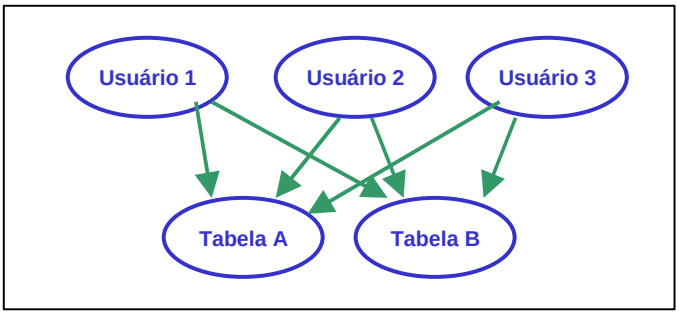


Figura 4-1: Usuários e privilégios sobre tabelas

A vantagem na utilização de *roles* é que se novas tabelas precisam ser acrescentadas, basta atribuir isso a *role* criada e todos os usuários passarão a ter esse privilégio. Além disso, se um novo usuário é acrescentado ao sistema, é necessário atribuir a ele apenas a role apropriada.

O uso de roles é muito similar ao uso dos privilégios. Primeiro de tudo é necessário criar a role. Exemplo, definir um papel (*role*) que permita a recuperação de dados e a atualização de dados da tabela Funcionário e atribuir ao usuário Pedro:

```
CREATE ROLE RL_FUNCIONARIO;
```

Criada a *role*, atribui-se a ela os privilégios sobre os objetos:

```
GRANT SELECT, UPDATE ON FUNCIONARIO TO RL_FUNCIONARIO;
```

Finalmente, atribui-se a role ao usuário;

```
GRANT RL_FUNCIONARIO TO PEDRO;
```

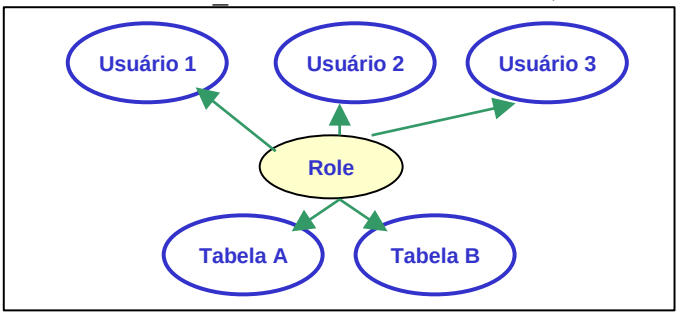


Figura 4-2: Usuários, privilégios sobre tabelas e *roles*

Também é possível atribuir uma *role* a outra *role*. Por exemplo, se agora fosse criada uma *role* RL_DEPENDENTES que permite selecionar, inserir e atualizar Dependentes de um Funcionário e que o usuário Pedro pode executar essas operações.

Pode-se fazer isso de duas formas:

```
GRANT RL_FUNCIONARIO, RL_DEPENDENTES TO PEDRO;
GRANT RL_DEPENDENTES TO RL_FUNCIONARIO;
```

4.1.4 Sinônimos

Um outro recurso oferecido pelo SGBD ORACLE é o de sinônimo. Um sinônimo tem por objetivo oferecer transparência no acesso aos objetos. Quando um objeto é acessado por um usuário que não é o proprietário ele deve ser referenciado pelo [nome_do_proprietário**.**nome_do_objeto].

O sinônimo cria um apelido para um objeto:

```
CREATE PUBLIC SYNONYM FUNCIONARIO FOR RH.FUNCIONARIO;
```

Foi criado um sinônimo público para a tabela Funcionário, que pertence ao esquema RH, chamado Funcionário. A partir da criação do sinônimo todos os usuários que tiverem privilégio sobre a tabela Funcionário poderão referenciá-la como Funcionário, apenas.

Esse capítulo apresentou alguns recursos que são oferecidos pelo SGBD Oracle, mas que podem estar presentes em outros SGBDs, também. Embora alguns desses recursos não estejam previstos na linguagem SQL, estes ainda assim foram apresentados para que você faça uma melhor utilização do SGBD nos exercícios práticos.